UNIVERSITÀ DEGLI STUDI DI FERRARA

Dottorato di Ricerca in
Ingegneria dell'Informazione

XIII Ciclo

Sede Amministrativa: Università di Modena e Reggio Emilia
Sedi Consorziate: Università di Ferrara

Solving Combinatorial Optimization Problems in Constraint Programming

Filippo Focacci

Il Coordinatore	**I Tutori**
Prof. Claudio Canali	Prof. Paola Mello
	Prof. Cesare Stefanelli

AA. AA. 1997–2000

A Franca e Carlo.

Contents

Acknowledgments

This thesis was carried on partially at the University of Ferrara and partially at ILOG S.A., Paris. The close relation with the Artificial Intelligence and Operations Research groups at the universities of Ferrara and Bologna, and Consulting and Research and Development groups at ILOG gave me the unique opportunity to work in very diverse environments, in contact with people, from whom I have never stopped learning. Always very challenging, and sometimes even stressful, working between Ferrara, Bologna and Paris, with some of the most important scientists from university and industry has influenced every aspect of my thesis and of my interpretation of Constraint Programming and Optimization. I am extremely grateful to everyone who made this collaboration possible: first of all to Paola Mello, Michela Milano and Jean Pommier who originally started this project. A particularly warm thank you to Jean Pommier because he gave me this great opportunity almost without knowing me, and never denied his support. I'm also very grateful to Cesare Stefanelli, Claude Fornarino, and Daniel Godard for their continuing support following the departure of Paola and Jean.

I would like to thank everyone in the Consulting and R&D group at ILOG for the many discussions that directly or indirectly greatly contributed to this thesis. The help and support I received from everyone is such that it would be too difficult to make a list.

I would like to thank Heinrich Braun, chief of the APO/OPT development, together with the whole R&D group from SAP because they gave me the opportunity to work on such an ambitious project, and to validate on this project many of the ideas presented here.

This project couldn't have been possible without the support of Paolo Toth and the entire Operations Research group at the University of Bologna, whose work inspired me many times, and who gave me invaluable help during the three years of research.

I would also like to honorably mention the following in assistance with my thesis: Chris Beck, Amedeo Cesta, Philippe Couronne, Bruno De Backer, Vincent Furnon, Marco Gavanelli, Daniel Godard, Ulrich Junker, Philippe Laborie, Francois Laburthe, Evelina Lamma, Daniel Min-Tung, Jean Francois Puget, Philippe Refalo, Luca Scarano, Paul Show, and Francis Sourd. A particular thank you to Wim Nuijten and Jerome Rogerie.

Finally, I would like to thank Michela Milano and Andrea Lodi who collaborated with me in most of my research projects.

List of Figures

List of Tables

Introduction

Optimization Problems deal with the minimization or maximization of a function of many variables subject to a set of constraints. These problems arise in an enormous variety of areas such as industry, logistics, finance, transportation, configuration, etc. Since the 60s, linear and mixed-integer programming technologies [44] have been employed with tremendous success for solving hundreds of problem types. Combinatorial Optimization Problems (COPs) deal with optimization problems where some or all variables are constrained to be integer. This class of problems is both extremely important and difficult to solve.

The Operations Research (OR) approach for solving hard combinatorial optimization problems was the only one used for a long time. It is based on a mathematical representation of the problem, which is typically modeled as an integer linear program where (integer) variables are linked by linear equalities and inequalities.

In recent years, a new programming paradigm evolved within the Computer Science (CS) and Artificial Intelligence (AI) community: Constraint Programming (CP) (a book on the subject has been recently published by Marriott and Stuckey [113], while for a survey see Jaffar and Maher [98]), has been successfully used for modeling and solving combinatorial optimization problems such as scheduling, planning, sequencing and routing problems [51, 52, 88, 161]. The seminal works on constraint-based languages can be traced back in the late 70s (ALICE [107] and CONSTRAINTS [156]), but the success of Constraint Programming is probably due to its introduction in the field of Logic Programming [110]. In the late 80s, Jaffar and Lassez [97] defined the Constraint Logic Programming (CLP) paradigm as a general scheme that can be specialized on different domains. CLP is a powerful programming paradigm combining the advantages of Logic Programming and the efficiency of constraint solving. More recently, constraint-based languages have been developed independently from Logic Programming (see, for instance, ILOG Solver [138, 139] which is based on C++), but maintaining a declarative programming style. More generally, in the following, we refer to Constraint Programming (CP) tools.

The Constraint Programming approach for combinatorial optmization problems models a problem as a set of variables taking their values on a finite domain of integers and linked by a set of constraints that can be mathematical or symbolic. Constraints embed constraint propagation algorithms that effectively reduce the search space by removing those combinations of variable-value assignments that are proven to be infeasible. CP has shown to be a highly effective technology using constraint propagation to efficiently solve problems that are highly combinatorial with highly logical content. The constraint programming methodology allows the natural expression of very complex relationships, including logical expressions. It is ideally suited for operational problems, which require fast, feasible answers. Users can also guide the search process with their own knowledge of the problem.

The industrial success of CP has, little by little, caught the attention of many OR researchers for this technology, while some clear failures of the first CP approaches w.r.t. pure OR ones has brought back to earth many AI researchers and pushed them to better understand the pros and cons of the two technologies. Currently, an increasing number of researchers, both from the OR and the AI communities, is investigating the possibility of integrating methodologies of the two fields for solving combinatorial optimization problems. Situated at the confluence of the two fields, Constraint Programming is an emerging discipline that has been recognized as a suitable environment for achieving such an integration.

At least two main and complementary streams for the integration of Constraint Programming and Operations Research can be considered:

- an algorithmic approach: several studies follow this stream on different directions. An incomplete list of direction is the following. OR methods can be used to remove uninteresting (infeasible or suboptimal) values from the domain of variables within a CP framework. Logical methods can be used to deduce valid inequalities within a Branch and Cut framework. Local search algorithms can be used within a CP framework to speed up the convergence towards good solutions. Consistency algorithms can be used to fix some variables in an Integer Linear Program (ILP) during a preprocessing step.

- an engineering approach: during the invited speech at the first international workshop on the integration of AI and OR techniques in CP for combinatorial optimization problems (CP-AI-OR'99), Jean-Francois Puget defined Constraint Programming as *software engineering applied to operations research.* New languages such as OPL [159] and Concert [93] propose a clear separation between problem modeling and problem solving; in these languages the model does not imply a solving technology, but it is just a description of the problem. To this purpose, a unique modeling language is defined independently on the solving technology. In general, software engineering practice can be applied to pure OR approaches leading to more flexible and modifiable code.

Concerning the algorithmic approach, this thesis proposes the integration of OR methods within *global optimization constraints.* The CP and OR approaches for combinatorial optimization problems are complementary: CP is based on the theory of constraint satisfaction to reduce the search space by removing the possibility of assigning combinations of values to variables that would lead to inconsistency with respect to the constraints. OR methods, by relaxing some constraints, define a new problem (relaxed problem) that can be optimally solved. The value of the optimal solution of the relaxed problem represents an optimistic evaluation of the optimal solution of the original problem, and it can be used to reduce the search space. Therefore, while in CP the reduction of the search space is obtained by reasoning on the feasibility of configuration of values, in OR it is obtained by reasoning on the optimality.

This thesis proposes a methodology to design *optimization constraints* integrating the two reduction techniques. This integration improves the performance of CP systems in optimization problems, and provides global constraints that can be used as software components for many different problems. The optimization constraints reduce the search space by removing values configurations that can be proved to be worse than already found solutions. This reduction is achieved by embedding an optimization component and a filtering algorithm in a CP global constraint. The optimization component calculates a *lower bound* (for minimization problems) and a gradient function. The lower bound gives an optimistic evaluation of the optimal solution of the problem, while the gradient function evaluates how much the lower bound increases if a given value is assigned to a given variable. Some of the OR techniques used to calculate lower bounds for global constraints are linear programming, cutting planes, polynomial algorithms for structured problems, and dynamic programming. Gradient functions have been defined using sensitivity analysis methods (e.g. reduced costs), and dynamic programming.

Optimization constraints have been used in many well-studied combinatorial optimization problems such as the Traveling Salesman Problem (TSP) and its time windows variant (TSPTW), timetabling problems, scheduling problems with sequence dependent setups, and min sum tardiness scheduling problems with release dates.

Concerning the engineering approach, this thesis proposes some extensions to the ILOG optimization libraries. Constraint Programming has shown to be a very successful tool for solving combinatorial optimization problems. It offers a variety of modeling and solving facilities; one of its major strengths consists in allowing the development of problem dependent search methods. Unfortunately, in some complex application, the type of optimization problem and the objective functions are known only at run time, and not during the development phase. This type of applications aims at solving a *class of optimization problems* instead of a *single optimization problem*. The extensions proposed add to the ILOG optimization libraries ideas and data structures, whose purpose is to give basic building blocks to handle this type of complex applications. The extensions proposed originate from the experience gained during the participation (together with the ILOG consulting group and the SAP research and development group) to the development of the Constraint based engine of the Detailed Scheduling (PPDS) module of SAP APO (Advanced Planning and Optimization). Most of the ideas presented in Chapter 5 have been inspired and validated on this project. Although examples throughout this chapter are focused on scheduling problems, the described concepts could easily be applied to a variety of optimization applications.

The first chapter of this dissertation provides the necessary background for the remainder of the thesis; combinatorial optimization problems are introduced together with a brief introduction of the solving methods that contributed to the research proposed. In particular, *Branch and Bound* methods based on Constraint Programming and Mathematical Programming are detailed.

Chapter 2 reviews the main directions of integration of Constraint Programming and Operations Research proposed in the literature.

Chapter 3 describes a contribution of this research for the integration of Constraint Programming and Operations Research with respect to the algorithmic approach: *global optimization constraints* are proposed embedding an optimization component and a filtering algorithm.

Constraint Programming based solvers using global optimization constraints are proposed for five types of combinatorial optimization problems in Chapter 4. The computational study demonstrates the advantages of the proposed approach.

Chapter 5 discusses several extension to the Constraint Programming framework for applications aiming at solving a *class of optimization problems* instead of a *single optimization problem*.

Finally, Chapter 6 describes three studies under development: symmetric combinatorial optimization problems, enhancement of the *edge finding* [125] algorithm for scheduling problems with sequence dependent setup times, and a new model for representing scheduling problems in Constraint Programming.

Chapter 1

Preliminaries

1.1 Combinatorial Optimization Problems

1.1.1 Optimization Problems

Many problems with important practical applications are concerned with the selection of a "best" configuration to achieve some objective criteria. Such problems are generally referred to as optimization problems, and can be stated as

$$min\{f(x) \,|\, x \in \mathcal{F} \subset \Re^n\} \tag{1.1}$$

where $x \in \Re^n$ is the vector of *problem variables*, $\mathcal{F}$ is the *feasible region* (the set of all feasible solutions), and $f : \mathcal{F} \to \Re$ is the *objective function*. Every $x \in \mathcal{F}$ is called a *feasible solution* to (1.1). If there is an $x^* \in \mathcal{F}$ satisfying

$$f(x^*) \le f(x), \forall x \in \mathcal{F}$$

then x^* is called the (global) optimal solution and $f(x^*)$ is called the (global) minimum with regard to (1.1).

Equivalently, an optimization problem can be stated as follows:

$$\begin{aligned} & min\ f(x), && \\ & subject\ to && \\ & g_i(x) \ge 0, && i = 1, \ldots, m && (1.2) \\ & h_j(x) = 0, && j = 1, \ldots, p && (1.3) \end{aligned}$$

where $g_i,\ h_j$ are functions $\Re^n \to \Re$, and represent the *constraints* of the optimization problem.

1.1.2 Combinatorial Optimization Problems

When an optimization problem has a finite number of candidate optimal solutions (i.e., the optimum can be found by comparing a finite number of solutions), the problem is defined *Combinatorial Optimization Problem.* Some important classes of Combinatorial Optimization Problems that will be extensively considered are the following:

- *Linear Program* (LP). A combinatorial optimization problem is a LP if f, g_i, h_j are linear functions.

- *Finite Domain Optimization Problem.* A combinatorial optimization problem is a Finite Domain problem if x *integer*, and $x_i \in [a_i, b_i]$, $i = 1, \ldots, n$.

- *Integer Linear Program* (ILP). A combinatorial optimization problem is a ILP if the objective function f is linear, and exactly one non linear constraint exists imposing x *integer*.

Linear Programs define a special class of combinatorial optimization problems; LP are very easy to solve, and are usually solved using standard well-known algorithms (e.g. the simplex method). In general, Finite Domain Optimization Problem, and Integer Linear Program are a lot harder. In the following, we will consider Integer Linear Program as a subclass of Finite Domain Optimization Problem. In other words, unbound Integer Linear Programs will not be considered. A general model of these problems involves a set of variables representing problem basic entities; variables can assume a given set of discrete values and are linked by a set of constraints. A *constraint* is a relation among variables that imposes that they assume values that satisfy the constraint itself. These problems are characterized by a finite (even if possibly large) number of solutions. One or more optimization criteria are defined that lead to prefer one solution, i.e., the *optimal solution*, among others. Optimization criteria are expressed through one *objective function*, which associates a value to each solution so as to compare different solutions and chose one. Finally, optimization problems are called either *minimization* problems when the objective function should be minimized, e.g., minimize the cost of the productive process, or *maximization* problems in the other case, e.g., maximize the profit derived by the productive process. Since maximization problems can be transformed in minimization ones by inverting the sign of the objective function, in the following we will only consider minimization problems.

1.1.3 Computational complexity

Combinatorial optimization problems arise in many real life applications, and are in general difficult to solve. The theoretical difficulty of a problem can be defined using the concept of computational complexity. In this section some notions of computational complexity will be given, see [77] for a complete treatment of this theory.

The *size* n of a problem instance I is defined as the number of symbols needed to encode I in a compact way. Given an instance I of a problem of size n, an algorithm is said to have a *time complexity* of $O(g(n))$ if the largest time required to execute the algorithm has an asymptotic upper bound of $k\,g(n)$ where k is a constant, and $g(n)$ is a function of n. An algorithm is a *polynomial-time* algorithm (e.g., $O(n), O(n^3\,log(n))$) if its time complexity is polynomial, otherwise it is *exponential-time* (e.g., $O(n!), O(2^n)$). Let c be the largest integer of a problem instance I. An algorithm is a *pseudo-polynomial* algorithm (e.g., $O(n\ c), O(n^3\ c^2)$) if its time complexity is polynomial on n, and c.

Each combinatorial optimization problem can be transformed into a corresponding decision problem (looking for a *yes*, *no* answer with respect to feasibility). It is easy to show that a decision problem DP is *not more difficult* than the corresponding optimization problem OP. Under the reasonable hypothesis that $f(x^*)$ can be represented with a number of symbols polynomial on the size of the problem, it is also easy to show that an optimization problem OP is *not more difficult* than the corresponding decision problem DP.

A decision problem is said to belong to the class $\mathcal{P}$ if any instance of the problem can be solved by a known polynomial-time algorithm (i.e., it can be solved in polynomial time by a Turing machine). A decision problem is said to belong to the class $\mathcal{NP}$ if any instance of the problem can be solved by a decision tree having known polynomial depth (i.e., it can be solved in polynomial time by a non-deterministic Turing machine). Obviously, $\mathcal{P} \subset \mathcal{NP}$; moreover, all the known decision problems are $\mathcal{NP}$. A decision problem P_A is *polynomially reducible* to a decision problem P_B if there exists a polynomial algorithm that transforms each instance of P_A into an instance of P_B having solution *yes* if and only if the instance of P_A has solution *yes*. This means that if P_B is polynomial then P_A is also polynomial. A problem P_A is $\mathcal{NP}$*-complete* if for every problem $P_B \in \mathcal{NP}$, P_B is polynomially reducible to P_A. A problem P_A $\mathcal{NP}$-complete is *not easier* than any other problem $\mathcal{NP}$, or, in other words, every problem $\mathcal{NP}$ is a special case of P_A. Let P_A^P be the restriction of the decision problem P_A to the subset of instances I for which the largest integer c in I is limited by a polynomial of its size; if P_A^P is $\mathcal{NP}$-complete, then P_A is defined *strongly* $\mathcal{NP}$*-complete.* The combinatorial optimization problem corresponding to a $\mathcal{NP}$-complete decision problem is called $\mathcal{NP}$*-hard.*

In conclusion, problems belonging to the $\mathcal{P}$ class are *easy* in the sense that there exist polynomial algorithms able to solve them; problems not belonging to the strongly $\mathcal{NP}$-complete class can be solved by pseudo-polynomial algorithms, while none of the problems belonging to the strongly $\mathcal{NP}$-complete class can be solved by polynomial or pseudo-polynomial algorithms unless $\mathcal{P} = \mathcal{NP}$. Linear Programs are $\mathcal{P}$, while most (non-linear) combinatorial optimization problems are strongly $\mathcal{NP}$-hard.

1.2 Solving techniques

When solving a combinatorial optimization problem, we can identify two conceptual steps:

- a constraint satisfaction part aimed at finding a *feasible* solution;
- an optimization part aimed at selecting, among feasible solutions, the one minimizing the objective function.

In general, the solution of this class of problems requires the exploration of a *search space.* The search space is represented by all the possible combinations of assignments of values to variables. Clearly, some parts of the search space are not *feasible* since they contain combinations of assignments that do not satisfy the constraints. The search space is explored to find a feasible and optimal solution. We introduce here the concept of *proof of optimality*: when the optimal solution has been found, we may want to prove that it is indeed optimal, this could be obtained, for instance, by exploring the remaining part of the search space demonstrating that a better solution does not exist.

Thus, techniques for *pruning* the search space should be defined. Conceptually, we can identify two kinds of pruning: pruning based on *feasibility* reasoning, and pruning based on *optimality* reasoning. Feasibility pruning (also called constraint propagation) removes combinations of assignments which do not lead to any feasible solution. Optimality pruning, instead, removes combinations of assignments which do not lead to better solutions with respect to the best one found.

A different type of optimality pruning is based on *dominance criteria*: combinations of values could be removed because it can be proven that they would lead to suboptimal solutions. Note that in this case combinations of values are removed because they would lead to a solution worse than others that are not found yet. In order to prove that a configuration is suboptimal w.r.t. a solution that was not found yet, *dominance criteria* based pruning makes strong hypothesis on the optimization problem at hand (i.e. it must know the whole set of constraints defining the problem). One of the characteristic of Constraint Programing is that the pruning algorithms, locally defined in each constraint, must remain globally valid independently on the set of other constraints defining the problem. For this reason, *dominance criteria* methods usually do not apply to CP filtering algorithms, and are not considered in this dissertation.

1.2.1 Exact versus Incomplete Methods

The fact that most combinatorial optimization problems are $\mathcal{NP}$-hard implies that, in general, there is no effective method to find optimal solutions for these problems and to prove optimality. Nevertheless, many combinatorial optimization problems of reasonable size can be solved up to optimality. Whenever optimality is out of reach incomplete methods can be used instead.[1] Incomplete methods are designed to provide *good* solutions but cannot guarantee optimality. They are used to solve approximately and efficiently large problems.

Exact and incomplete methods may share the same search methods. For instance, tree search can be used both in exact methods and for incomplete search; neighborhood search, normally used for incomplete methods, may, in some cases, be also used for exact methods. In the following some commonly used framework for solving $\mathcal{NP}$-hard combinatorial optimization problems will be briefly described.

1.2.2 The Branch and Bound Framework

A discrete combinatorial optimization problem P can be modeled by a set of variables $x_1, \ldots, x_n$, a set of constraints on these variables $c_1, \ldots, c_m$, and an objective function $z = f(x_1, , \ldots, x_n)$. Being D_i the set of values that can be taken by a variable x_i, a constraint c_k can be defined as a subset of the cartesian product of D_i, i.e., $c_k \subseteq D_1 \times D_2 \times \ldots \times D_n$.

In the branch and bound framework, a tree search is used to recursively decompose P in a disjunction of simpler subproblems. For example, suppose x_1 can assume values $1, 2$, or 3, then the optimal solution of P is equal to the best among the optimal solutions of the three simpler problems P_1, P_2, P_3 where $P_i = P$ with $x_1 = 1, 2, 3$. Problem P generated three *branches* $x_1 = i, i = 1, 2, 3$, and three new problems P_1, P_2, P_3. A subproblem P_k is not further decomposed when it is provably infeasible (either because it violate some constraints or because it leads to solutions worse than the current best one), or when all the decision variables are assigned to values, and a feasible solution is found.

This framework has been extensively used both by OR and AI communities; roughly speaking we can say that the OR community pays more attention to the mathematical properties of each subproblem, concentrating on the *node*, while the AI community pays more attention to heuristic decisions to find solutions, concentrating on the *branch*.

[1]In OR incomplete methods are often referred to as *heuristic* methods; we will always call them *incomplete* to avoid confusion with the term "heuristic" in tree search.

In optimization problems every time a feasible solution is found, such a solution imposes to the original problem P the additional constraint that any further solutions must have a better value of the objective function. Suppose there exists a method to know an optimistic evaluation of the value of the optimal solution of a problem P_k, if such a value is worse than the best solution found so far, there is no need solve problem P_k since there is no hope of improvement. The optimistic evaluation of the value of the optimal solution defines a *bound* (a lower bound for minimization problems) for the objective function. The method used to find optimistic evaluation of the optimal solution of a given problem Q is to solve a different problem $R(Q)$ (relaxed problem) having the characteristic that every solution of Q is also a solution of $R(Q)$, and, of course, $R(Q)$ is easier to solve. Since every solution of Q is also a solution of $R(Q)$, the optimal solution of $R(Q)$ cannot be better than the optimal solution of Q. In Section 1.4.2 some widely used methods to generate relaxed problems will be described.

A tree search can be explored in different ways depending on the order in which nodes are considered (see [123, 85] for a detailed description):

- *Depth First Search* is the most common search procedure: at each step there is one current open node P_k; if P_k is an infeasible problem or it does not have any more unexplored children, P_k is closed and a *backtrack* brings the search to the parent node which becomes the current open node; if P_k is a leaf node, a solution is found, the search may continue (to look for better solutions) by closing the node and backtracking to the parent node which becomes the current open node; if P_k has still unexplored children or it can be further decomposed, its leftmost child becomes the current open node.

- *Breadth First Search* at each step i there is a set S^i of current open nodes containing all the nodes of depth i. In step $i+1$, all nodes $P_k \in S^i$, that can be further decomposed, generate all their children, leading to the set S^{i+1}.

- *Best Bound First Search* at each step there is a set S of current open nodes; the node with the best bound is removed from S, and all its children (if any) that are feasible and that can be further decomposed are inserted in S.

- *Discrepancy Based Search* The original discrepancy based search procedure was proposed in [85]. In a binary tree the *discrepancy* of a node P_k is defined as its right depth, that is, the number of times the search has chosen the right branch from the root node P to P_k. Given a parameter d, a discrepancy based procedure explores in Depth First the subtree defined by all nodes with discrepancy less than d. After this exploration is complete, it will explore the subtree defined by all nodes with discrepancy between d and $2d$, etc. This tree search procedure has obtained very good results for many combinatorial optimization problems; the idea is that it is wise to explore nodes that follow the heuristic before nodes that contradict it; since the left branches are those suggested by the branching heuristic, discrepancy based search limits the number of times the right branches can be selected. Many variations on this original procedure have been proposed such as *Depth Bounded Discrepancy Search*, etc. ([85, 17, 131])

1.2.3 The Dynamic Programming Framework

Dynamic Programming (DP) is an optimization procedure that solves optimization problems by decomposing them into a nested family of subproblems. Dynamic Programming is based on the *principle of optimality* ([19, 54]) which states that the optimal sequence of decisions to go from a state A to a state B has the following property: whatever intermediate state C, the remaining sequence of decisions from the state C to B must be the optimal sequence of decision from that state C to B. Dynamic Programming is particularly applicable to problems requiring a sequence of interrelated decisions, and it has been applied to solve a wide variety of combinatorial optimization problems, as well as optimal control problems.

Consider for example the shortest path problem represented in Figure 1.1, which looks for the minimum cost path to go from point A to point B. Denoting as S_X the shortest path from a point X to the end point B, S_A can be defined as $S_A = min\{(1 + S_C), (0 + S_D)\}$; similarly, $S_C = min\{(5 + S_E), (4 + S_F)\}$, etc. The boundary condition for this recursive formulation is $S_B = 0$.

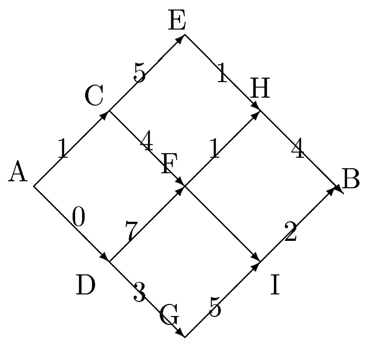

Figure 1.1: Shortest Path problem.

In Dynamic Programming a problem P is associated with a *state space graph* $SG = (S, T)$ where each element of the vertex set S is a state and each element of the arc set T represents a feasible transition from two states. The original problem is solved by solving a shortest path problem in the state space graph from an initial state to a final state (boundary condition)[2]. If the original problem is $\mathcal{NP}$-hard, a correspondent state space graph will have an exponential number of nodes.

Consider for example a Traveling Salesman Problem (TSP). A TSP looks for one tour of minimum cost visiting all nodes in a graph exactly once. A DP formulation for a TSP could be sketched as follows: let V the set of all nodes to be visited labeled from 1 to N. Let define $f_i(j, S)$ = *the length of the shortest path from city* 1 *to city* j *via the set of* i *intermediate cities* S. Clearly if $f_i(j, S)$ is known for every (j, S), the solution of the TSP is also known. The pair (j, S) represents a vertex in the state space graph, and the value of the optimal tour can be calculated by solving a shortest path problem in the state space graph. Clearly the number of vertices (j, S) in the state space graph is exponential, and so is the complexity of solving a TSP via Dynamic Programming.

1.2.4 Cutting Planes

The idea of solving integer (or mixed integer) linear programs by strengthening the initial LP relaxation through the iterative addition of inequalities called *cutting planes* is known in mathematical programming since the seminal work of Gomory [81] in 1958. Intuitively, given

[2]This definition of DP is somehow restrictive, but it is sufficient for the use of DP we make in this paper.

the ILP P, the cutting planes method is based on the *iterative* solution of a sequence of linear programs P_L^i each of them approximating the (non linear) ILP. Each LP P_L^i in the sequence improves the previous approximation P_L^{i-1}. The starting LP is the linear relaxation P_L^0 of P (i.e., the LP obtained by removing the constraint imposing the variables to be integer), the optimal solution of P_L^0, say x^*, provides a lower bound (for minimization problems) $z_{P_L^0} = cx^*$ on the optimal solution of P where, in general, the integrality constraint is violated. The aim of the cutting planes generation is to determine some *linear inequalities* that remove x^*, infeasible for P, without removing any integer solution of P. The problem of finding an inequality $\alpha x \leq \alpha_0$ satisfied by the feasible solutions of P, and such that $\alpha x^* > \alpha_0$, is called *separation problem.* In principle, one can iteratively add all such inequalities up to the definition of the convex hull of the set of solutions of P. However, there is no efficient way of generating the convex hull of a problem, since, if P is $\mathcal{NP}$-hard, the separation problem is $\mathcal{NP}$-hard as well [111].

1.2.5 Greedy constructive methods

A greedy algorithm generates a solution by considering each solution component (e.g. variable) one at a time. The search is guided by a heuristic function h: at each step of the construction process, h is evaluated for all possible choices, and the choice that minimizes h is considered the preferred decision. A greedy constructive algorithm could also be seen as a tree search algorithm where the preferred branch is systematically followed and no backtracking takes place. For pure optimization problems where feasibility is not an issue, such greedy algorithms yield a solution in polynomial time. In case of feasibility issues, the algorithm may produce a partial solution.

1.2.6 Neighborhood methods

A neighborhood search (or local search) method begins with a solution and successively improves it by modifying the solution itself. Local Search (LS) techniques are based on a simple and general idea. Let P be the combinatorial optimization problem we want to solve, and s a current solution which we assume to be feasible for P, and to have value $f(s)$. A *neighborhood* is defined for s with respect to a *move type* $\mathcal{N}$, i.e., a *function* mapping any solution s in a subset $\mathcal{N}(s)$ of the overall solution space. In other words, $\mathcal{N}(s)$ contains all the feasible solutions of P which can be reached from s by means of a move of type $\mathcal{N}$. Roughly speaking, the move is a manipulation of s whose effect is the transition to another solution $x \in \mathcal{N}(s)$. The LS framework explores the neighborhood by searching for the solution $x \in \mathcal{N}(s)$ such that $\delta f(x) = f(s) - f(x)$ is maximized (for minimization problems). If $\delta f(x) > 0$, then an improved solution has been found, and the process is iterated by considering x as new current solution. Otherwise, a local optimum has been reached, and several very effective techniques can be applied to escape from it.

If problem P presents relevant feasibility issues, i.e., it is not easy in practice to find a feasible initial solution, the same framework can be applied by using as current solution an infeasible one. In order to drive the local search process towards feasible solutions, the cost function needs to be modified in order to measure the infeasibility of the candidate solution. In this sense, the evaluation of a move can be more complicated (penalty functions are usually necessary), but the framework does not change substantially. LS methods can be *point-based* (e.g. simulated annealing [103], tabu search [80], etc.) or *population-based* (e.g. genetic algorithms [90]): the first maintain a current point (solution) of the search space and start from that point to move to a close one; the second maintain a set of solutions and define promising regions of the search space based on this set of solutions.

1.2.7 Hybrids

The list of solving methods previously described is not complete, but it covers all the techniques that have, in different ways, contributed to the research proposed in this thesis: indeed, one of the main goal of this thesis is to experiment different ways of combining the basic techniques described using a Constraint Programming Language as an environment for performing such a combination. None of them will be used in its original form (sketched here), but they will be combined with each other. For example, tree search methods will be used to explore a neighborhood; cutting planes, and dynamic programming will be used to calculate bounds in global constraints, etc.

1.3 Constraint Programming

Among CP languages, the one on finite domains is an effective tool for modeling and solving discrete combinatorial problems. It models a problem as a set of variables taking their values on a finite domain of integers and linked by a set of constraints that can be mathematical or symbolic. In particular, symbolic constraints have been recently recognized to play an important role in achieving efficiency of CP solvers. In fact, symbolic constraints [18] allow a concise and declarative modeling of a subproblem appearing in the original one, and embed powerful operational behavior that enables to effectively reduce the search space, i.e., the constraint propagation. Constraint propagation removes from the search space those combinations of variable-value assignments that are proven to be infeasible; it is an iterative process which involves all problem constraints and reaches a fix point when a certain level of consistency [158] is achieved.

In addition, an objective function can be defined on problem variables. However, pure CP solvers often do not effectively deal with objective functions since domain reduction is, in general, based on feasibility reasoning, but not on optimality reasoning.

Constraint Programming origins from the theory, developed within the AI community, of Constraint Satisfaction.

1.3.1 Constraint Satisfaction

Constraint Satisfaction Problems (CSP) play a central role in many fields of Artificial Intelligence and Computer Science in general (see, e.g., the book by Tsang [158]). A CSP can be defined on a set of variables $X_1, \ldots, X_n$ ranging on a finite domain $D_1, \ldots, D_n$, representing the possible values that variables can assume ($D_i = \{v_1, \ldots, v_k\}$), and a set of constraints $C_1, \ldots, C_m$, representing the relations among variables. A constraint $C_j(X_1, \ldots, X_n) \subset D_1 \times, \ldots, \times D_n$ is a relation whose tuple denote the legal combination of values for $X_1, \ldots, X_n$. Therefore, constraints limit the values that variables can simultaneously assume. Usually, research on CSPs is concerned with binary CSPs where only binary constraints are considered, i.e., constraints linking pairs of variables, e.g., $X_i > X_j$. However, some efforts [24, 25, 23, 22] have been performed in order to extend results obtained for binary CSPs to general CSPs, where non-binary constraints, i.e., constraints linking n variables, appear.

Each binary CSP can be represented as a graph where nodes are variables and arcs are binary constraints. Each node has an associated domain representing the domain of the variable. In the following we refer to nodes with their index i ($i = 1, \ldots, n$), to their domain with D_i and to arcs with the associated constraint. If an arc links variables (nodes) i and j, the associated constraint is indicated as C_{ij}. A loop in the CSP graph represents a unary constraint C_i, i.e. a constraint limiting the domain of a single variable.

A CSP can be solved by using tree search techniques by sequentially instantiating variables. If a partial instantiation violates any of the constraints, chronological backtrack is performed leading to a depth-first search of the solution space. It is well-known that a complete enumeration is inefficient since the search space grows exponentially with the dimension of the problem, and chronological backtracking suffers from trashing [78]. Trashing is essentially caused by the fact that an infeasible state is not immediately recognized as such, and the exhaustive exploration of its subtree is needed before backtracking on that decision which is the real cause of the infeasible state.

Many consistency techniques have been proposed to limit the trashing behavior of chronological backtracking, like node, arc, path consistency [75, 104, 112, 118, 119, 120], and the general technique of k-consistency proposed by Freuder [74]. In the following we introduce the definition of node and arc consistency which are the basic methods used in CP solvers for binary constraints.

Definition 1 Node Consistency: *A node i is node consistent iff for any value $x \in D_i$ the unary constraint $C_i(x)$ holds. A graph representing a CSP is node-consistent iff each node of the graph is node-consistent.*

Definition 2 Arc Consistency: *An arc linking variables i and j is arc-consistent iff for any node-consistent value $x \in D_i$, a node-consistent value $y \in D_j$ exists such that the binary constraint $C_{ij}(x, y)$ holds. A graph representing a CSP is arc-consistent iff each arc of the graph is arc-consistent.*

A general definition of k-consistency has been given by Freuder [74]: given values of $k-1$ variables, that satisfy all the constraints among these variables, then for any k-th variable, a value in its domain exists that satisfies all the constraints among these k variables. Moreover, Freuder proposed an algorithm for reaching whatever degree of consistency. It is easy to see that the arc-consistency corresponds to 2-consistency.

The node and arc consistency algorithms are frequently used also as basic propagation techniques in CP solvers. These algorithms do not solve a CSP completely, but remove local inconsistencies that cannot appear in any consistent global solution. In fact, a CSP can be arc-consistent, but globally infeasible. A simple example is the following: we have three variables X, Y and Z ranging on the same domain containing values $[a, b]$ linked by the following constraints: $X \neq Y$, $X \neq Z$ and $Z \neq Y$. The CSP is arc-consistent since for each value in the domain of each variable there exists in the domain of each other variable one value supporting it. However, the CSP is trivially inconsistent, since it is not $3 - consistent$.

A general result by Freuder [75] states that in general a CSP involving n variables is completely solved (i.e., a solution is found without any search) if a n-consistency algorithm is performed. Not surprisingly, the complexity of computing n-consistency is exponential in n (CPS are, in general, strongly $\mathcal{NP}$-complete). However, some CSPs have a special structure: if the maximum number of arcs connecting each node to others is k, then a $(k+1)$-consistency is enough to solve the CSP with no search.

As previously mentioned, some efforts have been devoted to the definition of similar properties for non-binary constraints. These constraints can be handled in two ways: either by transforming them in binary constraints [147] (this is always possible even if usually inefficient), or by extending the definitions and algorithms provided for binary constraints to non-binary ones. Concerning the last point, again, we have two possibilities: define general results, i.e., generalized arc-consistency for non-binary constraints [24, 25, 23, 22] or define special purpose propagation algorithms exploiting the semantics of the constraint, see, e.g., [18, 13, 36, 142, 144]. The first approach has in general a high complexity and is not used in practice, while the second approach is more appealing and is the one followed by CP solvers for defining the propagation of global constraints.

1.3.2 CP(FD) Basic Concepts

Constraint Programming on Finite Domain has been recognized as a suitable modeling and solving tool to face combinatorial (optimization) problems. In this section, we provide some basic notions on modeling, constraint propagation, search and optimization.

Modeling

There exists a one to one mapping between CSP concepts introduced in the previous section and CP(FD) syntactic structures. In fact, in a CP(FD) language we can define variables ranging on finite domain of values and a set of constraints involving those variables. Thus, CP(FD) benefits from all the results achieved for CSPs.

One of the characteristic of CP(FD) is the facility of modeling problems, which can be declaratively stated, extended and modified: expressivity and flexibility have always been considered important features of CP.

In CP(FD) languages, variables range over finite domains of integers. The variables' domains represent the values that variables can assume during the computation. For example, $X_1 :: [1..10]$ states that variable X_1 can assume one of the integer values from 1 to 10, $X_2 :: [3, 5, 9]$ states that variable X_2 is either 3 or 5 or 9. Concerning notation, given a variable X, in the following we will refer to the minimum (resp. maximum) value in its domain as `inf`(X) (resp. `sup`(X)). Moreover, the domain itself is referred to as `domain`(X), whereas once X has been instantiated its value is returned as `value`(X). Variables are linked by constraints that can be either mathematical constraints, such as $X_1 > X_2$, $X_1 < X_2$, $X_1 = X_2$, $X_1 \leq X_2$, $X_1 \geq X_2$, $X_1 \neq X_2$, or symbolic constraints. Symbolic constraints, which can also be defined by the user, are more powerful constraints with complex propagation mechanisms. A typical symbolic constraint is `element(varArray,X,Y)` available in most CP(FD) solvers like CHIP [53],ECL^{i}PS^e [56], and ILOG Solver [139]. Declaratively, the constraint `element(varArray,X,Y)` holds if and only if, given two domain variables `X`, and `Y`, and an array of domain variables `varArray`, `Y` assumes a value equal to `varArray[X]`.

The basic structure of a CLP(FD) program describing a CSP is the following:

```
solve(VarArray decisionVarArray) {
  makeDecisionVariables(decisionVarArray);
  makeProblemConstraints(decisionVarArray);
  search(decisionVarArray);
}
```

The solve parameter `decisionVarArray` is an array of variables representing the problem entities. In the `makeDecisionVariables` step, variables are defined with the corresponding domains. In the `makeProblemConstraints` step, the constraints defining the problem are posted. The underlying constraint solver propagates the constraints, thus reducing variable domain values. Constraint propagation is presented in Section 1.3.2. At the end of the propagation, either a solution is found (each variable is instantiated to a feasible value) or a failure is detected or a labeling step implementing the search strategy begins. This labeling process is iterated until all the variables are successfully instantiated, i.e., a solution is found. This aspect is discussed in Section 1.3.2.

For optimization problems, instead, the structure of the program can be slightly changed as follows:

```
solve(VarArray decisionVarArray, Var obj) {
  makeDecisionVariables(decisionVarArray, obj);
  makeProblemConstraints(decisionVarArray);
  minimize(decisionVarArray, obj);
}
```

An additional parameter is added for representing the objective function, i.e., `obj`, and a `minimize` predicate can be used in order to find the best among feasible solutions with respect to `obj`. The `minimize` predicate implements a simple form of Branch and Bound that is discussed in Section 1.3.2. In general CP languages have built-in features that implement tree search and branch and bound.

Constraint Propagation

In general, Constraint Satisfaction Problems can be solved by simple enumeration, but this approach is obviously not practical since the search space grows exponentially with the dimension of the problems. Thus, these problems greatly benefit from the reduction of the search space performed by constraint propagation, and some applications are solved more efficiently by using CP than Integer Programming approaches (see, e.g., [154, 45]).

During the computation, constraints are propagated in order to reduce variable domains by removing inconsistent values. If a domain becomes empty, a failure is raised and a backtrack forced. The basic propagation algorithm used for binary constraints is the arc-consistency algorithm [112]. Instead, symbolic constraints in general embed propagation algorithms which exploit the semantics of the constraint itself (see, e.g., [18, 142, 13, 144]). Consider, for example, the symbolic constraint `allDifferent`$(\mathtt{X}_1,\dots,\mathtt{X}_n)$. It declaratively holds if all variables are assigned to a different value and it is equivalent to a set of binary inequality constraints connecting each pair of values in $(\mathtt{X}_1,\dots,\mathtt{X}_n)$ (corresponding to $n(n-1)/2 \neq$ constraints). However, we can perform more global and informed reasoning on the set of variables. Suppose for example that we have an `allDifferent` constraint among three variables $(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3)$ whose domain is $\mathtt{D}_1 = \mathtt{D}_2 = [1,2]$ and $\mathtt{D}_3 = [1,2,4]$. While a set of binary inequalities cannot infer any value removal, the global constraint can reason on the cardinality of the sets of variables and values. Hence, we have a set of variables, i.e., $(\mathtt{X}_1,\mathtt{X}_2)$, whose common domains $\mathtt{D}_1 = \mathtt{D}_2 = [1,2]$ have the same cardinality equal to 2. Thus, values `1` and `2` are *reserved* for variables $\mathtt{X}_1$ and $\mathtt{X}_2$ (no matter which value is assigned to which variable), and are no longer feasible for variable X$_3$ (which is assigned to the remaining value 4).

An interesting feature of Constraint Programming is the interaction among constraints that cooperate through shared variables. A propagation algorithm is part of the constraint itself, and is triggered when an event is raised due to a domain modification of one variable involved in the constraint. The event is, in general, one of the following: removal of a value, reduction of a domain bound, instantiation of a variable (the domain is reduced to a single value). Thus, as soon as one constraint produces a modification on the domain of one variable, say X, all constraints involving X are triggered, and perform propagation on the basis of the current state of the variables' domains. At the end of the propagation each constraint is released, and waits for another event to occur. A constraint is considered solved when it is always satisfied, i.e., $X_1 < X_2$ where $D_1 = [1, 2, 4, 5]$ and $D_2 = [7, 8, 9, 10]$ is always satisfied for all pairs of values in the domains of X_1 and X_2. Thus, if a constraint is solved it does not need to be triggered again after modification in the domain of the variables.

In order to explain the interaction among constraints, let us consider the example depicted in Figure 1.2. The first propagation of `X = Y + 1` yields to the following domain reduction `X::[2..5], Y::[1..4], and Z::[1..5]`. Since `X = Y + 1` is not solved, it could be triggered again after modifications of the domain of some variables. Then, propagation of `Y = Z + 1` reduces domains as `X::[2..5], Y::[2..4], Z::[1..3]`. The domain of `Y` has been changed, thus the first constraint is awakened and removes value `2` from the domain of `X`. Finally, the propagation of `Z = X + 1` removes all values from the domain of `X` and a failure is detected. The order in which constraints are triggered and propagated does not affect the result, but can possibly affect the performances of the propagation process.

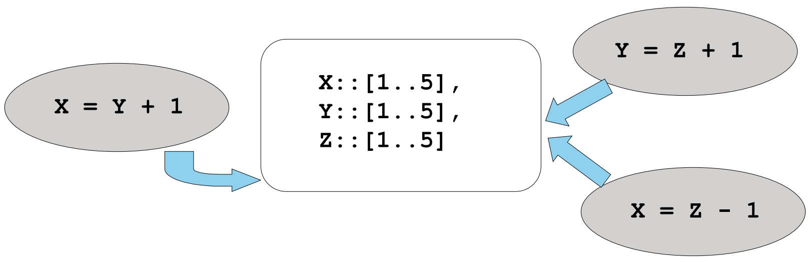

Figure 1.2: Interaction among constraints

Search

At the end of the constraint propagation process, we have three possible scenarios: (*i*) a domain becomes empty and a failure occurs; (*ii*) a solution is found, i.e., all variables are assigned to one value; (*iii*) some domains contain more than one value. In this third case, since constraint propagation is not complete, we need to explore the search space to look for solutions; this exploration is obtained by means of tree search (Branch and Bound for optimization problems), typically using a Depth First procedure. As mentioned, in the CP community, many problem dependent search strategies have been developed. The most commonly used and easiest one consists in iteratively selecting a non-instantiated variable and assigning it to a value among those belonging to its domain. Basically this step is quite similar to the branching strategies used in OR Branch and Bound algorithms with the difference that it does not rely on the computation of the optimal solution of a relaxed problem (bound). However, also different kind of branching constraints can be used in order to split the problem at a given node in simpler subproblems representing a partition of the original one. The way the search space is explored greatly influences the performances of the overall constraint-based computation. An important point to be clarified is that during the search, constraints are always taken into account and propagated in order to prune the search space. In fact, a variable instantiation raises an event awaking all the constraints involving that variable and a propagation process starts again. One important feature that Constraint Programming languages have inherited from their AI origin is the idea of encoding in solving methods knowledge on the problem in hands. All Constraint Programming languages have high level predicates allowing to easily write search heuristics (branching methods). This lead to the development, within the CP community of sophisticated branching methods for many type of problems allowing the development of effective solvers. It is worth mentioning that the first CP languages derived their tree search exploration method from Logic Programming languages such as Prolog implementing a form of Depth First Search (chronological backtracking). More recently, Constraint Programming languages have removed this limitation, and provide several built-in tree search exploration methods (such as Limited Discrepancy Search [85]); ILOG Solver 4.X [96] provides both built-in methods and a set of primitives that can be used to easily implement user-defined tree exploration methods.

Optimization

In some applications we are not looking for a feasible solution, but for an optimal one with respect to some objective function f defined on problem variables. CP systems use the Branch-and-Bound framework to find an optimal solution. The idea is to solve a set of decision (feasibility) problems (i.e., a feasible solution is found if it exists), leading to successively better solutions. In particular, each time a feasible solution z^* is found (whose associated cost is $f(z^*)$), a constraint $f(x) < f(z^*)$, where x is the vector of problem variables, is added to each subproblem in the remaining search tree. The purpose of the added constraint, called *upper bounding constraint*, is to remove portions of the search space which cannot lead to better solutions than the best one found so far. The problem with this approach is twofold: (i) CP does not rely on sophisticated algorithms for computing lower and upper bounds for the objective function, but derives these values from the variable domains; (ii) in general, the link between the objective function and the problem decision variables is quite poor and does not produce effective domain filtering.

Therefore, recently some efforts have been performed in order to embed in CP some techniques that take into account some form of *optimality reasoning.* For example, bounds deriving from the optimal solution of a (linear) relaxation have been used in CP based algorithms, e.g., [15, 146, 21, 82, 133, 27]. Different forms of relaxations have been also incorporated in symbolic constraints [71, 68, 67, 39, 38], thus achieving a form of pruning based on bounds and sensitivity analysis.

1.4 Mathematical Programming

In this section, we provide some elementary notions of Mathematical Programming (MP). In particular, we introduce the standard form adopted in Mathematical Programming to model Integer Linear Programs. Then we consider, as a solving technique, the Branch and Bound framework which is based on a two steps search procedure: the computation of the optimal solution of a relaxation, and a branching step. We describe some well known and widely used methods for defining and solving problem relaxations.

1.4.1 Integer Linear Programming

An ILP problem is a problem where we have to optimize a given linear objective function on a finite (possibly very large) set of solutions. The solution space is defined by a set of linear constraints (equalities and inequalities) and by the integrality constraint on problem variables. With no loss of generality, we consider minimization problems. ILP can always be written in the following standard form:

$$min\{c^T x | Ax = b,\ x \in Z^n_+\}$$

where Z^n_+ is the set of non-negative integer n-vectors, $x = \{x_1, \ldots, x_n\}$ are problem variables, c is an n-vector, A an $m \times n$ matrix, and b an m-vector. The set $S = \{x \in Z^n_+ | Ax = b\}$ is called *feasible region* and $x \in S$ *feasible solution*. The function $z(ILP) = c^T x$ is called *objective function*, and a feasible point $x^* \in S$ such that $c^T x^* \leq c^T x$ for all $x \in S$ is called *optimal solution*.

Integer (Linear) Programming problems, in general, belong to the NP-hard class of problems. An important methodology used to solve ILP is the Branch and Bound framework described in Section 1.2.2 where problem relaxations play a central role for effectively explore the search tree. As mentioned, the purpose of a relaxation in a Branch and Bound framework is to demonstrate that an NP-hard sub-problem does not need to be solved because its optimal solution cannot improve the best solution found so far. Therefore, the value of the optimal solution of a relaxation should be as close as possible to the value of the optimal solution of the original problem. In the following some widely used methods to generate relaxed problems will be described.

1.4.2 Relaxations

There are various method to generate a relaxation $R(P)$ of a problem P. In the following we describe some techniques for modifying a problem and obtaining a relaxation. These techniques can be applied one at a time, or be composed.

In the Branch and Bound scheme, a relaxation $R(P)$ is useful if it is easier to solve than P. In general, relaxations to ILP problems are focused at obtaining polynomially solvable problems. Linear Programs are often used as relaxation since finding the optimal solution of a LP problem has a polynomial time complexity, even if the most commonly used algorithm for LP is the *simplex algorithm* which is exponential in the worst case, but very effective in practice.

Relaxation obtained by constraint removal

The simplest method to generate a relaxation for a given problem P is to remove some of the constraints defining it. Consider for instance a Traveling Salesman Problem P; this problem looks for one tour of minimum cost visiting all nodes in a graph exactly once. By removing the constraint imposing that the nodes must be covered by only one tour the following relaxed problem can be stated: problem $R(P)$ looks for any number of tours visiting all nodes in a graph exactly once with minimum cost. This is the so called *Assignment Problem (AP)*; it is clearly a relaxation of the TSP, and it is a lot easier to solve (it belongs to the class $\mathcal{P}$).

Linear Relaxation

A special case of constraint removal relaxation is the *Linear Relaxation* or *Continuous Relaxation.* If the original problem P can be described by linear constraints over integer variables, the removal of the integrity constraints for all integer variables leads to a *Linear Relaxation.* The resulting LP has the form:

$$min\{c^T x | Ax = b,\ x \in R^n_+\}$$

and can be solved by Linear Programming techniques (e.g., the simplex algorithm).

If the optimal LP solution, say x^*, is integer, it is also feasible and optimal for the original ILP. In general, however, x^* violates the integrity constraint and contains fractional values. For some specific problems, the x^* is always integer. This is the case, for example, of problems with a matrix *A totally unimodular.* Thus, the ILP can be optimally solved by solving the corresponding linear relaxation.

Linear Relaxation and Cutting Planes

In many cases, it is convenient to tighten the formulation of the Linear Relaxation to obtain better bounds on the original problem. The Linear Relaxation can be tightened by iteratively generating some *cutting planes.*

As mentioned in Section 1.2.4 given the ILP problem P, the cutting planes method is based on the iterative solution of linear relaxations of P, say P_L. The optimal solution x^* of P_L provides a lower bound $z_{P_L} = c^T x^*$ on the optimal solution of P where, in general, some constraints of the original problem are violated. The aim of the cutting planes generation is to find some linear inequalities corresponding to violated constraints that remove x^*, infeasible for P, and do not prune any feasible solution of P. The problem of finding an inequality $\alpha x \leq \alpha_0$ satisfied by the feasible solutions of P, and such that $\alpha x^* > \alpha_0$, is called *separation problem.* In principle, one can iteratively add all such inequalities up to the definition of the convex hull of the set of solutions of P. However, there is no efficient way of generating the convex hull of a problem, since, if P is NP-hard, the separation problem is NP-hard as well [111].

Therefore, in general, the method used is to iteratively add a subset S_{cut} of inequalities, obtained by solving, possibly through heuristics, the separation problem for some *classes* of inequalities. Each addition of cutting planes (or simply cuts) originates a new linear relaxation P_L^{new} of P, and the optimal solution value of P_L^{new} typically increases with respect to P_L (P_L^{new} is a better approximation than P_L).

When the addition of cuts becomes ineffective (the bound does not increase any longer), the cutting plane generation stops, and a final LP represents a relaxation of the original problem P. The iterative generation of cutting planes within a Branch and Bound framework was introduced by Padberg and Rinaldi [129] for the Traveling Salesman Problem (TSP). The method has been applied to a wide variety of combinatorial optimization problems, often obtaining the state of the art results, and it is known with the name *branch and cut.*

Surrogate Relaxation

Another way to obtain a relaxation is to start from the original problem and relax some constraints in a surrogate way. The *Surrogate Relaxation* considers a set C of constraints of P, and substitutes them with a new set C' such that $C \rightarrow C'$.

An easy example of Surrogate Relaxation can be given by considering the following two constraints: $a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq a_0$, and $b_1x_1 + b_2x_2 + \ldots + b_nx_n \leq b_0$. The simplest surrogate relaxation corresponding to the set C composed by the two is the set C' containing the single constraint $(\pi a_1 + b_1)x_1 + (\pi a_2 + b_2)x_2 + \ldots + (\pi a_n + b_n)x_n \leq \pi a_0 + b_0$. The optimal solution of the problem in which C' replaces C is a lower bound for the original problem. Obviously, the best lower bound is the highest one over all possible $\pi \geq 0$.

Lagrangian Relaxation

In a Lagrangian Relaxation a set C of constraints of P is *dualized*, i.e., the constraints in C are removed from P and added to the objective function. The aim is to obtain an easier problem in which, however, the information associated to the removed constraints is not lost. In fact, the addition of the constraints in C to the objective function allows to penalize, proportionally to the violation, any solution that violates them.

More formally, consider the Lagrangian relaxation of a linear constraint $\alpha x \leq \alpha_0$, and let $\lambda \geq 0$ be the so-called Lagrangian multiplier associated to the constraint. The objective function of the relaxed Lagrangian problem $R_\lambda(P)$ is then $c^T x + \lambda(\alpha x - \alpha_0)$. In the objective function, we have a constant factor $\lambda\alpha_0$, whereas the cost matrix c is updated by adding the $\alpha\lambda x$ coefficients.

Since the term $\alpha\bar{x} - \alpha_0$ is positive, it is easy to see that every solution $\bar{x}$ violating the dualized constraint is penalized in the objective function.

Optimally solving $R_\lambda(P)$ leads to compute a lower bound value for P, where again the best lower bound is the highest one over all possible value of $\lambda \geq 0$.

Structured Relaxation

In some cases, relaxations can be defined using any combination of the previously mentioned methods in order to obtain a structured problem for which special purpose algorithms are available.

Consider, for example, the Traveling Salesman Problem (TSP), and in particular its Asymmetric version (ATSP). This problem looks for a single tour of minimum cost visiting all nodes in a directed graph exactly once. We have mentioned that an *Assignment Problem* (AP) can be defined by removing the constraint imposing that the nodes must be covered by only one tour. The AP is clearly a relaxation of the ATSP, and it appears a lot easier to solve, through special purpose algorithms (see, e.g., [33]).

Another structured relaxation for ATSP is the *Minimum Spanning r-Arborescence problem* (r-MSA) which calls for the determination of a minimum cost arborescence in a directed graph such that there is a path from a given vertex r to any other in the graph, and each vertex has in-degree equal to one (but r). This problem can be solved in polynomial time through special purpose algorithms (see, e.g., [63]), and it is easy to see that it is a relaxation of ATSP. In fact, if the arborescence plus the arc of minimum cost returning in r is a tour it is also optimal for ATSP.

Additive bounding method

Bounding procedures can be combined using the *additive bounding approach* [61]. Let $\mathcal{L}^1, \dots, \mathcal{L}^r$ be lower bound procedures for problem P:

$$min\{cx \mid x \in \mathcal{F} \subset \Re^n\}$$

Also suppose that, for $k = 1, \dots, r$, procedure $\mathcal{L}^k(\bar{c})$ is applied to an instance of P with cost vector $\bar{c}$, and it returns a lower bound value δ^k as well as a *residual cost* row vector $c^k \in \Re^n$ such that $c^k \geq 0$ and

$$\delta^k + c^k\, x \ \leq \bar{c}\, x, \ \ \forall x \in \mathcal{F}$$

A lower bound for P with the original cost vector c can be calculated by applying $\mathcal{L}^1(c)$, and $\mathcal{L}^k(c^{k-1})$, for $k = 2, \dots, r$. It can be shown (see [61]) that the sum $\sum_{k=1}^{r} \delta^k$ is a lower bound for P with the original cost vector c.

State Space Relaxation

State Space Relaxation considers the Dynamic Programming formulation of a combinatorial optimization problem, and modifies it to obtain a different DP formulation whose optimal solution is a lower bound of the problem. Proposed by Christofides et al. in [42], state space relaxation has been successfully applied to constraint variants of routing problems (see e.g. [116]). We will give here only a brief overview of state space relaxation; a formal and comprehensive description can be found in [42].

Consider the Dynamic Programming formulation of a problem P with the corresponding *state space graph* $SG = (S, T)$; a new state space graph $RSG = (RS, RT)$ is considered such that a function $w(s)$ exists mapping every state $s \in S$ into a state $w(s) \in RS$, and such that for every arc $(s_i, s_j) \in T$, $(w(s_i), w(s_j))$ is a feasible arc in RT. Moreover, the cost of $(rs_h, rs_k) \in RT$ is smaller or equal to the cost of $(s_i, s_j), \forall (s_i, s_j) \in T | (w(s_i), w(s_j)) = (rs_h, rs_k)$.

Then, if p is the shortest path from s_1 to s_i, and rp is the shortest path from $w(s_1)$ to $w(s_i)$, the cost of rp is less or equal to the cost of p. The relaxation represented by $RSG = (RS, RT)$ is, of course, useful if the number of vertices of the new graph is polynomial instead of exponential. Considering the TSP example in Section 1.2.3, let n be the number of nodes in the TSP, and c_{ij} the cost matrix. A possible relaxation is the following: $g_i(j) =$ *the length of the shortest path from city* 1 *to city* j *via* i *intermediate cities.* Clearly $g_i(j)$, $i, j = 1, \dots, n$ can be calculated in $O(n^3)$. The value $max_{j=1,\dots,n-1}\{g_n(j) + c_{j1}\}$ is a lower bound for the value of the optimal TSP solution.

1.5 ILOG Optimization suite

This thesis is the result of three years of research that was conducted in coordination and collaboration with the Consulting and Research and Development groups at ILOG. Founded in 1987, ILOG develops optimization and visualization libraries that are used by thousand of developers, and tens of thousands of end users in all areas of industry and business. ILOG's optimization suite includes the following:

- **ILOG Solver** is a C++ object-oriented Constraint Programming engine;

- **ILOG Scheduler** is a C++ library added on ILOG Solver that provides modeling objects and constraint-based scheduling algorithms for scheduling and resource allocation problems;

- **ILOG Dispatcher** is a C++ library added on ILOG Solver that provides Constraint Programming based Local Search methods for vehicle routing, and dispatching problems;

- **ILOG Planner** is a C++ library added on ILOG Solver enabling the integration of Linear Programming methods within Constraint Programming[3].

- **ILOG Configurator** is a C++ library added on ILOG Solver that provides modeling objects and algorithms for solving configuration problems, where parts are selected from a catalog in desired quantities and assembled to meet connection requirements and the desired configuration.

- **ILOG CPLEX** is a C library for Linear Programming and Mixed Integer Programming.

- **ILOG OPL Studio** is an interactive graphical environment and an high level modeling language for rapidly prototyping optimization applications.

- providing a set of C++ objects, **ILOG Concert Technology** is a modeling language for optimization problems allowing the separation between the model and the algorithms that are used to find solutions. A Concert model can be "extracted" to any of the C++ ILOG optimization libraries containing the algorithms used to solve the problem.

This dissertation rely on the ILOG optimization products for implementing the proposed algorithms. In particular, Chapter 5 proposes a set of C++ objects to be added to ILOG Scheduler library. For this reason some basic information on ILOG Solver, and Scheduler will be provided.

1.5.1 ILOG Solver: an overview

ILOG Solver is a C++ library for Constraint Programming. It includes predefined classes of variables; predefined classes of arithmetic, symbolic, and global constraints, together with a mechanism to implement new constraints; predefined search algorithms, together with a mechanism to write user defined tree search methods.

In Solver, domain variables are represented as objects with an associated domain. Solver provides integer variables, enumerated variables (variables whose domain is a finite set of C++ object addresses), floating point variables, boolean variables, and set variables (variables whose domain is a set of sets). Using the operator overload mechanism of C++, Solver provides all kind of arithmetical operators (such as $+, -, *, /$) on domain variables.

A Constraint is an object linking domain variables, with associated one (or more) propagation algorithm. Constraints can be *posted*, in this case the constraint is enforced, or can be *meta-posted*, i.e., treated as boolean variables and combined via boolean operators.

[3]With the recent introduction of ILOG Concert Technology, the functionality of ILOG Planner has been made available on ILOG Concert

Search for solution is typically performed via tree search; local search methods have been recently introduced as well. Solver provides a set of control primitives that allow its user to implement his/her own heuristic search algorithm. A (sub)tree is defined through an object of type `IlcGoal`. Goals can be composed via operators `IlcOr(g1,g2)`, and `IlcAnd(g1,g2)`; the *OR* goal operator sets a choice point between two goals (two subtree), and the *AND* goal operator defines a new subtree as the logical *AND* between the two goals. Consider, for instance, an array of k boolean variables $vars$; the following two goals can be used to define a tree search exploring all the combinations of values for $vars$.

```
ILCGOAL1(Instantiate, IlcIntVar, var) {
    return IlcOr(var = 0, var = 1);
}

ILCGOAL2(Generate, IlcIntVarArray, vars, int, index) {
    if (index == -1) return 0;
    return IlcAnd(Instantiate(vars[index]),
                  Generate(vars, index-1)
                  );
}
```

Calling the goal `Generate(vars,k)` leads to a Depth First complete exploration of the search tree where the sequence of generated solution (for $k = 2$) is $s_1 = [0,0]$, $s_2 = [0,1]$, $s_3 = [1,0]$, $s_4 = [1,1]$.

By default, the tree search is explored in Depth First Search. Solver provides also a set of control primitives to build user-defined search exploration methods (see [131, 96]).

1.5.2 ILOG Scheduler: an overview

ILOG Scheduler is a C++ library "add-on" to ILOG Solver. It allows the user to model scheduling problems and provides efficient propagation algorithms. ILOG Scheduler provides modeling objects such as **schedule**, **activity**, **resource**, etc. Activities are represented by four variables being its start time *start*, end time *end*, processing time *pt*, and duration *dur*. These variables are constrained by the relations imposing $start + dur = end$, and $dur \geq pt$. The processing time is equal to the duration minus the interruption due to brakes eventually defined on resources. Different types of resources exist. Unary resources are typically used to model machines, they have maximal capacity equal to one, and can be subject to sequence dependent transition times and costs. Discrete resources represent resources of discrete capacity; a profile of minimal and maximal capacity can be defined; each activity may require or provide capacity to the resource; they can, for instance, be used to model pool of workers. State resources represent resources which state can vary over time; for instance, they can be used to represent the temperature of an oven, or the current color in a painting machine. Energy resources represent amounts of energy available for a given time interval (e.g., days, months, etc.); they can be used, for example, to represent the global amount of man-hours available within a week. Finally, alternative resources are used to model resource allocation problems; they represent an alternative choice among different resources; for instance, a task may be executed on one resource to be chosen among a set of k resources within the given alternative set. For each type of resource, a time-dependent profile can be defined. For instance, a time-dependent profile in a discrete resource may represent changes in the maximal resource capacity.

Some of the predefined constraints include temporal constaints, resource utilization constraints, transition time and cost constraints. Predefined propagation algorithms are associated to scheduling global constraints to enforce a degree of consistency for the decision variables of the scheduling problem (see e.g. [13, 124]).

Chapter 2

Integrating Constraint Programming and Operations Research

2.1 Introduction

In recent years, the integration of techniques from Artificial Intelligence and Operations Research has shown to improve the solutions of complex and large scale combinatorial optimization problems, in terms of efficiency, scalability and optimality.

Situated at the confluence of the two fields, Constraint Programming is an emerging discipline that has been recognized as a suitable environment for achieving such an integration. This chapter briefly presents the integration directions explored in the literature, and provides some pointers to relevant work in these directions (see, [72, 70] for a recent review on the subject). CP is a programming paradigm exploiting constraint satisfaction techniques. The following discussion is limited to *Constraint Programming on Finite Domains* (CP(FD)) since it is particularly suitable for coping with combinatorial (optimization) problems. A comprehensive description of CP can be found in [113].

A first step toward the integration concerns comparisons between the two approaches aimed at identifying similarities, differences [86], and problem properties [45] leading to preferring one approach to the other.

For some problems, Constraint Programming seems more effective, see e.g., the progressive party problem [154], the warehouse location problem [160], the template design problem [136] and the change problem [86]. The main advantages of CP are: simple and concise models, semantically rich primitives to easily develop problem dependent search strategies, and powerful propagation algorithms to efficiently reduce the search space. Other problems, instead, are more easily solved using Integer Linear Programming (ILP) techniques, see e.g., the crew scheduling problem, and the flow aggregation problem [45].

In general, pure problems (e.g., problems without side constraints) are more likely to be approached with MP techniques able to exploit their (well studied) geometrical structure. When, however, problems are less pure, i.e., side constraints are added, CP seems to be more suitable. A typical example of this behavior is the Traveling Salesman Problem (TSP). State of the art Branch-and-Cut approaches for the TSP [4] achieve results that are several orders of magnitude better than pure CP approaches, while this gap is substantially reduced as soon as, for example, *Time Windows* constraints are considered (see, e.g., [6] and [68]).

Besides comparisons on different applications, some studies identify guidelines that help the designer to choose the best approach for the problem to solve. For this purpose, Bockmayr and Kasper [27] proposed a unifying framework, Branch-and-Infer, providing a uniform view of ILP and CP(FD). The framework has been used for comparing the two techniques both from the modeling and the solving perspective.

Branch-and-Infer identifies basic modeling and solving components where ILP and CP concepts can be accommodated and compared. From a modeling point of view, in a constraint language, the authors recognize two kind of constraints: *primitive* and *non primitive* ones. Roughly speaking, primitive constraints are those that are easily handled by the constraint solver, while non primitive ones are those for which no (complete) method exists for satisfiability, entailment and optimization that runs in polynomial time. In CP systems, primitive constraints are the following:

$$Prim = \{x \leq u, x \geq b, x \neq v, x = y, integer(x)\}$$

where x and y are variables, u, v, b are constants. All other constraints are *non primitive.* In ILP, instead, all linear equalities and inequalities are primitive while the integrality constraint is non primitive.

From a problem solving perspective, the purpose of a computation in a constraint-based system is to infer primitive constraints from non primitive ones. In CP, constraint propagation infers value removals and bound tightening on variable domains (primitive constraints for CP system). In ILP, cutting plane generation infers new linear inequalities (primitive constraints for ILP systems). Once no more primitive constraints can be inferred, both CP and ILP branch in order to partition the problem in a set of (easier) sub-problems.

To carry out optimization, CP implements a simple form of Branch and Bound where every time a feasible solution is found a constraint is imposed stating that any new solution must have a better value of the objective function. In ILP the Branch and Bound is based on the optimal solution of a relaxation of the current sub-problem. Such a solution provides an optimistic evaluation on the best solution that can be found for the sub-problem.

Unifying frameworks are thus useful to identify common concepts and differences and to provide the basis for integration, as widely discussed in [101].

In the next three sections we discuss in detail the main directions of integration of MP techniques in CP, problem modeling, filtering algorithms, and search methods.

2.2 Modeling

CP offers declarative, flexible and easily extensible languages for expressing satisfaction and optimization problems. Most industrial CP tools embed global constraints allowing a concise description of the problem, and effective propagation algorithms. Hence, one of the major strength of CP is the flexibility of models, which makes it easy to add application-oriented constraints (variants of the original problem).

On the other hand, a strong tradition relies on mathematical models suitable for MP solution techniques. Modeling problems through ILP is in general more complex, and involves a greater number of variables and constraints.

The first issue for integrating MP techniques in CP is to make the different models coexist, cooperate or even merge in a single language. In any case, a mapping between the two sides should be defined.

Three integration directions have been followed in the literature:

1. in some approaches CP is maintained as modeling language; both the MP model and the mapping between CP and MP is hidden from the user. An example is the work of Refalo [141], and Rodosek et al. [146]. In general, these approaches rely on the automatic transformation of CP constraints into a linear program. The advantage is that the integration is transparent to the user, and problem models are easily stated in terms of CP languages. On the other hand, no control on the MP side can be performed;

2. in some approaches the problem can be stated both using CP and MP models. The two models cooperate through shared variables in the constraint store. Examples of this second approach are given in [21, 87]. Obviously, the advantage consists on a greater flexibility with respect to the previous approach due to a better control over the MP side; the drawback is that the user has to write more complex problem models;

3. some approaches merge the CP and MP models, and provide a unique model embedding both the MP and CP side. This is done, for example, in [92, 126, 127]. Again, writing problem models is more complex than in pure CP approaches, but a uniform view of the problem could help to develop hybrid solutions.

Recently, two new languages *OPL* [159], and *Concert* [93] have been developed. In these languages the model is independent from the solving technology: a problem modeled using Concert can be solved combining CP and MP in arbitrary ways, in the sense that each constraint can be sent to a CP or MP solver or to both of them (in this last case the two solvers communicate through shared variables).

An important point on model integration concerns which part of the problem to model using MP and CP. Some approaches duplicate information by modeling a part of the problem (or the overall problem) in the two sides. For example, in automatic problem transformation the whole problem is duplicated in the two solvers. In [141] the MP solver works on the set of linear inequalities corresponding to the transformed constraints, while in [67] the MP solver works within a given global constraint. Finally, some approaches partition the problem in sub-parts and model each part in the solver that is best suited for it. For instance, the approach proposed in [57] for dynamic scheduling, partitions the problem so as to provide the MP solver with sub-problems that can be optimally solved in polynomial time, and leaves the CP solver to cope with the remaining part of the model.

2.3 Filtering algorithms

CP exploits constraint satisfaction techniques to solve combinatorial (optimization) problems. CP propagation has many similarities with Operations Research problem reductions. The problem at hand is considered as a *feasibility* problem, and constraint propagation is used to infer domain reduction and tighter bounds for the variables; as soon as the propagation reaches a fix point, a branching partitions the problem in independent sub-problems. However, this simple enumeration method does not take into account any form of *optimality* reasoning.

In the next two sections the possible integrations of MP techniques in CP with respect to feasibility and optimality issues will be discussed.

2.3.1 Looking for feasibility

The first integration of MP techniques in CP can be found in propagation algorithms of global constraints. A well-known example of the use of OR results for developing effective propagation algorithms is the so-called *edge finder* algorithm [125] included in ILOG Scheduler [95] and derived from the work of Carlier and Pinson [31].

Propagation algorithms (see, in addition the works of Régin [142, 144]) can be reinterpreted as special purpose OR algorithms embedded in CP software components. Two desirable characteristics of propagation algorithms consist in being *polynomial* (and fast) and *incremental*. These two characteristics are crucial for an algorithm to be effectively used in CP. Propagation algorithms must be fast since the same algorithm could be called several times within the same node of the search tree before the fix point is reached; moreover, a general tendency in CP is to rely on the branching scheme (using problem dependent heuristics) more than rely on complex and time expensive computation at each node. Propagation algorithms should be incremental, since they are called upon changes in the domains of their associated variables. Instead of re-solving the entire consistency problem, an incremental algorithm is able to re-establish a consistency situation by reasoning on the delta-changes from a situation of consistency.

Although incremental algorithms are not new in OR, they are not considered as important as they are in CP where the propagation framework enables to easily exploit them. In our opinion, incremental algorithms represent one very important research direction for integrating OR and CP, where people coming from the OR community could give great contributions.

2.3.2 Looking for optimality

The main limitation of CP systems appears when coping with objective functions in combinatorial *optimization* problems. As soon as, exploring the search tree, a leaf is reached and a new solution of value z^* is found, CP simply adds a new constraint imposing that further solutions must have a value better[1] than z^*.

The new constraint in general propagates poorly on the decision variables of the problem. For example, the Traveling Salesman Problem calls for the minimization of the sum of transition costs. Representing the problem with one variable per node identifying the next node in the optimal tour, the domain reduction of these decision variables loosely affects the bounds of the objective function and vice-versa. In these cases, in fact, only deep nodes can be fathomed (when almost all the branching have been performed), thus leading to huge search trees.

On the other hand, MP has developed since the Fifties very effective techniques for calculating lower bounds (for minimization problems) on the objective function value through the solution of relaxed problems. The integration of these techniques in CP has obviously received large attention, and two different levels of integration have been explored.

1. The first level of integration concerns the definition of relaxations for the complete problem. This corresponds to the transformation of the CP model into a linear program already discussed in Point 1 of Section 2.2. In this case we have two separate solvers (see, e.g., [82], and [21]), interacting through shared variables. The CP solver exploits the optimal solution of the relaxed linear program computed by a Linear Programming

[1]Greater (resp. smaller) than z^* for maximization (resp. minimization) problems.

solver both for updating the objective function bound and for guiding the search. Also, reduced costs can be used for filtering purposes in order to remove sub-optimal domain values. On the other hand, the LP solver exploits CP propagation for fixing or changing the bounds of variables.

2. The second level of integration concerns the calculation of bounds with respect to global constraints. These constraints model well-structured parts of the overall problem, thus relaxations on these parts can be effectively defined and solved. The solutions of these relaxations provide valid bounds and sensitivity analysis (e.g. LP reduced costs) for the overall problem, and they can often be computed through special purpose (fast) algorithms. Obviously, the quality of these bounds can be poor (in particular on high levels of the search tree) since they refer only to a specific part of the overall problem. However, their use in conjunction to propagation leads to interesting results within a CP framework (see, e.g., [38, 39], and [67, 68]).

Both methods can widely benefit from MP *cutting plane* generation techniques. When the whole problem (or part of it) is modeled as a LP, the iterative addition of cutting planes tightens the bound. Iterative cutting plane generation is also very useful when the LP model requires an exponential number of constraints (see, e.g., the classical linear relaxation of TSP with sub-tour elimination constraints) [69, 141].

The integration of cuts in CP has also been investigated to maintain a tight problem formulation during search. Recently, Ottosson et al. [128] and Refalo [140] have faced piecewise linear optimization problems and have shown how to preserve the piecewise structure as a global constraint. This allows a tight formulation which is maintained during the search by generating locally valid cutting planes.

The integration of cutting plane generation and CP is, however, not reduced to the issues discussed above. A significant amount of work has been devoted, in the last years, to clarify the relation between cutting planes and constraint propagation, and to recognize differences, similarities and possible cooperation methods, see, e.g., Bockmayr and Kasper [27]. Moreover, methods have been proposed to obtain (i) valid inequalities from constraint propagation, and (ii) powerful inference mechanisms through cutting planes (see, e.g., [14, 26], and [91, 92]).

2.4 Search methods

This section describes the integration of CP and OR methods with respect to search strategies. Both CP and ILP use the Branch and Bound framework; in this context, the CP/AI community has traditionally paid more attention to problem dependent heuristics, and it has recently proposed new and very promising tree exploration methods such as LDS [85]. On the other hand, the MP community uses the information of the optimal solution of problem relaxation to guide the search. Local Search is another search framework that has been used both in the CP and OR communities and that generated very interesting integration results. This section reviews the results obtained both with local search and tree search methods focusing on incomplete methods for real-world problems.

Real-world combinatorial optimization problems have two main characteristics which makes them difficult: they are usually large (see, for example, [29], which describes real-world crew scheduling applications), and they are not *pure*, i.e., they involve a heterogeneous set of side constraints (see, e.g., union contract regulations for crew scheduling and rostering

[29]). Hence, in most cases, exact approaches cannot be applied to solve real-world problems, whereas incomplete methods have been proved to obtain very good results in practice.

In the last few years, the development of CP based incomplete methods, and the integration of Constraint Programming and Local Search has produced exciting developments in the approximate solution of real-world combinatorial optimization problems. On one hand, CP enhances the capabilities of LS by providing flexibility in updating the model in case of adding or changing constraints, and, even more importantly, by proving to be very effective in dealing with large-size neighborhoods as required by these applications. In fact, both for the size of the problems and for the feasibility issues typically related to side constraints, large neighborhood exploration improves the quality of solutions obtained by local search. On the other hand, exploiting local search techniques leads CP approaches to converge faster to good solutions (see, [65] for a recent review on the subject).

2.4.1 Constrained Local Search

LS techniques are based on a simple and general idea. Let P be the combinatorial optimization problem we want to solve, and s a current solution which, for the moment, we assume to be feasible for P, and to have value $z(s)$. A *neighborhood* is defined for s with respect to a *move type* $\mathcal{N}$, i.e., a *function* mapping every solution s in a subset $\mathcal{N}(s)$ of the overall solution space. In other words, $\mathcal{N}(s)$ contains all the feasible solutions of P which can be reached from s by means of a move of type $\mathcal{N}$. Roughly speaking, the move is a manipulation of s whose effect is the transition to another solution $x \in \mathcal{N}(s)$. The LS framework explores the neighborhood by searching for the solution $x^* \in \mathcal{N}(s)$ such that $\delta z = z(s) - z(x^*)$ is maximized (for minimization problems). If $\delta z > 0$, then an improved solution has been found, and the process is iterated by considering x^* as new current solution. Otherwise, a local optimum has been reached, and several very effective techniques can be applied to escape from it.

Two important issues must be taken into account when dealing with real-world problems.

1. Huge problems or problems involving heterogeneous constraints require large neighborhoods. Their exploration can be computationally expensive.

2. Real-world applications typically lead to frequent update/addition of constraints (recall again union contract regulations), thus the algorithmic approach requires flexibility.

Thus many real-world combinatorial optimization problems may benefit from the efficiency of LS as well as from the flexibility of CP. In these cases, both the size of the problems and the presence of side constraints makes proving optimality impractical. However, even standard LS algorithms can get into troubles when the size of the neighborhood grows very fast and/or testing solution feasibility is expensive.

Small neighborhoods with side constraints

Fast LS algorithms typically use neighborhoods of small size that can be explored with a relatively small computational effort. Classic examples are the neighborhoods defined by a move, which simply exchanges a pair of assignments of the current solution. An example of this kind of move is the *2-opt*, used by [109] for the *Traveling Salesman Problem* (TSP). In the TSP case, given a Hamiltonian cycle (current solution), i.e., a sequence of edges connecting the cities in their order of visit, the 2-opt move simply deletes two of these edges by replacing them with two others in order to obtain a new feasible cycle (see Figure 2.1). The time complexity to find the *2-opt* move maximizing the improvement, i.e., to find the solution whose overall length is minimal among all neighbors of the current cycle has a time complexity of $O(n^2)$.

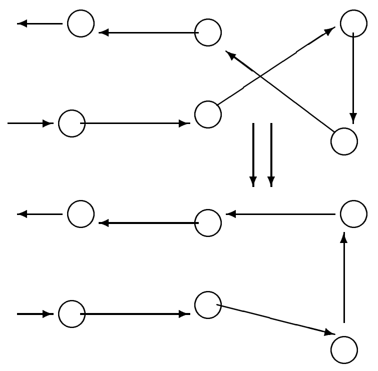

Figure 2.1: The 2-opt move.

Even if the neighborhood is in principle quite small, the addition of side constraints can considerably increase the computational effort required to explore it since it is often necessary to test feasibility.

A typical example is the time-constrained variant of TSP in which the salesman needs to visit the cities within specific *Time Windows* (TSPTW). In this case, in order to find the best 2-exchange move we need to test feasibility, which means testing, for each move, if the resulting solution violates some of the time window constraints. This is the standard way of addressing problems with side constraints in LS: the neighborhood of the pure problem is explored, and for each neighbor, the side constraints are iterated and, for each one of them, its satisfaction is tested. Thus, note that an algorithm must be associated to each constraint testing its satisfaction for a given solution. Note also that as soon as side constraints are added, the computational overhead of constraint checks for LS increases. However, checking feasibility only at the very end of the decision process is only a passive way of using constraints; constraints may be used in more active way by factoring out some of the constraint checks early on in the iteration. Therefore, single checks may discard (hopefully large) sets of neighbors, thus improving the overall efficiency of the neighborhood exploration.

In the example of the 2-opt neighborhood for the TSPTW, one check of time window constraints takes $O(n)$ time, therefore a straightforward implementation of 2-opt for the TSPTW takes $O(n^3)$ time. However, smart incremental computations can reduce the complexity of the TSPTW 2-opt to $O(n^2)$ by caching earliest arrival times and latest departure times [102]. Such optimized neighborhood explorations and constraint checks require specialized code that must be substantially changed whenever new side constraints are considered.

The main point concerns, however, the effectiveness of small neighborhood for real-world applications. With the addition of side constraints the number of feasible solutions of the neighborhood becomes smaller, thus the local optimization process is more likely to get trapped into local optima. Therefore real-world problems require larger neighborhoods, and exploring them by simple enumeration becomes ineffective.

Exploring large neighborhood with CP

As neighborhoods grow larger, finding the best neighbor becomes an optimization problem on its own, thus the use of global search is preferable over blunt enumeration.

We review two possibilities for implementing a LS algorithm, using CP. In both cases, we suppose that we have at hand a current feasible solution s and a CP model of the problem.

The first method consists in keeping a fragment of the solution s (keeping the value assignment for a subset of the variables), erasing the value of all variables outside that fragment and solving the subproblem defined by the uninstantiated variables. This technique was introduced by [5] for job-shop scheduling. The job sequence is kept on all machines but one, and the scheduling subproblem on that machine is solved to optimality by Branch and Bound. In this case, the fragment corresponds to the sequencing order on all machines but one. For scheduling, the approach was generalized to other fragments (time slice, ranks, sets of ordering decisions etc.) by [37] and [124]. The same approach was also applied to quadratic assignment problems by [114], and to vehicle routing by [151]. Such fragment-based LS methods are usually easy to implement once a first CP model has been built. A number of constraint based tools can be used to improve their efficiency:

- an optimization constraint can be set on the neighborhood imposing that only neighbors that strictly improve the objective value over the current solution are generated (see, e.g., [124]);
- fragment based neighborhoods can be explored in a Variable Neighborhood Search (VNS, see, [117] for its definition, and see, [37] for its application to fragment-based LS);
- in order to speed up the procedure, each neighborhood defined by a fragment can be explored by an incomplete search.

The second method, introduced in [132] consists in modeling the exploration of a neighborhood through CP variables and constraints. Roughly speaking, a CP model of the neighborhood is created such that every feasible solution of the CP problem represents a move that transforms the current solution into a neighbor solution. Consider a simple neighborhood defined by a $swap(x_i, x_j)$ move, i.e., a move that swaps the values of two variables x_i and x_j; this neighborhood can obviously be explored by two nested loops over indices i and j. Alternatively, the neighborhood may be defined by a CP model with two domain variables I, J and one constraint $I < J$. Every feasible solution (i, j) of the problem defined by variables I, J and constraint $I < J$ uniquely identifies a *swap* move. With such a model, the exploration of the neighborhood by means of iterators (two nested loops) can be replaced by a tree search (such as Branch and Bound for finding the best move).

Formally, the neighborhood $\mathcal{N}(s)$ of a solution s is described by a CP model N_P such that there is a one-to-one mapping between the set of solutions of N_P and the set of neighbors $\mathcal{N}(s)$. We refer to N_P as the neighborhood model and to its decision variables as neighborhood variables. In the framework proposed by [132], local search is then described as a sequence of CP tree search on auxiliary problems N_P.

While searching for a neighbor, two CP models are active: the original model for P, and the neighborhood model for N_P. The two models communicate through *interface constraints* linking variables across P and N_P.

In the example given before, the interface constraints are:

$$\begin{aligned} &x[I] = s[J] \land x[J] = s[I] \\ \forall k \quad &I \neq k \land J \neq k \Rightarrow x[k] = s[k] \end{aligned}$$

In addition, a cost function for the neighborhood model N_P can be defined, and Branch and Bound search can be used on N_P to find the best neighbor. These CP models supports fast neighborhood explorations. Indeed, constraints are used not only for testing the feasibility of solutions (neighbors) once they have been generated, but also for removing during the search, through propagation, sets of infeasible neighbors. For instance, the values of the already instantiated neighborhood variables may cause the reduction of the domains of the problem variables through the interface constraints and the domain reductions for the problem variables may, in turn, back-propagate on other not yet instantiated neighborhood variables, removing the possibility to generate infeasible neighbors. Propagation can also reduce the search space when only improving neighbors or only the best neighbors are looked for: the bounding constraint on the cost of the move can propagate out non optimal neighbors. Propagation is thus able to discard infeasible or uninteresting portions of the neighborhood without actually iterating those sets of neighbors. The larger and the more constrained the problem, the more significant the reduction in neighborhood search provided by propagation.

Several other advantages can be identified in such a CP approach. First, a clear separation between problem modeling and problem solving is maintained. Modeling constraints for P are kept separate from the neighborhood model. This supports, for example, the addition of side constraints to P without changing the neighborhood model nor the search methods. Second, any branching scheme may be used for building and exploring the neighborhood search tree. The simplest idea would only instantiate variables from N_P, but branching may also be performed on variables from P or on variables from both P and N_P. In addition, efficient exploration strategy like Limited Discrepancy Search may be used instead of Depth First Search. Few works have started taking advantage of this flexibility and the assessment of its interest is still an open research issue.

The main limitation of this approach lies in the overhead from the CP model and the propagation engine. CP models of the neighborhoods are of interest only when propagation produces a significant reduction of the search space; in such cases, the CP search of the neighborhood generates much fewer neighbors than the nested loop iteration. Moreover, in a variable selection - value selection branching strategy, the search tree exploration keeps instantiating and uninstantiating variables (upon backtracking); specific branching schemes have been proposed in [152] to reduce the overhead of this tree search exploration.

2.4.2 Tree Search

A search method based on tree search is identified by two characteristics: the branching scheme defines the "shape" of the tree, while the exploration method defines the subtree that will be actually explored. Focusing on the family of constructive algorithms, this section first briefly describes some commonly used branching schema, then it describes several techniques that, on the scale from greedy algorithms to complete global search, are useful for achieving interesting compromises between solution quality and search time.

Constructive algorithms

A global search algorithm produces a solution by taking decisions and backtracking on failure. The decisions taken in a branch amount to adding a constraint to the problem. Some general branching scheme will select any variable from the model (e.g., the one with the smallest number of values in its domain) and instantiate it: in such general cases, it is often difficult to interpret the state of the system before a solution has been reached. The situation is different for some branching schemes that are problem specific and where the decisions at each choice point build a small part of the final solution. For instance, in the case of vehicle routing, insertion algorithms consider customers one by one and decide the route that will visit them; for scheduling, ranking algorithms construct the schedule of a machine in a chronological manner by deciding which task should be sequenced first, which second, and so on; for time-tabling, assignment algorithms decide of the duty of a person (or a group of people) for one time-slot. Such global search algorithms are called *constructive search algorithms*: their states may indeed be interpreted as relevant partial solutions (routing plans for a subset of the customers, short-term schedules planning only a subset of the tasks or time-tables for a subset of the people) and it is easy to evaluate a bound of the objective function by adding the contribution from past decisions to an evaluation of the impact of the decisions to come.

The search in a constructive algorithm is guided by a heuristic[2]: at each choice point, a function h is evaluated for all possible choices and the choice are ranked by increasing values of h: the choice that minimizes h is considered the preferred decision. Although the heuristic used is almost always problem dependent, some generic principles can nevertheless identified.

First-Fail heuristic

The First-Fail heuristic [83] suggests to concentrate on solving first the most difficult part of a problem. For instance, in scheduling problems, if there exists a resource representing the bottleneck of the problem, it is wise to schedule this resource first, and deal with the others only after the bottleneck resource is solved. Very often, in a CSP, the First-Fail heuristic is interpreted as choosing the variable with the smallest domain and instantiate it to a value. Sometimes it is also interpreted as choosing the variable involved in the the highest number of constraints. This heuristic mainly focuses on the feasibility of a problem, ignoring any optimality issues.

Relaxation based heuristic

Typically, in ILP the branching strategy considers the optimal solution of the relaxed problem. If this optimal solution satisfies all the constraints of the problem, it is also a solution for the original problem. Otherwise, it violates some problem constraints. A frequently used heuristic branches on one violated constraint and strengthens it. For example, when the linear relaxation is used, the branching usually takes a variable x_i with non integer value in the optimal relaxed solution ($\neg\ integer(x_i^*)$), and imposes $x_i \geq \lceil x_i^* \rceil\ OR\ x_i \leq \lfloor x_i^* \rfloor$). The advantage is that in each generated branches one of the violated constraint of the parent node is removed. In addition, the optimal solution of the relaxed problem can be used as a *suggestion*. In the previous example, suppose that $x_i \in \{0, 1\}$, and suppose that $x_i^* = 0.1$;

[2]Note that, in the CP context, the word 'heuristic' does not refer to an approximation algorithm, but to a function used to compare different branches at a choice point. In the remainder of the chapter, heuristic will always refer to that meaning.

it is wise to branch on x_i by trying to assign the value 0 (the closest to the value of the optimal relaxed solution) first, and to assign the value 1 on backtracking. Alternatively, LP sensitivity analysis can be used to select the value that generates the smallest increase of the lower bound.

Regret based heuristic

A CP branching strategy that takes into account the objective function is the *max-regret* heuristic. The regret of a variable x can is defined as the additional cost to be paid over the optimum, if x is *not* assigned to its optimal value. Clearly, the regret cannot be calculated unless all the solutions of the problem are known; nevertheless, even an approximation of the regret can be a useful information for defining heuristics. The *max-regret* heuristic suggests to choose first the variable having the highest regret so as to minimize the risk of paying a high cost if the best assignment becomes infeasible due to wrong heuristic decisions.

Consider 2 variables x_1, x_2, with current domain $D_1 = D_2 = [0, 1]$. Suppose that assigning x_1 to 0 costs 10, while assigning it to 1 costs 100, assigning x_2 to 0 costs 20, while assigning it to 1 costs 40. The regret of x_1 can be calculated (heuristically evaluated) as $100 - 10 = 90$, while the regret of x_2 is $40 - 20 = 20$. The max-regret heuristic suggest to assign x_1 first because if its best value (0) becomes infeasible, the objective function will increase of 90. In Section 3.4.2 and [67] a method to evaluate the regret by using optimization constraints is presented.

Texture based heuristic

Texture measurement [16] consists in calculating statistical information on possible conflicts. This information is then used as a basis for the branching strategies. For example, in a scheduling problem, a curve describing the probability of each activity to appear in each point in time in each resource can be calculated. These curves give a good evaluation of the potential conflict in the schedule, and can be used to guide the search towards good solutions.

Lookahead heuristic

Simple heuristics for evaluating the interest of a branch are often “myopic” in the sense that they only assess a choice by some of its immediate consequences and not by long-term consequences on the planning process. For instance, in vehicle routing one may evaluate the insertion of a client in a route by the minimal distance between the client and any other client already in the route. A possibility for taking into account such far-reach effect consists in going down the branch, fully propagating the effects of the choice and evaluating a heuristic only thereafter. In the example of vehicle routing, this amounts to performing the insertion of the client at the best place in the route, propagating the consequences of the insertion, and returning the bound of the overall cost. One may also perform a deeper exploration of the subtree below each branch before evaluating and ranking them. In the field of combinatorial optimization, lookahead evaluation is a common way of improving greedy algorithms in vehicle routing (see, e.g., [40]) or scheduling (see, e.g., [48]). An example of lookahead heuristics employed in OR consists in calculating the optimal solution of the relaxed problem for each possible branch of the current node, and to choose first the branch leading to the minimal increase of the lower bound.

Greedy constructive algorithms

In a greedy constructive algorithm, the preferred branch is systematically followed and no backtracking takes place. For pure optimization problems where feasibility is not an issue, such greedy algorithms yield a solution in polynomial time. In case of feasibility issues, the algorithm may produce a partial solution (some customers are not assigned to a route, some tasks are not scheduled, some duties are not assigned in the time-table).

When the optimization system is granted more time than what is required by the greedy constructive algorithm, but not enough for performing a complete global search, local search may be an effective tool for improving the greedy solution. A first idea consists in using the greedy solution as starting point for a descent search or any random walk. However, interesting results can also be achieved by integrating notions from local search directly within the construction process.

The idea is that the construction process should explore a neighborhood of the greedy decision at each step of the construction process. Such a result can be reached in several ways: by considering a subset of the branches that are "close" to the best branch selected by the heuristic, or by trying to improve the current solution by a LS algorithm after each construction step.

Restricted Candidate Lists

At each choice point, the heuristic provides a preferred branch, as well as an indication of the quality of the other branches. In case of binary branching schemes, the heuristic may indicate how close both possibilities are, with situations ranging from near ties to definite choices between one good option and a terrible one. In the case of wider (non binary) branching, the heuristic may consider that some branches are serious competitors to the favorite branch while others are not. In any case, one can explore a subset of all solutions by following only those paths in the tree that never consider a poor branch according to the heuristic. Let $b_1, \ldots, b_k$ be the possible choices (branches), b_1 the preferred one ($h(b_1) = \min_i(h(b_i))$) and b_k the worst one ($h(b_k) = \max_i(h(b_i))$). The idea of Restricted Candidate Lists (RCL, see [60]) is to retain only the good branches and to discard the bad ones. More precisely, given a parameter α such that $\alpha \in [0, 1]$, only those b_i such that $h(b_i) \leq h(b_1) + \alpha(h(b_k) - h(b_1))$ are kept in a RCL while the others are discarded.

The introduction of RCL thus defines a subtree of the overall search tree. For $\alpha = 0$, this subtree covers all solutions reachable by a greedy algorithm; on the opposite $\alpha = 1$ yields a complete tree covering all possible solutions. For intermediate values of α, the subtree contains only solutions whose construction paths are located "around" greedy paths.

Such a subtree can be explored either systematically or not. In both cases it is important to control the global amount by which a solution path will diverge from the heuristic. It is indeed favorable to generate first solutions that diverge little from the heuristic (following good branches from the RCL) over solutions that systematically diverge from the heuristic (following always the worst branches from the RCL).

The GRASP method [60, 41] uses RCL within a randomized version of the greedy algorithm: a randomized version of the overall construction algorithm is run many times. For each construction, at each choice point, one branch is selected at random among the RCL and according to some probability distribution (with decreasing probabilities for $b_1, b_2, \ldots, b_k$). Thus, solutions that are globally closer to the heuristic are generated with an overall higher probability than solutions that are systematically far from the heuristic. Much room is left for tuning a GRASP algorithm, through the probability distributions or through the value of α. For instance, [135] showed that it is more efficient to consider varying values of α, starting with tight ones (around 0, in order to follow the heuristic), and progressively releasing their values to accept locally bad choices (higher values). Another possibility consists in allowing higher values of α early on in the construction process (at the first levels of the tree) and restricting the construction to quasi-greedy choices with low values of α in the end (deep in the tree); this is motivated by the fact that heuristics are often more reliable in the end of the construction process rather than at the beginning.

Discrepancy-based Search

The previous section introduced the notion of a subtree defined by RCL for a given heuristic. This subtree can either be explored by means of randomized construction procedures (GRASP) or by systematic search. This is, in essence, what discrepancy-based search procedures (xDS, [84]) do: this subtree is explored with construction moves and backtrack moves.

xDS add two notions to RCL:

- it keeps track of all nodes where the algorithm has diverged from the heuristic. Such cases when a branch b_i, $i > 1$ is followed are called discrepancies;

- it keeps track of the paths already generated in order to avoid visiting them twice and relying on backtracking to avoid recomputing many times common intermediate nodes.

This global account of the amount of discrepancy from the greedy heuristic is used to drive the exploration towards solutions that diverge little from the heuristic (i.e., following most of the times the branch b_1) before solutions that diverge more from the heuristic (i.e., following many b_i branches with $i > 1$). Thus, xDS methods in a way ensure by explicit discrepancy bounding what GRASP ensures on average, through cumulated probabilities. The underlying principle is the same in both methods: it is assumed that good solutions are more likely to be constructed by following always but a few times the heuristic b_1, rather that by diverging often from it.

The global account of discrepancies can be performed in several manners:

- counting the number of times K the search did not follow the heuristic. Limited Discrepancy Search (LDS, [85]) explores the tree by generating all solutions for increasing values of K;

- counting the number of times the search did not follow the heuristic up to a certain depth. Depth-bounded Discrepancy Search (DDS, [162]) explores the tree by generating all solutions that do not exceed a maximum number of discrepancies up to a certain depth, and then strictly follows the greedy algorithm;

- counting the total divergence in rank between the options taken and the preferred ones (with C denoting the sum of the rank discrepancy, i.e. $i - 1$, for all branches b_i that are followed along the path). For two consecutive choices, this method associates the same divergence to a path taking b_1 and then, b_3 and to a path taking twice the decision b_2. Credit Search, [17] generates all solutions for which C does not exceed a given limit.

Improving solutions

A last possibility for enhancing an (incomplete) global search algorithm consists in applying some local search steps.

- Local search can be applied at a leaf of the global search tree for improving a solution. This is a straightforward generalization of local search methods which build a solution by a greedy algorithm. Global search is simply used as a way to generate several initial solutions on which a local search improvement phase is applied. It is interesting to generate starting solutions that are different enough for the overall exploration to be rather diversified. LDS is an interesting way of generating such a diversified initial set of solutions.

- Local search can be applied at internal nodes of the global search tree for repairing or improving a partial solution. Designing such moves in the general case of any CP model may be hard. Indeed, such moves must handle partial assignments (producing a partial assignment similar to the current one). A simple idea of neighborhoods consists in selecting a set V_1 of variables that are instantiated in the current partial solution, produce a neighbor assignment of V_1, and apply a propagation algorithm on the overall problem to reduce the domains of variables not in V_1. The global search process can then continue from this improved partial state. Such methods have proven successful on routing problems. [149] introduced the idea of applying local moves every t steps of insertion: each local move tries to improve the partial plan (routes visiting a subset of the clients) by another partial plan, visiting the same clients, but in a different order, before continuing the insertion process. [40] compared the method that applies LS after each insertion step (Incremental Local Optimization, ILO) to the method that constructs a solution by greedy insertion and then improves it by LS. They showed that ILO was not only faster but also produced much better solutions.

A key element to be considered with LS is the evaluation of partial solutions. The quality of a move can easily be measured when LS is applied on completely instantiated solutions. On the other hand, when LS is applied on partial solutions, evaluating a move may be more difficult. It is usually interesting to consider bounds on the objective function, possibly with the addition of a term evaluating the difficulty of extending the partial solution into a complete one.

2.5 Conclusions

This brief overview has been undertaken to identify the main directions for the integration of MP techniques and CP(FD). MP techniques can be used to enhance efficiency, scalability and optimality convergence of CP methods. However, while MP can also benefit from such integration, few studies have been performed in this direction. From a problem solving point of view CP has been used [82] to generate a good feasible solution as *warm* start for the simplex algorithm. From a modeling perspective, CP offers a useful potential for helping MP techniques, as in the investigations of [93, 159], where a single modeling language is used for both CP and MP methods. Moreover, Kasper [101] proposes a strategy of embedding global ILP constraints within structured sub-problems that can be implemented and applied once for all by the user in an ILP program.

Hybrid combinations have been described for local search methods as well as for constructive global search methods. In these cases, CP provides the user with clean formalism to combine technologies: use propagation to reduce the size of neighborhoods, use global search techniques to explore a neighborhood, control the divergence of a construction process from a greedy path, etc. In a sense, CP supports a clean engineering of many "good old tricks" from Operations Research. The first clean integration has been the expression of neighborhoods with CP models. CP models could be introduced for many other algorithm engineering purposes such as objective function combinations for multi-criteria optimization (see, e.g., [64]), or solution fragment combinations for population-based optimization methods. All these are open research topics. The definite impact of CP to Operations Research in general, and to LS in particular, is the introduction of structuring objects that provide a way to easily combine techniques.

An interesting integration technique that was ignored in the discussion is *column generation*, e.g., Fahle et al. [58], where column generation techniques are used together with CP to solve crew assignment problems. The idea is to use Linear Programming to solve the master problem, while modeling the sub-problem as a constraint satisfaction problem using CP(FD).

The field of integrating CP and MP is quite young and many interesting directions for research are still open. Many MP techniques have yet to be explored for their possible incorporation within CP (for example, additive bound procedures, dynamic programming, surrogate constraint relaxations, and pricing techniques). Similarly, a variety of CP techniques remain incompletely examined for their potential application within MP (such as the use of local cuts derived from constraint propagation, special forms of incremental algorithms, and novel approaches for problem modeling). From a methodological point of view, a still open issue is to clearly determine which technique or which integration of techniques is more suitable for a given application.

Chapter 3

Global Optimization Constraint

In this chapter, the integration of various relaxations in Constraint Programming global constraints is proposed. Its main motivation is the introduction in CP languages of some form of *optimality reasoning.* CP languages provide effective and powerful tools for reducing those parts of the search space leading to infeasible solutions; however, they barely consider the problem objective function and implement a naive form of Branch and Bound that poorly reduces those parts of the search space leading to suboptimal solutions.

CP global constraints define subproblem abstractions that can be composed to model complex problems and exploit powerful filtering algorithms to reduce the search space. Many global constraints have a corresponding Integer Linear model that can be relaxed in various way; the optimal solution of the relaxation provides useful information that can be used to reduce the search space and guide the search. Thus, we propose to embed relaxations within global constraints, which are, therefore, enriched with an *optimization component* able to efficiently solve relaxations by exploiting different OR techniques.

The optimization component provides important information that can be used both for filtering purposes and for guiding the search. Filtering, in this setting, is aimed at removing, from variable domains, values that cannot lead to solutions whose cost is better than the cost of the best solution found. Having the optimal solution of the relaxation, the search can be guided toward most promising (in terms of costs) portions of the search space. Different forms of relaxation are discussed, i.e., linear relaxations, structured relaxations corresponding to well known problems, relaxations tightened by the generation of cutting planes, and relaxation based on Dynamic Programming. The general technique is exemplified through the definition of a global constraint, showing how the different types of relaxation can be used to solve TSP and TSPTW. Computational results for routing, timetabling and scheduling problems will be provided in Chapter 4.

Global constraints enable to perform effective pruning on the basis of *feasibility* reasoning; values are removed from domains when proven infeasible. In optimization problems global constraints can also be used for performing *optimality* reasoning: values are removed from domains if are proven sub-optimal. Works in this direction are, for example, [143], [38], [39], [67], [69].

We propose to embed in global constraints an *optimization* component, representing a proper relaxation of the constraint itself. This component provides three pieces of information: (i) the optimal solution of the relaxed problem s^*, (ii) the optimal value of this solution LB representing a lower bound on the original problem objective function, and (iii) a *gradient* *function* $grad(V, v)$ which returns, for each possible pair variable-value (V, v), an optimistic evaluation of the additional cost to be paid if v is assigned to variable V. The *gradient function* extends and refines the notion of regret used in [38] and [39]. We exploit this information both for propagation purposes, achieving the so called *cost-based domain filtering*, and for guiding the search toward promising (in terms of costs) areas.

The structure of the relaxation will be extensively discussed. Global constraints can often be linearized and have a corresponding Integer Linear model. These models can be relaxed in various ways: a commonly used relaxation is the *linear relaxation* obtained by removing the integrality constraint. The linear relaxation can be solved through a general purpose linear solver, like the simplex algorithm. In some cases, the linear relaxation has a special structure or can be transformed in order to obtain a structured problem that can be solved by special purpose solvers. Cutting planes techniques can also be used to strengthen the relaxation and obtain more accurate bounds on the problem. We investigate the use linear relaxation and cutting planes techniques within constraint programming; cutting planes are added to the relaxation in different ways, both directly to the problem or relaxed in a lagrangian way to obtain a structured problem. Finally, relaxations not based on Linear Programming are also considered and their integration into global constraint is proposed.

This approach has been implemented on two global constraints in ILOG Solver [139]: a constraint of difference and a path constraint. We have run experiments using different optimization components (a generic linear solver as well as specialized algorithms for well structured relaxation). The resulting constraints are used to solve Timetabling Problems, Traveling Salesman Problems, its time constrained variants, Scheduling Problems with setup times, and Min-Sum Tardiness Scheduling problem. The cost-based domain filtering technique significantly improves the computational results with respect to traditional CP approaches.

3.1 Motivations

In this section, we recall the general framework, Branch & Infer, proposed by Bockmayr and Kasper [27], which unifies and subsumes Integer Linear Programming (ILP) and Constraint Programming (CP). The optimization constraints are analyzed with respect to this framework, and the framework itself is extended to take into account the form of inference provided by the optimization component.

Branch & Infer is based on a set of transition rules of the form:

$$\_\_ : \frac{\langle P, S\rangle}{\langle P', S'\rangle}\ if\ Cond$$

representing the transition from the computation state $\langle P, S\rangle$ to the computation state $\langle P', S'\rangle$ where P and P' represent disjunction of combinatorial (sub)problems $\{C_1, ..., C_m\}$ and S and S' denote sets of feasible solutions.

In a constraint language, the authors recognize two kinds of constraints: *primitive* and *non primitive* ones. Roughly speaking, primitive constraints are those easily handled by the constraint solver, while non primitive ones are those for which it does not exist a (complete) method for satisfiability, entailment and optimization running in polynomial time. Thus, the purpose of a computation in a constraint-based system is to infer primitive constraints p from non primitive ones c through the transition rule called bi_infer in Figure 3.1.

Obviously, a complete reduction is, in general, not possible, so a partial reduction is performed and tree search (branching) is used to solve the problem. The corresponding

$bi\_infer : \frac{\langle (c \uplus C) \uplus P, S \rangle}{\langle p \uplus (c \uplus C) \uplus P, S \rangle}$	if $Prim(C) \land c \rightarrow p$ and $Prim(C) \not\rightarrow p$
$bi\_branch : \frac{\langle C \uplus P, S \rangle}{\langle \{(c_1 \uplus C), \ldots, (c_k \uplus C)\} \uplus P, S \rangle}$	if $C \equiv C \land (\bigvee_{i=1}^{k} c_i)$ and $Prim(C) \not\rightarrow c_i \; i = 1..k$
$bi\_clash : \frac{\langle C \uplus P, S \rangle}{\langle P, S \rangle}$	if $Prim(C) \rightarrow \perp$
$bi\_climb : \frac{\langle \{(C, C_1, \ldots, C_n\} \uplus P, \{s\} \rangle}{\langle \{(c \uplus C), (c \uplus C_1), \ldots, (c \uplus C_n)\} \uplus P, \{s^*\} \rangle}$	if $s^* = extract(Prim(C))$ and $f(s^*) < f(s)$ and $c \equiv (f(x) \leq f(s^*) - 1)$
$bi\_bound : \frac{\langle C \uplus P, \{s\} \rangle}{\langle P, \{s\} \rangle}$	if $f(s) \leq lb \leq min\{f(x) \mid x \in sol(C)\}$

Figure 3.1: Branch & Infer transition rules

transition is *bi_branch* in Figure 3.1. Splitting can be avoided if a sub-problem is infeasible; see the transition rule *bi_clash* in Figure 3.1.

When solving optimization problems, CP systems usually perform a naive form of branch and bound. In particular, each time a feasible solution s^* is found (whose cost is $f(s^*)$), a constraint $f(x) < f(s^*)$ is added to each subproblem in the remaining search tree. The purpose of the added *upper bounding constraint* is to remove portions of the search tree that cannot lead to better solution than the best one found so far. This approach has two main limitations: (i) there is no good information on the problem lower bound, and consequently, on the quality of the solutions found; (ii) the relation between the cost of the solution and the problem variables is in general not tight, in the sense that is represented by a non primitive constraints.

Many works have been proposed in order to solve the first problem by computing a lower bound on the problem, thus obtaining in CP a behaviour similar to the OR Branch and Bound technique. In global constraints, for example, a lower bound is computed on the basis of variable bounds involved in the constraint itself, see for instance [153]. Alternatively, Linear Programming can be used for this purpose as done for example in [21], [37], [146]. The corresponding transition rule is called *bi_bound* (see Figure 3.1).

The second problem arises from the fact that in classical CP systems primitive constraints are the following:

$$Prim = \{X \leq u, X \geq b, X \neq v, X = Y, integer(X)\}$$

where X and Y are variables, u, v, b are constants. All other constraints are *non primitive*. The Branch and Bound *a-la* CP would be very effective if the *upper bounding constraint* would be a primitive constraint. Unfortunately, in general, while the term $f(s^*)$ is indeed a constant, the function $f(x)$ is, in general, not efficiently handled by the underlying solver.

For example, in scheduling problems, the objective function may be the *makespan*, which is computed as the $max_{i \in Tasks}\{start_i + d_i\}$, where $start_i$ is a variable representing the start time of Task i and d_i its duration. There are also more complex cases where the objective function and the problem decision variables are even more loosely linked. For example, in matching, timetabling and traveling salesman problems, each variable assignment is associated with a cost (or a penalty), the objective function measures the sum of the assignment costs. Also in these cases, the objective function f leads to a non primitive upper bounding constraint.

The general idea proposed is to infer primitive constraints on the basis of information on costs. We enrich global constraints with an *optimization* components that is able to optimally (and efficiently) solve a proper relaxation of the problem (or exactly the same problem) represented by the global constraint itself. The optimization component provides (*i*) the optimal solution of the relaxed problem, (*ii*) its value and (*iii*) a gradient function computing the cost to be added to the optimal solution for each variable-value assignment.

The optimal solution of the relaxed problem can be used as heuristic information as explained in Section 3.4. The optimal value of this solution improves the lower bound of the objective function and prunes those portions of the search space whose lower bound is bigger than the best solution found so far. The corresponding transition rule is called *bi_bound* (see Figure 3.1). The gradient function evaluates the cost to be added to the optimal solution of the relaxed problem if a given variable-value becomes part of a solution; if this sum is greater than the best solution found so far, the value is removed by the domain of the variable. Thus, starting from a non primitive constraint c, we can infer primitive constraints of the kind $X \neq v$, and we prune the subproblem defined by the branching constraint $p = (X = v)$. This technique is related to the variable fixing [121] technique in OR. The advantage with respect to traditional OR variable fixing technique is that in our case domain filtering usually triggers propagation of other constraints through shared variables.

$$bi\_grad : \frac{\langle (c \uplus C) \uplus P, \{s\} \rangle}{\langle \neg p \uplus (c \uplus C) \uplus P, \{s\} \rangle} \; if \; lb + grad(p \uplus (c \uplus C)) \geq f(s)$$

The advantage of this approach is twofold. First, we exploit cost-based information for domain filtering in global constraints. Second, a proper relaxation of the original problem does not need to be defined for every problem, but a proper relaxation is associated to each global constraint written once for all, and that can be used for many optimization purposes.

Thus, on the one hand we have CP global constraints providing filtering techniques and general subproblem abstractions that enable to model complex problems in a declarative way. On the other hand, we have OR techniques providing powerful reasoning on the optimization side.

3.2 Integration of OR techniques in CP through global constraints

It is widely recognized that integrating inference in the form of constraint propagation and relaxation can yield substantial advantages in CP languages. In this setting, global constraints play an important role for the integration since they embed propagation algorithms and are declarative representation of subproblems that can be linearized/relaxed. In particular, concerning linear relaxations, two different ways of integrating a Linear Programming model as the result of the linearization of CP global constraints have been proposed.

The first relies on the automatic linearization of each single global constraint [146, 141]. The corresponding set of linear inequalities is stored in the so called LP store which represents a general Linear Program and can be solved by an LP solver like the simplex algorithm. The resulting LP represents the linearization of the original problem where all constraints are considered and linearized. The LP model is conceptually separated from the CP model even if the two models tightly interact through variable fixing, bound tightening and cutting planes generation [140], (see the conceptual architecture depicted in Figure 3.2).

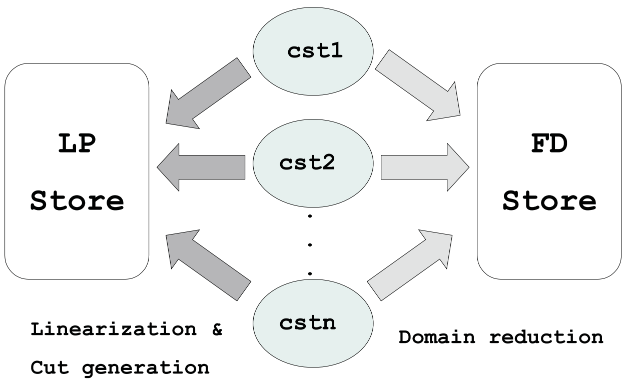

Figure 3.2: Linearization of all constraints

The second relies on the use of relaxation within each global constraint, obtaining what we call *global optimization constraint*. A global optimization constraint, beside a filtering algorithm, embeds an optimization component representing a relaxation (possibly a linear relaxation) of the constraint itself. The optimization component provides information on the problem bound and on the variable-value assignment cost that can be used for filtering purposes. However, this information is *local* to the constraint (see the conceptual architecture depicted in figure 3.3).

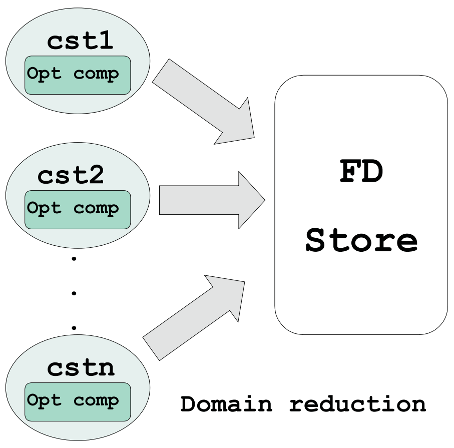

Figure 3.3: Optimization component in each constraints

Each approach has its advantages and drawbacks. Clearly, the first approach benefits from a *global view* of the problem. In fact, since the set of linear inequalities are considered and solved all together, the bound of the problem is in general tighter than the one obtained by considering the linear component of each constraint separately and taking the best bound. Moreover, cutting planes can be generated both considering the semantic of the global constraints involved, and by exploiting the structure of the overall problem. On the other hand, the resulting LP problem can be big and the LP computation can be too expensive with respect to the increased pruning achieved. Finally, the bound of the problem can only be calculated using an LP solver.

The benefits of the second approach are mainly due to the fact that the dimension of the relaxation of a single constraint is smaller than the relaxation of the overall problem, and, more important, that by embedding a relaxation within a global constraint, the structure of the problem represented by the constraint can be better exploited. The bound obtained could be tighter than the one obtained by considering the simple linear component of the constraint and it could be computed more efficiently. Consider, for example, the `allDifferent` constraint (see, e.g., Section 1.3.2), its linear representation is an Assignment Problem (AP) which is a structured problem that can be solved by means of special purpose algorithms more efficient than a generic LP solver. As in the first approach, in the second approach, the structure of the problem represented by the constraint enables the use of *global constraint specific* cutting planes generation techniques. For instance, for the TSP many valid inequalities can be added for tightening the LP relaxation. Thus, beside (or within) the optimization component, one or more cut generator can be embedded in the constraint.

In summary, the advantage of embedding an optimization component within a global constraint concerns the fact that we are not forced to use (*i*) a linear relaxation of the constraint, (*ii*) linear binary variables for modeling the constraint and (*iii*) a linear solver for solving it. For example, dynamic programming relaxation could be used instead.

The first approach is more *Operations Research oriented*, where the linear relaxation of the overall problem is often generated. The second approach is more *Constraint Programming oriented*, where each constraint represents a subproblem that can be considered locally and composed with other constraints in order to model and solve the problem. The filtering algorithm and the operational semantics of the constraint are hidden in the constraint in a software engineering fashion.

A drawback of the second approach is that the optimization component has only a local view on the entire problem. For example, the optimization component of a path constraint could be improved in case the path contains also information about precedence relations among nodes. When solving a TSPTW, we have indeed a path constraint and also a precedence graph constraint (see, e.g., [105, 95]) able to receive precedence relations from the user, or to deduce them from the time bounds of the visits. In this case information deriving from the two constraints could be exploited in order, say, to generate cutting planes (see the conceptual architecture depicted in figure 3.4). For example, the cutting planes described in Section 3.5.5, called SOP, have been designed explicitly for the Traveling Salesman Problem with Precedence constraints. In order to exploit these kind of cuts an additional constraint must be designed, merging the path constraint and the precedence graph and embedding a SOP cut generator. The design of new constraints for increasing the efficiency in solving real life problems is common practice in CP languages. The design of new constraints merging the a set of global constrains defines an intermediate approach between the two presented before.

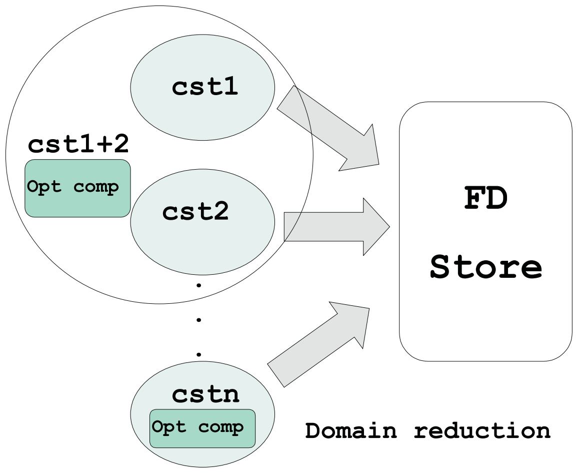

Figure 3.4: Merging of two constraints

3.3 Global optimization constraints

In this section, we discuss the structure of the global constraint, the interface of the optimization component, the mapping between CP variables and variables used by the optimization component, and the triggering events.

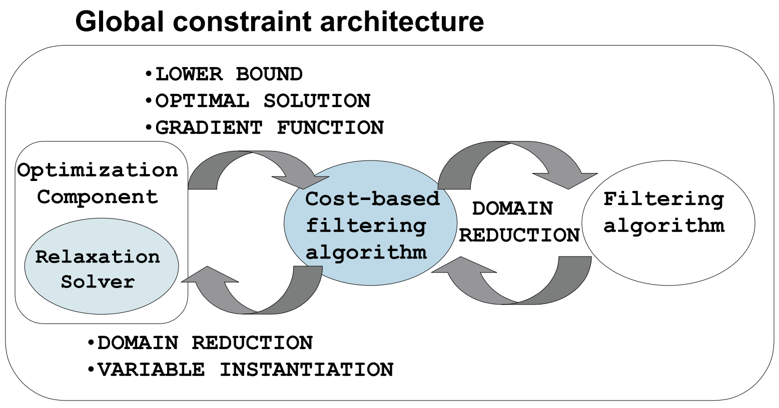

Figure 3.5: Global constraint architecture

3.3.1 Global constraint architecture

Beside the problem decision variables $x_1, \ldots, x_n$ involved in the constraint, the optimization constraint takes the variable z representing the problem objective, and the cost function.

$$globalOptCst(x_1, \ldots, x_n, z, cost)$$

Conceptually, a traditional CP global constraint embeds a software component containing a filtering algorithm pruning domains on the basis of *feasibility reasoning*: a value is removed from the domain of a variable if proven infeasible. Constraints interact with each other through the modification of the domains of the variables involved. Intuitively, we want to perform pruning on the basis of *optimality reasoning*. A value is removed from the domain of a variable if proven sub-optimal. For this purpose, the conceptual architecture of a CP global constraint is extended with two additional components: an optimization component and a cost-based filtering algorithm.

The optimization component is a solver able to compute the optimal solution of a relaxation of the problem represented by the global constraint. This relaxation depends on the declarative semantic of the global constraint and on the objective function of the problem. In general, the optimization component is based on OR techniques (either special purpose algorithms for structured problems, or general purpose LP algorithms).

In Section 3.5.2 several relaxations of a path constraint will be discussed, together with their corresponding optimization components. Here, we just provide an intuition: the path constraint can be seen as a declarative representation of the (feasibility side of the) Hamiltonian Path Problem. Given a set of internal nodes, one start node, and one end node, the path constraints holds if there exists a path starting from the start node, covering all internal nodes exactly once, and ending at the end node. The Hamiltonian Path Problem can be easily transformed in a Traveling Salesman problem (and vice-versa). For the TSP, a commonly used relaxation is the Assignment Problem (see Section 3.5.3). Thus, the optimization component of the path constraint can be an AP solver. Other relaxations for the TSP can be the Minimum Spanning Arborescence (see Section 3.5.4), relaxation based on Dynamic Programming (see Section 3.5.6), or Linear Programs strengthened with cutting planes (see Section 3.5.5) .

The optimization component can be the solver exploiting the most effective technique for the chosen relaxation of the global constraint. In the above mentioned example, a general purpose LP solver can be used, or alternatively, an AP solver, an MSA solver, etc.

Independently from the technique used in the optimization component, the optimization component is required calculating the following information:

- the optimal solution of the relaxed problem s^*;
- the optimal solution value LB. This value represents a lower bound on the problem corresponding to the global constraint, and therefore on the overall objective function value;
- a gradient function $grad(V, v)$ measuring the variable-value assignment cost.

These pieces of information are exploited both for filtering purposes and for guiding the search toward promising (in terms of costs) branches.

In this section we are interested in discussing the filtering aspect, while in Section 3.4 we will discuss how this information can be used for heuristic search.

The communication between the optimization component and the CP variables involved in the global constraint requires a mapping that depends on the chosen solver, and will be discussed in Section 3.3.2. The cost-based propagation algorithm interacts with the optimization solver through this mapping, and with all other feasibility-based algorithm thought the domain reduction of shared varibles.

3.3.2 Mapping

In this section, we define the mapping between variables and constraints used in the optimization component and those used in CP global constraints. The mapping allows the communication between the two components.

Defining a generic mapping between the CP model of the constraint and the model of its optimization component is clearly impossible since every possible optimization component could require a different type of modeling (based on Graph Theory, on Dynamic Programming, on Linear Programming, etc.).

As an example, the mapping between the CP formulation and the LP formulation previously suggested in [146] will be described. Consider a global constraint involving variables $x_1, ..., x_n$, ranging on domains $D_1, ..., D_n$, having cost c_{ij} of assigning value $j \in D_i$ to x_i. Obviously, the cost of each value not belonging to a variable domain is infinite. The problem is to find an assignment of values to variables consistent with the global constraint, and whose total cost is minimal. A mapping CP-LP could create LP variables $y_{ij} \in [0, 1]$; if the CP variable x_i is assigned to the value j, y_{ij} is equal to 1, $y_{ij} = 1 \leftrightarrow x_i = j$. Constraints $\sum_{j \in D_i}(y_{ij}) = 1$ is part of the mapping and imposes that there will not exist more LP variables corresponding to the same CP variable having value equal to 1. Note that, in general, for every solution of the CP constraint there is exactly one corresponding solution of the LP model, but there could be LP solutions of the constraint which do not correspond to any solution for the CP constraint. For example, all the LP solutions having non integer variable values do not have any corresponding CP solution.

Given the mapping between LP and CP variables, we know that the LP variable y_{ij} corresponds to the value j in the domain of the CP variable x_i. Thus, the reduced cost matrix $\bar{c}_{ij}$ provides information on CP variable domain values, $grad(x_i, j) = (1 - y^*_{ij}) * \bar{c}_{ij}$; where y^*_{ij} is the value of the optimal solution in the LP model.

3.3.3 The Cost-Based Propagation

In this section we describe a general filtering algorithm based on the information provided by the optimization component. We recall that the results of the optimization component computation are (i) the optimal solution of the relaxation; (ii) its value and (iii) a gradient function measuring the variable-value assignment cost.

A first (trivial) propagation is based on the optimal solution value LB of the relaxation. This value is a lower bound on the objective function z of the overall problem. Since the objective function in CP is represented by a domain variable, we impose the constraint $LB \leq z$. Since LB is a constant, it updates the lower bound of the domain of z. If LB is greater or equal than the upper bound of the domain of z, a failure occurs. This kind of propagation generates a yes/no answer on the feasibility of the current node of the search tree; therefore, it does not allow any real interaction with the other constraints of the problem.

More interesting is the propagation from the gradient function $grad(x, v)$ towards decision variables $x_1, ..., x_n$. The gradient function provides an optimistic evaluation on the cost of each variable-value assignment. Given this information, we can compute an optimistic evaluation on the optimal solution of a problem where a given variable is assigned to a given value, i.e., $LB_{x=v}$. For each domain value j of each variable x_i, we compute a lower bound value of the subproblem generated if value j is assigned to x_i as $LB_{x_i=j} = LB + grad(x_i, j)$. If $LB_{x_i=j}$ is greater or equal to the upper bound of the domain of z, j can be removed from the domain of z_i. We can impose:

$$LB \leq z \tag{3.1}$$
$$\forall i = 1, \dots, n, \forall j \in domain(x_i) \qquad LB + grad(x_i, j) \leq z \tag{3.2}$$

or, equivalently,

$$z_{min} \geq LB \tag{3.3}$$
$$\forall i = 1, \dots, n, \forall j \in domain(x_i) \qquad LB + grad(x_i, j) \geq z_{max} \Rightarrow x_i \neq j \tag{3.4}$$

As an example, suppose that at a certain node in the tree search the lower bound calculation returns a value equal to 96, and that the best solution found so far has a cost equal to 100 (i.e., $z_{max} = 100$). A lower bound of the problem where, in addition to the current state of the variables, the assignment $x_i = j$ is imposed, is $96 + grad(x_i, j)$. For each pair x_i , j where $grad(x_i, j)$ is greater than 3, value j can be immediately removed from the domain of x_i, since the assignment $x_i = j$ never leads to a solution better than the best one found so far.

This filtering algorithm performs a real back-propagation from z to x_i, and usually triggers other constraints imposed on shared variables; it appears therefore particularly suited for CP. Indeed, the technique proposed represents a new way of inferring primitive constraints starting from non primitive ones. In particular, primitive constraints added (of the form $x_i \neq j$) do not derive from reasoning on feasibility, but from reasoning on optimality. Note that the same constraints of the form $x_i \neq j$ are also inferred in some OR frameworks (variable fixing). However, this fixing is usually not exploited to trigger other constraints, but only in the next lower bound computation, i.e., the next branching node.

3.3.4 Propagation Events

When the constraint is stated for the first time, the optimization component is triggered and calculates the optimal solution of the relaxation. From this first computation, the lower bound of variable z representing the problem objective is updated with the value of the optimal solution of the relaxation.

The constraint is triggered each time a modification in the domain of one variable involved is performed. We have to distinguish two cases:

- the modification happens in the domain of a problem decision variable x_i.
- the modification happens in the domain of the objective function variable z.

In the first case, the feasibility-based filtering algorithm is triggered, possibly reducing other domain variables. The optimization component is triggered only if one of the values removed from the domain belongs to x^*, i.e., to the optimal solution of the relaxation. In fact, only in that case, the value of the optimal solution of the relaxation may change (increase). In all other cases, only the cost corresponding to the value removed is updated to infinite. At the end of the computation of the optimization component, the cost-based filtering algorithm is triggered since a new optimal solution of the relaxation and a new gradient function have been calculated. In the second case, the cost-based domain filtering algorithm is triggered if the upper bound of the objective function has changed (decreased), in this case, in fact, the domain of the variables may be reduced through the propagation based on the gradient function.

3.4 Heuristics

The optimal solution of a relaxed problem, the lower bound value, and the gradient function can be used for guiding heuristic decisions during the search for a solution. Different examples of such a use are described in the next chapter where four combinatorial problems are considered. The information made available by the optimization component can be used in several ways.

3.4.1 Select promising values

A branching decision often consists in assigning, to a given variable, one value out of the domain of possible ones. The value should be chosen considering feasibility and optimality issues. Although the optimization component relaxes some of the constraints of the problem, if a value belongs to the optimal solution of the relaxation it represents a good guess to be chosen as tentative value for the branching variable. Alternatively, the gradient function can be used to evaluate the impact of value-variable assignments on the objective function so as to avoid making heuristic decisions that increases too much the lower bound.

3.4.2 Regret Heuristic

Refining the concept of regret, the gradient function can be used to select the branching variable following a *max-regret* heuristic (see Section 2.4.2). The regret of a variable x can be defined as the additional cost (*regret*) to be paid over the optimum if a x is *not* assigned to its optimal value x^*. Clearly, this additional cost is not known. Being x^*_{rel} the optimal value in the relaxed problem, the gradient function gives a good evaluation of the regret by considering the minimum value of the gradient excluding the gradient of x^*_{rel}.

$$regret = \min_{v \in domain(x), v \neq x^*_{rel}} \{grad(x, v)\}$$

The *max-regret* heuristic is a variable selection heuristic that suggests to assign first the variable with the highest regret so as to minimize the risk of paying a high cost if the best assignment becomes infeasible due to wrong heuristic decisions.

3.4.3 Select promising area

In a large neighborhood framework (see Section 2.4.1), a good evaluation of the lower bound can be used to select promising areas for improvement. Let s be a current solution of problem P, of cost $z(s)$; a large neighborhood defines a subproblem $P_k(s)$ of P. The best improvement E_k that can eventually be obtained in $P_k(s)$ (by solving $P_k(s)$ up to optimality) is equal to the difference between $z(s)$ and the lower bound $LB_{P_k(s)}$ of z calculated in $P_k(s)$. In some cases it may be useful to quickly evaluate the possible improvement E_k of a set of neighborhoods, before trying to reoptimize on one of them. This method has been applied in [66, 30] for improving the solution of a crew rostering problem, and in [64] for scheduling problems with sequence dependent setup times.

3.4.4 Define subproblem

Restricted Candidate Lists methods (see Section 2.4.2) suggest to cut off uninteresting part of the search tree a priori, leading to incomplete tree search. The gradient function can be used to find those assignments that would increase the most the objective function and that could, for this reason, be cut. This method was used in the min sum tardiness problem (see Section 4.5). In this problem, each activity is assigned to a position in the schedule. For each activity, some position (the ones with the highest gradient) are removed, and the remaining subproblem is (quickly) solved using Branch and Bound. The best solution of this subproblem is typically a very good solution of the original problem and is used as starting point for a complete tree search.

3.5 A case study: the `Path` constraint

Most commercial Constraint Programming languages offer a global constraint used to represent paths in a graph. Consider the global constraint *Path* with the following syntax and semantics: `Path(next)` where `next` is an array of $N+2$ variables of index $0, 1, \dots, N, N+1$. Given a set of N internal nodes $0, 1, \dots, N-1$, an end node N, and a start node $N+1$, the path constraint ensures that there exists one path starting from node N, ending in node $N+1$, and connecting all internal nodes. Each internal node is visited only once, has exactly one direct predecessor and one direct successor. Each node i is modeled using a domain variable $next_i$ ($next_i = next[i]$) identifying its next node. If $next_i = j$, node j directly follows node i in the path.

Clearly, the constraint can be used to model TSP and its time constrained variant (TSPTW) by conventionally choosing a start node for the TSP and duplicating it to represent the end node.

In the following, we describe the ILP model of the path constraint and some relaxations that can be embedded in an optimization component to provide information on costs. The resulting constraint is an optimization constraint having the following syntax:

$$pathCost(next, z, cost)$$

where *next* is an array of domain variables representing the next node to be visited, *cost* is the cost matrix containing the costs of connecting each pair of nodes, and z is a variable $z = \sum_{i=0}^{N+1} cost_{i,next_i}$. The constraint is satisfied when the *next* variables form one path and the cost of the assignment is equal to z.

3.5.1 ILP model of the `Path` constraint

Let $G = (V, A)$ be a given digraph, where $V = \{1, \dots, n\}$ is the vertex set and $A = \{(i, j) : i, j \in V\}$ the arc set, and let $c_{ij} \geq 0$ be the cost associated with arc $(i, j) \in A$ (with $c_{ii} = +\infty$ for each $i \in V$). A *Hamiltonian Circuit* (*tour*) of G is a partial digraph $\bar{G} = (V, \bar{A})$ of G such that:

i) $|\bar{A}| = n$;

ii) for each pair of distinct vertices $v_1, v_2 \in V$, both paths from v_1 to v_2 and from v_2 to v_1 exist in $\bar{G}$ (i.e. digraph $\bar{G}$ is *strongly connected*).

An Integer Linear Programming formulation for path constraint is as follows:

$$v(TSP) = \min \sum_{i \in V} \sum_{j \in V} c_{ij}\, x_{ij} \tag{3.5}$$

$$\text{subject to} \quad \sum_{j \in V} x_{ij} = 1, \qquad i \in V \tag{3.6}$$

$$\sum_{i \in V} x_{ij} = 1, \qquad j \in V \tag{3.7}$$

$$\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq 1, \qquad S \subset V : S \neq \emptyset \tag{3.8}$$

$$x_{ij} \geq 0 \text{ integer}, \quad i, j \in V \tag{3.9}$$

where $x_{ij} = 1$ if and only if arc (i, j) is in the optimal solution. Constraints (3.6) and (3.7) impose out-degree and in-degree of each vertex equal to one, whereas constraints (3.8), imposing strong connectivity, are called *Subtour Elimination Constraints* (SECs). Note that the number of SECs grows exponentially with the size of the problem.

This model corresponds to the Traveling Salesman Problem in its *Asymmetric* version (ATSP). Such a problem is known to be strongly NP-hard and is one of the most important and studied problem in combinatorial optimization, finding many practical applications for example in vehicle routing, sequencing, and scheduling.

3.5.2 Relaxations

As mentioned, an important role in the solution of NP-Hard problems via branch and bound is played by the use of relaxations. Given a problem P and applying some relaxations to a subset C of its constraints, we obtain a problem R, relaxation of P, where the solution space of R contains the one of P.

3.5.3 The Linear Assignment Problem

Given a square cost matrix c_{ij} of order n, the linear Assignment Problem [47] is the problem of finding an assignment of a different column to each row, and vice versa, that minimizes the total sum of the row-column assignment costs.

This problem can be seen as the graph theory problem of finding a set of *disjoint* sub-tours on a digraph $G = (V, A)$ such that all the vertices in V are visited once and the sum of the costs of selected arcs in A is a minimum. An ILP model for AP is as follows:

$$z(AP) = \min \sum_{i \in V} \sum_{j \in V} c_{ij}\, x_{ij} \tag{3.10}$$

$$\text{subject to} \quad \sum_{i \in V} x_{ij} = 1, \qquad j \in V \tag{3.11}$$

$$\sum_{j \in V} x_{ij} = 1, \qquad i \in V \tag{3.12}$$

$$x_{ij} \geq 0 \text{ and integer}, \quad (i, j) \in A \tag{3.13}$$

where $x_{ij} = 1$ if and only if arc (i, j) is in the optimal solution. By considering the model of the TSP presented in Section 3.5.1, it is easy to see that the AP is a relaxation of the TSP where the SECs are relaxed (removed in this case).

The coefficient matrix of the AP is totally unimodular. Therefore, the optimal solution of the linear relaxation of the AP is also integer and thus optimal for the AP.

Therefore the AP can be optimally solved using a LP solver; however, one of the most effective special purpose algorithms for solving the AP is a *primal-dual* method, called Hungarian Algorithm [33] and it is based on graph theory. The time complexity of the Hungarian Algorithm is $O(n^3)$, whereas each re-computation of the optimal AP solution, needed in the case of modification of one value in the cost matrix, can be efficiently computed in $O(n^2)$ time.

The information provided by the Hungarian algorithm is the AP optimal solution and a reduced cost matrix $\bar{c}$. In particular, for each arc $(i, j) \in A$ the reduced cost value is defined as $\bar{c}_{ij} = c_{ij} - u_i - v_j$, where u_i and v_j are the optimal values of the dual variables ([33]) associated with the i-th constraint of type (3.11) and the j-th constraint of type (3.12), respectively. Each $\bar{c}_{ij}$ can be seen as a lower bound on the cost to be added to the optimal AP solution value if we force arc (i, j) in solution. Obviously, $\bar{c}_{ij} = 0$ if arc (i, j) is in the optimal solution. The reduced cost values are obtained through the Hungarian Algorithm without extra computational effort during AP solution.

The AP can be used as optimization component for the *Path* constraint previously described. The correspondent optimization constraint *pathCostAP(next,z,cost)* embeds the AP solver and performs propagation based on the value of the optimal AP solution and on the gradient function $grad(next_i, j) = \bar{c}_{ij}$.

A better gradient function can be defined as follows: we want to evaluate if value j could be removed from the domain of variable $next_i$. Let $next_i = k$ and $next_l = j$ in the optimal AP solution. In order to assign $next_i = j$, a minimum augmenting path (see [2] for the definition of augmenting path), say PTH, from l to k has to be determined since l and k should be re-assigned. Thus, the cost of the optimal AP solution where $next_i = j$ is $LB + \bar{c}_{ij} + cost(PTH)$, where $cost(PTH)$ is clearly the cost of path PTH. Two bounds on this cost can be computed by considering the subset of edges starting from l, say S, and the subset of edges arriving in k, say T. By first considering the subset S and denoting with $a = min_{h \in S \setminus \{j\}} \bar{c}_{lh}$, it is easy to prove that a first lower bound on the cost of PTH is:

$$lb_1 = a + min\{c_{lk} - a, min_{h \in T \setminus \{i,l\}} \bar{c}_{hk}\}$$

Analogously, considering subset T and denoting with $b = min_{h \in T \setminus \{i\}} \bar{c}_{hk}$, the corresponding lower bound is:

$$lb_2 = b + min\{\bar{c}_{lk} - b, min_{h \in S \setminus \{j,k\}} \bar{c}_{lh}\}$$

Both lb_1 and lb_2 are valid lower bounds for $cost(PTH)$, hence $grad(next_i, j) = LB + \bar{c}_{ij} + max\{lb_1, lb_2\}$. Note that the information required to compute the improved gradient function can be obtained and stored during the AP solution in $O(n^2)$.

3.5.4 The Minimum Spanning r-Arborescence

The *Minimum Spanning r-Arborescence problem* (r-MSA) is the graph theory problem of finding a partial graph $\bar{G} = (V, \bar{A})$ of a given digraph $G = (V, A)$ such that the in-degree of each vertex is exactly one and there is a path from a given vertex, say $r \in V$, to each other vertex. An ILP model for r-MSA is as follows:

(3.14) $$z(r\text{-}MSA) = \min \sum_{i \in V} \sum_{j \in V} c_{ij}\, x_{ij}$$

(3.15) $$\text{subject to} \quad \sum_{j \in V} x_{ij} = 1, \qquad i \in V$$

(3.16) $$\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq 1, \qquad S \subset V : r \in S$$

(3.17) $$x_{ij} \geq 0 \text{ and integer}, \quad (i,j) \in A$$

where again $x_{ij} = 1$ if and only if arc $(i,j) \in A$ belongs to the optimal solution. Since the SECs (3.8) can be always re-written as in (3.16), the r-MSA corresponds to the TSP relaxation in which the out-degree constraints (3.6) have been relaxed. This problem can be polynomially solved in $O(n^2)$ time (see Fischetti and Toth [63]). It is worth noting that the above relaxation can be different depending on the chosen node r. Finally, the associated reduced cost matrix can be also computed in $O(n^2)$ time.

Also the r-MSA can be used as optimization component for the *path* constraint previously described. The correspondent optimization constraint *pathCostMSA(next,z,cost)* embeds the MSA solver and performs propagation based on the value of the optimal MSA solution and on the gradient function $grad(next_i, j) = \bar{c}_{ij}$.

3.5.5 Strengthening the Linear Relaxation by Cutting Planes

Linear Relaxation strengthened by cutting planes can also be used as optimization component for the `path` constraint.

The AP obtained by eliminating constraints (3.8) and the integrality constraint is considered as initial linear relaxation. The addition of cutting planes is aimed at strengthening (i.e., increasing) the lower bound provided by the AP. Many different classes of valid inequalities have been studied for the TSP and its variants, and each of them can be used to improve the bound, typically in simultaneous ways.

We use the class of subtour elimination constraints (already included in the ILP formulation, constraints (3.8)). In addition, we will discuss the use of *sequential ordering* inequalities (SOPs) ([9]) for the extension of the `path` constraint with precedence relations among nodes.

It is well known that the separation problem (see Section 1.4.2) for SECs can be solved in polynomial time by computing the *minimum capacity cut* in the graph induced by x^*, and the same holds for each following LP obtained by the iterative addition of cuts. The associated minimum cut problem is solved by using the ANSI C implementation [100] of the Padberg and Rinaldi *micut* algorithm [130]. At each iteration, if a violated subtour exists, the procedure returns the set of nodes S involved in the subtour (constraint 3.8). At the end of the iterative process, these inequalities define the subtour polytope of the TSP, i.e., a relaxation in which no subtour is violated, but the solution is possibly fractional. Obviously, the optimization over the subtour polytope provides a valid lower bound for TSP (and TSPTW) which is typically much better than the AP solution value.

In the extension of the `pathCost` constraint with precedence relations among nodes, we also used additional classes of inequalities studied by Balas, Fischetti and Pulleyblank [9] for the *Sequential Ordering Problem* (SOP). Obviously, all the inequalities for this relaxation are also valid for the TSPTW, provided the information on precedences among nodes. In particular, we consider: (i) *Predecessor inequalities* (π inequalities). Let $S \subseteq V$, $\bar{S} := V \setminus S$, then

$$\sum_{i \in S \setminus \pi(S)} \sum_{j \in \bar{S} \setminus \pi(S)} x_{ij} \geq 1 \tag{3.18}$$

where $\pi(S)$ indicates the set of nodes which *precede* the nodes in S (see, [9] for more details); (ii) *Successor inequalities* (σ inequalities). Let $S \subseteq V$, $\bar{S} := V \setminus S$, then

$$\sum_{i \in \bar{S} \setminus \sigma(S)} \sum_{j \in S \setminus \sigma(S)} x_{ij} \geq 1 \tag{3.19}$$

where $\sigma(S)$ indicates the set of nodes which *follow* the nodes in S (see, [9]).

The separation problem is solved by using the heuristic procedure proposed in [9] and the corresponding code by Ascheuer, Fischetti and Grötschel [6].

Integration of cutting planes in global constraints

The main idea is to embed in global constraints, beside an optimization component, one or more cut generators that provide valid linear inequalities depending on the global constraint declarative semantics and the optimal solution of the corresponding linear relaxation. These cuts can be added to the linear formulation, which, in turn, produces better bounds that enable to perform more powerful domain filtering.

Depending on the semantics of the global constraint and its relaxation, we can use different sets of cuts. We have used Subtour Elimination for the path constraint, and we have additionally used Sequential Ordering cuts for the extension of the path constraint with precedence relations. We have followed two different approaches: in the first, cuts are generated and added when the constraint is posted for the first time, but no longer updated during the search; in the second, instead, cuts are generated and added at each node.

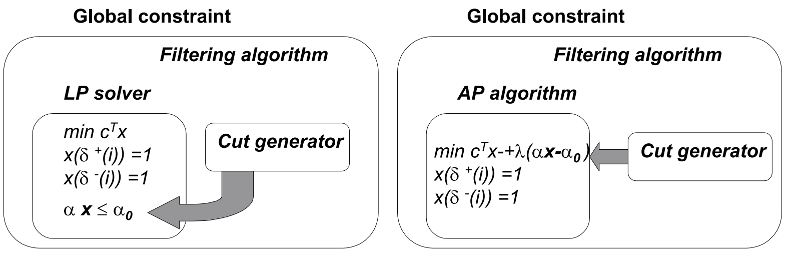

Figure 3.6: Addition of cuts in optimization constraints

Adding cuts at the root node

When cuts are added only at the root node, we have experimented three alternative approaches: (i) cuts can be simply added to the initial LP if the optimization component considered in the global constraint is a general Linear Programming solver able to handle any kind of Linear Problems, with no special structure; (ii) cuts can be relaxed in a Lagrangian way in order to obtain a structured problem (see Section 1.4.2) if the optimization component is a special purpose algorithm aimed at solving a problem with a special structure, say the Assignment Problem; (*iii*) purging techniques can be applied to the second approach in order to remove no longer needed cuts.

Adding cuts to the problem formulation

This first integration can be performed when the optimization component in the global constraint is a general purpose Linear Programming solver. Thus, the computed cuts can be directly added to the problem formulation as shown in the left hand side of Figure 3.6. The resulting linear problem is solved at each node during the whole search space. A linear solver (ILOG Planner [94] in our case) is used to compute the optimal solution of the LP at each node.

In the case of the path constraint for solving TSPs, the starting problem considered in the optimization component is the problem P_{Lin} composed by constraints (3.7)-(3.6) of Section 3.5.1 plus the bounds on variables (which is the constraint set of AP). This problem is an Assignment Problem which is a linear problem. A set of subtours inequalities S^{cut} are generated by the cut generator at each iteration, and the subtour polytope is finally derived. The resulting problem P^{cut}_{Lin} is a Linear Problem composed by P_{Lin} together with S^{cut}. The structure of P^{cut}_{Lin} is no longer an AP, but a general Linear Problem.

Clearly, during the search some added cuts may no longer be effective if they are trivially satisfied by the current partial solution of the problem. In principle, the satisfied cuts can be removed to reduce the linear formulation of the problem. However, leaving these cuts does not affect the accuracy of the bound.

During search, the resulting LP, P^{cut}_{Lin}, is changed at each node when variables are fixed and constraint propagation removes values from variable domains, but not with the addition of new cuts.

A disadvantage of this technique is that the resulting LP can be quite big for non-trivial problems and the LP re-computation at each node of the search tree can be expensive.

Lagrangian relaxation of cuts

An alternative method can be applied when the optimization component in the global constraint is a special purpose algorithm able to solve structured problems. Consider, for example, as optimization component an Assignment Problem solver such as the Hungarian algorithm [33]. The Hungarian algorithm has a polynomial time complexity and an incremental behavior, thus is more suitable in a CP framework with respect to general LP solvers.

In this case, we generate cuts (set S^{cut}) as described in the previous approach. At the end of the cut generation phase we have the Linear Problem P^{cut}_{Lin} composed by the initial Assignment Problem P_{Lin} and the set of cuts S^{cut}. The relaxed problem P^{cut}_{Lin} can be transformed into an AP having exactly the same objective function value, say LB_{Root}, by relaxing in Lagragian way the linear inequalities of set S^{cut}. The optimal Lagrangian multipliers for the cuts are the dual values (associated to the cuts) computed by the linear solver. The problem obtained, say AP$_{Lagr}$, is an AP with a different objective function cost vector, and, as mentioned, by solving AP$_{Lagr}$ (through the Hungarian algorithm) we have an integer solution whose value is LB_{Root}.

More formally, consider the Lagrangian relaxation of the cut $\alpha x \leq \alpha_0$ in the objective function min $c^T x$. We call λ ($\lambda \geq 0$) the optimal Lagrangian multiplier of the cut, i.e., the dual value associated to the cut in the LP solution[1]. The objective function of the Lagrangian relaxation becomes: min $c^T x + \lambda(\alpha x - \alpha_0)$, i.e., $-\lambda\alpha_0+$ min $(c^T + \lambda\alpha)x$. In the objective function, we have a constant factor and we can change each value in the cost matrix c by adding the coefficient corresponding to x (see the right hand side of Figure 3.6).

The AP$_{Lagr}$ solution can be seen as an advanced starting point for the CP algorithm which is now applied exactly as in 3.5.3 to complete the search. AP$_{Lagr}$ is considered at each node as relaxation and modified through variable fixing derived at each step, but no more cuts nor Lagrangian multipliers are re-computed.

The addition in Lagrangian way of the generated cuts to the objective function can have some drawback for the remaining nodes of the search tree. In fact, the Lagrangian multipliers associated to these cuts are fixed to values that are optimal at the root node, but could be very "far" from the optimal ones during search, and no re-optimization (subgradient optimization) is performed. In particular, if during the search, a cut $\alpha x \leq \alpha_0$, which was tight at the root node (i.e., $\alpha x = \alpha_0$), becomes trivially satisfied with respect to the current partial instantiation of CP variables, i.e., $\alpha x < \alpha_0$, its contribution to the objective function represents a penalty with respect to the same solution where the cut is removed. In fact, the term $\alpha x - \alpha_0 < 0$ added to the objective function of the original AP, i.e., min $c^T x$, leads to a worse bound (since $\lambda \geq 0$). Thus, while at the root node the bound produced by the Lagrangian relaxation is in general better (higher) than the bound produced by the AP solution (without cuts), it decreases during the search as cuts become no longer tight, and, at some point, it becomes worse than the AP bound. In Figure 3.7 we depict the qualitative trend of the lower bound produced by the Lagrangian relaxation LB_{Lagr} versus the bound produced by the AP, LB_{AP}.

However, the optimization over AP$_{Lagr}$ is always a valid lower bound for the problem, and the above limitation can be removed either by calculating both LB_{AP} and LB_{Lagr} and keeping the best of the two, or by performing some kind of *purging* to disable, during the search, those cuts which are no longer necessary, i.e., both not tight and trivially satisfied.

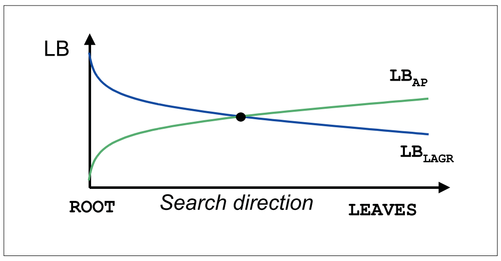

Figure 3.7: Trend of lower bounds in different relaxations

Purging

During the search, the current instantiation of CP variables and the reduction of their domains can lead the corresponding linear variables to assume values that trivially satisfy some of the

[1]Since the cuts are in ≤ 0 form, the corresponding dual variables are non-positive, thus they must be inverted to be used as Lagrangian multipliers.

cuts. Suppose, for example, that a generated cut has the form: $x_{12}+x_{21}+x_{13}+x_{31}+x_{23}+x_{32} \leq 2$ and removes the subtour visiting nodes 1, 2 and 3. Moreover, suppose that during search the CP variable $next_1$ is instantiated to the value 2 (thus, $x_{12} = 1$ and $x_{13} = 0$), values 1 and 3 are removed from the domain of variable $next_2$ (thus, $x_{23} = 0$ and $x_{21} = 0$) and values 1 and 3 are removed from the domain of variable $next_3$ (thus, $x_{31} = 0$ and $x_{32} = 0$). The cut is satisfied and can be removed to reduce the linear formulation of the problem. Moreover, in case of Lagrangian relaxation, the added and satisfied cut can produce a penalty in the objective function and it is preferable to remove it. Note that the concept of purging is standard for branch-and-cut (cuts which are no longer "effective" are removed and included in the *Cut Pool*). However, in case of Lagrangian relaxation the performed operation is even more crucial since it affects the value of the bound.

Thus, during search, each time a linear variable x_{ij} is fixed all added cuts are taken into account and removed if they are trivially satisfied. In particular, in the Lagrangian relaxation the contribution of the cut to the objective function $-\lambda\alpha_0 +$ min $(c^T + \lambda\alpha)\,x$ should be removed, by restoring the cost matrix to its original values and removing $\lambda\alpha_0$ from the objective function.

A drawback of the purging technique is that each time the cost matrix is changed, i.e., one or more cuts are removed, the AP plus the remaining cuts (always relaxed in a Lagrangian way) should be re-computed from scratch leading to an $O(n^3)$ algorithm instead of $O(n^2)$.

An interesting point concerns the exploitation of the problem specific knowledge on cuts. Consider the above cut involving nodes in the set $S = \{1, 2, 3\}$. We know that as soon as an arc starting (resp. ending) from (resp. in) one of the nodes 1, 2 or 3 and ending (resp. starting) in a node not contained in S belongs to the current solution, the subtour is not violated. This conclusion can be drawn despite the values of the other variables are still unbound. Thus, it is much more powerful to exploit information on the special structure of cuts since we can remove trivially satisfied cuts earlier. The cuts have been removed when variable values lead the constraint to be trivially satisfied. An interesting extension is currently investigated: we can associate to the cut generator an event that is raised each time some variables in the cut are changed (due to CP propagation). The event triggers an evaluation procedure which determines if the cut should be removed or not on the basis of its semantics.

Adding cuts at each node

An alternative use of cuts, closer to the traditional Operations Research branch and cut technique, is to add cuts at each node. During search, we acquire more information on the portion of the search tree we are exploring, thus we can generate more *informed* cuts. We have experimented the addition of cuts at each node after the computation of the optimal solution of the LP corresponding to that node.

An important point concerns the use of local or global cuts. The SECs are globally valid cuts, while SOPs are locally valid since they may correspond to precedence relations deduced as consequences of branching decisions. Thus, while SECs can be left in the problem formulation when backtracking is performed, the SOPs must be removed. However, we have decided to remove also added SECs upon backtracking in order to limit the size of the generated linear problem.

3.5.6 State Space Relaxation

DP model

Consider the Dynamic Programming model of a TSP. Let $G = (V, A)$ be a digraph, and c be the cost matrix that associates the cost c_{ij} to the arc $(i, j) \in A$. D_i is the subset of arcs $(i, j) \in A$, and E_i is the subset of arcs $(j, i) \in A$. Let $0, 1, \ldots, n-1$ be the vertices in V, and $V' = V \setminus \{0\}$.

For any subset of vertices $Y, Y \neq \emptyset, Y \subseteq V'$, and a vertex $i \in Y$, let define $f_k(Y, i)$ as the cost of the shortest path from vertex 0 to vertex i passing by all vertices in $Y \setminus i$ exactly once. The index k in the definition of $f_k(Y, i)$ represents the cardinality of the set Y, and it could be omitted.

Let $SG = (S, T)$ be the *state space graph* associated to the DP formulation of TSP. A state $s \in S$ is represented by the couple (Y, i) and can be interpreted as the family of all feasible paths starting from vertex 0, visiting every vertex in Y and ending in vertex i. The cost of a feasible arc in the state space graph $((Y, i), (Y \cup \{i\}, j))$ is equal to c_{ij}. Knowing the following boundary conditions:

$$\begin{aligned} f_1(\{i\}, i) &= c_{0i} \qquad \forall i \in D_0 \\ f_1(\{i\}, i) &= \infty \qquad \forall i \in V' \setminus D_0 \end{aligned}$$

The following recursive relation can be stated:

$$f_k(Y, i) = min_{j \in (Y \setminus \{i\}) \cap E_i} \{f_{k-1}(Y \setminus \{i\}, j) + c_{ji}\} \qquad \forall Y \subseteq V', |Y| \geq 2, \ \forall i \in Y$$

The value of optimal solution is given by

$$z = min_{i \in E_0} \{f_{n-1}(V', i) + c_{i0}\}$$

Given this formulation, it is possible to calculate the optimal solution of the problem by recursively computing $f_k(Y, i)$ for every Y and i. In fact, the value of $f_k(Y, i)$ where $k = 1$ is known for every i; knowing $f_1(Y, i)$, it is possible to calculate every $f_2(Y, i)$, and so on until $f_{n-1}(V', i)$. The number of states (vertices of the state space graph) is of course exponential in the number of nodes of the TSP. Therefore the time and space complexity is exponential in the number of nodes of the TSP.

We recall that given a problem P with an associated *state space graph* $SG = (S, T)$ a state space relaxation aims at defining a new state space graph $RSG = (RS, RT)$ such that a function $w(s)$ exists mapping every state $s \in S$ into a state $w(s) \in RS$, and such that for every arc $(s_i, s_j) \in T$, $(w(s_i), w(s_j))$ is a feasible arc in RT. Then, if p is the shortest path from s_0 to s_i, and rp is the shortest path from $w(s_0)$ to $w(s_i)$, the cost of rp is less or equal to the cost of p. The relaxation represented by $RSG = (RS, RT)$ is of course useful if the number of vertices of the new graph is polynomial instead of exponential.

In the TSP example, let consider $w(s) = w((Y, i)) = (|Y|, i)$; a state rs for the relaxed state space is therefore described by the couple (k, i). To each state $s = (Y, i) \in S$ is associated a state $rs = (|Y|, i)$. The cost of an arc in RSG $((k, i), (k+1, j))$ is equal to c_{ij}.

Let define $g(k,i)$ as the cost of the shortest path from vertex 0 to vertex i passing by k vertices in V. Boundary conditions are:

$$\begin{aligned} g(1,i) &= c_{0i} && \forall i \in D_0 \\ g(1,i) &= \infty && \forall i \in V' \setminus D_0 \end{aligned}$$

The recursive relation is:

$$g(k,i) = min_{j \in E_i}\{g(k-1,j) + c_{ji}\} \qquad \forall i \in V'$$

The value of optimal solution is:

$$z' = min_{i \in E_1}\{g(n-1,i) + c_{i1}\}$$

It is easy to prove that $z' \leq z$, i.e., the new formulation gives a lower bound for the original problem. Moreover, it is easy to prove that the space complexity of the new formulation is $O(n^2)$, while its time complexity is $O(n^3)$.

State Space Relaxation using Constraint Programming

The State Space Relaxation model described for the TSP can be easily implemented in Constraint Programming. Let first perform the usual transformation from a TSP into an hamiltonian path, duplicating node $n-1$ and considering it both end and start node of the path (named $n-1$ and n). Nodes $0,\ldots,n-2$ are internal nodes, node $n-1$ is the end node, while node n is the start node.

The TSP model is described by $(n+3)$ domain variables per node, the domain variable $prev_i$ represents the node directly preceding i in the path; the domain variable pos_i represents the position of node i in the path (i.e. pos_i is equal to the number of nodes preceding i); a domain variable $cumul_i$ represents the cumulative cost in the path from the start node n to node i; and n domain variables $cumul2_{pi}$ represent the cost of the path from the start node n to i if node i is in position p (i.e. p nodes exist in the path from n to i). Note that variables $cumul2_{pi}$ is a sort of conditional variable; it is, in fact, possible that node i is not in position p; in this case the value of $cumul2_{pi}$ is meaningless.

Let M be any upper bound of the TSP, the state space relaxation described can be modeled as follows:

$$\begin{aligned} & \forall i \in \{0,\ldots,n\} && prev_i \in [0..n] \\ & \forall i \in \{0,\ldots,n\} && cumul_i \in [0..M] \\ & \forall i \in \{0,\ldots,n\} && pos_i \in [0..n] \\ & \forall i,p \in \{0,\ldots,n\} && cumul2_{ip} \in [0..2\,M] \\ (3.20)\quad & \forall i \in \{0,\ldots,n\} && cumul_i = cumul2[i, pos_i] \\ (3.21)\quad & \forall i,p \in \{0,\ldots,n-1\} && cumul2_{ip} = cumul2[prev_i, p-1] + c[prev_i, i] \\ (3.22)\quad & \forall i,p \in \{0,\ldots,n\} && pos_i \neq p \Leftrightarrow cumul2_{ip} > M \\ (3.23)\quad & && pos_n = 0 \\ (3.24)\quad & && cumul_{n0} = 0 \end{aligned}$$

The conditional cumulative costs $cumul2_{p\,i}$ are linked to the cumulative cost $cumul_i$ by means of the element constraint $cumul_i = cumul2_{p\,i}[pos_i]$. The objective function is given by the cumulative cost variable of the end node $n-1$, $z = cumul_{n-1}$.

The gradient function could simply be calculated as:

$$grad(i,j) = \begin{cases} \infty & \text{if } i \notin prev_j \\ \min_{p=0,\dots,n-1}\{\texttt{inf}(cumul2_{ip}) + c_{ij} - \texttt{inf}(cumul_{j\,p+1})\} & \text{otherwise} \end{cases}$$

Alternatively, a weaker gradient function can be defined as:

$$grad(i,j) = \begin{cases} \infty & \text{if } i \notin prev_j \\ \texttt{inf}(cumul_i) + c_{ij} - \texttt{inf}(cumul_j) & \text{otherwise} \end{cases}$$

The advantage of this second formulation is that it can be obtained in constant time, and does not require to be stored in a specific data structure. Note that a gradient function can also be defined with respect to the position variable pos. The function $gradPos(i,p)$ evaluates the additional cost to be paid if node i is in position p:

$$gradPos(i,p) = \begin{cases} \infty & \text{if } p \notin pos_i \\ \texttt{inf}(cumul2_{ip}) - \texttt{inf}(cumul_i) & \text{otherwise} \end{cases}$$

The CP model of the state space relaxation, has the advantage that it makes use of *element* constraints, leading to an incremental version of the dynamic programming recursion. The drawback is that it is not possible to control the order in which the element constraints propagate. Therefore the time complexity of the (CP based) dynamic programming recursion depends on the actual implementation of the constraint queue of the chosen CP system. On the other hand, this drawback can easily be avoided by encapsulating the model in a global constraint responsible for triggering the element constraints in the "correct" order. This solution leads to an incremental algorithm with worst case time complexity of $O(n^3)$.

Several improvement on this model can be studied; clearly the optimal path of the relaxed formulation contains subcycles and it may also go through the same node more than once. A different formulation, could, for example, avoid all subcycles of cardinality two, and could consider both the backward cumulative variable (cost of a path from the start node n to node i) and a forward cumulative variable (cost of a path from node i to the end node $n-1$).

Dynamic Programming and Conditional Variables

The CP based State Space Relaxation proposed has some serious drawback that lead to lack of propagation. A variable $cumul2_{ip}$, initialized with a domain $[0..2M]$, is "pushed" to values greater than M when the node i is not in position p. This trick tries to mimic the idea of having a *conditional variable* whose values are meaningful if and only if a certain condition holds. Consider constraint (3.20); the (range) propagation of this element constraint is usually enforced by the following rules:

$$\begin{aligned} \texttt{inf}(cumul_i) &\geq \min_{p \in pos_i}\{\texttt{inf}(cumul2_{ip})\} \\ \texttt{sup}(cumul_i) &\leq \max_{p \in pos_i}\{\texttt{sup}(cumul2_{ip})\} \\ \texttt{sup}(cumul2_{ip}) < \texttt{inf}(cumul_i) &\Rightarrow pos_i \neq p \\ \texttt{inf}(cumul2_{ip}) > \texttt{sup}(cumul_i) &\Rightarrow pos_i \neq p \\ pos_i = p &\Rightarrow cumul_i = cumul2_{ip} \end{aligned}$$

the bounds of $cumul_i$ are propagated to $cumul_{ip}$ *only* when $pos_i = p$. Therefore the information of these bounds may not be propagated by the constraint (3.21) until very late in the search space.

This drawback can be avoided by introducing the concept of *Conditional Variable*. A *Conditional Variable* $cVar$ is a variable depending on a constraint; it is defined by a the pair $cVar = (Dom, cst)$ where Dom is a domain of possible values, and cst is a constraint, and it has the following semantic:

$$cst \Leftrightarrow Dom \neq \emptyset$$

A conditional variable $cVar = (Dom, cst)$ is defined *true* if its definition constraint cst is true. It is defined *false* if its definition constraint cst is false. The constraint cst of $cVar$ is identified by `cst(`$cVar$`)`.

The concept of conditional variables has been used in constraint programming for instance to implement constructive disjunctions ([89]). In the following we will present some examples of constraints that could be defined using conditional variables.

The equality constraint can be defined between one conditional variable and a variable, and between two conditional variables: let $cVar = (Dom, cst)$ be a conditional variable, and var be a variable. The constraint $cVar = var$ holds if cst is false or if cst is true and the two variable assume the same values.

The equality constraint could propagate as follows:

$$\begin{array}{r}
i \notin \texttt{domain}(var) \Rightarrow cVar \neq i,\ \forall i \in cVar \\
cst \Rightarrow (i \notin \texttt{domain}(cVar) \Rightarrow var \neq i,\ \forall i \in var)
\end{array}$$

Similarly, let $cVar_1 = (Dom_1, cst_1)$, and $cVar_2 = (Dom_2, cst_2)$ be two conditional variables. The constraint $cVar_1 = cVar_2$ holds if $cst_1 \wedge cst_2$ is false or if $cst_1 \wedge cst_2$ is true and the two variable assume the same values.

Conditional variables can be combined via arithmetical operators: for instance, two conditional variables $cVar_1 = (Dom_1, cst_1)$, and $cVar_2 = (Dom_2, cst_2)$ can be combined as $cVar_3 = cVar_1 + cVar_2$, where $cVar_3$ is defined as $cVar_3 = (Dom_3, cst_1 \wedge cst_2)$, where Dom_3 is defined by the usual propagation rule of the sum operator.

We mentioned that conditional variables have been used to implement constructive disjuction; for example, consider an activity act requiring one out of k resources. ILOG Scheduler implements this disjunction as if k "conditional activities" require each possible resource (see Section 4.3.4, or [95]).

A global constraint used to implement constructive disjunction (in this setting we are only interested in exclusive disjunction) can be defined as follows: $xunion(cVars, y, x)$ where x, y are regular variables, and $cVars$ is an array of k conditional variables. Exactly one out of the k conditional variables $cVars[i]$ is a true variable (say $cVars[i^*]$), and the index i^* of the only true variable $cVars[i^*]$ is equal to x ($x = i^* \mid$ `cst(`$cVars[i^*]$`)`). Formally, the constraint $xunion(cVars, y, x)$ holds iff:

$$\begin{array}{r}
XOR_{i=1,\dots,k-1}\texttt{cst}(cVars[i]) \\
cVars[x] = y
\end{array}$$

From the definition, it is easy to show that $\texttt{cst}(cVars[i])$ is equivalent to $x = i$, for all $i = 0, \ldots, k-1$. The following propagation rules can be enforced on the bounds of the variables[2]:

$$\begin{array}{lll}
(3.25) & & \texttt{inf}(y) \geq \min\limits_{i \in \texttt{domain}(x)} \{\texttt{inf}(cVars[i])\} \\
(3.26) & & \texttt{sup}(y) \leq \max\limits_{i \in \texttt{domain}(x)} \{\texttt{sup}(cVars[i])\} \\
(3.27) & \forall i \in \texttt{domain}(x) & \texttt{inf}(cVars[i]) \geq \texttt{inf}(y) \\
(3.28) & \forall i \in \texttt{domain}(x) & \texttt{sup}(cVars[i]) \leq \texttt{sup}(y) \\
(3.29) & \forall i \in \texttt{domain}(x) & \texttt{inf}(y) > \texttt{sup}(cVars[i]) \Rightarrow x \neq i \\
(3.30) & \forall i \in \texttt{domain}(x) & \texttt{sup}(y) < \texttt{inf}(cVars[i]) \Rightarrow x \neq i \\
(3.31) & \forall i = 0, \ldots, k-1 & i \notin \texttt{domain}(x) \Leftrightarrow \neg\texttt{cst}(cVars[i]) \\
(3.32) & \forall i = 0, \ldots, k-1 & i = x \Leftrightarrow \texttt{cst}(cVars[i])
\end{array}$$

Using conditional variables, the constraint (3.20) can be replaced by $xunion((cumul2_{ip})_p, cumul_i, pos_i)$ (where $(cumul2_{ip})_p$ is clearly the array of conditional variables $cumul2_{ip}, p = 0, \ldots, n-1$); the advantages of this formulation are mainly two: first, the *xunion* constraint propagates more than the element constraint; second, other constraints can now *safely* modify both bounds of the conditional variables $cumul2_{ip}$, while in the previous model the upper bound had to remain *open* to enable the variable to be pushed over value M to mimic the condition "node i not in position p".

We now propose a generalization of the constructive disjunction that enables to easily write dynamic programming recursion in CP. The generalization proposed provides a constructive (exclusive) disjuction between a conditional variable and an array of conditional variables and can be defined as follows: $xunion(cVars, y, x)$ where x, y are conditional variables, and $cVars$ is an array of k conditional variables. The new constraint has the same semantic as the previous one if $\texttt{cst}(x) \wedge \texttt{cst}(y)$ holds. Formally, the constraint $xunion(cVars, y, x)$ holds iff:

$$(\texttt{cst}(x) \wedge \texttt{cst}(y)) \Rightarrow (XOR_{i=1,\ldots,k-1}\texttt{cst}(cVars[i]) \wedge cVars[x] = y)$$

The following propagation rules can be enforced on the bounds of the variables:

$$\begin{array}{lll}
(3.33) & & \texttt{cst}(y) \Rightarrow (0 \leq x \leq k-1) \\
(3.34) & & \texttt{cst}(x) \Rightarrow (\texttt{inf}(y) \geq \min\limits_{i \in \texttt{domain}(x)} \{\texttt{inf}(cVars[i])\}) \\
(3.35) & & \texttt{cst}(x) \Rightarrow (\texttt{sup}(y) \leq \max\limits_{i \in \texttt{domain}(x)} \{\texttt{sup}(cVars[i])\}) \\
(3.36) & \forall i \in \texttt{domain}(x) & (\texttt{cst}(x) \wedge \texttt{cst}(y)) \Rightarrow \texttt{inf}(cVars[i]) \geq \texttt{inf}(y) \\
(3.37) & \forall i \in \texttt{domain}(x) & (\texttt{cst}(x) \wedge \texttt{cst}(y)) \Rightarrow \texttt{sup}(cVars[i]) \leq \texttt{sup}(y) \\
(3.38) & \forall i \in \texttt{domain}(x) & \texttt{cst}(y) \Rightarrow (\texttt{inf}(y) > \texttt{sup}(cVars[i]) \Rightarrow x \neq i) \\
(3.39) & \forall i \in \texttt{domain}(x) & \texttt{cst}(y) \Rightarrow (\texttt{sup}(y) < \texttt{inf}(cVars[i]) \Rightarrow x \neq i)
\end{array}$$

[2]Similar rules can be enforced considering the entire domain of the variables, instead of the bounds.

(3.40) $\forall i = 0, \ldots, k-1 \quad (\texttt{cst}(x) \wedge \texttt{cst}(y)) \Rightarrow (i \notin \texttt{domain}(x) \Rightarrow \neg\texttt{cst}(cVars[i]))$

(3.41) $\forall i = 0, \ldots, k-1 \quad \texttt{cst}(y) \Rightarrow (\neg\texttt{cst}(cVars[i]) \Rightarrow x \neq i)$

(3.42) $\forall i = 0, \ldots, k-1 \quad (\texttt{cst}(x) \wedge \texttt{cst}(y)) \Rightarrow (i = x \Rightarrow \texttt{cst}(cVars[i]))$

(3.43) $\forall i = 0, \ldots, k-1 \quad \texttt{cst}(y) \Rightarrow (\texttt{cst}(cVars[i]) \Rightarrow x = i)$

In most cases, the $xunion(cVars, y, x)$ constraint on conditional variables x, y is only interesting if both x and y are conditioned by the same constraint, i.e., $\texttt{cst}(x) = \texttt{cst}(y)$. This is the case of dynamic programming; in fact, in DP the variable x represents the transition from one conditional state to another, this transition is itself a conditional variable. For example, the cumulative cost to reach node i on position p is equal to the cumulative cost to reach node j on position $p-1$ plus c_{ji} where j is the node directly preceding i "if i is in position p". In the DP model for TSP, we introduce the conditional prev variables $prev_{ip}$ representing the node directly preceding i, if i is in position p. Using the newly introduced conditional variable, we could rewrite constraint 3.21 as

$$xunion((cumul2_{j,p-1} + c_{j,i})_j, cumul2_{ip}, prev_{ip}), \forall i, p = 0, \ldots, n$$

where $(cumul2_{j,p-1} + c_{j,i})_j$ is the array of conditional variables $cumul2_{j,p-1} + c_{j,i}$ for $j = 0, \ldots, n$. Clearly, the conditional variables $cumul2_{ip}$ and $prev_{ip}$ share the same conditioning constraint $pos_i = p$.

In general, Dynamic Programming formulations link a conditional variable y to arrays of conditional variables $vars$ through a transition function represented by a conditional variable x; the transition function conditional variable x, and the conditional variable y share the same constraint.

When the $xunion(cVars, y, x)$ constraint is stated on conditional variables x, y, and x and y are conditioned by the same constraint, the propagation rules can be enhanced as follows:

(3.44) $0 \leq x \leq k-1$

(3.45) $\texttt{inf}(y) \geq \min_{i \in \texttt{domain}(x)} \{\texttt{inf}(cVars[i])\}$

(3.46) $\texttt{sup}(y) \leq \max_{i \in \texttt{domain}(x)} \{\texttt{sup}(cVars[i])\}$

(3.47) $\forall i \in \texttt{domain}(x) \quad \texttt{cst}(y) \Rightarrow \texttt{inf}(cVars[i]) \geq \texttt{inf}(y)$

(3.48) $\forall i \in \texttt{domain}(x) \quad \texttt{cst}(y) \Rightarrow \texttt{sup}(cVars[i]) \leq \texttt{sup}(y)$

(3.49) $\forall i \in \texttt{domain}(x) \quad \texttt{inf}(y) > \texttt{sup}(cVars[i]) \Rightarrow x \neq i$

(3.50) $\forall i \in \texttt{domain}(x) \quad \texttt{sup}(y) < \texttt{inf}(cVars[i]) \Rightarrow x \neq i$

(3.51) $\forall i = 0, \ldots, k-1 \quad \texttt{cst}(y) \Rightarrow (i \notin \texttt{domain}(x) \Rightarrow \neg\texttt{cst}(cVars[i]))$

(3.52) $\forall i = 0, \ldots, k-1 \quad \neg\texttt{cst}(cVars[i]) \Rightarrow x \neq i$

(3.53) $\forall i = 0, \ldots, k-1 \quad \texttt{cst}(y) \Rightarrow (i = x \Rightarrow \texttt{cst}(cVars[i]))$

(3.54) $\forall i = 0, \ldots, k-1 \quad \texttt{cst}(cVars[i]) \Rightarrow x = i$

Beside the increase in the propagation, the DP formalization that uses conditional variables enable a full exploitation of the information calculated in the DP recursion. In fact, the bound of the conditional variables could be updated by other constraints, and this modification would be transferred to the DP recursion through shared variables, as in the style of Constraint

Programming. For example, in TSPTW, typically two different types of *cumul* variables are defined: the first type accumulates the cost from the start node to node i, the second one accumulates the time from the start node to node i. Sharing the same conditional transition variables, the two sets of *xunion* constraints transparently communicate respective deductions.

TSP State Space Relaxation using Conditional Variables

The State Space Relaxation model for the TSP can be implemented in Constraint Programming using the Conditional Variables.

The TSP model is described by $(2n+3)$ domain variables per node, the domain variable $prev_i$ represents the node directly preceding i in the path; the domain variable pos_i represents the position of node i in the path (i.e. pos_i is equal to the number of nodes preceding i); a domain variable $cumul_i$ represents the cumulative cost in the path from the start node n to node i; and n conditional variables $cumul2_{pi}$ (conditioned by $pos_i = p$) represents the cost of the path from the start node n to i if node i is in position p; and n conditional variables $prev2_{pi}$ (conditioned by $pos_i = p$) represents the node directly preceding i if i is in position p. The state space relaxation described can be modeled as follows:

$$\begin{array}{lrl}
 & \forall i \in \{0,\dots,n\} & prev_i \in [0..n] \\
 & \forall i \in \{0,\dots,n\} & cumul_i \in [0..M] \\
 & \forall i \in \{0,\dots,n\} & pos_i \in [0..n] \\
 & \forall i,p \in \{0,\dots,n\} & cumul2_{ip}(pos_i = p) \in [0..M] \\
 & \forall i,p \in \{0,\dots,n\} & prev2_{ip}(pos_i = p) \in [0..n] \\
(3.55) & \forall i \in \{0,\dots,n\} & xunion((cumul2_{ip})_p, cumul_i, pos_i) \\
(3.56) & \forall i \in \{0,\dots,n\} & xunion((prev2_{ip})_p, prev_i, pos_i) \\
(3.57) & \forall i,p \in \{0,\dots,n-1\} & xunion((cumul2_{j,p-1} + c_{ji})_j, cumul2_{ip}, prev2_{ip}) \\
(3.58) & & pos_n = 0 \\
(3.59) & & cumul_{n0} = 0
\end{array}$$

The gradient functions can be calculated as before:

$$grad(i,j) = \begin{cases} \infty & \text{if } i \notin prev_j \\ \min_{p=0,\dots,n-1 \mid i \in prev2_{j,p+1}} \{\texttt{inf}(cumul2_{ip}) + c_{ij} - \texttt{inf}(cumul_{j\,p+1})\} & \text{otherwise} \end{cases}$$

$$grad(i,j) = \begin{cases} \infty & \text{if } i \notin prev_j \\ \texttt{inf}(cumul_i) + c_{ij} - \texttt{inf}(cumul_j) & \text{otherwise} \end{cases}$$

$$gradPos(i,p) = \begin{cases} \infty & \text{if } p \notin pos_i \\ \texttt{inf}(cumul2_{ip}) - \texttt{inf}(cumul_i) & \text{otherwise} \end{cases}$$

By encapsulating the model in a global constraint responsible for triggering the *xunion* constraints in the “correct” order, an incremental algorithm can be designed having worst case time complexity of $O(n^3)$. Moreover all the conditional variables are available to be used by other constraints, or for defining search heuristics.

Chapter 4

Applications

This chapter gives a few examples of applications where optimization constraints improve the performances of a standard Constraint Programming approach. The following four different optimization contraints will be used:

- $pathCost(next, z, cost)$
- $pathCostPC(next, succ, z, cost)$
- $multiPathCost(next, path, z, m, cost)$
- $allDiffCost(var, z, cost)$

Given a set of nodes $V = \{0, \ldots, n+1\}$, the constraint $pathCost(next, z, cost)$ enforces that one path exists starting from node $n+1$ ending at node n, and covering all nodes $0, \ldots, n-1$ exactly once. Nodes n and $n+1$ are called end and start node respectively, while nodes $0, \ldots, n-1$ are called internal nodes. Variable $next_i$, associated to node i, identifies the node next of i. Conventionally, the next of the end node is the start node, i.e., $next_n = n+1$. $cost$ is a transition cost matrix, i.e., $cost_{ij}$ defines the cost from node i to node j, $\forall i, j \in V$. The variable z represents the sum of transition costs along the path, i.e., $\sum_{i=0}^{n+1} cost_{i\, next_i}$. The constraint $pathCost$ embeds a solver calculating optimal solution of a relaxation for the constraint, and a gradient function. The optimal solution of the relaxation updates the lower bound of z; the gradient function reduces the domain of the variable $next_i$.

The constraint $pathCostPC(next, succ, z, cost)$ is a generalization of $pathCost$ modeling precedence constraint among internal nodes. A set variable $succ_i$ ($\forall i = 0, \ldots, n-1$) is defined for every internal node, and identifies the set of nodes that are successors of node i in the path from the start node $n+1$ to the end node n [1].

A different generalization of $pathCost$ can be obtained by allowing multiple paths, and a path dependent cost matrix. Consider a set of nodes $V = \{0, \ldots, n+2m-1\}$ partitioned in three sets S, I, E. $I = \{0, \ldots, n-1\}$ represents the set of internal nodes, $S = \{n+m, \ldots, n+$

[1] An integer set variable is a variable whose domain contains sets of integer values. In analogy with the definition of lower and upper bounds on integer variables, lower and upper bounds can also be defined for an integer set variable: the lower bound (or required set) is the intersection of all sets belonging to the domain of the variable; the upper bound (or possible set) is the union of all sets belonging to the domain of the variable. The variable is considered instantiated when the domain contains a single set of integer values thus the required set is equal to the possible set.

$2m-1\}$ represents the set of start nodes, and $E = \{n, \ldots, n+m-1\}$ represents the set of end nodes. The constraint $multiPathCost(next, path, z, m, cost)$ enforces that exactly m disjoint paths exist, each of them starting from a node in S, ending in a node in E, and covering some nodes in I. Moreover each internal nodes $i \in I$ is covered by exactly one path.

Variable $next_i$, associated to node i, identifies the node next of i. Conventionally, the next of each end node is its corresponding start node, i.e., $next_{n+k} = n+m+k$, $\forall k = 0, \ldots, m-1$. Variable $path_i$, associated to node i, identifies the path to which node i belongs to. $cost$ is a transition cost matrix, i.e., $cost^k_{ij}$ defines the cost from node i to node j on path k, $\forall i, j \in V, \forall k = 0, \ldots, m-1$. The variable z represents the sum of transition costs along the paths, i.e., $\sum_{i=0}^{n+2m-1} cost^{path_i}_{i\, next_i}$. The constraint $multiPathCost$ embeds a solver calculating the optimal solution of a relaxation for the constraint, and a gradient function.

Given the array of variables var, the constraint $allDiffCost(var, z, cost)$ enforces each variable var_i to be assigned to a different value. Moreover a cost $cost_{ij}$ is considered if var_i is assigned to the value j. The variable z represents the sum of the assignment costs $\sum_{i=0}^{n-1} cost_{i\, var_i}$.

4.1 Traveling salesman problem

TSP concerns the task of finding a tour covering a set of nodes exactly once with a minimum cost. The problem is strongly NP-hard, and has been deeply investigated in the literature (see [99] for a survey). Although CP is far from obtaining better results than the ones obtained with state of the art OR methods, it is nevertheless interesting to build an effective CP based solver for TSP. Indeed, many problems contain subproblems that can be described as TSPs; in these cases the flexibility and the domain reduction mechanism of Constraint Programming languages can play an important role, and hybrid CP-OR systems could outperform pure OR approaches (as shown in [133, 68]).

Within the Constraint Programming community, several global constraints have been developed to cope with this problem since many real-life problems contain subproblems that can be described as TSPs, whereas very little work has been devoted to solving pure TSPs. Caseau and Laburthe [38] have solved small TSP instances with constraints by using a set of simple, but effective, propagation algorithms, and cost-based reasoning deriving from the use of the Lagrangian relaxation of the problem. Methods based on optimization constraints are proposed in [71, 67, 69].

4.1.1 CP Model

A TSP can be transformed into a minimum cost Hamiltonian Path Problem (HPP) by duplicating a node. The generated HPP calls for the minimum cost hamiltonian path starting from the first copy of the duplicated node, and ending at the second copy.

A CP model of the HPP considers a set of nodes $V = 0, 1, 2, \ldots, n$, where nodes $0, \ldots, n-2$ represent internal nodes, and nodes $n-1$ and n represent the end and start nodes respectively. For each pair of nodes i and j, $c_{i\,j}$ is the travel cost from i to j. The CP model for a HPP is the following:

$$\begin{array}{rl}
\min & z \\
\text{on} & \\
 & z \geq 0 \\
\forall i \in \{0, \ldots, n\} & next_i \ :: \ [0..n] \\
\text{subject to} & \\
 & pathCost(next, z, c)
\end{array} \tag{4.1}$$

Domain variables $next_i$ identify the node visited after node i. A feasible solution of the TSP is an assignment of a different value to each variable (a successor in the path) by avoiding subtours. An optimal solution of the problem is the one minimizing the sum of the travel costs $\sum_{i=0}^{n-1} c_{i\;next_i}$.

4.1.2 Relaxations

Five relaxations have been compared:

- the Assignment Problem referred to as **AP** (see Section 3.5.3);

- the Minimum Spanning Arborescence referred to as **MSA** (see Section 3.5.4);
- the Assignment Problem generated by calculating the subtour elimination cuts (SEC) relaxed in a Lagrangian way at the root node referred to as **LAGR** (see Section 3.5.5);
- the Linear Program generated by the Assignment Problem with the addition of the subtour elimination cuts (SEC) only at the root node, referred to as **LP** (see Section 3.5.5);
- the Linear Program generated by the Assignment Problem with the addition of the subtour elimination cuts (SEC) at every node, referred to as **LP-CN** (see Section 3.5.5).

The different techniques mentioned are sorted by increasing complexity. The **MSA** has a complexity of $O(n^2)$; the **AP** has an initial complexity of $O(n^3)$ whereas each following re-computation is performed in $O(n^2)$. The method **LAGR** has the same complexity as **AP**, but the initial complexity is increased by the fact that a Linear Problem is solved through the simplex algorithm providing the optimal Lagrangian multipliers. Moreover, whenever the Lagrangian method is used, two optimization constraints run, one using the Lagrangian AP, one using the AP with the original cost matrix. Finally, the last two approaches are solved by using the LP solver at each node which has an exponential worst case complexity (using the simplex algorithm).

4.1.3 Branching strategies and heuristics

Beside the increased constraint propagation, an additional advantage in using lower bounding procedures within Constraint Programming is the possibility of exploiting the information given by the relaxed problem to build smarter heuristics.

A very simple heuristic is based on the *first-fail* principle to select the branching variable, and on the information of the optimization constraint to select the tentative value. The variable $next_i$ with the smallest domain is selected as branching variable; the optimal value of the relaxed problem is selected as tentative value.

Caseau and Laburthe [38] propose a heuristic combining *first-fail* and *max regret* for the variable selection obtaining good results. The branching variable is selected as the one minimizing a weighted sum of regret and domain size.

When the optimization component is an AP solver, a subtour elimination branching rule can be defined (see, e.g., [11]). The subtour elimination branching rule is often used in OR-based Branch and Bound algorithm for the TSP. At any node of the search tree, the solution of the AP is considered, and an infeasible tour identified by arcs $x_{i_1 i_2}, x_{i_2 i_3}, \ldots, x_{i_s i_1}$ is selected. Branching is obtained by imposing the following disjunction:

$$(x_{i_1 i_2} = 0) | (x_{i_1 i_2} = 1 \wedge x_{i_2 i_3} = 0) | \ldots | (x_{i_1 i_2} = x_{i_2 i_3} = \ldots = x_{i_{s-1} i_s} = 1 \wedge x_{i_s i_1} = 0)$$

This branching strategy can be combined with a *first-fail, max regret* for the selection of the tour. *First-fail* ideas can be applied by selecting the "minimum size tour" where the size of the tour is defined as the sum of the domain's size of all the variable in the tour. *Max regret* ideas can be applied to define the branches on the selected tour: the variables within the tour are sorted by decreasing values of their *regret* (i.e., the evaluation of the AP solution obtained by eliminating from the domain of each variable the value corresponding to the AP optimal solution). Therefore, the branching strategy will break last those arcs that will cause the highest increase of the lower bound.

4.1.4 Computational Results

In this section several optimization constraints are tested; the results show that the constraint solver using the optimization constraint significantly outperforms any previously proposed CP approach. Moreover, the *optimization oriented* constraints are able to solve problems whose dimensions are one order of magnitude larger than those solved by the predefined `IlcPath` constraint of ILOG Solver.

This computational section tries to achieve three goals: first, we prove the effectiveness of optimization constraints within a CP framework; second, we try to understand how different combinations of constraints perform together; third, sophisticated cutting plane techniques are evaluated within the CP framework proposed.

Performance of the method on TSPs

The results have been obtained using the `IlcAllDiff` constraint of ILOG Solver44 together with the optimization constraint `PathCost` based on the AP relaxation, and using the primal-dual Hungarian Algorithm, and, in particular, a C++ adaptation of the APC code [33].

The tests run on a Pentium II, 300 MHz, while the computing times reported in Table 4.2 and referring to [38] (CL97) are on a Sun Sparc 10. The run time values are expressed in seconds; letters k and M, following a run time or a number of fails indicate thousands and millions (of seconds or fails), respectively.

We consider two sets of instances: a set of asymmetric randomly generated TSP instances with up to 250 nodes, and a set of six small symmetric instances from TSP-LIB.

In the first group of results, we run the algorithm on randomly generated problems whose size range from 20 and 250 (columns labelled with n-ATSPs where n is the number of nodes). These problems have been generated according to the class A described in [62]. The cost c_{ij} are uniformly random in the range [1, 1000]. The results are reported in Table 4.1 in which each entry represents an average value over five instances, and concern the solution of the problems by using the subtour elimination strategy.

Problem	20-ATSPs	40-ATSPs	80-ATSPs	250-ATSPs
Time	0.01	0.23	0.39	2.03
Fails	1.0	8.3	8.5	3.9

Table 4.1: Results on randomly generated ATSP with the subtour elimination strategy

On this class of problems the AP is known to be very effective, thus our method is very fast, but note that with a pure CP code exploiting the Branch and Bound implemented in most CP systems we cannot solve the 20-ATSPs in less than 8 hours.

The second group of results, reported in Table 4.2, concerns symmetric TSP from TSPLIB. The size of the problems is self evident from the name of the instance; both the run time and the number of backtracks are reported. The first row (CL97) reports the results published in [38]; the second row (AP_CST1) reports the results obtained by using `PathCost` and a branching strategy based on combination of *first-fail*, and *max-regret* similar to the one described in [38]; finally, the third row (AP_CST2) reports the results obtained by using `PathCost` and the subtour elimination branching strategy described in Section 4.1.3.

Problem	gr17		gr21		gr24		fri26		bayg29		bays29	
	Time	Fails	Time	Fails	Time	Fails	Time	Fails	Time	Fails	Time	Fails
CL97	3.10	5.8k	7.00	12.5k	6.90	6.6k	930.00	934k	4.4k	4.56M	1.2k	1.1M
AP_CST1	0.64	1.3k	0.02	9	4.45	7.5k	1.80	1.8k	171.00	213k	58.90	74k
AP_CST2	0.53	616	0.06	12	0.18	105	2.30	1.6 k	10.30	8k	14.70	19k

Table 4.2: Results on small symmetric TSP instances

Although symmetric TSP instances are known to be difficult for exact methods using AP as a bound, the results in Table 4.2, show the effectiveness of the optimization constraint with respect to other CP approaches.

By comparing rows AP_CST1 and AP_CST2 in Table 4.2, we can distinguish the contribution of the lower bound and the gradient function propagation from the contribution of the tour finding part. In fact, in AP_CST2 the branching strategy used is more TSP-oriented since it is devoted to remove subtours. In general, it outperforms both in the number of fails and in the computing time the AP_CST1 which uses a more general-purpose heuristic.

Combination of constraints

In Constraint Programming, different techniques can be applied to a problem; these techniques smoothly interact through the domain reduction mechanism. In the TSP example, it can be observed that the global constraint `IlcPath` reduces domain of variables that would not be reduced by the `PathCost` constraint. `IlcPath` may remove a value that was part of the optimal solution of the relaxed problem, thus causing an immediate recalculation of the lower bound. Vice versa, the optimization constraint may remove a value feasible w.r.t. the `IlcPath` constraint.

In order to show the behaviour differences depending on the used constraints, we consider the problem Bays29. All the results in Table 4.3 were generated using `PathCost` based on AP. Each row corresponds to the results obtained by adding the corresponding constraints written in the table.

The addition of a `PathCost` constraint based on the MSA relaxation is also considered (column MSA_CST). The first column does not use MSA_CST, the second one uses MSA_CST with a gradient function defined by the reduced costs.

Comparison among different combinations of constraints is very interesting because in all the cases reported the heuristic and the branching scheme are the same. In general, the use of the MSA-based constraint (MSA_CST) together with the AP-based constraint reduces the number of backtracks, but increases the run time, even when, in the best case, such a reduction saves up to the 60% of backtracks. More or less the same can be said for the `IlcPath` constraint; while the combination of `IlcPath` and MSA_CST does not generate a further significant reduction of backtracks.

Quite surprisingly, the combination of AP_CST with `IlcAllDiff` is often the most effective technique. This may, at first glance, seems strange since the AP computation never returns a solution that violates an `IlcAllDiff` constraint; thus, it could be argued that

`IlcAllDiff` never reduces the search. This is of course not true. Suppose, in fact, that you have the following configuration of array *next*: $next = [[2,3],[1,3],[1,2,4],[5,6],[4,6],[4,5]]$. The assignment $next_3 = 4$ never appears in any AP solution, but since the AP performs pruning only based on value of the optimal solution and gradient function, it may not detect that $next_3 = 4$ is infeasible until such assignment is performed. In this case a failure and backtrack is generated. `IlcAllDiff`, on the other hand, immediately eliminates the value 4 from domain of $next_3$, therefore saving an AP calculation and a backtrack.

	no MSA_CST		MSA_CST	
	Time	Fails	Time	Fails
-	32.36	40k	45.93	17.9k
IlcAllDiff	14.74	19k	27.74	6.9k
IlcPath	21.47	8.6k	34.74	6.3k
IlcAllDiff IlcPath	21.20	8.0k	33.60	5.9k

Table 4.3: Comparison of different constraint combinations with the subtour elimination strategy on Bays29

Note that the first row represents a pure OR, AP-based, depth first approach, and that such combination results the worst performing one. The insertion of any CP constraint improves the performance of the solver engine.

Applying Cutting Planes techniques

The overall approach has been implemented using the ILOG Optimization Suite [139], and runs on a Pentium II 200 MHz with 64MB of RAM. A time limit of 30 minutes has been imposed.

The results reported in Table 4.3 concern symmetric TSPs from TSPLIB. The lower bound at root node, the run time (in seconds) and the number of backtracks are reported. The branching strategy is very simple: the variable $next_i$ with the smallest domain is selected, and the optimal solution of the relaxed problem is chosen as tentative value.

From the results we can see that for small problems, it is not worth adding cutting planes either in the Lagrangian relaxation or in the LP. Although the AP is not a good lower bound for symmetric TSPs, it is nevertheless good enough to quickly solve small problems. On the other hand, it is known that pure OR Branch and Cut is the best exact method to solve large TSPs. For medium size problems the AP with Lagrangian relaxation performs better than

Inst.	OPT	AP			LAGR			LP			LP-CN		
		LB	Time	Fail	LB	Time	Fail	LB	Time	Fail	LB	Time	Fail
gr17	2085	1652	1,31	1926	2085	1,38	59	2085	0,84	15	2085	0,82	15
gr21	2707	2420	0,22	165	2707	0,3	62	2707	0,47	19	2707	0,53	19
gr24	1272	1052	0,39	300	1272	0,86	133	1272	36,4	1793	1272	44,0	1294
fri26	937	833	2,36	2863	937	2,46	1191	937	180,1	8342	937	102	2969
bays29	2020	1764	2,87	2803	2014	1,67	601	2014	82,6	3320	2014	138	2892
dantzig42	699	532	*	*	697	6,19	1277	697	181,9	3810	697	106	700

Table 4.4: Results on small symmetric TSP instances

the AP since the increased complexity of adding cuts is balanced by a great reduction in the number of fails. Dantzig42 could not be solved by the AP (indeed the optimal solution was found but the proof of optimality falls outside the time limit), whereas it is solved in few seconds with the Lagrangian relaxation. The AP does not provide in general a good lower bound for symmetric TSPs since many cycles of length 2 are created. Thus, the addition of SECs in general improves the bound significantly.

Concerning the LP approach, for small instances, it always performs worse than all the other approaches. This is mainly due to the fact that the simplex algorithm in general does not behave as efficiently as a special purpose (incremental) algorithm. In addition, the chosen branching strategy is certainly not the most effective for the LP. An alternative, more effective strategy considers the optimal solution of the linear relaxation and branches on fractional variables.

4.2 Traveling salesman problem with time windows

The Traveling Salesman Problem with Time Windows (TSPTW) is the problem of finding a minimum cost tour visiting a set of cities where each city is visited exactly once. In addition, each city must be visited within a given time window even if early arrivals are allowed. In general, the cost is computed as the sum of travel costs from each couple of cities in the path. The problem is NP-hard, and Savelsberg [150] showed that even finding a feasible solution of TSPTW is NP-complete. TSPTW has important applications in routing and scheduling. For this reason, it has been extensively studied in the OR field [50], and also previously faced via Constraint Programming [133].

The TSPTW can be seen as the union of two combinatorial optimization problems, i.e., the Traveling Salesman Problem (TSP) and a scheduling problem. In TSPs *optimization* usually results to be the most difficult issue. In fact, although the feasibility problem of finding an hamiltonian cycle in a graph is also NP-hard [77], it is usually easy to find a feasible solution for a TSP, while the optimal solution is very hard to find. TSPs have been efficiently solved by using OR methods such as Branch and Cut [4] and Local Search [150]. Some OR Branch and Bound approaches have also been proposed, see for instance [106]. All the OR exact methods (i.e. Branch and Cut, and Branch and Bound) for TSP optimally solve a relaxation of the original problem.

On the other hand, scheduling problems with release dates and due dates may contain difficult feasibility issues involving disjunctive, precedence and resource capacity constraints. The solution of scheduling problems is probably one of the most promising areas of application to date [13, 36, 37], of constraint technology. This is also due to the definition of powerful propagation techniques, such as the *edge finding* [125], in global constraints.

We propose a set of techniques for solving the TSPTW using optimization constraints for coping with the optimization perspective, and CP propagation algorithms for the feasibility viewpoint.

The synergy between OR optimization techniques embedded in global constraints, and CP constraint solving techniques makes the resulting framework a technique of choice for the TSPTW. In fact, the proposed algorithm outperforms previous CP approaches, is competitive with state of the art OR approaches, and can be easily adapted to variants of the problem, e.g., TSPTW with precedence constraints, with pickup and delivery, with multiple time windows.

4.2.1 TSPTW: definition and previous approaches

The Traveling Salesman Problem with Time Windows is a time constrained variant of the Traveling Salesman Problem. It consists in finding the minimum cost path to be travelled by a vehicle, starting and returning at the same depot, which must visit a set of n cities exactly once. Each couple of cities has an associated travel time tt_{ij}, and travel cost c_{ij}. The service at a node i should begin within a time window $[a_i, b_i]$ associated to the node. Early arrivals are allowed, in the sense that the vehicle can arrive before the time window lower bound. However, in this case the vehicle has to wait until the node is ready for the beginning of service. This problem can be found in a variety of real life applications such as routing, scheduling, manufacturing and delivery problems. For this reason, the problem has been studied both in the Operations Research and Constraint Programming fields.

Within the OR community, despite of the difficulty of the problem (as mentioned even finding a feasible tour to the TSPTW is an NP-complete problem), many exact algorithms have been presented.

The first approaches are due to Christofides, Mingozzi and Toth [42] and Baker [7] and consider a variant of the problem where the total schedule time has to be minimized (MPTW). The first paper presents a Branch and Bound algorithm based on a state space relaxation approach, whereas the second one is again a Branch and Bound algorithm exploiting a time constrained critical path formulation. Langevin et al. [106] addressed both MPTW and TSPTW by using a two-commodity flow formulation within a Branch and Bound scheme. Dumas, Desrosiers, Gélinas and Solomon [55] proposed a dynamic programming approach for the TSPTW that extensively exploits elimination tests to reduce the state space. More recently, a dynamic programming algorithm has been presented by Mingozzi, Bianco and Ricciardelli [116]. The algorithm embeds bounding functions able to reduce the state space (derived by a generalization of the state space relaxation technique [42]) and can be also applied to TSPTW problems with precedence constraints.

The most recent approaches are due to Ascheuer, Fischetti and Grötschel [6] and to Balas and Simonetti [10]. The first paper considers several formulations for the asymmetric version of the problem and compares them within a Branch and Cut scheme, whereas the second one is again a dynamic programming framework.

Quite surprisingly, despite the importance of the problem, few efforts have been devoted to the study of the TSPTW in the Constraint Programming community. Caseau and Koppstein [35] have faced a large task assignment problem where a set of small TSPTW instances are solved with a good average performance. The combination of scheduling and routing models allows to benefit from the advantages of each formalization when one view of the problem dominates the other.

More recently, TSPTW has been solved by enriching a simple CP model with redundant constraints [133]. These additional constraints embed arc-elimination and time window reduction algorithms previously proposed in the OR field in [106] and [49] respectively.

A variant of the TSPTW, called TSP with Multiple Time Windows, has been solved in [134]. The authors underline the CP flexibility by showing that this variant of the TSPTW can be solved with exactly the same algorithm used for the original problem by slightly changing the model.

In the Constraint Programming community, some work has been done concerning pure TSPs. Caseau and Laburthe [38] have solved small TSP instances with constraints by using a set of simple, but effective, propagation algorithms, and cost based reasoning deriving from the use of the Lagrangian relaxation of the problem. We have solved TSP using optimization constraints as shown in Section 4.1, and [71, 67].

4.2.2 CP Model

$$\begin{array}{rl}
\min & z \\
\text{on} & \\
 & vehicle = UnaryResource(tt) \\
\forall i \in \{1,\ldots,N\} & start_i \; :: \; [a_i..b_i] \\
\forall i \in \{1,\ldots,N\} & service_i = Activity(start_i, d_i, i) \\
 & z \geq 0 \\
\forall i \in \{1,\ldots,N+2\} & next_i \; :: \; [0..n+1]
\end{array}$$

$$\forall i \in \{1,\dots,N\} \quad succ_i \ :: \ [\{\}..\{1,\dots,n-1\}]$$

subject to

$$\forall i \in \{1,\dots,N\} \quad service_i \text{ requires } vehicle \tag{4.2}$$
$$precedenceGraph(vehicle, succ, next) \tag{4.3}$$
$$pathCost(next, z, c) \tag{4.4}$$
$$pathCostPC(next, succ, z, c) \tag{4.5}$$

Most Constraint Programming languages provide modeling objects such as *Activities*, *Resources*, etc. Using such objects rather than only variables yields more concise models. Even if the actual syntax of such objects may vary depending on the specific CP language at hand, we may safely assume that the model proposed can easily be coded using most CP languages. We are given a *vehicle*, and n *visits* that must be performed within a *time window.* The vehicle is a *UnaryResource* object containing the information on the travel time among locations (cities). The service at each city is an *Activity* object defined by a variable start-time, a constant duration, and a location. Constraint (4.2) $service_i$ requires *vehicle* enforces that from $start_i$ to $start_i+d_i$ the *UnaryResource vehicle* is used by *Activity* $service_i$ without interruption. A *UnaryResource* cannot be simultaneously used by more than one *Activity*, and it cannot be used during the time needed to move from location to location. An equivalent model (when the triangular inequality on tt holds) is that for every pair of *Activity* $service_i$, $service_j$ $(start_i \geq start_j + d_j + tt_{ji}) \bigvee (start_j \geq start_i + d_i + tt_{ij})$. The scheduling model represented by the set of activities and the resource correctly models the TSPTW. On the other hand a routing model is also necessary to permit explicit reasoning on the sequence of cities. As in the TSP model, the variable $next_i$ identifies the next city visited after city i; in addition to the variables identifying direct successor in the path $next$, it is also possible to consider set variables $succ_i$ identifying the set of successor of node i [2].

The routing model is described by the constraint *pathCost*. Alternatively, the constraint *pathCostPC*, considering both the next and the succ relations can be used.

Constraint *precedenceGraph* (4.3) links the scheduling model and the routing model. The propagation of constraint *precedenceGraph* is described in 4.3.4, and [95].

4.2.3 Relaxations

We have compared three relaxations:

- *pathCost* using the Assignment Problem, referred to as **AP** (see Section 3.5.3);
- *pathCostPC* using the Assignment Problem generated by relaxing cuts in a Lagrangian way at the root node, referred to as **LAGR** (see Section 3.5.5);
- the same relaxation as **LAGR** where purging techniques have been used to remove no longer necessary cuts, referred to as **LAGR-Pg** (see Section 3.5.5).

[2]the actual implementation we used is not based on set variables, but on the precedence graph object of ILOG Scheduler (see [95])

4.2.4 Heuristics

We have already mentioned that the TSPTW can be seen as a mix of TSP and Scheduling Problem. For this reason we have tested different branching strategies suited for one or the other aspect of the problem. In this section, we present two branching techniques: the first is more suitable for the TSP model, while the second is a branching strategy often used in scheduling problems, thus being more focused on time windows feasibility.

The most commonly used branching scheme for Branch and Bound algorithms for TSP which uses the AP as relaxation is the *sub-tour elimination strategy* (see Section 4.1.3 and [11]). At any stage of the search tree, we consider the solution of the AP, and we choose a tour identified by arcs $(i_1, i_2), (i_2, i_3), \ldots, (i_s, i_1)$. The *sub-tour elimination* strategy imposes the following disjunction:

$$(next_{i_1} \neq i_2) | (next_{i_1} = i_2 \wedge next_{i_2} \neq i_3) | \ldots | (next_{i_1} = i_2 \wedge \ldots next_{i_{s-1}} = i_s \wedge next_{i_s} \neq i_1)$$

The tour chosen $(i_1, i_2), (i_2, i_3), \ldots, (i_s, i_1)$, infeasible for the TSP, will not appear in any of the generated branches. This search strategy has an important drawback when applied to TSPTW: the branching decision $(next_{i_k} \neq i_{k+1})$ has often no influence on the *start* variables, i.e., on the scheduling view of the problem. The use of CP enables us to easily obtain a variant of the sub-tour elimination strategy for the TSPTW. In fact, instead of partitioning the search space by fixing and removing values, we can use constraints. Again, in any stage of the search tree, we consider the solution of the AP, and we choose a tour identified by arcs $(i_1, i_2), (i_2, i_3), \ldots, (i_s, i_1)$. We impose the following disjunction:

$$(i_2 \in succ_{i_1}) | (i_1 \in succ_{i_2} \wedge i_3 \in succ_{i_2}) | \ldots | (i_1 \in succ_{i_2} \wedge \ldots i_{s-1} \in succ_{i_s} \wedge i_1 \in succ_{i_s})$$

Where $k \in succ_i$ introduces a temporal constraint between the visits at nodes h and k. Also in this case, the tour chosen $(i_1, i_2), (i_2, i_3), \ldots, (i_s, i_1)$, infeasible for the TSP, will not appear in any of the generated branches, since the temporal constraint will propagate both on the *next* variables and on the *start* variables. This search strategy appeared to be very effective for TSPTW with large time windows since in this case the TSP view dominates the scheduling one.

The second heuristics tested is a typical scheduling branching strategy. When dealing with scheduling problems, very often a solution is built in chronological order. At each node of the search tree the activity with the minimum earliest start time is chosen and scheduled. In our case, we sequence all nodes starting with assigning a node to the initial depot (we fix its variable *next*), until the last node is assigned to the final depot. For each given node, we choose the value to be assigned as follows: among all values having zero gradient (we know that there exists at least one of such values), we select the one corresponding to the node with the smallest latest start time. This strategy is always able to quickly find solutions for the TSPTW (maybe of not very good quality), and it is therefore particularly suited when the scheduling part is dominant.

4.2.5 Results

The overall approach has been implemented using ILOG Solver and Scheduler, and runs on a Pentium II 200 MHz. We tested the algorithm on three different sets of problems from the literature in order to obtain a strong indication of the effectiveness of the proposed approach with respect to both OR and CP literature.

The first set of problems considered was proposed in [106]. In particular, Langevin et al. solved randomly-generated symmetric instances with up to 60 cities (see [106] for a detailed description of the problem generation method). The optimization constraint is based on the Assignment Problem.

n	w	subtour		sequence		Langevin [106]
		time	fails	time	fails	time
40	20	0.05	3.2	0.066	3.2	1.7
	30	0.078	7.4	0.078	8.4	4.4
	40	0.08	12	0.122	15.6	7.3
	60	0.186	26	0.22	31.6	-
60	20	0.22	7.75	0.19	7.25	7.9
	30	0.27	15.75	0.2075	15.25	15.0
	40	0.412	30	0.465	51.5	17.6
	60	0.78	63.75	2.0475	195	-
80	20	0.538	14.6	0.45	15.8	-
	30	0.912	47.2	1.43	71.4	-
	40	1.846	86	17.04	1247	-
	60	93.822	4682.6	4/5 in 15 mins		-

Table 4.5: Results on instances from Langevin et al. [106]

Results are shown in Table 4.5 where n is the number of cities and w the size of the time windows. We report computing times (in seconds) and number of fails of the sub-tour elimination strategy and the chronological scheduling strategy on an average of 5 randomly generated problems. Since the problems do not present feasibility issues, the sub-tour elimination strategy outperforms the sequence one. The results reported in the last column are taken from [106] and were obtained on a SUN Sparc Station 2. Note that, the solved instances are obviously not the same from [106], but the same generation policy has been maintained. Also Mingozzi et al. [116] solved instances from the same set by Dynamic Programming and claim to obtain computing times on average five times smaller than those achieved by Langevin et al.

The second set of problems is derived from the RC instances of Solomon [155] for the Vehicle Routing Problem with Time Windows (VRPTW). In particular, the TSPTW instances are obtained by considering the single-vehicle decomposition deriving from VRPTW solutions in the literature (i.e., Rochat and Taillard [145] and Taillard et al. [157]). This decomposition generates instances with up to 40 cities and has been solved by Pesant et al. [133]. It is easy to see that the optimization of the route of a single vehicle is in general very interesting because it directly leads to an improvement of the overall VRPTW solution. The optimization constraint is based on the Assignment Problem.

The results (reported in Table 4.6) for these instances in terms of computing times, number of failures and quality of the solution are compared with the ones of the CLP approach by Pesant et al. [133] (where the CPU time is given with respect to a SUN SS1000). Note that our results are obtained by solving the problem from scratch, while [133] solves the problems by imposing an objective function upper bound equal to the best solution for the problem taken from Taillard et al. [145] (reported in column VRP) which is always very close to the optimal one.

Inst.	n	Subtour			Sequence			Pesant et al. [133]			VRP
		best	time	fails	best	time	fails	best	time	fails	best
rc201.0	25	378.62	0.1	19	378.62	0.15	25	378.62	51	607	378.7
rc201.1	28	374.70	3.21	778	374.70	51.5	13.5k	374.7	311	4.03k	376.6
rc201.3	19	232.54	0.09	25	232.54	0.11	35	232.54	6.5	77	232.6
rc202.0	25	246.23	9.81	2.96k	246.22	5.9	1.83k	246.22	4.16k	32.2k	246.3
rc202.1	22	206.54	1.77	553	206.53	3.4	1.07k	206.53	1.80k	19.5k	206.6
rc202.2	27	341.77	1.11	257	341.77	0.93	265	341.77	696	5.15k	345.1
rc202.3	26	367.85	33.35	8.06k	367.85	325	111k	367.85	11.1k	65.3k	369.8
rc205.0	26	251.65	6.88	1.74k	251.65	4.4	1.45k	251.65	652	4.77k	251.7
rc205.1	22	271.22	0.09	17	271.22	0.16	54	271.22	128	1.32k	271.3
rc205.2	28	434.69	847	237k	434.69	1.21k	375k	436.64	24h	-	442.5
rc205.3	24	361.24	3.81	1.04k	361.24	6.78	2.27k	361.24	7.02k	90.9k	362.9
rc207.0	37	-	2h	-	436.69	757	112k	436.69	24h	-	439.6
rc207.1	33	396.39	2.7k	484k	396.36	1.70k	332k	396.36	24h	-	396.4
rc207.2	30	246.41	284	62k	246.41	160	40k	246.41	6.79k	62.9k	246.5

Table 4.6: Results on rc2 instances from Solomon et al. [155]

We find the optimal solution for each instance and prove optimality, while Pesant et al. [133] are not able to prove optimality of instances rc205.2, rc207.0 and rc207.1 in 24h. Moreover, we are able to find the optimal solution to problem rc205.2, which was never found before. From Table 4.6 we can see that, while the sub-tour elimination strategy for random instances (Table 4.5) always outperforms the chronological one, in this case, depending on the problem structure, the chronological strategy can lead to better results. For example, in the rc207.0 instance, the sub-tour elimination strategy fails to find any feasible solution in 2 hours.

The third set of instances derives from real-world problems with up to 233 cities originally proposed by Ascheuer et al. [6]. These instances are asymmetric and derive from stacker crane routing applications [3]. The optimization constraint is based on the Assignment Problem.

The results for a subset of these asymmetric instances are reported in Table 4.7 and are compared with the ones obtained by the branch-and-cut approach of Ascheuer et al. [6] (on a Sun Sparc Station 10). In the second column, we report either the value of the optimal solution (when known) or the lower and upper bounds of the objective function from Ascheuer et al.. Numbers in round brackets refer to results obtained by the dynamic programming approach by Balas and Simonetti [10].

In this set of instances, the search strategy that gave the best results is the chronological heuristics, while we could not achieve good results by using the sub-tour elimination strategy. In addition, since in rbg instances the scheduling component dominates the TSP one, we have tested the *Limited Discrepancy Search (LDS)* [85, 131], previously proved to be effective in scheduling problems. LDS was first defined in [85] and is based on the assumption that the left branches are more likely to be successful than right ones. In particular, with the LDS strategy we obtain the best known solution for the rbg048a instance.

A final remark concerns the use of a pure CP approach with ILOG Solver and Scheduler. We did not report results in detail, since they are disappointing. For the 14 instances of Table

[3]The complete set can be downloaded from the web page:
`http://www.zib.de/ascheuer/ATSPTWinstances.html` along with the best known solutions.

Instance	Ascheuer et al. [6]		Sequence			Sequence LDS		
	best	time	best	time	fails	best	time	fails
rbg010a	149	0.12	149	0.03	11	149	0.02	11
rbg016a	179	0.2	179	0.04	12	179	0.03	12
rbg020a	210	0.2	210	0.03	10	210	0.04	10
rbg027a	268	2.25	268	0.19	33	268	0.18	33
rbg031a	328	1.7	328	279	99.5k	328	318	89.2k
rbg048a	[456-527]	5h	503	2h	-	487	2h	-
rbg049a	[420-501] (486)*	5h	505	2h	-	497	2h	-
rbg050b	[453-542] (518)*	5h	546	2h	-	548	2h	-
rbg050c	[508-536]	5h	573	2h	-	542	2h	-

Table 4.7: Results on rbg instances from Ascheuer et al. [6]

4.6, for example, this approach is able to prove optimality only in 8 cases within 30 minutes, and the computing times are at least 10 times larger than those obtained by our approach.

The last set of results compares the performance of the optimization constraint based on the Assignment Problem with the performance of the optimization constraint integrating cutting planes.

The set of problems is derived from the RC instances of Solomon [155] for the Vehicle Routing Problem with Time Windows (VRPTW). In particular, the TSPTW instances are obtained by considering the single-vehicle decomposition deriving from VRPTW solutions. This decomposition generates instances with up to 40 cities and has been solved by Pesant et al. [133].

The results for these instances are reported in Table 4.8 in terms of computing times (in seconds) and number of failures, with a time limit of 30 minutes.

Inst.	n	AP		LAGR		LAGR-Pg	
		time	fails	time	fails	time	fails
rc201.0	25	0.08	26	0.44	21	0.23	21
rc201.1	28	37.36	14.5k	24.08	7.3k	23.7	6.14k
rc201.3	19	0.12	42	0.24	42	0.18	6.5
rc202.0	25	2.78	1334	1.87	93	1.60	86
rc202.1	22	3.08	1.33k	4.43	1.21k	4.27	1.21k
rc202.2	27	0.92	377	1.50	349	1.68	345
rc202.3	26	292.35	153k	173.03	89k	173.79	62k
rc205.0	26	5.49	2.28k	1.10	124	1.16	118
rc205.1	22	0.61	214	0.88	193	0.93	189
rc205.2	28	898	483k	1.5k	483k	1.4k	483k
rc205.3	24	4.78	1.86k	7.58	1.78k	7.09	1.68k
rc207.0	37	752	165k	832	141k	788	139k
rc207.1	33	1.5k	453k	174	32.8k	165	31.6k
rc207.2	30	163.62	51k	2.09	238	1.81	168

Table 4.8: Results on rc2 instances from Solomon et al. [155] with cutting plane

The quality of the LAGR and LAGR-Pg strongly depends on the quality of the bound produced at the root node. For instance, in rc207.1, the bound produced by the optimization at the root node using cuts is very tight (close to the optimal solution), while the AP bound is poor. Instead, for rc205.2 the improvement of the bound produced by the cut generation is small with respect to the gap between the bound and the optimal solution (in fact the number of failures is the same in the three approaches). In this case, the AP approach benefits from a lower computational complexity.

4.2.6 Conclusions

We have previously tackled pure TSPs in [71], by exploiting lower bound calculation and reduced costs fixing for calculating the gradient function. On pure TSPs, we have achieved the best known results which are, however, far to be comparable with state-of-the-art OR branch and cut approaches to the problem. Here, we have adopted the same techniques for performing cost-based domain filtering for the time-constrained variant of the problem. We achieve results that are comparable, in some cases even better, than those found via sophisticated Branch and Cut approaches. This comes from the flexibility of Constraint Programming models that allow to cope with variants of the same problem in an easy way, while traditional OR approaches require much effort if the problem specifications slightly change. Moreover, we are able to find the optimal solution for instances which have never been solved to optimality before.

4.3 Scheduling problem with setup times and alternative resources

In this Section a general shop scheduling problem with sequence dependent setup times and alternative resources is considered, where optimization criteria are both makespan and sum of setup times. Two cooperative models for the problem based on Constraint Programming are proposed. The first is used to model the scheduling constraints, while the second is a multi-path model used for setup optimization. Integrating lower bounding techniques for the sum of setup times, the multi-path model performs propagation based on reduced cost fixing. A solution method based on a two phase algorithm is described, and a computational study is performed both on instances known from literature as on newly proposed instances. It is shown that the cooperation of the two models significantly improves performance. Although the aim of this work is to study the problem including alternative resources, for several known instances without alternative resources, we were able to improve on the best known results.

4.3.1 Introduction

The scheduling problem studied is a general shop problem with sequence dependent setup times and alternative resources for activities. The setup time between two activities act_1 and act_2 is defined as the amount of time that must elapse between the end of act_1 and the start of act_2, when act_1 precedes act_2. If an activity can be scheduled on any one resource from a set S of resources, we say that S is the set of alternative resources for that activity. We consider two criteria as objective functions: makespan and sum of setup times. A large part of the motivation for this study was found in our experience with industrial applications. There we found that both sequence dependent setup times and alternative resources are very commonly encountered properties of scheduling problems. What's more, both properties are also often important in the sense that not considering them leads to unacceptable solutions. The importance of setup times and setup costs has also been investigated in several other studies. Allahverdi, Gupta, and Aldowaisan, in reviewing the research on scheduling involving setup considerations [3], discuss the importance of scheduling with sequence dependent setups in real world applications, and encourage researchers to work on the subject. Our experience with industrial applications also formed the basis for the motivation for considering both makespan and sum of setup times, and, more specifically for trying to find a schedule with a good makespan and a minimal sum of setup times. We chose to use Constraint Programming as it has been proven to be a very flexible framework to model and solve scheduling problems. Numerous industrial applications have been developed using constraint-based scheduling both because of its modeling capability and for the efficiency of specialized Operations Research algorithms that are embedded in global constraints. For more detailed discussions of constraint-based scheduling we refer to [125, 12, 37, 108, 124, 16]. We introduce two CP based models capturing two different aspects, i.e., the scheduling aspect and the multi-path aspect. The constraints representing the two models and a constraint linking them are described. The scheduling representation is essentially used to enforce feasibility, while the multi-path representation is used to effectively minimize the sum of setup times using OR techniques, i.e., lower bound calculation and reduced costs fixing. We propose a two phase procedure to solve the scheduling problem. In the first phase a good solution with respect to the makespan is found, after which, in the second phase, a local improvement method aims at minimizing the sum of setup times while maintaining a limit on the maximal makespan equal to the best makespan found during the first phase.

4.3.2 Problem Definition

We are given a set of n activities $act_1, ..., act_n$ and a set of m unary resources (resources with maximal capacity equal to one) $res_1, ..., res_m$. Each activity act_i has to be processed on a resource res_k for p_i time units. act_i can choose Resource res_k within a given subset RS_i of the m resources. Sequence dependent setup times exist among activities. Given a setup time matrix s^k (square matrix of dimension n), s^k_{ij} represents the setup time between activities act_i and act_j if act_i and act_j are scheduled sequentially on the same resource res_k. In such a case, $start_j \geq end_i + s^k_{ij}$. There may furthermore exist a setup time su^k_j before the first activity act_j can start on resource res_k and a teardown time td^k_i after the last activity act_i finishes on resource res_k. Activities may be linked by precedence relations $act_i \rightarrow act_j$. In this case activity act_j cannot start before the end of activity act_i. Constraints of the problem are therefore defined by the resource capacity, the transition times, the temporal relations, and the time bounds of the activities (release date and due date). The goal we will follow is to first find a schedule with the best possible makespan after which we will try to minimize the sum of setup times.

4.3.3 Related Work

A comprehensive review of the research on scheduling involving setup considerations was given in [3]. The authors review the literature on scheduling problems with sequence dependent and sequence independent setup times, on single machine and parallel machines. They finally suggest directions for future research in the field. Here we follow some of these directions, i.e., emphasis on multi-machine scheduling problems, on multi criteria objectives, and on a generalized shop environment. An important reference for the work we propose is the paper of Brucker and Thiele [28] where the authors propose a branch and bound algorithm for a scheduling problem with sequence dependent setup times. We generalize the problems described in this paper by considering alternative machines. Moreover, while for the authors the makespan is the only objective that is taken into account, we consider both makespan and sum of setup times. For general considerations on cost-based filtering used in the setup optimization we refer to [67].

4.3.4 CP Model

$$
\begin{array}{rl}
\min & z \\
\text{on} & \\
\forall k \in \{0, \ldots, m-1\} & res_k = UnaryResource(s^k, su^k, td^k) \\
\forall i \in \{0, \ldots, n-1\} & start_i \; :: \; [a_i..b_i] \\
\forall i \in \{0, \ldots, n-1\} & act_i = Activity(start_i, p_i, i), \\
 & z \geq 0 \\
\forall i \in \{0, \ldots, n+2m-1\} & next_i \; :: \; [0..n+2m-1], \\
\forall i \in \{0, \ldots, n-1\} & selectedRes_i \; :: \; RS_i \\
\text{subject to} &
\end{array}
$$

$$
\begin{array}{lll}
(4.6) & \forall i \in \{0,\dots,n-1\} & act_i \text{ requires } res[selectedRes_i] \\
(4.7) & \forall i,j \in \{0,\dots,n-1\} | pc_{ij} = 1 & end_i < start_j \\
(4.8) & \forall k \in \{0,\dots,m-1\} & precedenceGraph(res_k, next) \\
(4.9) & & multiPathCost(next, selectedRes, z, s')
\end{array}
$$

Scheduling Model

The scheduling part of the problem is modeled by using ILOG Scheduler [95]. Each activity act_i is represented by an activity object using two variables being its start time $start_i$ and its end time end_i. These two variables are constrained by the relation $end_i - start_i = p_i$. Each resource res_k is represented by a unary resource object.

Each activity act_i is to be processed on resource res_k chosen among the resources belonging to RS_i. $selectedRes_i :: RS_i$ denotes the variable whose value represents the index k of the resource $res_k \in RS_i$ that will be chosen for activity act_i[4].

Setup times are represented in the scheduling model via a n square matrix s^k, and two vectors su^k, td^k associated with each resource res_k. Precedence constraints and time bound constraints are posted on the variables $start_i$ and end_i of activities. As we will see in section Scheduling Constraints, the scheduling model allows the propagation of precedence, time bound, setup times and resource availability constraints over the variables of activities $start_i$, end_i and $selectedRes_i$.

Path Model

A path model is used as a relaxation of the scheduling problem (see Section 4). Each internal node i represents an activity act_i. Each path represents a different resource. It starts in the start node of the resource, traverses a sequence of internal nodes, and ends in the end node of the resource. Variable $selectedRes_i$ is used both to identify the selected resource in the scheduling model and the selected path in the routing model. If an internal node i has its next variable assigned to another internal node j, activity act_i directly precedes activity act_j; if an internal node i has its next variable assigned to an end node $n+k$, activity act_i is the last activity scheduled on resource res_k. The transition cost function of the path model represents the transition time (setup time) among activities, where s'^k is the matrix defined "merging" s^k, su^k, td^k; therefore the minimization of $\sum_{i=0}^{n+2m-1} s'^{path_i}_{i\ next_i}$ in the path model corresponds to the minimization of the sum of setup times in the scheduling model.

In Figure 4.1 a schedule of 6 activities on 2 resources is shown with its correspondent path model. The scheduling model and the path model can be linked together and can cooperate for the solution of the problem by exploiting different views of the same problem.

Scheduling Constraints

In this section, we briefly describe the scheduling constraints that are used to perform propagation on the scheduling model.

[4]The same set RS_i is used both to identify a set of resources, and a set of resource indexes used to define the initial domain of $selectedRes_i$

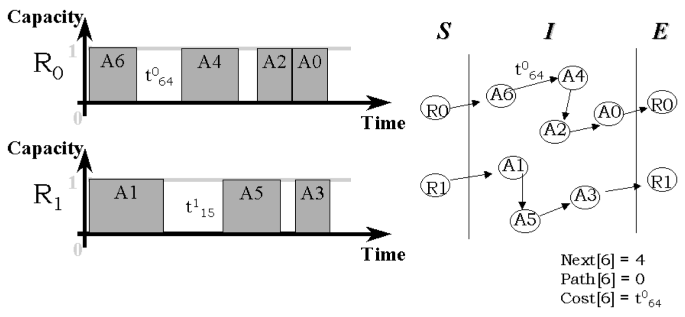

Figure 4.1: Models

Temporal Constraint

The temporal constraints represent the precedences between activities given in the problem definition. The precedence constraint between two activities act_i and act_j is propagated as a constraint $end_i \leq start_j$. If a precedence graph is used (see section Precedence Graph Constraint), these precedence constraints are also taken into account by the precedence graph.

Disjunctive Constraint

The disjunctive constraint aims at discovering new precedences by looking at pairs of activities that require the same unary resource. If act_i is an activity of the problem, we respectively denote $smin_i$, $smax_i$, $emin_i$ and $emax_i$ the earliest start time, latest start time, earliest end time and latest end time of activity act_i. Let act_i and act_j be two activities that require the same unary resource res_k. If $emin_j + s^k_{ji} > smax_i$, it means that activity act_j will not have enough time to execute before activity act_i. See an illustration on Figure 4.2 where we suppose that the setup time between act_1 and act_2 is 5. Thus, as both activities require the same unary resource res_k, act_i will have to be processed on res_k before act_j and the following domain reduction can be performed[5]: $smin_j = \max(smin_j, emin_i + s^k_{ij})$ and $emax_i = \min(emax_i, smax_j - s^k_{ij})$. Considering the example of the figure, the disjunctive constraint leads to a new earliest start time of 20 for activity act_2 and a new latest end time of 15 for activity act_1. Whenever the earliest end time or the latest start time of an activity act_i changes, the disjunctive constraint traverses the set of activities act_j that are to be processed on the same resource as act_i in order to perform this propagation.

Edge-Finding Constraint

The edge-finding constraint is more powerful than the disjunctive constraint. It propagates the start and end time of one activity with respect to a subset of other activities. In general the edge-finding constraint can deduce that one activity act_i is to be scheduled before or after a set S of activities that are all to be scheduled on the same resource. In Figure 4.3 we give an example of such a deduction. For a more detailed description we refer to [125]. In the

[5]If a precedence graph is used (see section Precedence Graph Constraint), this domain reduction will be performed by the precedence graph constraint. In that case, the disjunctive constraint only adds the new precedence to the graph.

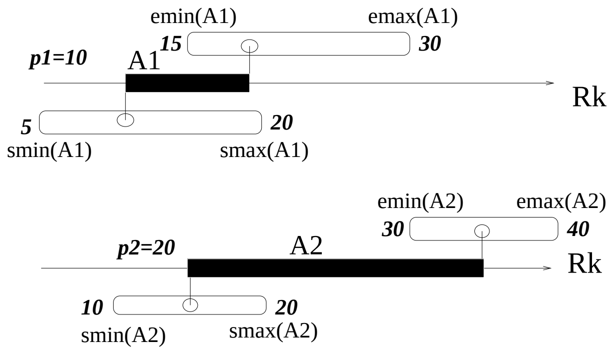

Figure 4.2: Disjunctive Constraint

example of Figure 4.3, the sum of the processing times on resource res_k on the time interval $[0, 30)$ is $p_1 + p_2 + p_3 = 25$. Thus, on this resource not enough slack time exists to allow the processing of activity *act* (whose duration is 20) on the interval $[0, 30)$. Activity *act* must thus be processed on res_k after activities act_1, act_2, and act_3 and its new propagated earliest start time is 25. A similar reasoning allows the edge-finding constraint to restrict the domain of the possible end times of activities by proving that a given activity must be processed before a subset of activities.

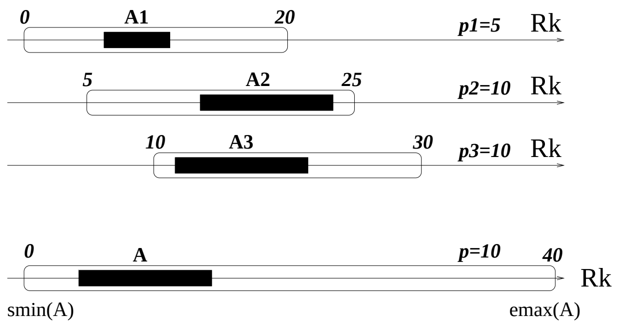

Figure 4.3: Edge-Finding Constraint

Alternative Resource Constraint

As seen in the scheduling model, each activity act_i may be processed on a resource res_k chosen within a given set of possible alternative resources $RS_i = \{res_{i,1}, ..., res_{i,p}\}$. Alternative resources are propagated as if the activity act_i was split into p alternative activities $act_{i,k}$ where each activity $act_{i,k}$ requires resource $res_{i,k}$ (see an illustration on Figure 4.4 for an activity act_i that must be processed either on res_1 or res_2). The alternative resource constraint maintains the constructive disjunction between the alternative activities $act_{i,k}$ that is, it ensures that:

- $smin_i = \min_{k=1..p}\ smin(act_{i,k})$

- $smax_i = \max_{k=1..p}\ smax(act_{i,k})$
- $emin_i = \min_{k=1..p}\ emin(act_{i,k})$
- $emax_i = \max_{k=1..p}\ emax(act_{i,k})$

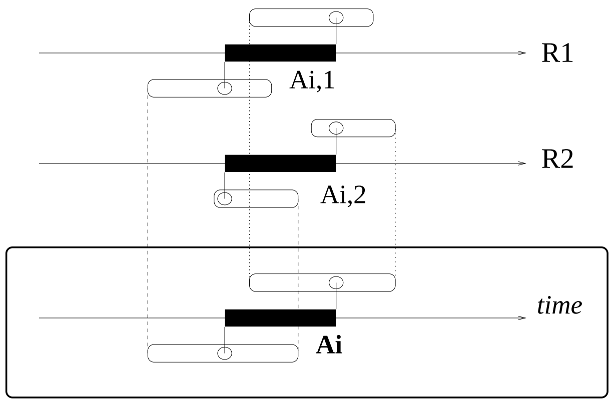

Figure 4.4: Alternative Resource Constraint

The scheduling constraints (disjunctive, edge-finding constraint) deduce new bounds for the alternative activities $act_{i,k}$ on the alternative resources res_k. Whenever the bounds of an activity $act_{i,k}$ turn out to be incoherent, the resource res_k is simply removed from the set of possible alternative resources for activity act_i. This is done by removing k from the possible values of the variable $selectedRes_i$ that represents the resource on which activity act_i will be processed.

Path Optimization Constraint

In this section, we describe the cost-based domain filtering algorithms previously used in [68] and [67] for TSPs, TSPTW, and Matching Problems, and proposed as a general technique in [67]. The idea is to create a global constraint embedding a propagation algorithm aimed at removing those assignments from variable domains which do not improve the best solution found so far. Domain filtering is achieved by optimally solving a problem which is a relaxation of the original problem.

Here, we consider the Assignment Problem (AP) [47] as a relaxation of the Path Model described (and consequently of the global scheduling problem).

The Path Model (PM) looks for a set of m disjoint paths each of them starting from a start node and ending into the corresponding end node covering all nodes in a graph. Considering each end node connected to the correspondent start node, the Path Model looks, in other words, for a set of m disjoint tours each of them containing a start node. A correspondent AP can be formulated on the graph defined by the set of nodes in PM and the set of arcs (i, j) such that $j \in next_i$. The cost on arc (i, j) is the minimal transition cost $min_k\{t_{ij}^k\}$. The optimal solution of the AP is obviously a lower bound on the optimal solution of the PM. The *primal-dual* algorithm described in [33] provides an optimal integer solution for the AP with a $O(n^3)$ time complexity. The AP relaxation provides: the optimal AP solution, i.e., a variable assignment; the value of the optimal AP solution which is a lower bound LB on the original problem; a reduced cost matrix $\bar{c}$. Each $\bar{c}_{ij}$ estimates the additional cost to be added to LB if variable $next_i$ is assigned to j. We have used these results to perform domain filtering and to define branching strategies as defined in Section 3.3.3.

Precedence Graph Constraint

We now need a way to link the path model and the scheduling model. This is done through a precedence graph constraint. This constraint maintains for each resource res_k an extended precedence graph G_k that allows to represent and propagate temporal relations between pairs of activities on the resource as well as to dynamically compute the transitive closure of those relations [105]. More precisely, G_k is a graph whose vertices are the alternative activities $act_{i,k}$ that may execute on resource res_k. A node $act_{i,k}$ is said to *surely contribute* if resource res_k is the only possible resource on which act_i can be processed. Otherwise, if activity act_i can also be processed on other resources, the node $act_{i,k}$ is said to *possibly contribute*. Two kind of edges are represented on G_k:

- A *precedence edge* between two alternative activities $act_{i,k} \rightarrow act_{j,k}$ means that if resource res_k is chosen for both activities act_i and act_j, then act_j will have to be processed after act_i on res_k.
- A *next edge* between two alternative activities $act_{i,k} \Rightarrow act_{j,k}$ means that if resource res_k is chosen for both activities act_i and act_j then, act_j will have to be processed directly after act_i on res_k. No activity may be processed on res_k between act_i and act_j.

The first role of the precedence graph is to incrementally maintain the closure of this graph when new edges or vertices are inserted, i.e., to deduce new edges given the ones already posted on the graph. The following two rules give a flavor of how this closure is computed[6]:

1. If $act_{i,k} \rightarrow act_{l,k} \rightarrow act_{j,k}$ and $act_{l,k}$ surely contributes then $act_{i,k} \rightarrow act_{j,k}$ (Transitive closure through contributor).
2. If $act_{l,k} \Rightarrow act_{i,k}$ and $act_{l,k} \rightarrow act_{j,k}$ and $act_{l,k}$ surely contributes then $act_{i,k} \rightarrow act_{j,k}$ (Next-Edge closure on the left).

As shown in Figure 4.5, new edges are automatically added on the precedence graph G_k by the scheduling constraints (precedence, disjunctive, edge-finding constraints) and by the path optimization constraint (whenever a variable $next_i$ is bound a new Next-edge is added). Besides computing the incremental closure, the precedence graph also incrementally maintains the set of activities that are possibly next to a given activity $act_{i,k}$ given the current topology of G_k. It allows to effectively reduce the domain of the variables $next_i$ in the path model. Furthermore, the precedence graph constraint propagates the current set of precedence relations expressed on G_k on the start and end variables of activities.

4.3.5 Problem Solving

The problem is solved in two phases: during the first phase a *good* solution w.r.t. makespan is searched for. Let the best makespan found in this phase be m^*. In the second phase a

[6]For reasons of space, the set of rules we describe here is not complete. The set of rules for ensuring a complete closure contains 5 rules [105].

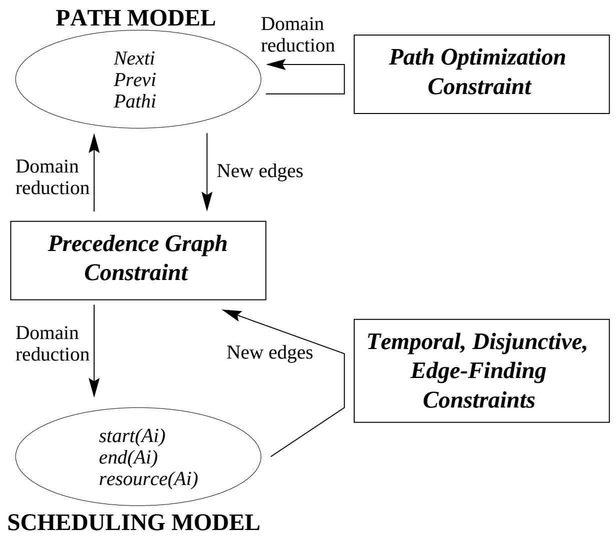

Figure 4.5: Architecture

constraint is added to the system imposing that any further solution will have a makespan smaller or equal to m^*. Local improvement methods are used to minimize the sum of setup times.

First Phase Heuristic

A time-limited, incomplete branch and bound method is used to find solutions trying to minimize the makespan. At each node of the search tree we administer for each resource which activity has been scheduled last. The set B contains all these activities. By analyzing the precedence graph, we choose an activity act_i among the set of activities that are still unscheduled and can be next to one of the activities in B. We branch on the relative position of act_i forcing act_i to be either *next* or *successor but not next* of one activity in B. Among all activities that can be chosen we select the one having the earliest possible start time and, in case of ties, the one having the smallest latest end time. In the left branch of the tree, imposing act_i next to one activity in B, we also need to choose the resource assignment for act_i. If one or more resource assignments are feasible we heuristically choose one of them, otherwise we backtrack. We choose the resource assignment that schedule the activity as early as possible, and in case of ties, the one which generates the smallest setup time.

Setup Optimization

In the setup optimization phase, given a solution having a makespan equal to m^*, and a total setup equal to s^*, we call for solutions having makespan less or equal to m^*, minimizing the sum of setup times. The improvement technique used is a time window based local optimization procedure. A time window $[TW_L^k, TW_U^k]$ defines a subproblem P_{TW^k} in the following way: in every resource, all activities on the left of the window are fixed (their start times and resource assignments are fixed); all activities on the right of the time window have their resource assignment fixed, and the sequence of the activities is also fixed (the variable *next* is fixed); all activities within the current window are completely free. On each subproblem a time limited branch and bound search is used to (possibly) find the optimal solution for P_{TW^k}. The branch and bound technique used to minimize the sum of setup times can effectively exploit the optimization constraint to reduce the search space, and eventually guide the search. In fact, the computational results will show that when the optimization constraint is used most subproblems are quickly solved up to optimality. Two

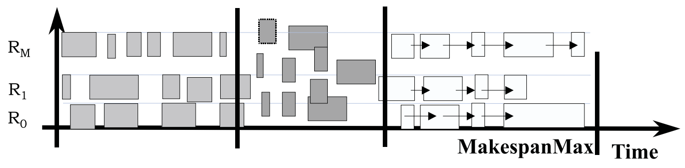

Figure 4.6: Setup optimization

different methods have been used to select the current window: a simple gliding window method and a lower bound based method.

Gliding Window

When the gliding window method is used, given a fixed size of the window W_{size}, and a window offset W_{delta}, we start the setup optimization at P_{TW^0} defined by window $[0, W_{size}]$, we optimize the problem, then we move the right and left bound of the window of W_{delta} to the left.

$$P_{TW^{k+1}} \leftarrow [TW_L^k + W_{delta}, TW_U^k + W_{delta}] \tag{4.10}$$

We repeat this until the end of the schedule is reached. At the end of each loop, the window size and offset can eventually be modified and another loop can be performed.

LB-based Window Selection

The method described here is based on the idea to work first on the part of the schedule where we can hope to obtain the highest improvement. For a given subproblem P_{TW^k}, defined by window $[TW_L^k, TW_U^k]$ we can calculate the expected improvement on the objective function E_{TW^k} as the difference between the current sum of setup times in that window, and the lower bound calculated in that window. After subproblem P_{TW^k} is defined, variable z identifying the sum of setup times contains the information of the lower bound calculated by the optimization constraint together with the precedence graph constraint and the scheduling constraints. Therefore if s^* is the total setup value of the current best solution found, E_{TW^k} is simply equal to $s^* - z_{min}$. In the LB-based window selection we first calculate the expected improvement on the objective function E_{TW^k} for a certain number of subproblems (depending on parameters similar to W_{size} and W_{delta}), and then sort the subproblems in descending order of E_{TW^k}. All subproblems that may lead to an improvement of the objective function are labeled as improvable. We run the branch and bound algorithm on the first ranked subproblem that can lead to an improvement, and change the label of the window. If a better solution is found, the current solution is updated, and the values E_{TW^k} of all windows on the right of the modified one are recalculated since they may have been changed by the new solution. Also, the labels of the windows on the right are updated. The windows are then re-sort and the procedure is repeated until no window exists that is labeled as improvable.

4.3.6 Computational Results

In the computational results we primarily try to show that the integration of lower bounding OR techniques in Constraint-Based Scheduling can improve performances both in terms of computation time and quality of solutions. We show that the large neighborhood defined by a time window containing between 30 and 60 activities can very effectively be solved by means of the interaction among the optimization constraint based on a lower bound calculation and reduced cost fixing, the precedence graph, and the scheduling constraints available in ILOG Scheduler. We have tested the proposed models on real world applications, moreover, the definition of the problem itself, and of the objective functions is a direct consequence of the real applications considered. In the following we will describe the generation process of the instances that we used to be able to compare our results with, hopefully, different approaches that could be presented. We will finally show the computational results.

Instance generation

Following the computational studies in [28], we run experiments on open-shop, general-shop and job-shop problems. We use the instances of Brucker and Thiele (available on the Web at http://www.mathematik.uni-osnabrueck.de/research/OR) to test the approach on scheduling problems with setup times without resource alternatives. We then duplicate and triplicate these instances to generate scheduling problems with setup times and alternative resources. In order to generate an instance with the alternative choice of k resources, for each activity and each resource in the original instance k activities and k resources are created. If in the original instance activity act_i requires resource res_j, in the k-multiplied instance each one of the act_{ih} identical activities requires one out of the k identical resources res_{jh}. The temporal constraints among activities are also duplicated such that if a temporal constraint exists in the original instance $act_i \rightarrow act_j$, the set of temporal constraints $act_{ih} \rightarrow act_{jh}$ exist in new instance. For all instances without alternative resources we can qualitatively compare our results with the results published in [28]. Nevertheless, a real comparison cannot be done since in [28] the objective is the minimization of the makespan, while we want to minimize the sum of setup times in a problem constrained by a maximal makespan. The open-shop problems considered contain, in the non-alternative instances, 8 machines and 8 jobs (16 machine and 16 jobs for the 2-alternatives instances etc..). The general-shop problems considered also contain, in the non-alternative instances, 8 machines and 8 jobs, and derives from the open-shop problems with the addition of the temporal constraints described in [28]. The job-shop problems considered contain 5 machines and 20 jobs in the non-alternative instances.

Results

Tables 4.9 to 4.11 report results on open-shop, general-shop, and job-shop instances. Table 4.9 reports results for the original instances from [28]. Table 4.10 and 4.11 report results for the 2 and 3-multiplied instances respectively, generated as described above, as alternative resources problems. For each problem we report the results obtained by the first solution phase, and the setup optimization phase in terms of sum of setup times (*su*), and makespan (*mks*). We finally report the results published in [28] in terms of makespan for all the instances without alternative of resources. All tests run on a Pentium II 300 MHz. The results published in [28] were obtained on a Sun 4/20 workstation where a time limit of 5 hours was set for open-shop and general-shop problems, and a time limit of 2 hours was set for the job-shop problems.

Column *FirstSol* reports results in terms of makespan and sum of setup times of the best solutions obtained using the time limited branch and bound strategy described in section First Phase Heuristics. The time limit given was 60 seconds, and a Limited Discrepancy Search tree exploration was used, see [85, 131]. The solution obtained after this phase is thought to be a good solution w.r.t. makespan minimization. For example, for all the problems without alternative resources the makespan obtained is very close to the best known solution published in [28]. In half of the instances considered, the makespan found in the first solution phase improves the best known published in [28]. These results were used as starting point for setup optimization.

In the setup optimization phase we fixed an initial window size of 30 activities (i.e. each subproblem has 30 completely free activities), and we used a 5 seconds time limited branch and bound algorithm to minimize the sum of setup times in each subproblem. In order to compare the results obtained with and without the optimization constraint, we used always the same very simple branching strategy: we choose the variable *next* with the smallest domain and we branch on its possible values starting from the one generating the smallest setup time. Given an initial window size, the setup optimization methods (columns *noLB GW*, *LB GW*, *LB Rank*) are called until a local minimum is reached, then the window size is increased 20% (e.g. from 30 free activities to 36 free activities), and the procedures are repeated until a global time limit is reached. Column *noLB GW* and column *LB GW* report the results obtained by the gliding window method described in section Gliding Window. The algorithm used for column *noLB GW* does not calculate the lower bound on the sum of setup times, while the algorithm used for column *LB GW* makes full usage of the pruning based of the lower bound calculation and the reduced cost fixing. Column *LB Rank* reports the results obtained by the method described in section LB-based Window Selection.

For the open-shop and general-shop instances of Table 4.9 (containing 64 activities each) the global time limit used is 30 seconds. For the job-shop instances of Table 4.9 (containing 100 activities each), for the open-shop, and general-shop instances of Table 4.10 (containing 128 activities each) the global time limit used is 60 seconds. For the job-shop instances of Table 4.10 (containing 200 activities each), for the open-shop, and general-shop instances of Table 4.11 (containing 192 activities each) the global time limit used was 120 seconds. For the job-shop instances of Table 4.11 (containing 300 activities each) the global time limit used is 240 seconds.

When the optimization constraint is used in collaboration with scheduling propagation algorithms the solutions obtained are always a lot better than the ones obtained without the optimization constraint. The improvement in the solution quality is particularly important for problems with two and three alternative resources. Problems without alternative resources are easier and even without the optimization constraint, in each window, the local optimal solution can often be found. Nevertheless, even in these cases, when the optimization constraint is used the subproblems are solved up to optimality in a shorter time.

More difficult is the comparison between the LB-based Window Selection and the Gliding Window method since the best solutions are equally distributed between the two methods. We cannot at this point claim that the LB-based Window Selection method outperforms the simpler Gliding Window method. Indeed, if the scheduling problem is small enough to

	FirstSol		noLB GW		LB GW		LB Rank		BT96
open-shop									
	su	mks	su	mks	su	mks	su	mks	mks
TAIBS81	2680	942	1740	928	1620	919	1480*	936	914*
TAIBS85	3480	1113	2180	985	1280*	1108	1280*	1108	899*
TAIS81	1460	699	980	693	890*	690*	980	693	713
TAIS85	1850	755	1260	748	790*	748	850	754	747*
general-shop									
	su	mks	su	mks	su	mks	su	mks	mks
TAIBGS81	1680	763	1410*	763	1410*	763	1470	759*	837
TAIBGS85	2010	869	1150	862	870*	867	870*	867	762*
TAIGS81	1510	734*	1190	734*	1150*	734*	1190	734*	858
TAIGS85	1540	749	1210	745*	1010*	747	1160	747	783
job-shop									
	su	mks	su	mks	su	mks	su	mks	mks
T2-PS12	1710	1450	1640	1450	1640	1450	1530*	1448*	1528
T2-PS13	1930	1669	1640	1667	1640	1667	1430*	1658	1549*
T2-PSS12	1480	1367	1300	1367	1300	1367	1220*	1362*	1384
T2-PSS12	1290	1522	1220	1522	1140*	1522	1220	1518	1463*

Table 4.9: Original instances from Brucker & Thiele 1996.

	FirstSol		noLB GW		LB GW		LB Rank	
open-shop								
	su	mks	su	mks	su	mks	su	mks
TAIBS81	3920	908	3760	903	2760*	904	2840	905
TAIBS85	4520	942	4260	940	2580	939	2380*	942
TAIS81	2220	723	2060	723	1540*	723	1590	723
TAIS85	2280	690	2110	689	1730*	690	1950	689
general-shop								
	su	mks	su	mks	su	mks	su	mks
TAIBGS81	2220	1023	2140	1023	1270*	1008	1490	1017
TAIBGS85	2640	1031	2350	1019	1300	1020	1150*	1026
TAIGS81	2510	766	2430	764	1900	766	1720*	756
TAIGS85	2490	748	2450	748	1810	743	1710*	748
job-shop								
	su	mks	su	mks	su	mks	su	mks
T2-PS12	3410	1562	2980	1537	2330*	1552	2510	1551
T2-PS13	2890	1593	2670	1593	2270	1584	2240*	1593
T2-PSS12	2090	1515	1820	1479	1610	1505	1540*	1515
T2-PSS12	2120	1578	1720	1576	1520*	1574	1590	1545

Table 4.10: Instances with alternative of two resources.

allow one or several complete gliding window loops, the LB-based method may loose some interest: if all windows are considered, the order in which they are solved may not be too important. On the other hand, for very large problems a complete gliding window loop may not be possible within the CPU time available. In such a case, a fast evaluation of the most promising area for improvement may play an important role. Further analysis of the relative advantages of the two methods will be subject of future work. In Figures 7-9, the plot of Table 4.9 to 4.11 is reported; the x-ax represents the problem instance, while the ratio between the final sum of setup times, and the sum of setup times of the first solution is reported on the y-ax.

It is interesting to look in more details at the results obtained by the optimization constraint compared to the ones obtained without the lower bound calculation. In Table 4.12 we report, for each type of problem, the average number of subproblems created, the average number of subproblems solved up to optimality (in percentage), and the average time spent in each window, when one single gliding window loop is performed. We recall that the time limit for each window is 5 seconds, therefore whenever the average time spent per window is close to 5 seconds it means that most windows could not be solved up to optimality.

We can see that when the optimization constraint is used we can solve, on average, 80%

	FirstSol		noLB GW		LB GW		LB Rank	
	open-shop							
	su	mks	su	mks	su	mks	su	mks
TAIBS81	4780	1002	4380	999	3320*	999	3520	986
TAIBS85	5320	875	5280	875	4180	870	4160*	865
TAIS81	2910	802	2440	802	2190	800	2090*	802
TAIS85	2660	758	2540	758	2020	755	1690*	757
	general-shop							
	su	mks	su	mks	su	mks	su	mks
TAIBGS81	2230	1083	2140	1067	1380*	1079	1540	1083
TAIBGS85	2240	1280	2080	1280	1550	1268	1410*	1268
TAIGS81	2470	887	2430	887	1740	887	1670*	885
TAIGS85	2900	789	2710	789	1760*	789	1850	787
	job-shop							
	su	mks	su	mks	su	mks	su	mks
T2-PS12	2870	1593	2740	1593	2640	1587	2210*	1585
T2-PS13	2600	1585	2600	1585	2400*	1585	2500	1585
T2-PSS12	2500	1455	2360	1455	2240*	1455	2290	1455
T2-PSS12	2100	1562	1850	1562	1770	1562	1730*	1562

Table 4.11: Instances with alternative of three resources.

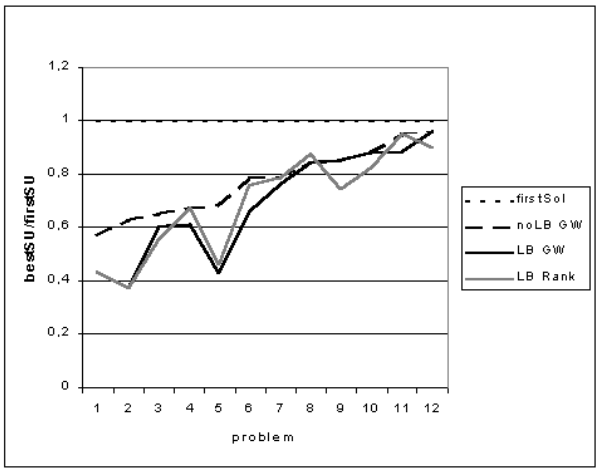

Figure 4.7: Instances without alternative

of the subproblems up to optimality (with the proof of optimality) within the 5 seconds allocated. On the other hand, when the optimization constraint is not used, the percentage of subproblems solved up to optimality quickly drops from an average of 70% for the problems without alternative resources to an average of less than 10% for the problems with alternatives of three resources. Moreover, the average time spent on each window when the optimization constraint is used always remains very small. Figure 10 and 11 compare the performance of solving (and proving optimality) of a setup minimization problem with and without the LB calculation for increasing problem size (in number of activities). Instances with up to 60 activities were easily solved within one minute.

One small remark on the fact that although the aim of the methods proposed is to study the problem including alternative resources, for several known instances without alternative resources, we were able to improve on the best known results.

Finally, some tests were run on small instances without alternative of resources taken form [28]. Open-shop and general-shop problems contain 4 machines and 4 jobs, while job-shop

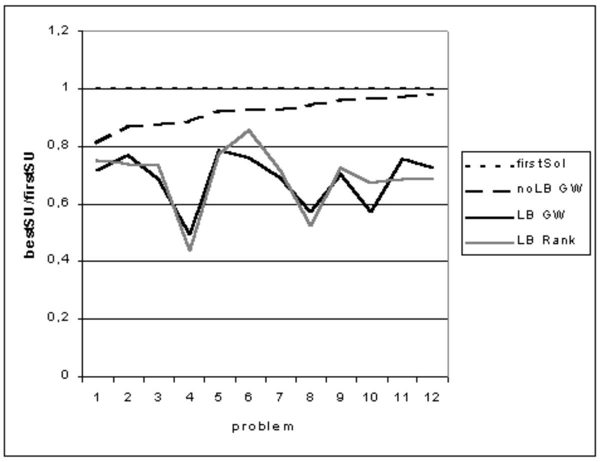

Figure 4.8: Instances with two resources alternative

	win	opt	time	win	opt	time	win	opt	time
	1 instance			2 instances			3 instances		
	open-shop								
noLB	3	50%	2,6	7	21%	4,2	10,7	9%	4,8
LB	3	100%	0,2	7	100%	0,3	10,7	90%	0,9
	general-shop								
noLB	3	83%	1,4	6,2	20%	4,4	9,2	13%	4,7
LB	3	100%	0,3	6,2	84%	1,3	9,2	94%	0,6
	job-shop								
noLB	5,2	80%	1,7	12	8%	4,9	18,3	3%	5
LB	5,2	90%	1,1	12	56%	3	18,3	73%	2,3

Table 4.12: Windows statistics.

contain 5 machine and 10 jobs. For each problem two results are reported in Table 4.13. Column BestMk - BestSu reports the results obtained by optimizing the makespan in a first phase, and the sum of setup times in a second phase where the makespan was limited by the best value found in the first phase. Column BestSu - BestMk reports the results obtained by optimizing the sum of setup times in a first phase, and the makespan in a second phase where the sum of setup times was limited by the value found in the first phase.

In all cases where a computation time is reported, the optimal solution could be proven in both phases. For example, in problem TAIGS05 the optimal makespan is 384, and given such a makespan, the optimal sum of setup times is 300; on the other hand, the optimal sum of setup times also for problem TAIGS05 is 160, and given a limit on the sum of setup times equal to 160, the optimal makespan is 546. Where the time is not reported, optimality could not be proven within 30 minutes.

Two consequences can be taken from these results: the first consequence is that the minimization of only one objective between makespan and sum of setup times may generate poor quality solutions for the other objective. For this reason we think it is necessary, in practice, to consider multi criteria objectives. The second consequence is that the algorithm proposed, for small problems, is able to fix any limit for one objective and find the optimal solution for the other objective. Therefore the method could be used to optimize any combination of makespan and sum of setup times, and to find a set of pareto-optimal solutions.

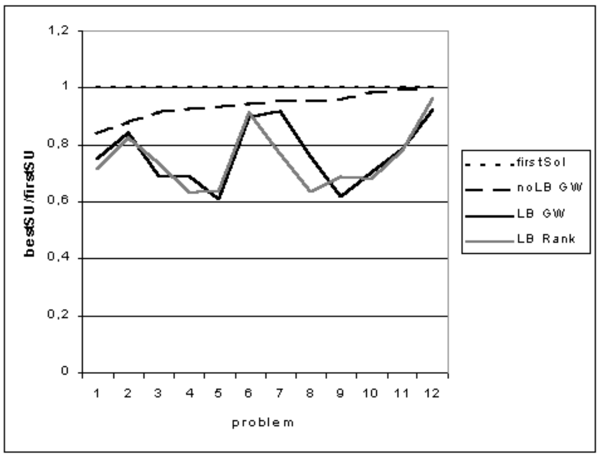

Figure 4.9: Instances with three resources alternative

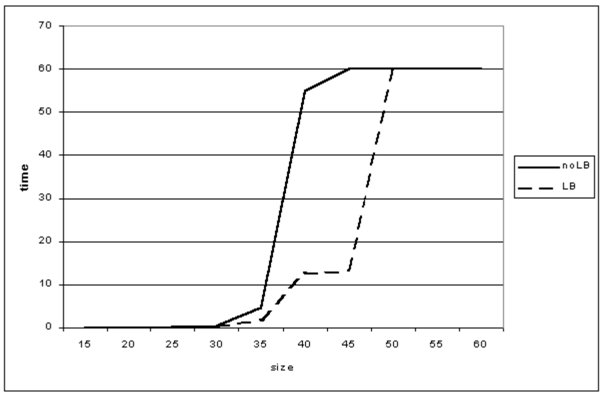

Figure 4.10: Instances without alternative

4.3.7 Conclusion and Future Work

A general scheduling problem with a multi criteria objective function was defined which, to our experience, is of great practical interest. The problem was modeled using a CP approach based on ILOG Solver and Scheduler. A multi-path model was defined to take care of the sequence dependent setup view of the problem. We integrated OR lower bounding techniques and reduced cost fixing in the multi-path constraint in order to effectively prune the search space. A large neighborhood for setup optimization was proposed and we showed that the local optimal solution within the neighborhood can effectively be reached. We generated new problem instances to test the described approach. The computational results show that the cooperation between the scheduling and multi-path model can effectively be used to minimize the sum of setup times while maintaining the makespan constrained to be under a given threshold. Although the aim is to study the problem including alternative resources, for several known instances without alternative resources, we were able to improve on the best

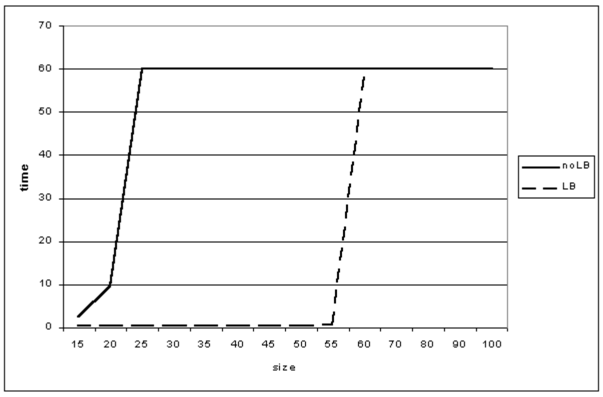

Figure 4.11: Instances with three resources alternative

open-shop						
	BestMk - BestSu			BestSu - BestMk		
	su	mks	time	su	mks	time
TAIBS01	380	306	0.38	320	384	2.58
TAIBS05	440	395	0.22	320	563	0.72
TAIS01	190	249	0.28	160	344	1.93
TAIS05	220	348	0.27	160	523	0.33
general-shop						
	BestMk - BestSu			BestSu - BestMk		
	su	mks	time	su	mks	time
TAIBGS01	280	322	0.16	160	527	0.06
TAIBGS05	280	491	0.33	160	747	0.06
TAIGS01	230	285	0.44	160	362	0.39
TAIGS05	300	384	0.22	160	546	0.16
job-shop						
	BestMk - BestSu			BestSu - BestMk		
	su	mks	time	su	mks	time
T2-PS01	710	798	-	250	2368	-
T2-PS02	630	784	88,42	250	2221	-
T2-PS03	550	749	144,2	250	1932	-
T2-PS04	670	730	388,31	250	1665	-
T2-PS05	710	691	30,43	250	1899	-

Table 4.13: Results on small instances.

known results. We plan to extend our approach in several directions. We are working on the definition of several different neighborhoods, and are experimenting on the combination of them. Moreover, the use of the optimization constraint could also be exploited for generating more sophisticated branching strategies and heuristics.

4.4 Timetabling problem

The timetabling problems considered have been described in [39]. The problems consist in producing a weekly schedule with a set of classes whose duration goes from 1 to 4 hours. Each week is divided in 4-hours time slots (a morning and a afternoon time slot), and each class should be assigned to one time slot. The problem involves disjunctive constraints on classes imposing that two classes cannot overlap and constraints stating that one class cannot spread on two time slots. The objective function to be minimized is the sum of weights taking into account penalties associated to pairs class-slot. More formally, we are given N classes with duration $d_i, i = 1, \ldots, N$, $L = 10$ time slots (2 slots $\times$ 5 days), and a cost matrix $c_{i\ell}$ defining the cost of assigning class i to the time slot ℓ. The problem aims at finding the minimum cost assignment of classes to slots given that the the disjunctive constraints must be satisfied and that each class cannot spread on two time slots.

4.4.1 CP Model

$$
\begin{array}{lrl}
 & \min & z \\
 & \text{on} & \\
 & & classroom = UnaryResource() \\
 & \forall i \in \{1,\ldots,N\}, k \in \{1,\ldots,d_i\} & hour_{ik} \ :: \ [1..4L] \\
 & \forall i \in \{1,\ldots,N\} & class_i = Activity(hour_{i1}, d_i) \\
 & \forall i \in \{1,\ldots,N\} & slot_i \ :: \ [1..L] \\
 & & z \geq 0 \\
 & & cost1 \geq 0 \\
 & & cost2 \geq 0 \\
 & \text{subject to} & \\
(4.11) & \forall i \in \{1,\ldots,N\} & class_i \text{ requires } classroom \\
(4.12) & \forall i \in \{1,\ldots,N\}, \ell \in \{1,\ldots,L\} & hour_{i1} \neq [4\ell - d_i + 2..4\ell] \\
(4.13) & \forall i \in \{1,\ldots,N\} & slot_i = \left\lceil \frac{hour_{i1}}{4} \right\rceil \\
(4.14) & \forall i \in \{1,\ldots,N\}, k \in \{1,\ldots,d_i - 1\} & hour_{i\,k} = hour_{i\,k+1} + 1 \\
(4.15) & & z = \sum_{i=1}^{L} c_{i\,slot_i} \\
(4.16) & & allDiffCost(hour, cost1, c1) \\
(4.17) & & allDiffCost((slot_i)_{i|d_i \geq 3}, cost2, c) \\
(4.18) & & z \geq cost1 \geq cost2 \\
(4.19) & \forall i \in \{1,\ldots,N\} \,|\, d_i \geq 3 & hour_{i1} = 4 \times (slot_i - 1) + 1 \\
(4.20) & \forall i,j \in \{1,\ldots,N\} \,|\, (d_i \leq 2 \wedge d_j \leq 2 \wedge i < j) & slot_i = slot_j \Rightarrow hour_{i1} < hour_{j1}
\end{array}
$$

The classroom is represented with a *UnaryResource* object, while each class is represented with an *Activity* object. Constraints (4.11) enforce the disjunction among classes. For each hour of course of each class, a variable represents the time at which that hour is taught. *hour* is the array of arrays containing all these variables, and constraint (4.14) forces the hours of every class to be consecutive. Each class i has also a variable $slot_i$ representing the slot at which the class is taught. Constraints (4.19) and (4.20) are used to break the symmetries of the problem. In fact, the order in which classes are taught within the same slot is not important, and can be imposed a priori. Within the same slot, classes of 3 and 4 hours are always scheduled before classes of 1 and 2 hours (4.19), while classes of 1 and 2 hours are scheduled within the same slot in lexicographical order.

4.4.2 Relaxations

Two different matching problems representing two relaxations of the timetabling problem have been modeled by two constraints of difference embedding an optimization component. The first one is the linear assignment relaxation arising when classes are considered interruptible involving variables *hour*. The cost of assigning each *hour* variable to a value t is the cost of assigning the corresponding course to the time slot divided by the duration of the class, i.e., $c1_{it} = c_{ik}/d_i, \mid k = \lceil t/4 \rceil$. The second relaxation considers variables *slot* for classes lasting 3 and 4 hours. The corresponding problem is an assignment problem since two 3 or 4 hours courses cannot be assigned to the same slot for limited capacity. The interesting point here is that different problem relaxations coexist and easily interact through shared variables.

4.4.3 Heuristics

The search strategy used assigns first a slot to all classes of 3 and 4 hours (its start time is then fixed by constraint (4.19)). Then it assigns a start time to all remaining classes. The information provided by the optimization constraint have been used for guiding the search. Defining the regret of a variable as the difference between the cost of the best assignment and the cost of the second best, a good heuristic consists in selecting first variables with high regret. In [39] the regret has been heuristically evaluated directly on the cost matrix as the difference between the minimum cost and the second minimum of each row (despite of the fact that these two minimum could not be part of the first best and the second best solutions). The gradient function provided by the optimization constraint gives a more accurate computation of the regret. The regret is then combined in a weighted sum with the size of the domain (following the first-fail principle), and such a weighted sum is used in the variable selection strategy. Concerning the value selection strategy for variable $hour_{i1}$, the value of the optimal solution of the relaxation in the optimization constraint is tried first.

4.4.4 Results

In Table 4.14 we report the result obtained using a pure CP approach, and the result obtained using the two optimization constraints described. The computing times (given in seconds on a Pentium II 200 MHz) and number of fails are reported. The last row of Table 4.14, refers to row 4 of Table 6 of [39], and the corresponding computing times are given in seconds on a Pentium Pro 200 MHz.

Problem	Problem 1		Problem 2		Problem 3	
	Time	Fails	Time	Fails	Time	Fails
Pure-CP	3.77	5.4k	5.50	8.5k	11.20	14.5k
ST2	0.70	199	0.10	30	4.00	1.3k
CL [39]	29.00	3.5k	2.60	234	120.00	17k

Table 4.14: Results on timetabling instances.

4.5 Min-sum tardiness problem

The Min-Sum Tardiness Scheduling Problem considers a set of N activities $act_0, ..., act_{N-1}$ that have to be scheduled on a single unary resource (resource with maximal capacity equal to one) res. Each activity act_i has to be processed on res for p_i time units, has a release date r_i, and a due date d_i. Activities cannot start before their release date, but can end before or after their due dates. If activity act_i ends after its due date d_i a penalty cost $tard_i$ has to be considered, with $tard_i = w_i * (end_i - d_i)$ where w_i is given, and end_i is the end time of activity act_i. If act_i ends before its due date d_i, $tard_i = 0$. The problem consists in finding a feasible schedule which minimizes

$$totTard = \sum_{i=0}^{N-1} tard_i = \sum_{i=0}^{N-1} w_i * max\{(end_i - d_i), 0\}$$

4.5.1 Related Work

A survey on Min-Sum Tardiness Scheduling Problem on single machine can be found in [1]. More recently, the Min-Sum Tardiness Scheduling Problem with release date has been solved by Chu ([43]) where the author proposes lower bounding procedure, dominance properties, branching strategies.

4.5.2 CP Model

$$\begin{array}{lrl}
 & \text{min} & totTard \\
 & \text{on} & \\
 & & res = UnaryResource() \\
 & \forall i \in \{0,\dots,N-1\} & start_i \geq r_i \\
 & \forall i \in \{0,\dots,N-1\} & act_i = Activity(start_i, p_i) \\
 & \forall i \in \{0,\dots,N-1\} & pos_i \ :: \ [0..N-1] \\
 & & totTard \geq 0 \\
 & \forall i,p \in \{0,\dots,N-1\} & start^*_{ip} \geq r_i \\
 & \forall i,p \in \{0,\dots,N-1\} & tard^*_{ip} \geq 0 \\
 & \text{subject to} & \\
(4.21) & \forall i \in \{0,\dots,N-1\} & act_i \text{ requires } res \\
(4.22) & & position(res, pos, start^*)
\end{array}$$

$$\text{(4.23)} \qquad \forall i,p \in \{0,\dots,N-1\} \qquad tard^*_{ip} = \max\{(start^*_{ip} + p_i - d_i), 0\}$$
$$\text{(4.24)} \qquad allDiffCost(pos, totTard, tard^*)$$

An original approach based on Constraint Programming is sketched here. The idea is to associate to each activity act_i a variable pos_i identifying the position the activity can assume in the schedule. For example, if $pos_i = 4$, act_i has 4 activities preceding it, and $N-3$ following it; if $pos_i = [2..5]$, act_i has at least 2, at most 5 activities preceding it. For each activity act_i we also maintain conditional bounds for its start variable when act_i is scheduled all its possible positions: $start^*_{ip}$ is a conditional variable representing the start time of act_i if it is scheduled in position p. $start^*_{ip}$ is a conditional variable in the sense that for such a variable an empty domain does not trigger a failure, but simply means that p is not a feasible position for activity act_i. Several propagation rules can be designed to enforce consistency among the conditional variables of different activities. For example, if the variable $prev_i$ containing all activities that can be scheduled immediately before act_i is available, for each activity act_i, the following propagation rule applies:

$$\texttt{inf}(start^*_{ip}) \geq \min_{j \in prev_i,\ p-1 \in pos_j} \{\texttt{inf}(start^*_{j\,p-1}) + p_j\}$$

A complete description of these propagation rules (enforced by the global constraint (4.22)) is out of the scope of this Section and can be found in Section 6.3. Similarly, conditional variables $tard^*_{ip}$ can be defined as the tardiness of act_i if it is scheduled in every possible position (4.23).

In every feasible solution each activity will be scheduled in a different position. Therefore the objective function $totTard$ can be linked to the position variables by mean of an $allDiffCost$ constraint (4.24).

4.5.3 Relaxations

The $allDiffCost$ constraint calculates a lower bound LB on the total cost, and performs the gradient based propagation from the objective variable $totTard$ to the position variables pos_i. The lower bound LB is clearly calculated using a cost matrix obtained by considering the current lower bound of each conditional variable $tard^*_{ip}$.

The gradient function can also be used to perform propagation from the objective variable $totTard$ to the end variables of the activities. In fact, the upper bound of $tard^*_{ip}$ can be updated as follows:

$$\texttt{sup}(tard^*_{ip}) = \texttt{inf}(tard^*_{ip}) + \texttt{sup}(totTard) - LB - grad(i,p)$$

Then the constraint (4.23) updates $start^*_{ip}$ as follows:

$$\texttt{sup}(start_{ip}) \leq (\texttt{sup}(tard^*_{ip}) + w_i * d_i)/w_i - p_i$$

Finally, constraint (4.22) eventually updates the $start_i$, and end_i variables based on the new bounds of $start_{ip}$.

4.5.4 Heuristics

Two search strategies have been tested: the first one is a standard ranking strategy; the schedule is built in chronological order, selecting at each branch the unscheduled activity with the smallest due date; ties are broken by selecting the activity with the minimal earliest start time. The second branching heuristic uses the newly introduced position variables as decision variables. Once all position variables are instantiated, the sequence of tasks is known, and the corresponding left shifted schedule is optimal. The branching strategy selects the position variable having the largest domain, and splits it two halves. The left branch removes the half of the possible position with highest gradient on average.

In the proposed model the evaluation of the additional cost $grad(i,p)$ to be paid if activity act_i is scheduled in position p is available for every $i, p = 0, \dots, N-1$. The idea is to use this information to try to "exclude" bad solutions. In other word, we want to add a constraint to the model that will heuristically remove bad solutions. Let $MaxGrad$ be the maximum of $grad(i,p)$, i.e., $MaxGrad = \max_{i,p=0,\dots,N-1}\{grad(i,p)\}$ calculated at the root node. Let B_i a set of positions p of activity act_i with "high" value of $grad(i,p)$

$$B_i = \{p | grad(i,p) \geq \alpha MaxGrad\}$$

for some $0 < \alpha < 1$. The constraint HC excluding bad solutions can be defined as follows:

$$pos_i \neq p, \; \forall i = 0, \dots, N-1, \forall p \in B_i$$

Note that this constraint enforces heuristic decisions that could be incorrect, and could even exclude the optimal solution. On the other hand, the search space obtained after the addition of this constraint is a lot smaller than the search space of the original problem P, and the resulting problem $P \uplus HC$ can be quickly solved up to optimality by branch and bound. The optimal solution of $P \uplus HC$ is obviously a feasible solution of P.

4.5.5 Results

The section on computational result gives only an idea of the advantages of the use of optimization constraint for the min sum tardiness problem; in fact, the optimization constraint is based on the information of the $start^*$ pseudo variable; these variables are calculated by the constraint $position(res, pos, start^*)$ that is in prototype version, and results therefore very inefficient. For this reason, although the run time improves the performance of a pure CP approach, this section will only give results in terms of number of fails, and lower bound values, without giving any information about the run time of the method.

We tested min sum tardiness problems on randomly generated problem instances of 10 tasks described in [43]. In Figure 4.12, we report the curves representing the number of fails (in logarithmic scale) of three approaches. The first approach uses the ranking heuristic without the optimization constraint; the second one also uses the ranking heuristic, but with the optimization constraint; while the third one uses the optimization constraint both for propagation and for the heuristic decisions.

On average, the number of fails of the pure CP approach (no LB, Rank) is 60 times more than the approach that uses the optimization constraint (LB Rank); on average, the number of fails of the pure CP approach (no LB, Rank) is 700 times more than the approach that uses the optimization constraint both for filtering and in the heuristic (LB Pos).

$$fails(no\ LB\ Rank)/fails(LB\ Pos) = 700$$
$$fails(no\ LB\ Rank)/fails(LB\ Rank) = 60$$

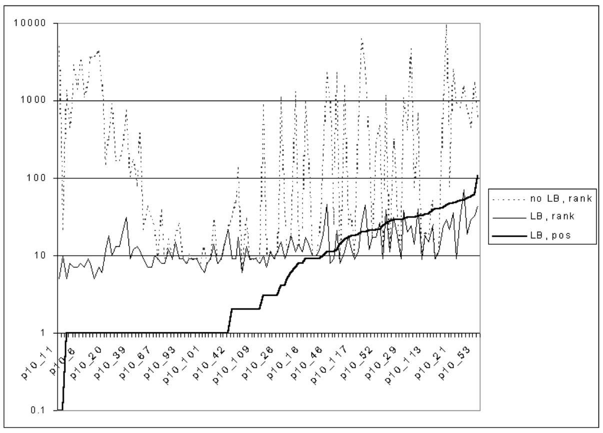

Figure 4.12: Instances from Chu

Since for some instances the optimal solution has value zero ($OPT = 0$), it is therefore impossible to provide a ratio LB/OPT; in general, the measure of the quality of the lower bound is greatly influenced by the absolute value of the optimal solution: when OPT is very small, the relative mean deviation could be very large, while the absolute mean deviation is very small. For this reason we evaluated the quality of the lower bound as follows: we run tests on 120 problem instances described in [43]; on these problem instances the smallest optimal solution has value equal to 0, while the biggest has value equal to 229. The average optimal solution is 48,7. The quality of the lower bound has been calculated after translating, for each instance, the optimal solution to the value $OPT' = 48,7$. The lower bound has been adjusted accordingly: $LB' = LB + OPT' - OPT$. On the average of the 120 instances, the ratio LB'/OPT' results equal to $94,5\%$; on 53 instances the calculated lower bound is equal to the optimal solution, while the minimal ratio LB'/OPT' is equal to 61%.

A final remark concerns dominance relations. Several dominance relations have been found in the literature for the Min Sum Tardiness Problem; as mentioned in Section 1.2 we voluntarily omit dominance criteria in the construction of optimization constraints since their use would lead to constraints that would apply only to a particular problem. In our case instead, it would be very easy to extend this approach for problems with sequence dependent setup times, or any other constraint. In the continuous of this study, we plan to use dominance criteria for this particular problem during the branching strategy.

Chapter 5

Building complex applications

5.1 Motivations

The purpose of this Chapter is to extract some lessons learnt during the development of the constraint based engine of the Production Planning and Detailed Scheduling (PPDS) module of SAP APO. APO (Advanced Planning and Optimization) is the supply chain management solution of SAP. Roughly speaking, the Detailed Scheduling module receives a set of planned orders and decides the execution of such orders in terms of space (which resource will execute which task), and time (when each task will be executed). The execution of all tasks belonging to all planned orders is subject to many constraints (release dates, deadlines, temporal constraints, capacity constraints, etc.) and is aimed at minimizing a weighted sum of several objective functions (makespan, total tardiness, resource assignment cost, sum of setup times/costs, etc.). The scheduling model of PPDS, proposed by SAP, is extremely generic, and it is able to represent scheduling problems arising in many different industries, from discrete to process industries. In a way, it is not designed to solve *one scheduling problem*, but *a large class of scheduling problems*. In this sense, we define *complex applications* those optimization applications aiming at solving a large class of combinatorial optimization problems. The PPDS module solves problems with up to 50,000 activities by decomposing it in a sequence of smaller related subproblems; the optimization engine that will be considered here is tailored to solve problems with up to one thousand activities.

The PPDS constraint based engine is based on ILOG optimization products [96, 95]; see Section 1.5 for some basic notions of such products.

The constraint model and the search procedures used in PPDS will not be described. Instead, we will describe (i) some generic software components that, extensively used in the project, could be applied to many other complex optimization applications, (ii) a framework for dynamically composing several solvers. A special emphasis will be given to the difference between the construction of optimization software dedicated to a specific problem and the construction of applications dedicated to solve a large class of problems whose characteristics are known only at run time.

5.1.1 One problem, one solution

Constraint Programming offers facilities for modeling and solving combinatorial optimization problems.

Concerning the modeling, Constraint Programming tools offer a modeling language to easily describe optimization problems in terms of domain variables and constraints. Global constraints define recurrent abstractions (see e.g., a global disjunctive constraint representing a machine with maximal capacity equal to one); arithmetic and symbolic constraints can be used to represent simple relations or can be combined to define more complex relations among variables; finally, user defined constraints can be implemented to represent problem specific characteristics. Moreover, a tool like ILOG Scheduler offers modeling facilities in terms of complex objects such as *Activity*, *Resource*, solution objects, etc.

Concerning the solving, Constraint Programming tools offer many facilities:

- constraints embed propagation algorithms aimed at removing from the domains of variables those values that are inconsistent with the constraints;
- a tree search mechanism is available and easily programmable; not only is it possible to describe a tree search in terms of sequence of branching decisions, but it is also possible to explore the tree by choosing among different predefined search methods (Depth First, Best Bound First, Discrepancy Search, etc.) or by defining custom tree exploration methods [96, 131];
- more recently, local search mechanisms have been made available. Predefined classes representing LS move operators and neighborhoods provide the building blocks for local search based methods within Constraint Programming [96, 46].

The flexibility of CP based models, together with the facilities offered for writing problem specific search methods, made Constraint Programming a very successful tool for solving real world combinatorial optimization problems.

5.1.2 A large class of problems, one application

One of the key ingredients of the success of many CP based applications is the fact that the optimization developer can easily write sophisticated problem dependent heuristics. When developing an application that deals with a large class of problems, some of the advantages of Constraint Programming (e.g., model flexibility) are even more important; on the other hand it is not possible to count on a single problem dependent heuristic for the success of the application.

In fact, in these cases, the model of the problem is very generic and several objective functions must be considered. The specific characteristics of the problem instance to solve and the objective function (or the set of objective functions) that must be minimized are known only at run time. For example, the PPDS scheduling solver may receive an instance of a Job Shop Problem, as well as an instance of an Asymmetric Hamiltonian Path Problem (modeled as a scheduling problem with sequence dependent setup times), an instance of a Generalized Assignment Problem, etc.

In order to handle the generality of the model, during the development of PPDS, we tried to achieve the following goals:

- we tried to define some principles for a standard way of building optimization applications; we tried to define reusable objects, by coding frequently used optimization techniques and following an object oriented software engineering approach to isolate complexity by means of composability, and flexibility;

- we tried to build a set of solvers able to find good solutions for some classes of problems; we tried to define a way to combine those solvers; we tried to define a way to fine tune the application (changing its behaviour) via a scripting language.

The rest of the chapter describes ideas, data structures, and methodologies that should help to achieve these goals; such ideas, presented with scheduling examples, are indeed quite generic and could easily be applied to a variety of optimization applications.

5.2 Architecture

The architecture of an optimization application, in general, can be structured using three layers (see [59]):

- a *data model* often represented by database tables;
- a *decision model* containing the solving objects, i.e., variables, constraints, solution objects, solvers, heuristics, etc. and representing the *optimization engine*;
- an *object model* representing the bridge between the data model and the decision model, it is responsible for the end-user interface and the data flow.

In this chapter, we will only consider the *decision model*. All issues related to the connections among decision model, object model, and data model will be ignored. The optimization engine of an application is represented by an object of type **Optimizer**. Moreover, in this discussion the **Optimizer** will be considered as a *Batch Engine*, meaning that the interaction between the optimization engine and the user is very limited. The optimizer communicates with the object model for:

- reading the data defining the optimization problem
- reading some initialization control parameters (max time available, settings, etc.)
- reading and delivering some dynamic information (explicit stop from the user, sending back notice of solution found, etc.)
- delivering one or more solutions to the problem

Since the object model is separated from the optimizer, everything concerning connection with Graphical User Interface (GUI), database, or any other client module is ignored by the optimizer. Typically, in a batch engine based on Constraint Programming an optimization problem is solved in three steps:

```
create all variables;
create all constraints;
search for solutions
```

As soon as the complexity of the problem at hand increases, this simple structure shows some limitations. First, it is important to explicitly separate the problem definition (decision variable and constraints) from the objective functions. Second and more important, a single search method will not be enough, and more search methods will have to be used. Third, the search methods could be based on different optimization techniques (local search, global search), tailored to improve a certain subset of objectives, aimed at finding a first solution or at improving an already good solution. Therefore, the implementation of the filtering algorithms of the constraints should depend on the strategy that is currently used. Finally, it is worth defining explicitly the objects used to represent one or more solutions. The previous structure should be rewritten as:

```
create all modeling objects;
  - create all variables;
  - create all constraints;
  - create all the objective functions;
  - create solution objects;
create all solving objects;
search for solutions by combining solvers
```

Note that the new modeling language ILOG Concert pushes CP users in this direction by clearly separating the model from the solving technology. A Concert model defines only the semantic of the problem, the actual operational behavior of the constraints is defined only once a specific solver is instantiated. If the solver is based on Constraint Programming, a filtering algorithm will be chosen. If the solver is based on Linear Programming the set of linear inequalities corresponding to the model constraints will be defined.

The rest of the chapter describes some C++ objects that are useful for the construction of flexible optimization applications. These objects are not "self contained", but are used in conjunction with ILOG Solver and Scheduler. They can be seen as utilities for improving the management of some parts of the development of an optimization application. They concern four different and weakly related topics:

- multiple objective functions;
- tree search;
- large neighborhood search;
- combination of solvers.

5.3 Multi objectives problems

5.3.1 Introduction

Real world problems are typically multi objective. In Scheduling some very common objective functions to be considered are *makespan*, *sum (and max) of tardiness*, *sum (and max) of earliness*, *sum of setup costs*, *resource assignment costs*, *resource utilization*, etc.

Being $z_1, \ldots, z_n$ the n objective functions to consider for a problem P, the following four methods are commonly adopted to compare the quality of different solutions.

Weighted objective

n weights $w_1, \ldots, w_n$ are given, and a global weighted objective function $z = \sum_{k=1}^{n} w_k z_k$ is considered and minimized. Given two solutions s_1, s_2,

$$s_1 \ \mathit{better}\ s_2 \Leftrightarrow \sum_{k=1}^{n} w_k z_k(s_1) < \sum_{k=1}^{n} w_k z_k(s_2)$$

Hierarchical objective

The n criteria are considered hierarchically. Given two solutions s_1, s_2,

$$s_1 \ \mathit{better}\ s_2 \Leftrightarrow \exists k | (z_k(s_1) < z_k(s_2) \wedge z_h(s_1) \leq z_h(s_2) \forall h = 1, \ldots, k-1)$$

Interactive hierarchical objective

The n criteria are considered sequentially, and n different single objective problems are solved. First problem P is considered minimizing z_1. Let z_1^* be the optimal value of the objective function z_1 on problem P. At this point a new problem P_1 is defined as $P_1 := P \uplus (z_1 \leq g_1(z_1^*))$ where g_1 is an arbitrary function given (for example $g_1(z_1^*) = 1.1\ z_1^*$), and z_2 is minimized. Let z_2^* be the optimal value of the objective function z_2 on problem P_1. Again a new problem P_2 is defined as $P_2 := P_1 \uplus (z_2 \leq g_2(z_2^*))$ where g_2 is an arbitrary function given, and so on.

Pareto optimality

The n criteria define the set of pareto-optimal solutions. Given two solutions s_1, s_2,

$$s_1 \ \mathit{better}\ s_2 \Leftrightarrow \exists k | (z_k(s_1) < z_k(s_2) \wedge z_h(s_1) \leq z_h(s_2) \forall h \neq k$$

Bounding Constraint

In general, every solution found imposes a *bounding constraint* $ofCst$ that modifies the bounds of the objective function. This modification depends on the value of the objective function of the solution and should impose z *better* z^*, where the term *better* may have different meanings depending on the type of optimization needed. A particularly important type of *bounding constraint* has the following simple form:

$$ofCst = \bigwedge_{k=1,\ldots,n} (z_k \leq g_k(z_k^*))$$

All the optimization methods mentioned, with the exception of the pareto optimality (when all the pareto-optimal solutions are needed), can be implemented using this type of *bounding constraint*. The bounding constraint can be used in two ways. It can be used during the exploration of a search tree as a *non backtrackable* constraint. In this case, every time a solution is found, the bounds are updated and such updates remain valid for the rest of the tree. It can also be used during an iterative solving procedure to impose limits on the objective functions at each solving iteration.

For example, in the interactive hierarchical optimization n problems are solved sequentially; for each problem, a non backtrackable bounding constraint is used to minimize the current objective function, together with a bounding constraint used to impose limits on all the objectives already optimized.

5.3.2 Dealing with multi objectives

For the type of applications considered here (real world applications characterized by large size problems, and several objectives) quickly finding good solutions is the most important task. Therefore it is often convenient to force large improvements in the quality of the solution in early stages of the optimization process, and leave smaller improvements for later stages. If this is true for single objective problems (consider, for instance, optimization methods based on dichotomic search on the objective function), it is even more important when several objectives must be considered simultaneously. The definitions proposed in the following are meant to help in dealing with multi criteria objectives.

During tree search, when a single objective function z is considered, every time a solution s with objective value $z^* = z(s)$ is found, for the remaining search tree the bounding constraint enforces $z \leq z^* - step$. The step defines the granularity of the objective function and it is often set to 1. The bounding constraint can be generalized as follows: every time a solution s with objective value $z^* = z(s)$ is found, for the remaining search tree the bounding constraint enforces $z \leq g(z^*)$, where g is an arbitrary function. Using this generalization it is, for example, possible to enforce an improvement of the objective function expressed in percentage on the current upper bound or on the difference between the upper bound and the lower bound (e.g. $z \leq 0.9\, z^*$). The second generalization of the standard minimization method considers an array of objective functions instead of a single one. Being $z_1, \ldots, z_n$ the n objective functions to consider, every time a solution s with objective value $z_k^* = z_k(s), k = 1, \ldots, n$ is found, for the remaining tree search the bounding constraint enforces $\bigwedge_{k=1,\ldots,n}(z_k \leq g_k(z_k^*))$, where g_k $(k = 1, \ldots, n)$ is an arbitrary function.

The generalization can be proposed for iterative search, where a sequence of solvers run on problems whose objectives are constrained to improve the previously found solutions. In the first case, the bounding constraint is used as a non-backtrackable constraint, while in this second case it is used as a regular constraint.

5.3.3 Classes

The proposed generalization can be implemented by defining the following data structures:

- A class **Objective** is defined containing the objective variable, the best known lower bound, and the value of the best solution found w.r.t. this objective function.
- The pure virtual class **ObjectiveLimit** is used to encapsulate the function g defining the bounding constraint. Any **ObjectiveLimit** subclass must overload the pure virtual function

  ```
  IlcFloat ObjectiveLimit::getLimitMax(IlcFloat value) const;
  ```

  An **ObjectiveLimit** object can be used to implement the bounding constraint during a branch and bound tree search. Every time a solution is found, the bounding constraint is updated by calling, for every objective z_k, the function `getLimitMax(IlcFloat value)` where `value` is z_k^*. The same **ObjectiveLimit** object can be used to limit the value of one objective function, for example, during an iterative search process.

Some examples of **ObjectiveLimit** are:

```
ObjectiveUBLimit(IlcFloat upperBound)
ObjectiveStepLimit(IlcFloat step)
ObjectiveRelativeLimit(IlcFloat pct)
ObjectiveMinLimit(ObjectiveLimit l1, ObjectiveLimit l2)
ObjectiveMaxLimit(ObjectiveLimit l1, ObjectiveLimit l2)
```

For instance, the class **ObjectiveUBLimit** implements the function $g(value) = upperBound$; the class **ObjectiveRelativeLimit** implements the function $g(value) = (100 - pct) * value/100$, etc.

- The class **ObjectiveBoundingProfile** represents the conjunction of a set of bounding constraints defined for each objective variable. It is indeed a container class that stores for each objective a list of **ObjectiveLimit** objects.

5.3.4 Result

Branch and Bound

Using the described classes it is possible to run a branch and bound procedure that tries to minimize several objectives simultaneously, e.g., using a percentage based step, as showed in the following code.

```
IlcManager m = ...;
Objective makespan = ...;
Objective tardiness = ...;
Objective setupCost = ...;
IlcGoal searchGoal = ...;
ObjectiveBoundingProfile profile(m);
profile.addLimit(makespan, ObjectiveStepLimit(1));
profile.addLimit(makespan, ObjectiveRelativeLimit(5));
profile.addLimit(tardiness, ObjectiveStepLimit(1));
profile.addLimit(tardiness, ObjectiveRelativeLimit(5));
profile.addLimit(setupCost, ObjectiveStepLimit(0));
minimize(m,searchGoal,profile);
```

Every time a solution s is found having objective values m^*, t^*, s^* for makespan, tardines, and setup respectively, the bounding constraint imposes for the remaining part of the tree search the following constraints:

$$\begin{aligned}
&makespan <= m^* - 1\\
&makespan <= (100 - 5)m^*/100\\
&tardines <= t^* - 1\\
&tardines <= (100 - 5)t^*/100\\
&setupCost <= s^*
\end{aligned}$$

The optimal solution of the branch and bound procedure described here guarantees that no solution exists that is *simultaneously* better than 5% m^*, better than 5% t^*, and better than

s^*. Imposing improvement on several objectives simultaneously forces large improvements and it is therefore an interesting minimization method for early stages of the optimization procedure. The drawback is that the generated branch and bound method is incomplete: as soon as one objective cannot be further improved, the minimization ends even if some other objective is very far from the optimum. For example, consider a problem where the weighted objectives $z = z_1 + z_2$ has to be minimized, and suppose the only optimal solution of the problem corresponds to $z_1 = 80, z_2 = 20, z = z_1 + z_2 = 100$. If a branch and bound uses the optimization constraint

$$(z_1 \leq z_1^* - 1) \wedge (z_2 \leq z_2^* - 1)$$

and it gets trapped, for example, in the solution $z_1 = 70, z_2 = 50$, it will never find the optimum.

Iterative Search

Within an iterative search framework, we could write the following code similar to the one used in the Branch and Bound example:

```
IlcManager m = ...;
Objective makespan = ...;
Objective tardiness = ...;
Objective setupCost = ...;
IlcGoal searchGoal = ...;
ObjectiveBoundingProfile profile(m);
ObjectiveBoundingProfile profBBMks(m);
ObjectiveBoundingProfile profBBTard(m);
profileMks.addLimit(makespan, ObjectiveRelativeLimit(5));
profBBTard.addLimit(tardiness, ObjectiveRelativeLimit(5));
profile.addLimit(setupCost, ObjectiveStepLimit(0));
profile.addLimit(tardiness, ObjectiveRelativeLimit(-10));
profile.addLimit(makespan, ObjectiveRelativeLimit(-10));
IlcGoal profileGoal = makeProfileGoal(profile);
IlcGoal goal = IlcAnd(profileGoal,searchGoal);
bool improved = true;
while(improved) {
    bool impMks = minimize(m,goal,profBBMks);
    bool impTard = minimize(m,goal,profBBTard);
    improved = impTard || impTard;
}
```

two branch and bound algorithms run iteratively trying to optimize makespan and tardiness. Each algorithm runs on the original problem with the additional bounding constraint defined by `profileGoal`, which imposes that none of the three objective can decrease. Suppose, for example, that the first run of `minimize(m,goal,profBBMks)` finds a solution with $makespan = m^*, tardiness = t^*, setupCost = s^*$. Then the subsequent algorithm (`minimize(m,goal,profBBTard)`) runs with the bounding constraint imposing

$$(makespan \leq 110m^*/100) \wedge (tardiness \leq 110t^*/100) \wedge (setupCost \leq s^*)$$

As shown in this last example, objects of type **ObjectiveBoundingProfile** are used simultaneously to impose limits on objective functions during an iterative improving method, and to implement a non backtrackable constraint during branch and bound.

5.4 Tree search for scheduling problems

5.4.1 Introduction

A global search algorithm finds solutions by taking decisions and backtracking on failure. The decisions taken in a branch amount to adding a constraint to the problem. CP offers primitives to easily define branching schemes that are problem specific where, at each choice point, the decisions build a small part of the final solution. For instance, in the case of vehicle routing, insertion algorithms consider customers one by one and decide the route that will visit them; for scheduling, ranking algorithms construct the schedule of a machine in a chronological manner by deciding which task should be sequenced first, which second, and so on; for time-tabling, assignment algorithms decide of the duty of a person (or a group of people) for one time-slot. Such global search algorithms are called *constructive search algorithms.* Their states may indeed be interpreted as relevant partial solutions (routing plans for a subset of the customers, short-term schedules, planning only a subset of the tasks, or time-tables for a subset of the people).

In a constructive algorithm, the search is guided by a heuristic: at each choice point, a function h is evaluated for all possible choices, and the choice are ranked by increasing values of h. The choice that minimizes h is considered the preferred decision. Depending on the amount of backtracking allowed, a constructive algorithm ranges from greedy search to complete search. In a greedy constructive algorithm, the preferred branch is systematically followed and no backtracking takes place. When some backtracking is allowed, the construction process could consider a subset of the branches that are "close" to the best branch selected by the heuristic. At each choice point, the heuristic provides a preferred branch, as well as an indication of the quality of the other branches. In case of binary branching schemes, the heuristic may indicate how close both possibilities are, with situations ranging from near ties to definite choices between one good option and a terrible one. In the case of wider (non binary) branching, the heuristic may consider that some branches are serious competitors to the favorite branch while others are not. In any case, one can explore a subset of all solutions by following only those paths in the tree that never consider a poor branch according to the heuristic. Let $b_1, \ldots, b_k$ be the possible choices (branches), b_1 the preferred one ($h(b_1) = \min_i(h(b_i))$) and b_k the worst one ($h(b_k) = \max_i(h(b_i))$). The idea of Restricted Candidate Lists (RCL, see [60], and Section 2.4.2) is to retain only the good branches and to discard the bad ones. More precisely, given a parameter α such that $\alpha \in [0,1]$, only those b_i such that $h(b_i) \leq h(b_1) + \alpha(h(b_k) - h(b_1))$ are kept in a RCL while the others are discarded.

The introduction of RCL thus defines a subtree of the overall search tree. For $\alpha = 0$, this subtree covers all solutions reachable by a greedy algorithm; on the opposite $\alpha = 1$ yields a complete tree covering all possible solutions. For intermediate values of α, the subtree contains only solutions whose construction paths are located "around" greedy paths.

Such a subtree can be explored either systematically or not. In both cases it is important to control the global amount by which a solution path will diverge from the heuristic. It is indeed favorable to generate first solutions that diverge little from the heuristic (following good branches from the RCL) over solutions that systematically diverge from the heuristic (following always the worst branches from the RCL).

The GRASP method ([60]) uses RCL within a randomized version of the greedy algorithm: a randomized version of the overall construction algorithm is run many times. For each construction, at each choice point, one branch is selected at random among the RCL and according to some probability distribution (with decreasing probabilities for $b_1, b_2, \ldots, b_k$). Thus, solutions that are globally closer to the heuristic are generated with an overall higher probability than solutions that are systematically far from the heuristic.

5.4.2 Selector: a generic structure

In most constructive algorithms, it is possible to recognize a common structure independently on the problems they are trying to solve (scheduling, routing, timetabling, etc.), and also independently on the type of algorithm used (Greedy, GRASP, Discrepancy Search, Depth First Search, etc.).

Most algorithms consist in repeatedly selecting a decision object, and taking decisions on that object. Consider the classic variable selection - value selection scheme in CSPs. At each node of the search tree, a variable *var* is selected from a given container (the set of all variables in the problem). A decision is taken consisting on selecting a value v from a value container (the domain of *var*), and taking a decision (for example branching on $var = v\ OR\ var \neq v$).

The same holds, for example, for a ranking strategy in job shop problems. At each node of the search tree, an activity *act* is selected from a given container (the set of all activities in the problem). A decision is taken consisting in ranking *act first* in the required resource or ranking *act not first.*

The interesting point is that the common part of this structure can be extracted into a generic *selector* object. A *selector* iterates on the set of decision objects. Some of these objects are *selectable*, some are not. A *filter* object states whether a given decision object can be selected or not (for example, already instantiated variables cannot be selected for instantiation). The purpose of a selector is, of course, to select one decision object among the set of possible ones. This requires the ability to compare several objects and return one of them following a certain heuristic criterion. Once the heuristic criterion is evaluated, a generic *compare* object is able to compare all objects and return the one minimizing the heuristic criterion. Identifying the set of *selectable* decision objects with a set $I \subset \mathcal{N}$, the heuristic is a function $h : I \rightarrow \Re$. A *compare* object is responsible for comparing the heuristic value $h(i) \in \Re$ of the decsion object i with respect to the heuristic values of other decision objects. Therefore, it only needs to know the value associated to each object, it does not need to know the heuristic itself.

Let **DecisionObject** be the class implementing the decision object, and let **DecObjIterator** be the class implementing a decision object iterator. We suppose that each decision object has a unique integer identifier.

The filter class is implemented using a template class **SelectorFilter<DecisionObject>**; a subclass of **SelectorFilter** must overload the function **bool isValid(DecisionObject)** used to filter out the decision objects that cannot be selected by the selector class.

The class **Compare** will be described in details in Section 5.4.4; for the moment it is enough to say that an object of type **Compare** maintains the current best decision object as its internal state, and is able to return its integer identifier using the function **int getBest()**.

The template class **Selector<DecisionObject,DecObjIterator>** represents a selector. It contains an object of type **SelectorFilter** and an object of type **Compare**. A subclass of

Selector must overload the function **bool select(DecObjIterator, DecisionObject&)**; this function returns true if an object was selected (and it returns the selected object by reference), false otherwise.

5.4.3 A simple example

Consider a job shop scheduling problem, and the following simple ranking heuristic: select the not yet ranked activity having the minimum earliest start time. Branch trying to rank it first or not first. Two classes **ActSelectorFilter** and **ActSelector** can be implemented as follows:

```
bool ActSelectorFilter::isValid(Activity act) {
    if (act.isPossibleFirst()) return true;
    return false;
}

bool ActSelector1::select(ActivityIterator ite, Activity& act) {
    _compareObj.reset();
    for (ite.begin(); ite.ok(); ++ite) {
        Activity currAct = *ite;
        if (! _filter.isValid(currAct)) continue;
        int currEST = currAct.getStartMin();
        int id = currAct.getId();
        _compareObj.compare(currEST,id);
    }
    if (! _compareObj.hasBest()) return false;
    int bestId = _compareObj.getBest();
    act = ite.get(bestId);
    return true;
}
```

For each activity act_i the heuristic h returns its earliest start time ($h(act_i) = startMin(act_i)$). This probably represents the simplest example of selector. It is so simple that it definitely does not justify the introduction of the new classes. Nevertheless, this simple selector can become more and more complex by considering the following extensions:

- instead of comparing objects by considering only one characteristic (the earliest start time), several characteristics should be considered simultaneously (e.g. earliest start time, due date, and setup time/cost);
- the characteristics of the decision objects could be compared hierarchically, by comparing its weighted sum, by finding a set of non dominated decision objects (using ideas from pareto-optimality), etc.
- the selected object should be chosen randomly within a set of more or less equally good choices;
- in order to implement a restricted candidate list algorithm, the selector should also identify a set of more or less equally bad choices that should immediately be discarded to avoid choosing them in backtracking.

- the characteristics of the decision objects to be considered should be passed as a parameter (i.e. we want to be able to change them at run time);

With the exception of the last extension proposed, all the others extensions are used in practice very often. The last one becomes important in the framework presented here where the type of problem and the objectives are known only at run time. In this case, instead of writing several different selectors and dynamically choose one of them, it may be useful to be able to dynamically change the behavior of one selector.

5.4.4 Comparing objects

The extensions proposed require the generalization of the definition of heuristic. Nevertheless, their increased complexity can be nicely encapsulated in the compare object. Identifying the set of *selectable* decision objects with a set $I \subset \mathcal{N}$, the heuristic is a function $h : I \to \Re^k$. A *compare* object is then responsible for comparing the heuristic value $h(i) \in \Re^k$ of the decision object i with respect to the heuristic values of other decision objects. Figure 5.1 represents a 2-dimensional heuristic for a set of decision object labeled from $1, \ldots, 11$. These 11 decision objects will have to be compared based on the 2-dimensional value associated to each of them. Moving from a comparison in a one-dimensional space to a k-dimensional one, it is now clear that (i) the structure of the compare object becomes more complex, (ii) there exist several different ways of comparing sets of points in a k-dimensional space, leading to several possible definitions of compare (sub)classes.

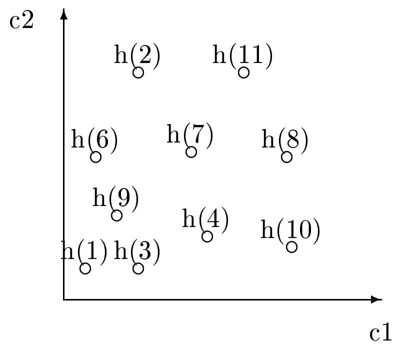

Figure 5.1: 2-dimensional heuristic.

A generalization that allows the use of GRASP and RCL techniques requires the compare object to analyze the set of decision objects and to partition them in three subsets: *good*, *neutral* and *bad* objects. Moreover, the best decision object is maintained.

Borrowing ideas and terminology from multi-criteria optimization, we can define hierarchical compare objects, weighted compare objects, pareto compare objects. Indeed, it is as if the compare object solved a multi-criteria optimization problem where each problem solution is represented by a decision object, and the set of objective values of each solution is represented by the k-tupla of values returned by the application of the heuristic function to each decision object. Clearly the rank of these solutions with respect to the objective functions depends on the type of optimization we are interested in, such as hierarchical optimization, interactive hierarchical, weighted objective function, pareto-optimality, etc.

The class **Compare** is defined as a pure virtual class with the following member functions:

```
void Compare::reset();
void Compare::setDelta(int argNb, float deltaG, float deltaB);
bool Compare::compare(float arg0, ..., float argN, int id) const;
bool Compare::hasBest() const;
int Compare::getBest() const;
int Compare::getBestRandom() const;
IlcIntSet Compare::getSetOfWorst() const;
IlcIntSet Compare::getSetOfBest() const;
```

The **reset()** function is used to reset the internal state of the object; the **compare()** function notifies of a new decision object defined by the integer identifier `id`, and having values `arg0, ..., argN` for the N different characteristics to consider. The function **compare()** updates the internal state of the object recalculating the sets of *good*, *neutral*, and *bad* objects. The compare objects calculate sets of equally good (resp. bad) decision objects by defining an equivalence relation. For example, consider the simple case where a set of activities is compared w.r.t. their earliest start time. Two values Δ^g, Δ^b, could be defined such that any two activities act_1, act_2 are *equally good* if

$$|act_1.getStartMin() - act_2.getStartMin()| \leq \Delta^g$$

and are *equally bad* if

$$|act_1.getStartMin() - act_2.getStartMin()| \leq \Delta^b$$

Clearly, two objects can be both equally good, but not equally bad, or vice versa. Although this seems contradictory, it turns out to be convenient to keep two independent relations since the two sets (of *good*, and *bad* elements) have quite different meaning and usage. The first is used to choose randomly the current best, while the second is used to implement incomplete search such as restricted candidate list methods. The values Δ_i^g, Δ_i^b for each characteristic i are set using the function **setDelta(**i,Δ_i^g,Δ_i^b**)**.

The following Compare objects have been defined:

- **HCompare** Hierarchical compare: N characteristics are compared hierarchically. N values $\Delta_i^g, i = 1, \dots, N$ define the *equally good* relation between two decision objects with respect to each characteristic i. Similarly N values $\Delta_i^b, i = 1, \dots, N$ define the *equally bad*. Given two decision objects o_1, o_2, with characteristics $p_1(o_1), \dots p_N(o_1)$, and $p_1(o_2), \dots p_N(o_2)$ respectively,

$$o_1 \textit{ better } o_2 \Leftrightarrow \exists k | (p_k(o_1) < p_k(o_2) - \Delta_k^g \wedge |p_h(o_1) - p_h(o_2)| \leq \Delta_h^g \forall h = 1, \dots, k-1)$$

$$o_1 \textit{ worse } o_2 \Leftrightarrow \exists k | (p_k(o_1) > p_k(o_2) + \Delta_k^b \wedge |p_h(o_1) - p_h(o_2)| \leq \Delta_h^b \forall h = 1, \dots, k-1)$$

- **WCompare** Weighted compare: N characteristics are compared in a weighted sum. N values $w_i, i = 1, \dots, N$ define the weights of the characteristics; the value $\Delta^g = \sum_{k=1}^{n} w_k \Delta_k^g$ defines the *equally good* relation between two decision objects with respect to the weighted sum of their characteristics; similarly the values $\Delta^b = \sum_{k=1}^{n} w_k \Delta_k^b$ defines the *equally bad* relation. Given two decision objects o_1, o_2, with characteristics $p_1(o_1), \dots p_N(o_1)$, and $p_1(o_2), \dots p_N(o_2)$ respectively,

$$o_1 \; better \; o_2 \Leftrightarrow \sum_{k=1}^{n} w_k p_k(o_1) < \sum_{k=1}^{n} w_k z_k(o_2) - \Delta^g$$

$$o_1 \; worse \; o_2 \Leftrightarrow \sum_{k=1}^{n} w_k p_k(o_1) > \sum_{k=1}^{n} w_k z_k(o_2) - \Delta^b$$

- **PCompare** Pareto compare: N characteristics are compared following ideas from pareto optimality. N values $\Delta_i^g, i = 1, \dots, N$ define the *equally good* relation between two decision objects with respect to each characteristic i; similarly N values $\Delta_i^b, i = 1, \dots, N$ define the *equally bad* relation. Given two decision objects o_1, o_2, with characteristics $p_1(o_1), \dots p_N(o_1)$, and $p_1(o_2), \dots p_N(o_2)$ respectively,

$$o_1 \; better \; o_2 \Leftrightarrow \exists k | (p_k(o_1) < p_k(o_2) - \Delta_k^g \wedge |p_h(o_1) - p_h(o_2)| \leq \Delta_h^g \forall h \neq k)$$

$$o_1 \; worse \; o_2 \Leftrightarrow \exists k | (p_k(o_1) > p_k(o_2) + \Delta_k^b \wedge |p_h(o_1) - p_h(o_2)| \leq \Delta_h^b \forall h \neq k)$$

5.4.5 Defining selectors

The interest in using compare, filter, and selector objects, relies on the fact that the complexity of the code for a selector does not grow much with respect to the simple example showed before. The implementation of a selector considering several characteristics simultaneously, a random selection among a subset of the possible objects, and complex comparative methods is straightforward. Moreover, the compare objects are completely independent from the problem at hand and from the heuristic used in the selector. They are therefore extremely generic and interchangeable. The definition of selectors via composition of generic and interchangeable elements increases the flexibility and the ability to experiment many variant of the heuristic.

The example of activity selector previously proposed is now modified to take into account sequence dependent setup time, and tardiness cost:

```
bool ActSelector2::select(ActivityIterator ite, Activity& act) {
    _compareObj.reset();
    _compareObj.setDelta(1,50,100);
    _compareObj.setDelta(2,10,10);
    _compareObj.setDelta(3,50,100);
    for (ite.begin(); ite.ok(); ++ite) {
        Activity currAct = *ite;
        if (! _filter.isValid(currAct)) continue;
        int currSetupTime = evaluateSetup(currAct);
        int currTardiness = evaluateTardiness(currAct);
        int currEST = currAct.getStartMin();
        int id = currAct.getId();
        _compareObj.compare(currEST,currSetupTime,currTardiness,id);
    }
    if (! _compareObj.hasBest()) return false;
    int bestId = _compareObj.getBestRandom();
    act = ite.get(bestId);
    return true;
}
```

Note that the type of comparison (depending on the implementation class `_compareObj` belongs to) is not known in the function `select`. The object `_compareObj` could actually be passed as a parameter of the `ActSelector2` constructor.

5.4.6 Changing priorities

In this last example, the characteristics taken into consideration are earliest start time, setup time, and tardiness. The compare object considers them in that order. An additional step in the direction of flexibility consists in letting the decision object characteristics and the order in which they have to be considered, to be parameters that could be changed at run time.

In most problems the set of possibly interesting characteristics of decision objects is very small. In scheduling problems **Activity** objects typically play the role of decision objects, and the characteristics that are commonly considered for a given activity are: earliest start time, latest completion time, duration, relative slack (e.g. latest completion time minus earliest start time) due date, setup time, setup cost, resource assignment cost, etc. While selecting the activity on which some decisions will be made, we could imagine to calculate the values of all the characteristics we could possibly need. An object is delegated to decide which subset of characteristics will be used in the current selection call and on which order the characteristics will be considered. In order to obtain such a behavior the class **SelPermutation** is defined. The object `permutation` of type **SelPermutation** reads the characteristics of a decision object and *dispatches* them in the right order depending on the way it was configured. In other words, once it has been configured, the object `permutation` is able to read the characteristics of a decision object in any order and send them to a compare object in the right order.

The class **SelPermutation** is defined with the following member functions:

```
//during configuration
void reset();
void setParInPosition(SelParam par, int pos, float deltaG, float deltaB, bool sign);
void close();
// in the selector
void setValueInPar(float value, SelParam par);
int getNbOfPositionsUsed() const;
float getValueOfPosition(int position) const;
float getDeltaGOfPosition(int position) const;
float getDeltaBOfPosition(int position) const;
bool isParUsed(SelParam par) const;
```

where SelParam is a typedef of int; each characteristic must have a unique identifier of type SelParam.

In the configuration phase the identifier of each characteristic that has to be considered is passed to the setParInPosition function together with the position on which it will be read by the compare object, its deltas, and its sign (true if the characteristic should be minimized, false otherwise).

In order to better clarify the usage of **SelPermutation**, let us consider an example where again the decision objects are of type **Activity** and the characteristics on which the heuristics will base their decisions are earliest start time, tardiness, and setup time. The following constants must be defined having each a unique identifier:

```
extern SelectorParam SP_est;
extern SelectorParam SP_tard;
extern SelectorParam SP_st;
```

The selector, containing a _permutation object of type **SelPermutation**, will look like this:

```
bool ActSelector3::select(ActivityIterator ite, Activity& act) {
    _compareObj.reset();
    for (int i = 0; i < _permutation.getNbOfPositionsUsed(); ++i) {
        float deltaG = _permutation.getDeltaGOfPosition(i);
        float deltaB = _permutation.getDeltaBOfPosition(i);
        _compareObj.setDelta(i,deltaG,deltaB);
    }
    for (ite.begin(); ite.ok(); ++ite) {
        Activity currAct = *ite;
        if (! _filter.isValid(currAct)) continue;
        int id = currAct.getId();
        if (_permutation.isParUsed(SP_est)) {
            int currEST = currAct.getStartMin();
            _permutation.setValueInPar(currEST,SP_est);
        }
        if (_permutation.isParUsed(SP_tard)) {
            int currTardiness = evaluateTardiness(currAct);
            _permutation.setValueInPar(currTardiness,SP_tard);
        }
        if (_permutation.isParUsed(SP_st)) {
            int currSetupTime = evaluateSetup(currAct);
            _permutation.setValueInPar(currSetupTime,SP_st);
        }
        int arg0 = _permutation.getValueOfPosition(0);
        int arg1 = _permutation.getValueOfPosition(1);
        int arg2 = _permutation.getValueOfPosition(2);
        _compareObj.compare(arg0,arg1,arg2,id);
    }
    if (! _compareObj.hasBest()) return false;
    int bestId = _compareObj.getBestRandom();
    act = ite.get(bestId);
    return true;
}
```

Note that the code of this selector does not have explicit knowledge neither of the way the decision objects are compared (encapsulated in the compare object), nor of how many, and which characteristics of the decision objects are considered (encapsulated in the permutation object). The behavior of the selector is therefore completely parameterized and can be changed at any time, even through a text file (using a very simple parser), or through a scripting language.

5.4.7 Using selectors

Suppose we want to build a schedule for a unary resources R subject to sequence dependent setup times. The activities requiring R have due dates, release dates, and deadlines. The objective is to minimize makespan, sum of setup times, and maximal tardiness on the due dates. Using ILOG Solver, and Scheduler, the following search goal can be written:

```
ILCGOAL (Rank, Resource, res, ActivityIterator, acts, ActivitySelector, sel) {
    IlcManager m = getManager();
    Activity selectedAct;
    bool found = sel.select(acts, selectedAct);
    if (! found)
        retrun 0;
    IlcIntSet removeSet = sel.getSetOfWorst();
    for (IlcIntSetIterator ite(removeSet); ite.ok(); +ite) +
        int actId = *ite;
        Activity removeAct = getActivity(actId);
        removeAct.rankNotFirst();
    }
    return IlcAnd(IlcOr(RankFirst(m,selectedAct),
                        RankNotFirst(m,selectedAct)
                        ),
                  Rank(m,res, acts, sel)
                  );
}
```

The schedule is built by ranking all activities. The selector is of type `ActivitySelecor3` previously defined and is passed as a parameter. It returns the tentative activity to be scheduled next and also a set of activities that should not be chosen (even in backtrack) to be ranked first. These activities are committed to be ranked *not first*. The goal can be used as in the following example where the same selector is used a first time with a first set of parameters, and it is reused a second time after changing the parameters (i.e. the behaviour of the selector):

```
IlcManager m = ...;
Resource res = ...;
```

```
ActivityIterator acts = ...;

ObjectiveBoundingProfile profBBMks = ...;
ObjectiveBoundingProfile profBBTard = ...;

SelPermutation permutation();
permutation.reset();
float deltaGEst = 10;
float deltaBEst = 100;
permutation.setParInPosition(SP_est, 0, deltaGEst, deltaBEst, true);
float deltaGTard = 10;
float deltaBTard = 100;
permutation.setParInPosition(SP_tard, 1, deltaGTard, deltaBTard, true);
float deltaGSetup = 50;
float deltaBSetup = 200;
permutation.setParInPosition(SP_st, 2, deltaGSetup, deltaBSetup, true);
permutation.close();

ActSelectorFilter filter;
HCompare compObject();
ActivitySelector3 selector(filter, compObject, permutation);
IlcGoal searchGoal = Rank(m,res, acts, selector);

minimize(m,searchGoal,profBBMks);

permutation.reset();
permutation.setParInPosition(SP_est, 1, deltaGEst, deltaBEst, true);
permutation.setParInPosition(SP_tard, 0, deltaGTard, deltaBTard, true);
permutation.setParInPosition(SP_st, 2, deltaGSetup, deltaBSetup, true);
permutation.close();

minimize(m,searchGoal,profBBTard);
```

The selector considers earliest start time, setup time, and due date of activities. The `compObject` compares activity characteristics hierarchically. In the first minimization step the selector consider as first criterion the earliest start time and tries to minimize the makespan. In the second minimization step the selector consider as first criterion the due date, and tries to minimize the tardiness.

5.5 A CP framework for Large Neighborhoods Search

5.5.1 Introduction

In CP, large neighborhood search often consists in keeping a fragment of the solution s (keeping the value assignment for a subset of the variables), erasing the value of all variables outside that fragment and solving the subproblem defined by the uninstantiated variables (see Section 2.4.1). This technique was introduced in [5] for Job-Shop Scheduling. The job sequence is kept on all machines but one, and the scheduling subproblem on that machine is solved to optimality by branch-and-bound. In this case, the fragment corresponds to the sequencing order on all machines but one. For scheduling the approach was generalized to other fragments (time slice, ranks, sets of ordering decisions etc.) by [37] and [124]. The same approach was also applied to quadratic assignment problems by [114] to vehicle routing by [151]. Such fragment-based LS methods are usually easy to implement once a first CP model has been built. A number of constraint based tools can be used to improve their efficiency :

- an optimization constraint can be set on the neighborhood imposing that only neighbors that strictly improve the objective value over the current solution are generated;
- fragment based neighborhoods can be explored in a variable neighborhood search (i.e., dynamically changing the size of the neighborhood);
- in order to speed up the procedure, each neighborhood defined by a fragment can be explored by an incomplete search.

Large neighborhood search in Constraint Programming offers the following interpretation of local search. Let P be the combinatorial optimization problem we want to solve, x the set of variables of P, and s a current solution with value $z(s)$. A set of constraints $\mathcal{C}(x,s)$ defines a neighborhood structure. A *neighborhood* $\mathcal{N}(s)$ is defined as the set of all feasible solutions in $P \uplus \mathcal{C}(x,s)$ where the symbol $\uplus$ identifies the addition of a set of constraints to a problem. Basically, $\mathcal{C}(x,s)$ are additional constraints imposing that part of the problem P is fixed at values depending on the solution s. Clearly, problem $P \uplus \mathcal{C}(x,s)$ must be easier than the original problem P. Note that the so defined Large Neighborhoods perform the opposite operation of a relaxation. A problem relaxation R of P basically consists in removing some of the constraints of P. A relaxation makes sense if R is easier to solve than P and the optimal solution of R gives a lower bound (for minimization problems) on the optimal solution of P. A neighborhood problem $P \uplus \mathcal{C}(x,s)$, as defined here, consists in adding some constraints to P. A neighborhood problem makes sense if it is easier to solve than P and its optimal solution gives an upper bound (for minimization problems) on the optimal solution of P.

5.5.2 Elementary Neighborhood

Most commonly used large neighborhoods have a simpler structure than the generic $\mathcal{C}(x,s)$. Let $x_1,\dots,x_n$ be the variables of problem P. If every constraint c of a neighborhood structure $\mathcal{C}(x,s)$ can be written as $c = c(x_i,s)$, then we define the large neighborhood as *elementary large neighborhood*. In scheduling problems, neighborhoods based on time window, on resources decomposition, or on ranks are all elementary large neighborhoods. In the following we will show some basic properties and a generic mechanism to create and combine elementary large neighborhoods.

The first property is that an elementary neighborhood structure $\mathcal{C}(x,s)$ can always be written as $\mathcal{C}(x,s) = \bigwedge_{i=1,\dots,n} c_i(x_i,s)$. The second property is that elementary neighborhood structures can be combined to generate other elementary neighborhood structures. In fact, given two elementary neighborhood structures $\mathcal{C}^1(x,s)$ and $\mathcal{C}^2(x,s)$, a new elementary neighborhood structures can be obtained by taking some constraints $c_i^1(x_i,s)$ from $\mathcal{C}^1(x,s)$, and some constraints $c_i^2(x_i,s)$ from $\mathcal{C}^2(x,s)$.

Since in practice many large neighborhoods impose the same constraints to several problem variables, it is useful to partition the set of problem variables into disjoint subsets such that in each subset, the same constraints apply to all the problem variables. An elementary neighborhood structures can therefore be defined as follows:

$$\mathcal{C}(x,s) = \bigwedge_{k=1,\ldots,K} (\bigwedge_{i=1,\ldots,n_k} c_k(x_i^k, s))$$

An elementary neighborhood problem can be defined by a partition of the problem variables and, for each set R_k in the partition, by a constraint (a conjunction of constraints $\bigwedge_{i=1,\ldots,n_k} c_k(x_i^k, s)$) that is applied to all the variables x_i^k, $k = 1, \ldots, K$ in the set.

5.5.3 A template for elementary neighborhood structures

The purpose of this section is to propose a set of C++ classes that can be used to easily build elementary neighborhoods. Designed for scheduling problems, the mechanism described could easily be adapted for any type of combinatorial optimization problem. It is based on two assumptions. First, all the decision variables are part of an activity object **Activity** that is indeed considered as *decision object*. Second, the number of different types of constraints that will be used is limited (i.e., typically for one given activity we may want to fix its start, its end, its resource assignment, its position, etc.). The mechanism is implemented by means of the three virtual classes **Policy**, **LargeNhood**, and **LargeNhoodIterator**. We defined an elementary neighborhood structure as:

$$\mathcal{C}(x,s) = \bigwedge_{k=1,\ldots,K} (\bigwedge_{i=1,\ldots,n_k} c_k(x_i^k, s))$$

using this formulation, $c_k(x_i^k, s)$ represents a constraint applied to a variable x_i^k, and whose definition depends on the current solution s. Typically, $c_k(x_i^k, s)$ represents a conjunction of *elementary constraints*, say $c_k(x_i^k, s) = \bigwedge_{h=1,\ldots,H_k} c_k^h(x_i^k, s)$, where an elementary constraint is defined as a constraint that cannot be further decomposed in conjunction of constraints. The pure virtual class **Policy** is used to define elementary constraints $c_k^h(x_i^k, s)$. The pure virtual function **Policy::restore(Activity, Solution)** defines and applies an elementary constraint. An object of type **Policy** is therefore a template for elementary constraints $c_k^h(x_i^k, s)$. The class **PolicySet** is a set of object of type **Policy** and it represents a conjunction of constraints $\bigwedge_{h=1,\ldots,H_k} c_k^h(x_i^k, s)$ to be applied to all variables $x_i^k, i = 1, \ldots n_k$ belonging to the k^{th} set in the partition.

The pure virtual class **LargeNhood** is a constraint (subclass of **IlcConstraint**) representing an elementary neighborhood structure $\mathcal{C}(x,s)$. The application of **LargeNhood** transforms a problem P into a problem $P \uplus \mathcal{C}(x,s)$. An object of type **LargeNhood** must define a partition of the set of the decision objects and, for each partition subset, must define the constraint (the set of elementary constraints) to be applied to the decision objects of the subset.

An object of type **LargeNhood** contains an array of set of policies **policySetArray**. It also contains a pure virtual function **PolicySet getPolicySet(Activity)** as implicit definition of a partition. In other words, in order to define a neighborhood it is sufficient to define a partition of the decision objects through the function **getPolicySet(Activity)** and, for each partition subset, the set of policies that will be applied to every decision object in the set.

The pure virtual class **LargeNhoodIterator** is simply used to define iterators of objects of type **LargeNhood**.

5.5.4 A time window example

Let P be a scheduling problem whose decision variables are contained in a set of n activities $activity_1, \dots, activity_n$; suppose that the decision variables of each activity $activity_i$ are its start time and resource assignment identified by the variables $start_i$ and res_i. Let s be a solution of P encoded through the values $sStart_i$ and $sRes_i$ representing the values of the variables $start_i$ and res_i in solution s. Consider a time window neighborhood defined by time $time_1$ and $time_2$. All activities ending in solution s before $time_1$ or starting after $time_2$ should be fixed at their start time in s ($sStart_i$) and at their resource assignment ($sRes_i$). All the others should remain unconstrained. The neighborhood defines a partition of the set of activities in two subsets: the subset of activities inside and the one outside the window. For each subset a constraint must be defined. The constraint for the activities outside the window consists in the conjunction of two elementary constraints: the first fixes the start time and the second fixes the resource assignment. The constraint for the activities within the time window does nothing.

This neighborhood could be implemented by defining two **Policy** classes. The class **FixStartPolicy** (subclass of **Policy**) applies constraints of type $start_i = sStart_i$, while the class **FixResPolicy** applies constraints of type $res_i = sRes_i$.

The class **TWNhood** (subclass of **LargeNhood**) has an array *policySetArray* containing two sets of policies: an empty set to be applied to the activities scheduled within $[time_1, time_2]$ in s and a set containing two objects of type **FixStartPolicy** and **FixResPolicy** for the activities scheduled in s outside the window (see Figure 5.2). The function **PolicySet TWNhood::getPolicySet(Activity** *act***)** returns one set of policies or the other depending on the partition *act* belongs to.

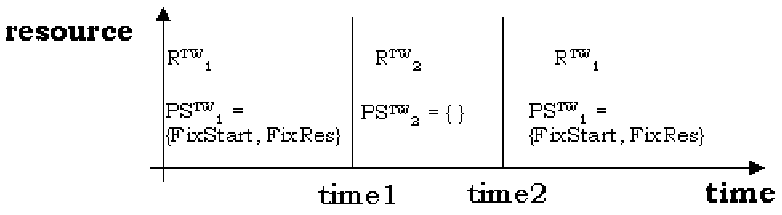

Figure 5.2: Time Window Large Neighborhood.

A gliding window improving method can therefore be implemented by creating a class **GWIterator** (subclass of **LargeNhoodIterator**) that updates the time bounds $time_1$, $time_2$ of an object of type **TWNhood** at each iteration step.

5.5.5 Elementary neighborhood composition

A resource-based neighborhood (say **RESNhood**) can be defined based on a set of resources *resSet*. Activities that were scheduled in a resource belonging to *resSet* are completely free, while all the other activities must keep their start time and resource assignment. Both time window and resource-based neighborhoods partition the set of decision objects in two (see Figure 5.3). A combined time window, resource-based neighborhood partitions set of decision objects in four sets corresponding to the intersection of two partitions having two subsets each. In general, given two neighborhoods LN_1 and LN_2, partitioning the set of decision objects in K_1 and K_2 sets, a combination of the two neighborhoods generates a neighborhood LN_{12} partitioning the set of decision objects in $K_{12} = K_1 \times K_2$ sets. For each set in the generated partition, a new set of policies is to be defined. Typically we desire these set of policies to be dependent on the set of policies of the two original partitions.

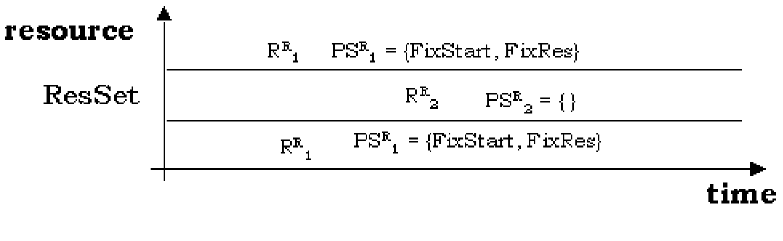
resource
ResSet
R^R_1 PS^R_1 = {FixStart, FixRes}
R^R_2 PS^R_2 = {}
R^R_1 PS^R_1 = {FixStart, FixRes}
time

Figure 5.3: Resource Decomposition Large Neighborhood.

Let R_1^{TW}, R_2^{TW} be the sets of decision objects of the partition defined in **TWNhood**, and PS_1^{TW}, PS_2^{TW} the sets of policies to apply in R_1^{TW}, R_2^{TW} respectively. Let R_1^R, R_2^R be the sets of decision objects of the partition defined in **RESNhood**, and PS_1^R, PS_2^R the sets of policies to apply in R_1^R, R_2^R respectively. A combined neighborhood will have four partitions $R_{11}, R_{12}, R_{21}, R_{22}$, and four corresponding sets of policies $PS_{11}, PS_{12}, PS_{21}, PS_{22}$. Typically, we desire the set of policies of a generated set of decision objects to be dependent on the set of policies of the two original sets. By default they are built as the union of the set of policies corresponding to the original partitions:

$$\begin{array}{ll}
R_{11} = R_1^{TW} \cap R_1^R & PS_{11} = PS_1^{TW} \cup PS_1^R \\
R_{12} = R_1^{TW} \cap R_2^R & PS_{12} = PS_1^{TW} \cup PS_2^R \\
R_{21} = R_2^{TW} \cap R_1^R & PS_{21} = PS_2^{TW} \cup PS_1^R \\
R_{22} = R_2^{TW} \cap R_2^R & PS_{22} = PS_2^{TW} \cup PS_2^R
\end{array}$$

A class **BinaryNhood** (subclass of **LargeNhood**) is defined with the purpose of easily combining neighborhoods. Given two objects LN_1, LN_2 of type **LargeNhood**, a **BinaryNhood** object defines the partition combining LN_1 and LN_2. By default the set of policies for each set in the new partition is generated as the union of the corresponding set of policies of LN_1 and LN_2. The default can be changed either by specifying the set operator that must be used (union, intersection, difference, etc.) or by defining the set of policies explicitly. Moreover a class **BinaryNhoodIterator** (subclass of **LargeNhoodIterator**) is defined taking two **LargeNhoodIterator** objects $LNIte1, LNIte2$.

Suppose, for example, that the two neighborhoods **TWNhood**, and **RESNhood** have been defined; two different combined neighborhoods could be built as follows:

```
TWNhood twNhood(time1,time2);
RESNhood resNhood([res1,res2,res3]);
BinaryNhood combined1(twNhood,resNhood);
BinaryNhood combined2(twNhood,resNhood);
```

```
combined2.setPolicySet(1,2,intersect);
combined2.setPolicySet(2,1,intersect);
```

The set R_1^{TW} of **TWNhood** contains the activities that were scheduled outside the window $[time1, time2]$, while R_2^{TW} contains the ones that were scheduled within the window. The two policies **FixStartPolicy** and **FixResPolicy** are applied to R_1^{TW} (i.e., $PS_1^{TW} = \{FixStartPolicy, FixResPolicy\}$), while no policy is applied to partition R_2^{TW} (i.e., $PS_2^{TW} = \emptyset$). The set R_1^R of **RESNhood** contains the activities that were scheduled outside the set of resources $[res1, res2, res3]$, while R_2^R contains the ones that were scheduled in one of these resources. The two policies **FixStartPolicy** and **FixResPolicy** are applied to R_1^R (i.e., $PS_1^R = \{FixStartPolicy, FixResPolicy\}$), while no policy is applied to R_2^R (i.e., $PS_2^R = \emptyset$).

Both combined neighborhoods generate a partition of four sets (see Figure 5.4). In `combined1` the sets of policies are the union of the sets of policies corresponding to the neighborhoods **TWNhood** and **RESNhood**. Therefore, `combined1` fixes the start and resource assignment of all the activities outside the time window $[time1, time2]$ or outside the set of resources $[res1, res2, res3]$ (see white area on the top Figure 5.4). An activity that was scheduled within $[time1, time2]$, but outside $[res1, res2, res3]$ is therefore fixed in time and resource assignment.

$$\begin{aligned}
PS_{11} &= PS_1^{TW} \cup PS_1^R = \{FixStartPolicy, FixResPolicy\}\\
PS_{12} &= PS_1^{TW} \cup PS_2^R = \{FixStartPolicy, FixResPolicy\}\\
PS_{21} &= PS_2^{TW} \cup PS_1^R = \{FixStartPolicy, FixResPolicy\}\\
PS_{22} &= PS_2^{TW} \cup PS_2^R = \emptyset
\end{aligned}$$

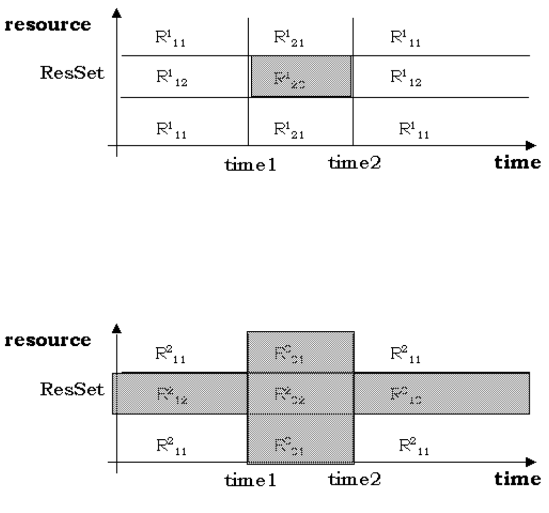

Figure 5.4: Combining Large Neighborhoods.

In `combined2` the four sets of policies are the following:

$$PS_{11} = PS_1^{TW} \cup PS_1^R = \{FixStartPolicy, FixResPolicy\}$$

$$PS_{12} = PS_1^{TW} \cap PS_2^R = \emptyset$$
$$PS_{21} = PS_2^{TW} \cup PS_1^R = \emptyset$$
$$PS_{22} = PS_2^{TW} \cap PS_2^R = \emptyset$$

Therefore, `combined2` fixes the start and resource assignment only of the activities outside the time window $[time1, time2]$ and outside the set of resources $[res1, res2, res3]$ (see white area on the bottom Figure 5.4). An activity that was scheduled within $[time1, time2]$, but outside $[res1, res2, res3]$ is not fixed (see dark area on the bottom Figure 5.4).

5.6 Putting more solver together

5.6.1 Introduction

Constraint Programming tools such as ILOG Solver and Scheduler, have shown to be well suited to write algorithms able to effectively solve combinatorial optimization problems. One important ingredient of the success of CP tools is the possibility to easily write ad-hoc search strategies using the heuristic knowledge of the problem at hand. In order to effectively solve problems whose characteristics and objectives are not known a priori, one must write a library of search strategies, choose a subset of the available strategies, and combine them.

The classes defined hereafter are designed with the purpose of facilitating the task of mixing different solving methods.

5.6.2 Programs and Program operators

The base class is the **Program** class. It only contains two pure virtual functions **bool Program::run()**, and **bool Program::isEnabled()**. In general, the function **isEnabled()** is called within the function **run()** and it represents a precondition for the execution of **run()**. If a given program is not enabled, it is ignored. Objects of type **Program** can be mixed via operators defined as **Program** subclasses: for example, **LoopProgram** is a subclass of **Program** whose constructor takes a **Program** object p and whose function **bool LoopProgram::run()** calls the function `p.run()` until it returns the value true. Similarly, the binary operator (classes) **AndProgram, OrProgram, SequenceProgram** can be defined. The **AndProgram** takes two programs p_1,p_2. It runs p_1, and if p_1 returns true it runs p_2. **AndProgram::run()** returns the logical AND of the results of the runs of p_1 and p_2. The **OrProgram** takes two programs p_1,p_2. It runs p_1, and if p_1 returns false it runs p_2. **OrProgram::run()** returns the logical OR of the results of the run of p_1 and p_2. The **SequenceProgram** takes two programs p_1,p_2. It runs p_1 and p_2 in sequence. **SequenceProgram::run()** returns the logical OR of the results of the runs of p_1 and p_2.

Consider the following example. A problem P has to be solved. Two strategies `fsp_1` and `fsp_2` are designed aimed at finding a first solution for P. Three improvement strategies `lip_1, lip_2`, and `lip_3` are designed to try to improve the current solution, and a final strategy `ppp` is designed to do some post processing on the final solution. A simple way to combine these strategies consists in looking for a first solution, running the three improvement strategies until some better solution can be found, and finally running the post processing strategy. Using the defined operator a strategy `program` can be defined as:

```
first = OrProgram(fsp_1, fsp_2);
```

```
improve = LoopProgram(SequenceProgram(lip_1,SequenceProgram(lip_2, lip_3)));
program = AndProgram(first,SequenceProgram(improve,ppp));
```

The combination of `fsp_1, fsp_2, lip_1, lip_2, lip_3`, and `ppp` is run by calling `program.run()`.

5.6.3 Basic Strategies

The **Program** class and its operator classes offer a very simple mechanism to combine solvers. Nevertheless these classes are not able to completely satisfy the needs for a true flexible mechanism to combine solvers. Three pure virtual classes are defined in order to catch some common characteristics of solvers: **Strategy, AtomicStrategy, CompositeStrategy**. Both **AtomicStrategy** and **CompositeStrategy** inherit from the class **Strategy**, which inherits from the class **Program** so that all the defined program operator can be used on these new classes. Despite of the method used to solve an optimization problem, a solver can be either atomic or composite. An atomic solver is a solver that does not contain other solvers, while a composite solver calls other solvers. Both atomic and composite solvers should maintain information about the number of runs that terminated with a success and with a failure, and the number of last consecutive successes and failures. Moreover the limit condition must be maintained in terms of *time limit*, and of *number of solution limit.* Finally, it is possible to add a hook (of class **StrategyHook**) to a **Strategy** class that is executed at the end of every run. Typically the hook is used to modify the characteristics of a strategy as a function of its outcome (for example, it can be used to disable a strategy that did not produce any improving solution for some consecutive calls).

The class **CompositeStrategy** contains an array of **Strategy** objects. The strategies contained in a composite one can be combined both by using the program operators, and by using C++ code overloading the function **bool CompositeStrategy::run()**.

The class **AtomicStrategy** is used to write solvers aimed at solving problem P. Still, no hypothesis are made on the solving technique used (Constraint Programming, Local Search, Linear Programming, Genetic Algorithms, etc.). It is possible to modify problem P with the introduction of limits on the objective functions (using the classes **ObjectiveLimit** and **ObjectiveBoundingProfile** defined in Section 5.3). Moreover, for each objective function z_i, being $z^*_{i,old}$, $z^*_{i,new}$ the values of z_i in the best solution before and after the execution of the strategy. The following information on the quality of the outcome of the strategy are maintained:

- the improvement Δz^*_i obtained in the last success run; $\Delta z^*_i = z^*_{i,new} - z^*_{i,old}$;
- the best value of the improvement Δz^*_i obtained among all the execution of the strategy;
- the sum of the values of the improvement Δz^*_i obtained in all the execution of the strategy;
- the best value z^{**}_i of the objective function z_i obtained among all the execution of the strategy.

The simple pseudo-code example that follows is meant to show how these new classes help to separate the complexity of a generic optimization engine and to write flexible solvers. Suppose a scheduling problem P has to be solved and that the weighted sum of makespan and tardiness has to be minimized. Let P be described by the object `optimizer` and let `makespanObjective`, `tardinessObjective`, and `weightedObjective` be three objects of type **Objective** defining the three objective functions of P. The two weights for the sum of tardiness and the makespan are set by the user at runtime.

Suppose the following strategies have been defined: `jobShopStgy` is meant to solve job shop problems; `fsStgy` is meant to find a first solution for the generic scheduling problem (no solutions have been found yet); `tardiImprStgy` is tailored to improve the tardiness; `makespanImprStgy` is tailored to improve the makespan; `genericImprStgy` is a generic strategy not tailored for any specific objective or problem structure.

The class **MainStrategy** can be defined as subclass of **CompositeStrategy** and contains all the strategies above specified. The **run()** method of **MainStrategy** could look like this:

```
bool MainStrategy::run()
{
    if (optimizer.isJobShopLike())
        return jobShopStgy.run();
    fsStgy.setNbSolutionLimit(1);
    if (! fsStgy.run())
        return IlcFalse;
// at this point a solution exists
    if (weightOfTardiness == 0) tardiImprStgy.disable();
    tardiImprStgy.setTimeLimit(15);
    tardiImprStgy.addLimit(tardinessObjective,ObjectiveRelativeLimit(5));
    tardiImprStgy.addLimit(makespanObjective,ObjectiveStepLimit(-10));
    tardiImprStgy.addLimit(weightedObjective,ObjectiveStepLimit(1));
    tardiImprStgy.addHook(StopAfterConsecutiveFailures(2));
    if (weightOfMakespan == 0) makespanImprStgy.disable();
    makespanImprStgy.setTimeLimit(15);
    makespanImprStgy.addLimit(makespanObjective,ObjectiveRelativeLimit(5));
    makespanImprStgy.addLimit(tardinessObjective,ObjectiveStepLimit(-10));
    makespanImprStgy.addLimit(weightedObjective,ObjectiveStepLimit(1));
    makespanImprStgy.addHook(StopAfterConsecutiveFailures(2));
    genericImprStgy.setTimeLimit(15);
    genericImprStgy.addLimit(weightedObjective,ObjectiveRelativeLimit(5));
    genericImprStgy.addLimit(tardinessObjective,ObjectiveStepLimit(0));
    genericImprStgy.addLimit(makespanObjective,ObjectiveStepLimit(0));
    genericImprStgy.addHook(StopAfterConsecutiveFailures(2));
    Program tryAll = LoopProgram(SequenceProgram(genericImprStgy,
    SequenceProgram(makespanImprStgy,tardiImprStgy)));
    tryAll.run();
    Strategy rerunStrategy;
    float rMakespanStrat =
        makespanImprStgy.getNbOfSuccess()/ makespanImprStgy.getNbOfRuns();
    float rTardinessStrat =
        tardiImprStgy.getNbOfSuccess()/ tardiImprStgy.getNbOfRuns();
    float rGenericStrat =
        genericImprStgy.getNbOfSuccess()/ genericImprStgy.getNbOfRuns();
    if (rGenericStrat >= rMakespanStrat)
        rerunStrategy = genericImprStgy;
    else if (rMakespanStrat >= rTardinessStrat)
        rerunStrategy = makespanImprStgy;
    else
        rerunStrategy = tardiImprStgy;
    rerunStrategy.unsetTimeLimit();
    rerunStrategy.clearHooks();
    rerunStrategy.run();
}
```

This function first checks if the problem is a job shop. If this is the case, it calls the job shop special purpose strategy, otherwise it looks for a first solution, and then it iteratively calls the three improvement strategies. At first, a time limit of 15 seconds is imposed for every run of every improvement strategy. Moreover, if a strategy does not find an improving solution twice consecutively, it is disabled via the hook **StopAfterConsecutiveFailures**. Each improvement strategy has different limits on the component of the objective function. For example, the strategy tailored for improving the tardiness looks for a solution improving the tardiness of at least 5%. On the other hand, the makespan cannot decrease by more than 10 and the weighted objective function must improve. The sequence of the three strategies is run in a loop until some improvements can be still obtained. At the end the strategy that performed best is rerun without any time limit (i.e. with the time limit imposed on the MainStrategy).

5.6.4 Specialized Strategies

In the definitions given so far no hypothesis was made on the optimization methods used. Since this framework was developed using Constraint Programming, the strategy classes have been specialized to be used with CP optimization methods (ILOG Solver and Scheduler). The abstract class **CPStrategy** is used as root class for two classes implementing CP-based optimization methods: **GSStrategy** and **LSStrategy**, which are used to do global search (tree search) and local search respectively.

The class **CPStrategy** uses the information that the problem is represented through a constraint model and introduces two ideas:

- the constraint model should *define* the problem, but the actual implementation of the propagation algorithms should depend on the solving method used. For instance, if local search is used to solve the problem the implementation of a constraint model should only check for consistency. If global search is used to solve, for example, a makespan minimization scheduling problem, it could be a good idea to implement the constraint model using edge finding propagation algorithms. The **CPStrategy** class offers the possibility to specify an object (a subclass of IlcGoal) with the purpose of specializing the constraint model by setting the right propagation parameters for the strategy;

- instead of solving the original problem P, it is sometimes wise to add some constraints $\mathcal{C}$ to the problem. The additional constraints $\mathcal{C}$ could define a subproblem of P, for instance, if $\mathcal{C}$ is a **LargeNHood** class. Alternatively, $\mathcal{C}$ could implement some restrictions that are expected helping the search of good solutions (see, the application in Section

4.5). In a tardiness minimization scheduling problem, we may decide to impose precedence constraints between pairs of activities where one activity has shorter duration, smallest due date and smallest release date. Although this constraint could remove feasible (and optimal) solutions, we would expect that good solutions will respect such a configuration.

The class **GSStrategy** solves problem P (or $P \uplus \mathcal{C}$) using CP tree search. The following parameters can be specified to describe the type of tree search to be used:

- a tree search goal *searchGoal*;
- a fail limit (maximum number of fails allowed in the tree search);
- an objective profile **ObjectiveBoundingProfile** (see Section 5.3) to be used during the tree search;
- a minimization method (Branch and Bound, Random Restart, Dichotomized Search);
- a tree search exploration method (Limited Discrepancy Search, Depth First Search);

Each type of search may be furthermore adapted to the specific needs by changing some default parameter. For instance, when a Random Restart minimization method is chosen, one can specify a number of fails after which the restart is forced. When a LDS search tree is chosen one can specify the discrepancy parameters. The search goal *searchGoal* could also have a set of parameters (such as specific *Selector* objects, see Section 5.4) applying to the search goal on the particular strategy.

The use of the **GSStrategy** class limits the flexibility of a tree search method coded using ILOG Solver, on the other hand it provides the choice of several standard tree search methods and many parameters can be used to adapt it. If, however, the full flexibility of ILOG Solver is needed, one can directly create a subclass of **CPStrategy** and redefine a proper method `run()`.

The class **LargeNHoodStrategy** is the subclass of **GSStrategy** that is used to solve the sequence of problems P_i (subproblems of P), defined by the application of a sequence $\mathcal{C}_i(x,s)$ of large neighborhoods of type **LargeNHood** (see Section 5.5). An object of type **LargeNHoodStrategy** requires a large neighborhood iterator of type **LargeNhoodIterator** and a starting solution s for P (by default, the best current solution will be taken). In addition, some search limits can be defined to be applied to each single subproblem. The method **LargeNHoodStrategy::run()** iteratively defines a subproblem $P \uplus \mathcal{C}_i(x,s)$ and solves it using tree search.

The class **LSStrategy** is the subclass of **CPStrategy** using the new local search features of ILOG Solver5.0. The actual implementation of this class is subject of future development. Figure 5.5 shows the hierarchy of classes proposed.

5.7 Work in progress

This last section briefly describes some current and future development that heavily rely on the structures defined in the previous sections.

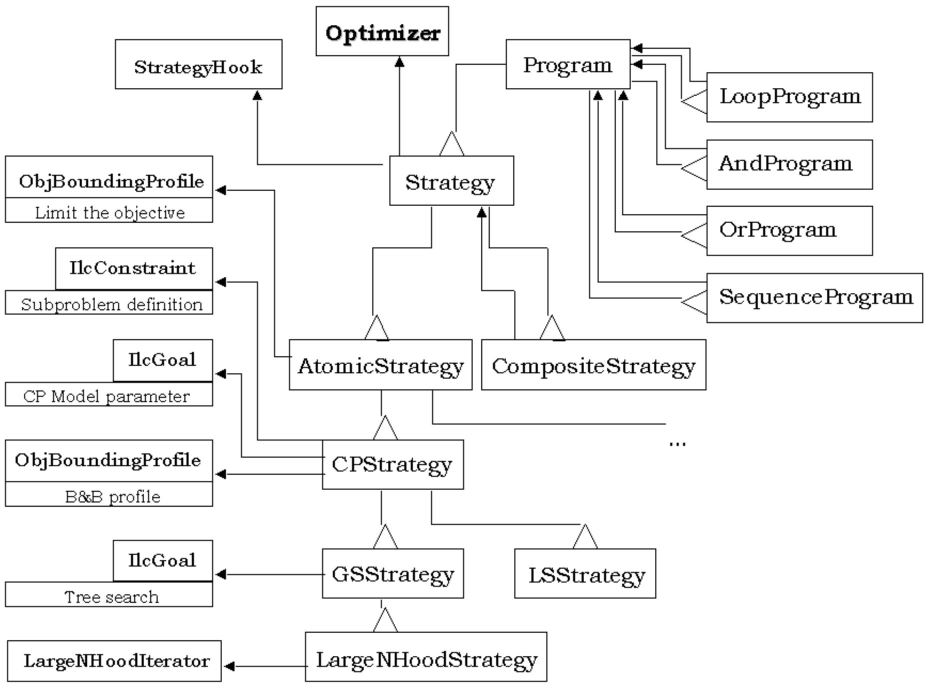

Figure 5.5: Strategy class hierarchy.

5.7.1 A library of strategies

We mentioned that the construction of a library of solvers is necessary for building complex optimization applications where both the problem structure and the objectives are known only at run time. We have shown some structures that are necessary for building such a library. Very little was shown of this library. Even if many strategies already exists, the further development of additional strategies is one of the most important task under way. Moreover, this library should be structured in such a way that strategies can be catalogued following different criteria:

- a strategy may be tailored to find a first solution, look for strong improvements, fine tune good (quasi-optimal) solutions, or prove optimality;
- strategies may tailored to optimize specific subsets of objective functions;
- strategies may be tailored to solve specific types of problem;

We should be able to query for subsets of the set of strategies having some characteristics. For instance, we may want to know which are the strategies that could be used to solve Job Shop type of problems calling for the minimization of tardiness. Within this set we may want to (*i*) find any solution, (*ii*) achieve strong improvements, (*iii*) try to improve further the solutions or to prove optimality (if some time is still available).

5.7.2 Mapping problem instances to strategies

Clearly, the fact that we want to characterize strategies by the type of problems that they are able to solve, means that we want to be able to associate a type to a given problem instance. This may not be an easy task and it requires at least the definition of the concept of problem type and the definition of a function assigning one or more problem types to a problem instance. The easiest approach consists in defining the type by negating groups of constraints in the generic model. For instance, problems without reservoirs (resources representing stocks), problems without max-delay temporal constraints, problems without alternative choice of resources, etc. Each strategy may require some of the constraint types to be missing. A different way of characterizing problems may be based on a documented description of type definitions. Each strategy must declare whether it is able to solve each type of problem or not. A problem instance may be explicitly classified either by the user or by a module able to analyze it.

5.7.3 Scripting language

Once a mechanism to combine strategies is available, we immediately are eager to go a step further and export such a mechanism into a scripting language. Through the scripting language we could be able to redefine the **run()** function of the main strategy, and therefore to change the behavior of the engine via an interpreted language or via simple text file. This would allow a very sophisticated customization of the engine. On the one hand, from a technical point of view, the creation of scripting language is not difficult (using a tool like ILOG Script this would require very little effort). On the other hand, deciding the set of features that will be exported is not an easy task. A scripting language should allow a basic customization, but should prevent from building new atomic strategies. A first approximation of scripting language definition is that everything that we may want to do on the method **run()** of a **CompositeStrategy** should be possible to be done via the scripting language. Tests to identify the good set of features to export is under way.

5.7.4 Auto-tuning

Another way to customize the application for a specific client is based on the hypothesis that for each given client the *type* of optimization problem does not change. The problem instances may change at each run, but all of them have a *quid* in common that defines the type of the problem. If this hypothesis holds, we could imagine to run a very large number of strategies on a limited number of problem instances (training set), and to collect information on the performance of all the run strategies. The analysis of the collected information could identify the subset of strategies (and their parameters) that should be used in production for that client.

Chapter 6

Work in progress

6.1 Removing symmetries through global cuts

In this paper, we propose a general technique for removing symmetries in CSPs. The idea is to record, during the exploration of the search tree, some information concerning nodes whose symmetric counterpart could be removed. This information, called *global cut seed*, can be used in order to generate *global cuts*, i.e., constraints that hold in the remaining part of the search tree. The propagation of global cuts allows to remove symmetric configurations with respect to already visited states. We present a general filtering algorithm for global cuts, correctness and completeness results. The main advantages of the proposed approach are that it is not intrusive in the problem-dependent search strategy and that we can treat symmetries in an additive way since global cut seeds are symmetry independent. Finally, we show that many relevant previous approaches for removing symmetries can be seen as special cases of our filtering algorithm.

6.1.1 Introduction

Constraint Satisfaction Problems (CSPs) occur widely in Artificial Intelligence and are used for solving many real life applications. A CSP is defined on a set of variables taking their values from a finite domain and linked by a set of constraints. Many search and propagation algorithms have been designed for solving CSPs [88]. Propagation algorithms remove combinations of assignments which cannot appear in any consistent solution. In this paper, we focus on symmetric CSPs. A CSP is symmetric when there exists a mapping that transforms a state in another equivalent to the first. Symmetries [137] have been identified as a source of inefficiency since much computational time is spent in visiting equivalent states.

In recent years, symmetry removal methods have interested many researchers; three main approaches have been identified to remove symmetries. The first concerns the imposition of additional constraints to the model of the CSP; in this fashion, the work by Puget [137] defines valid reductions for the original problem obtained by imposing additional constraints, e.g., ordering constraints among variables, that enable to avoid permutations. If a valid reduction of a given CSP is proven to be unsatisfiable, the original CSP is unsatisfiable as well. This approach seems appealing, but presents two drawbacks: first, it is not always simple to find proper symmetry-breaking constraints in case of general symmetries; second, the search strategy can be influenced by the additional constraints. In fact, in many problems, imposing a static order on problem variables contradicts the search strategy suitable for the problem.

The second approach starts from a different idea: constraints able to prune symmetric solutions are introduced during search. In this way, if a given state is encountered, symmetric states can be pruned since they represent the same piece of information. In this setting, [79] and [8] use respectively the notion of conditional constraints and entailment in order to avoid the search in branches leading to symmetric solutions. They add constraints to nodes of the search tree and consider the constraints valid globally in order to remove symmetric configurations upon backtracking. In [148] the authors detect and exploit *intensional permutation* symmetries. Then, they use this information each time they detect a failure, by removing the value causing the failure from the domain of permutable variables. In [115], a similar pruning after failure filtering algorithm is proposed. Similar approaches are based on general notions of interchangeble values [73] and syntactical symmetries [20] which partition variable domains in equivalence classes: when a failure is detected for a given value, all values belonging to the same equivalence class are removed since they would lead to a failure as well.

A third way of coping with symmetries is to define a search strategy that breaks symmetries as soon as possible in the search space. In [115] the authors propose a symmetry breaking strategy that selects first variables involved in the greatest number of local symmetries. Again, the problem dependent search strategy is affected, but this method can be very effective if symmetries are the prevailing feature of the problem.

We focus on the second approach and propose a general, problem-independent way of coping with symmetries. We propose to collect information during search, called global cut seeds, representing states whose symmetric conterpart, if any, should be removed. An interesting property of global cut seeds is that they are independent from the symmetries of the CSP we are solving. Thus, we can treat many symmetries in an addittive way by exploiting the same set of global cut seeds. We then use global cut seeds to generate *global cuts*, i.e., constraints that, deduced at a given node of the search tree, are valid in the remaining part of the tree. The propagation of global cuts enables to avoid exploring branches leading to nodes symmetric to the ones found so far.

We finally show that many relevant previous approaches for removing symmetries can be seen as special cases of our filtering algorithm.

6.1.2 Symmetric CSPs

In this section, we first provide preliminary notions on symmetric CSPs. A CSP is a triple (V, D, C) where V is a set of variables $X_1, X_2, \ldots, X_n$ ranging respectively on finite domains $D(X_1), D(X_2), \ldots, D(X_n)$. A constraint $c_i(X_{i_1}, \ldots X_{i_k})$ defines a subset S_i of the cartesian product of $D(X_{i_1}) \times \ldots \times D(X_{i_k})$, containing those configurations of assignments *allowed* by the constraint. An element τ of $D(X_{i_1}), \ldots, D(X_{i_k})$ is called *tuple* on variables involved in the constraint c_i. A tuple is consistent with the constraint if it belongs to S_i, i.e., it is allowed by the constraint.

CSPs may exhibit symmetries; during search, if two or more states are related by a symmetry, they represent equivalent states. Thus, only one of them could be visited, while the other discarded. This can be very useful in reducing the size of the search tree, and consequently in improving performances. More formally a symmetry is a set of bijective mappings $\{\sigma, \sigma_1, ..., \sigma_n\}$, where $\sigma : V \rightarrow V$ and $\sigma_i : D(X_i) \rightarrow D(\sigma(X_i))$ that preserves constraints. Preserving constraints means that by applying the symmetry to the variables involved in the constraint, if a tuple τ is consistent with the constraints, also $\sigma(\tau)$ is consistent.

Notice that when σ_i is the identity function for every i, then symmetry $\{\sigma, \sigma_1, ..., \sigma_n\}$ is a symmetry on variables. When σ is the identity function, then symmetry $\{\sigma, \sigma_1, ..., \sigma_n\}$ is a symmetry on values. In the general case considered in this paper, $\{\sigma, \sigma_1, ..., \sigma_n\}$ can be a symmetry on variable and values.

6.1.3 Global Cuts

We propose a general method for pruning symmetric configurations in CSPs. It is based on the use of *global cuts*, i.e., constraints deduced during search and that hold in the remaining part of the search tree. Intuitively, cuts use the information contained in cut seeds, which can be collected on each node N of the search tree. In a way, cut seeds are related to *nogoods* [76]. The propagation of global cuts removes configurations which are symmetric w.r.t. the one in N.

Let $V = \{X_1, \ldots, X_n\}$ be the set of variables of a CSP P. A branching strategy BS partitions problem P in a given node N by imposing additional constraints. With no loss of generality, we can suppose that the branching strategy imposes on the left branch the *positive* constraint c, and on the right branch the *negative* constraint $\neg c$. All constraints imposed from the root node to a node N are called *branching constraints* $BC(N)$.

A node N is described by a triple (DS_{old}, DS_{new}, BC), where DS_{old} is a set of domains prior to propagation and DS_{new} is the set of domains (possibly) shrunk thanks to constraint propagation, BC is the set of branching constraints imposed to reach the node. If we call $f(N)$ the node father of N, DS_{old} of N corresponds to DS_{new} in $f(N)$. At root node, DS_{old} corresponds to the set of initial domains. If one domain in D_{new} is empty, a failure is detected and a backtrack forced. A node can be: (i) a solution when all variables have D_{new} containing exactly one value; (ii) a failure when at least one D_{new} is empty; (iii) an intermediate node when all domains D_{new} contain at least one value and at least one domain contains more than one value.

Intuitively, each time a node is visited, we can collect a data structure independent from the problem symmetries, called *global cut seed*, needed for generating *global cuts*. Clearly, the more information we collect at each node, the better pruning we can achieve. However, there should be a tradeoff between the information stored and the pruning achieved.

Definition 1: Global Cut Seed: Given a CSP P with n variables, a global cut seed is a n-tuple $\Delta = \{\delta_1, \delta_2, ..., \delta_n\}$ where each δ_i is a set of values, and such that each n-tuple $\{v_1, \ldots, v_n\}$, $v_1 \in \delta_1, ..., v_n \in \delta_n$ is either an already found solution or an infeasible configuration for P.

Proposition 1: Cut Seed Group Given two Global Cut Seeds $\Delta_1 = \delta_{11}, \ldots, \delta_{1n}$ and $\Delta_2 = \delta_{21}, \ldots, \delta_{2n}$ with $n-1$ sets in common, i.e., for all $i = 1..n, i \neq j$ $\delta_{1i} = \delta_{2i} = \delta_i$ and $\delta_{1j} \neq \delta_{2j}$, $\Delta_{(1+2)} = \delta_1, \ldots, \delta_{1j} \cup \delta_{2j}, \ldots, \delta_n$ is also a Global Cut Seed. The new Global Cut Seed subsumes both the previous ones.
Proof: Each tuple belonging to the newly generated Global Cut Seed is part of a cut seed and is thus part of a solution or of an infeasible tuple for problem P.

Definition 2: Symmetry Removal Cuts: Given a CSP on variables $V = \{X_1, ..., X_n\}$, a global cut seed $\Delta = \{\delta_1, ..., \delta_n\}$ and a symmetry set Σ, a Symmetry Removal Cut is a constraint that holds in the remaining part of the search tree and has the form: $remove\_symmetric(\Delta, V, \Sigma)$. The constraint imposes that configuration symmetric to Δ with respect to Σ are not explored in the remaining search tree.

6.1.4 Filtering Algorithm for Pruning Symmetries

In this section, we define a filtering algorithm exploiting global cuts. Given a global cut seed $\Delta = \{\delta_1, \dots, \delta_n\}$, we have to remove from variable domains all configurations of assignments representing symmetric counterparts with respect to the values contained in the cut seed. For this purpose, we transform it by applying the symmetry to each value belonging to each δ_i[1]. Let $i = idx(X_i)$ be the index of variable X_i. At each node N of the search tree, if there exists a subset M_j of variables X_i whose cardinality is $n-1$ such that for each couple $(X_i, \delta_{idx(\sigma(X_i))})$, $i = [1..n], i \neq j$, $D_{new}(X_i) \subseteq \sigma_i(\delta_{idx(\sigma(X_i))})$, we can perform a pruning on the domain of the free variable X_j. All values $v_k \in \sigma_j(\delta_{idx(\sigma(X_j))})$ can, in fact, be removed from $D_{new}(X_j)$. Note that if the subset of variables for which the condition holds has cardinality equal to n, the search fails and backtracks.

As a simple example, consider a problem where we have three variables subject to a constraint of difference [142], i.e., they should be assigned to different values. Their initial domain is $D(X_1) = D(X_2) = D(X_3) = I = [1,2,3,4]$ and they are subject to permutation symmetries. Now suppose we find the first feasible solution $S1 = \{X_1 = 1, X_2 = 2, X_3 = 3\}$, which represents a global cut seed by definition. Proceeding depth first, we find the second solution $S2 = \{X_1 = 1, X_2 = 2, X_3 = 4\}$. We can group the first two cut seeds in the global cut seed $\Delta = \{\{1\},\{2\},\{3,4\}\}$. Consider the symmetry σ which maps X_1 in itself, X_2 in X_3 and vice-versa. Now, upon backtracking, we find a node where variable $X_1 = 1$, $X_2 = 3$ and the domain of variable X_3 contains values $\{2,4\}$. Thus, we can find a matching of size $n-1=2$ which maps X_1 in $\delta_1 = \{1\}$ since $D_{new}(X_1) \subseteq \delta_1$ and X_2 in $\delta_3 = \{3,4\}$, since $D_{new}(X_2) \subseteq \delta_3$. The domain of the free variable X_3 (for which $D_{new}(X_3) \nsubseteq \delta_2$) can be pruned by removing all values belonging to $\delta_2 = \{2\}$. In this way we have removed the symmetrical solution $S3 = \{X_1 = 1, X_2 = 3, X_3 = 2\}$.

Correctness and completeness

In this section, we prove that, given the constraint $remove\_symmetric(\Delta, V, \Sigma)$, the filtering algorithm proposed is both correct and complete with respect to the symmetry elimination task.

Proposition 2: The filtering algorithm is sound, i.e., it does not remove any configuration which is not symmetric with respect to a previously found configuration in $\Delta = \{\delta_1, \dots\delta_n\}$.
Proof sketched Suppose we remove a configuration N which is not a symmetric counterpart of a node already visited. The triggering condition of the filtering algorithm is that we have a set of $n-1$ variables whose domain $D_{new}(X_i)$ $i = 1..n, i \neq m$ is contained in $\sigma_i(\delta_{idx(\sigma(X_i))})$ where $\sigma \in \Sigma$. If the node we are removing is not symmetrical with respect to an already found configuration, it means that there exists for the free variable X_m a value we are removing $v_{j_m} \in D_{new}(X_m)$ that does not belong to $\sigma_m(\delta_{idx(\sigma(X_m))})$. But, by construction of the filtering algorithm, we remove only values belonging $\sigma_m(\delta_{idx(\sigma(X_m))})$. This fact contradicts the hypothesis.

Proposition 3: The filtering algorithm is complete, i.e., it removes all configurations symmetric with respect to a previously found configuration.

[1]The application of a symmetry to a set $\delta_i = \{v_1, \dots, v_n\}$ is a set where each element is computed as the result of the application of the symmetry to each value belonging to the original set, i.e., $\sigma_i(\delta_i) = \{\sigma_i(v_1), \dots, \sigma_i(v_n)\}$.

Proof sketched Suppose we do not remove a node N which is a symmetric counterpart of a node already visited whose information (node configuration) is contained in Δ, i.e., there exists a n-size mapping from the symmetric node to the global cut seed $\delta_1, ..., \delta_n$. When at least $n-1$ variables are ground, we have a mapping of size $n-1$ which maps $D_{new}(X_i)$ in $\sigma_i(\delta_{idx(\sigma(X_i))})$ for $i = 1..n$ $i \neq m$. Suppose there exists a value v_m in the domain of X_m which is symmetric to a previously visited configuration. Thus, v_m is contained in $\sigma_m(\delta_{idx(\sigma(X_m))})$ and thus removed by the filtering algorithm.

6.1.5 Node Configurations

We will now show how Global Cut Seeds can be collected during search. At each node, we collect data structures, called Node Configurations representing the node. A node can be a solution, a dead-end, or an intermediate node where propagation is performed. We show that in every case, the information collected is a global cut seed. However, in practice, it is not convenient to store and analyze for pruning too much data. Thus, only some nodes (say, solutions and dead-ends) could be stored. We will also show that in case of depth first search, the information can be effectively *compressed* and efficiently stored.

We will show that Node Configurations represent a set of global cut seeds if the branching constraints are always unary constraints, i.e., remove a set of values from the domain of a variable. More formally, let $c^{(N)}$ and $\neg c^{(N)}$ be the branching constraints at node N. Then $c^{(N)}$ (resp. $\neg c^{(N)}$) defines a subset of the domain of a single variable X_k (branching variable), i.e., $c^{(N)} = c^{(N)}(X_k) = (I_k \setminus S_k)$, being I_k the initial domain of X_k, and $S_k \subseteq I_k$. When $c^{(N)}$ (resp. $\neg c^{(N)}$) is posted on the branching variable X_k, all values S_k (resp. $(I_k \setminus S_k)$) are removed from the domain of X_k

Definition 3: Node configurations Let $V = \{X_1, \dots, X_n\}$ be the set of variables of a CSP. Let $BC(f(N))$ be the set of branching constraints used to reach the node father of node N from the root node. All constraints in $BC(f(N))$ should be unary constraints for which propagation is complete, i.e. all incompatible are removed by the branching. Let c_k be the unary constraint on variable X_k generating node N from $f(N)$, let S_k be the set of values incompatible with c_k for variable X_k. We refer to X_k as *branching variable.*

Each node N of the search tree generates a set of at most n Node Configurations $\Delta_1^{(N)}, \dots, \Delta_n^{(N)}$, one for each variable X_i having $D_{new}(X_i) \neq D_{old}(X_i)$.

The configuration associated to X_i is a n-tuple $\Delta_i^{(N)} = \{\delta_{i1}^{(N)}, \dots, \delta_{in}^{(N)}\}$ where each element $\delta_{ij}^{(N)}$ is a non-vacuous subset of the domain $D_{old}(X_j)$. In particular, $\delta_{ij}^{(N)} = D_{old}(X_j)$ for $j \neq i, j \neq k$, $\delta_{ii}^{(N)} = D_{old}(X_i) \setminus D_{new}(X_i)$, $\delta_{ik}^{(N)} = D_{old}(X_k) \setminus S_k$.

The configuration associated to the branching variable X_k is a n-tuple $\Delta_k^{(N)} = \{\delta_{k1}^{(N)}, \dots, \delta_{kn}^{(N)}\}$ where each element $\delta_{kj}^{(N)}$ is a non-vacuous subset of the domain $D_{old}(X_j)$. In particular, $\delta_{kj}^{(N)} = D_{old}(X_j)$ for $j \neq k$, $\delta_{kk}^{(N)} = D_{old}(X_k) \setminus S_k \setminus D_{new}(X_k)$.

If a node represents a dead-end, all the configuration nodes are equal, $\Delta^{(N)} = \{\delta_1^{(N)}, \dots, \delta_n^{(N)}\}$ where $\delta_j^{(N)} = D_{old}(X_j)$ for $j \neq k$, $\delta_k^{(N)} = D_{old}(X_k) \setminus S_k$.

Notice that since in Constraint Programming, after a solution is found, a fail is triggered in order to continue the search, a node configuration could also contain already found solutions, and all already found solutions are contained in some Node Configurations.

Proposition 4: Given a node N if the node represents a solution, the corresponding node configuration is a global cut seed.
Proof: By definition of global cut seed.

Proposition 5: Given a node N if the node represents a failure or an intermediate node, every $\Delta_j^{(N)}$ associated to the node represents a global cut seed if the branching strategy partitions the search space through unary constraints on which the propagation is complete.
Proof: Let consider node $f(N)$, father of node N, and node N; let $D_{old}(X_1)$, ..., $D_{old}(X_n)$ be the domains prior to the application of branching constraint c_k at node $f(N)$. Every possible tuple $v_1, \ldots, v_n$, such that $v_i \in D_{old}(X_i)$, is consistent with respect to $BC(f(N))$ because of the hypothesis on the branching strategy. Let S_k be the set of values inconsistent with constraint c_k, $S_k \subset D_{old}(X_k)$. Then, every possible tuple $v_1, \ldots, v_n$ such that $v_i \in D_{old}(X_i), i \neq k, v_k \in D_{old}(X_k) \setminus S_k$ is consistent with respect to $BC(N)$ since it is consistent with $BC(f(N))$ and the last constraint imposed removes S_k.

Therefore, by definition of configuration node, every tuple $cn_1, \ldots, cn_n$ belonging to a configuration node $\Delta_i^{(N)} = \{\delta_{i1}^{(N)}, \ldots, \delta_{in}^{(N)}\}$ is consistent with respect to $BC(N)$. On the other hand, tuple $cn_1, \ldots, cn_n$ represents a failure for $P \cup BC(N)$. In fact, by definition, $\delta_{ii}^{(N)}$ contains only values that are inconsistent with every value belonging to $\delta_{ij}^{(N)} \forall j \neq i$. Finally, since $cn_1, \ldots, cn_n$ is consistent w.r.t. $BC(N)$, and inconsistent w.r.t. $P \cup BC(N)$, it is therefore inconsistent w.r.t. the original problem P.

Starting from the general definition of Global Cut Seed, with the additional hypothesis of unary branching constraints, we have defined a Node Configuration, a data structure that could, in practice, be collected during search and used for pruning. Nevertheless, at each node a very large number of Node Configurations could be created; generating all of them would not be practical, choosing only a subset of them or merging them could be difficult.

Let now introduce another restrictive (yet commonly satisfied) hypothesis: let suppose the search space is explored in Depth First Search. Under this condition, we can introduce a more powerful Global Cut Seed that can be used in practice to develop efficient and effective cuts. Two are the main ideas behind this new Global Cut Seed: the first one (which does not use the additional hypothesis of Depth First Search) is to distinguish between value removal due to propagation, and value removal due to branching constraints. When a value is removed from a domain because of propagation, the value is inconsistent with the current configuration of variable domains; when a value is removed from a domain because of a branching constraint, the value is not necessary inconsistent. The second idea (which uses the additional hypothesis of Depth First Search) makes use of the knowledge that whenever a right branch is selected (a negative branching constraint is used), the correspondent left branch has already been explored and has lead to a failure[2]. Some other strategies, e.g., Least Discrepancy Search, satisfy this requirement.

Definition 4: Extended Node Configuration Let $V = \{X_1, \ldots, X_n\}$ be the set of variables of a CSP. Let N be a node where a failure is detected. Let $RP^{(N)} = \{RP_1^{(N)}, \ldots, RP_n^{(N)}\}$ be the set containing, for each $RP_i^{(N)}$, the set of values removed by the application of all positive branching constraints (i.e. the left branches constraints) of $BC(N)$ on variable X_i from

[2]In this setting, also solutions are considered since a failure is forced to find other solutions upon backtracking.

the root node to the current node[3]. Then, at node N, an Extended Node Configuration $\Delta^{(N)}$ can be defined as follows: $\Delta^{(N)} = \{(I_1 \setminus RP_1^{(N)}), \ldots, (I_n \setminus RP_n^{(N)})\}$ where I_i is the initial domain of each variable.

Proposition 6: An Extended Node Configuration is a Global Cut Seed.
Proof: Let $S_k^{(N)}$ be the set of values removed by the positive branching constraint c_k applied on the branching variable X_k to go from node $f(N)$ to the failure node N. By definition, $\Delta1^{(N)} = \{\delta1_1^{(N)}, \ldots, \delta1_n^{(N)}\}$ where $\delta1_i^{(N)} = D_{old}(X_i), i \neq k$, $\delta1_k^{(N)} = D_{old}(X_k) \setminus S_k^{(N)}$, is the node configuration of the dead-end node N, and therefore it is a Global Cut Seed.

Since all the propositions and theorems of section 6.1.3 do not depend on the particular filtering power of the constraints, we can suppose that all constraints defining the problem are implemented in such a way that no pruning is performed until all variables are bound (in a *generate and test* way)[4]. Then for each variable X_i the domain, at node N, $D_{old}(X_i)$ is equal to $(I_i \setminus R_i^{(f(N))})$, being $R_i^{(f(N))}$ the set of values removed by all branching constraints (positive or negative) from root node to node $f(N)$. Therefore, $\Delta1^{(N)} = \{\delta1_1^{(N)}, \ldots, \delta1_n^{(N)}\}$ where $\delta1_i^{(N)} = (I_i \setminus R_i^{(f(N))}), i \neq k$, $\delta1_k^{(N)} = (I_k \setminus R_k^{(f(N))} \setminus S_k^{(N)})$ is a Global Cut Seed. Notice that, up to now, the hypothesis on Depth First Search was not used, and therefore the sufficient hypothesis for $\Delta1^{(N)}$ to be a Global Cut Seed is that branching constraints are unary constraints.

To prove that $\Delta^{(N)} = \{(I_1 \setminus RP_1^{(N)}), \ldots, (I_n \setminus RP_n^{(N)})\}$ is a Global Cut Seed we use the result that $\Delta1^{(N)}$ is a Global Cut Seed, and the hypothesis of Depth First Search.

We consider the path for reaching a dead-end node N starting from the root node. We have imposed a set of positive branching constraints and a set of negative branching constraints. The hypothesys on the search strategy ensures that each time we choose a negative branching constraint in the path, the branch corresponding to the positive one has already been explored, lead to a failure and has generated a cut seed. We call $N_1, \ldots, N_k$ to identify the dead-end nodes opposite to the k negative branching constraints used to reach N from the root node.

$\Delta1^{(N)}$ is a Global Cut Seed generated at node N, and $\Delta1^{(N_1)}, \ldots, \Delta1^{(N_k)}$ are Global Cut Seeds generated at nodes $N_1, \ldots, N_k$ where $f(N_1), \ldots, f(N_k)$ are the nodes where the negative constraints were used to reach N. The general idea is that, by merging $\Delta1^{(N)}$ with $\Delta1^{(N_1)}, \ldots, \Delta1^{(N_k)}$ we can create a new Global Cut Seed where only the values removed by positive branching constraints are considered.

In fact, let $C(S_j^{(N_i)})$ be the set of values removed by the negative constraint on node $f(N_i)$ acting on variable X_j ($C(S_j) = I_j \setminus S_j$). By considering $\Delta1^{(N_i)}$ and $\Delta1^{(N)}$, it is easy to see that for each $l, l = 1, \ldots, n, l \neq j$ $\delta_l^{(N)} \subseteq \delta_l^{(N_i)}$. Then, by using Proposition 1, $\Delta1^{(N_i)}$ and $\Delta1^{(N)}$ can be merged and the union of $\delta_j^{(N_i)}$ and $\delta_j^{(N)}$ can be considered.

$\delta_j^{(N_i)} = (I_j \setminus R_j^{(f(N_i))} \setminus S_j^{(N_i)})$, $\delta_j^{(N)} = (I_j \setminus C(S_j^{(N_i)}) \setminus R1)$ where $R1 = R_j^{(f(N_k))} \setminus C(S_j^{(N_i)})$; their merge leads to $(I_j \setminus R1)$. Therefore by merging $\delta_j^{(N)}$ and $\delta_j^{(N_i)}$ we could remove the dependence of $\Delta1^{(N)}$ from the negative branching constraint acting on node $f(N_i)$. By removing, one after the other, all dependencies of $\Delta1^{(N)}$ from the negative branching constraints acting on $f(N_k), \ldots, f(N_1)$, $\Delta1^{(N)}$ can be transformed into $\Delta^{(N)} = \{(I_1 \setminus$

[3]If a variable X_j has not been involved in a branching constraints, the corresponding $RP_j^{(N)}$ is empty.

[4]Basically we suppose the constraints defining problem P are implemented in *generate and test*, while the branching constraints BC(N) partition the domain of the branching variable.

$RP_1^{(f(N_k))}), \ldots, (I_i \setminus RP_i^{(f(N_k))} \setminus S_i^{(N_k)}), (I_{i+1} \setminus RP_{i+1}^{(f(N_k))})\}, \ldots, (I_n \setminus RP_n^{(f(N_k))})\}$. Finally, since for each index j corresponding to a non branching variable $RP_j^{(f(N))} = RP_j^{(N)}$, for the index i corresponding to the branching variable $RP_i^{(f(N))} \setminus S_i^{(N)} = RP_j^{(N)}$, we showed that $\Delta^{(N)} = \{(I_1 \setminus RP_1^{(N)}), \ldots, (I_n \setminus RP_n^{(N)})\}$ is a Global Cut Seed. Notice that this is valid also if the search tree is explored in Limited Discrepancy Search.

Proposition 7: The number of non-subsumed Extended Node Configuration is less or equal to the depth of the search space.
Proof: By construction, every Extended Node Configuration $\Delta^{(h0)}$ subsumes the Extended Node Configuration $\Delta^{(h)}$ if $h0$ is a dead-end node ancestor of h. In fact, each element of $\Delta^{(h0)}$ is a superset of the corresponding element of $\Delta^{(h)}$.

Therefore, once an Extended Node Configuration $\Delta^{(h)}$ is generated, all Δ generated in the subtree of h can be removed from the pool, because although they remain still valid, will not lead to any pruning that cannot be deduced also by $\Delta^{(h)}$.

6.1.6 Specialization of the Filtering Algorithm and Related Approaches

We will now describe some simple specializations of the general filtering algorithm. Some of these specialization lead to new symmetry removal algorithms; some others instead show that already known results in the field of symmetry removal can be seen as special cases of the general algorithm given in Section 4.

First Specialization (CUTS1)

The first specialization considers problems having a set of variables subject to permutation symmetries. Without loss of generality, we suppose that the entire set of variables is subject to permutation symmetries, and that all symmetric variables have the same initial domain. In this specialization, we suppose that the branching strategy chooses a variable and a value for the variable. In this situation, the filtering algorithm outlined in section 6.1.4 can be easily implemented.

Intuitively, we can see that if a value *val* for a given variable X_i is infeasible, being all variables symmetric, *val* will also be infeasible for any other variable X_j. Thus, upon backtracking on the branching choice $X_i = val$, we can remove *val* from the domain of any other variable not yet bound by branching.

To prove that this simple specialization is a special case of the general filtering algorithm proposed, consider a node h, on which a failure was generated by the assignment $X_i = val$. Being *Selected* the set of the k variables already assigned by branching, and being X_i the current branching variable, the node h has the following extended node configuration: $\Delta^{(h)} = \{\delta_1^{(h)}, \ldots, \delta_n^{(h)}\}$. For each $s, s \in Selected$, $\delta_s^{(h)} = \{v_s\}$; for each $j, j \notin Selected$, $\delta_j^{(h)} = I \setminus RP_j^{(h)} = I$, being I the initial domain of all variables; finally $\delta_i^{(h)} = \{val\}$.

At node $f(h)$, $D_{new}(X_s) = \{v_s\}$ for $s \in Selected$. There exist exactly $n-k$ different matchings M_l (for each $l \notin Selected$) of cardinality $n-1$ between the node configuration and the domain set. In matching M_l, $D_{new}(X_s)$ is associated to $\delta_s^{(h)}$ for $s \in Selected$; $D_{new}(X_i)$ is associated to $\delta_l^{(h)}$; $D_{new}(X_j)$ is associated to $\delta_j^{(h)}$, for $j \notin Selected, j \neq l$; $D_{new}(X_l)$ and $\delta_i^{(h)}$ are not matched. Therefore the matching M_l allows to remove $\delta_i^{(h)} = \{val\}$ from $D_{new}(X_l)$. This pruning can be done for each $l \notin Selected$.

Using different arguments, Roy and Pachet [148] found this same algorithm for pruning in case of permutation symmetries. Here we deduce the algorithm from the general one described in section 6.1.4, and we make a minor correction w.r.t. the one of [148]. In fact, in [148] the value that generated a failure *val* is removed from every non bound variable, while we show that *val* can be removed from every not yet bound by branching variable even if such a variable was bound by propagation.

An extension of this algorithm for removing (local) symmetries is that described in [115]. It is based on the idea that if the instantiation $X_i = val$ is tried without success, we can remove $\sigma_i(val)$ from the domain of $\sigma(X_i)$. In fact, given the same extended node configuration as before[5], we can find in node $f(h)$ only one mapping M_l of size $n-1$ where $l = idx(\sigma(X_i))$. Thus, we can remove from the domain of $\sigma(X_i)$ the symmetrical of *val*.

Second Specialization (CUTS2)

The second specialization still considers problems having a set of variables subject to permutation symmetries. Again, without loss of generality, we suppose that the entire set of variables is subject to permutation symmetries, and that all symmetric variables have the same initial domain. In this case we only suppose that the branching constraints are unary constraints (e.g., $X < val$, $X \geq val$, $X \neq val$). We are, in a way, generalizing the results presented in [148].

Given $\Delta^{(h)} = \{(I \setminus RP_1^{(h)}), \ldots, (I \setminus RP_n^{(h)})\}$ generated after a failure at node h, the domain of variables can be reduced if there exists a matching of cardinality $n-1$ between the set of domains and $\Delta^{(h)}$. The left subtree of node $f(h)$ is the fail node h; let consider the right subtree of node $f(h)$. In the right subtree defined by node $f(h)$, for each $i, i \neq k, D_{new}(X_i) \subseteq \delta_i^{(h)}$, therefore for each j such that $D_{new}(X_k) \subseteq \delta_j^{(h)}$, a matching M_j of cardinality $n-1$ exists, and $\delta_k^{(h)}$ can be removed from $D_{new}(X_j)$.

As shown in Proposition 7, once a Global Cut Seed $\Delta^{(h)}$ is generated, all Δ generated in the subtree of h can be removed from the pool, because although they remain still valid, will not lead to any pruning that cannot be deduced also by $\Delta^{(h)}$.

Whenever a new Global Cut Seed $\Delta^{(h)}$ is inserted into the pool, the filtering algorithm needs to check for each j if $D_{new}(X_k) \subseteq \delta_j^{(h)}$; at most $sizeOf(I)$ values are removed from $D_{new}(X_j)$. Therefore, the initial propagation of each Global Cut Seed can be obtained in $O(n\, sizeOf(I)^2)$. For each variable there exist at most $sizeOf(I)$ cuts to be considered, if all the instantiations of different values to a variable generated a non-subsumed Global Cut Seed. Whenever a set of values is removed from a domain of a variable, at most n checks should be performed for each cut. Each time a check $D_{new}(X_k) \subseteq \delta_j^{(h)}$ is successful, at most $sizeOf(I)$ values are removed from $D_{new}(X_j)$. Therefore, the worst case complexity of the filtering algorithm on one single cut for the entire set of permutation symmetries is $O(n\, sizeOf(I))$; the computational results will show that the mean case complexity is a lot lower.

Third Specialization (CUTS3)

We will now give a specialization of the algorithm for axial symmetries in order to show that it is independent from the particular symmetry we are considering. Consider a CSP with a y axis symmetry. An example is the symmetry appearing in the n-queen problem: for

[5]The node configuration is independent from the symmetry considered.

the 5 queen, we have that a solution is $X_1 = 2, X_2 = 5, X_3 = 3, X_4 = 1, X_5 = 4$ which is symmetrical to a solution where $X_1 = 4, X_2 = 1, X_3 = 3, X_4 = 5, X_5 = 2$. For each $X_i = k$ in the first solution, we have $X_i = n - k + 1$ in the symmetrical one. Thus, the symmetry on variables is the identity function, while the symmetries on values map k in $n - k + 1$.

Global Cut Seeds can be built at every failure, leading to $\Delta^{(h)} = \{(I_1 \setminus RP_1^{(h)}), \dots, (I_n \setminus RP_n^{(h)})\}$ at node h. I_i is the initial domain of variable X_i, and $RP_i^{(h)}$ is the set of values removed from variable X_i by positive branching constraints.

A matching of cardinality $n - 1$ exists iff for all variables but one $D_{new}(X_i) \subseteq \sigma_i(\delta_i)$ $i \neq j$ and there exists one and only one variable X_j such that $D_{new}(X_j) \not\subseteq \sigma_j(\delta_j)$. Note that $\sigma_j(\delta_j) = \{n - k | k \in \delta_j\}$.

In such a case for all values $k \in \delta_j$, we can remove their symmetrical counterpart $(n - k)$ from $D_{new}(X_j)$.

Each time a value is removed from a variable, each cut is triggered if the value removal produces a matching of cardinality $n-1$. At most $SizeOf(I_i)$ Global Cut Seeds exist for every variable X_i; every time a matching of cardinality $n - 1$ is found, at most $SizeOf(I_i)$ values are removed. Therefore, being $S = max_{i=1,\dots,n}\{SizeOf(I_i)\}$ the worst case complexity of the propagation algorithm is $O(n\ S)$ for a single cut. By incrementally maintaining information on the state of each Global Cut Seed, the mean case complexity can be greatly reduced.

Forth Specialization (CUTS4)

We will now give a specialization of the algorithm for symmetries defined by interchangeable values [73]. Let consider a CSP with n variables having interchangeable values. Given n variables $X_1, \dots X_n$, with initial domain I_i, suppose I_i can be partitioned in k_i subset of values $S_1^i, \dots, S_{k_i}^i$ such that for any couple of values belonging to the same partition, $va \in S_j^i, vb \in S_j^i$ any two solutions $v_1, \dots, v_{i-1}, va, v_{i+1}, \dots, v_n$ and $v_1, \dots, v_{i-1}, vb, v_{i+1}, \dots, v_n$ are equivalent.

Global Cut Seeds can be built at every failure, leading to $\Delta^{(h)} = \{(I_1 \setminus RP_1^{(h)}), \dots, (I_n \setminus RP_n^{(h)})\}$ at node h. $RP_i^{(h)}$ is the set of values removed from variable X_i by positive branching constraints.

Let X_k be the branching variable at node $f(h)$. In the right subtree defined by node $f(h)$, for each $i, i \neq k, D_{new}(X_i) \subseteq \delta_i^{(h)}$. Thus, a matching of cardinality $n - 1$ exists with $\delta_k^{(h)}$ and $D_{new}(X_k)$ unmatched. Therefore, for each value $v* \in \delta_k^{(h)}$ all values v such that v and $v*$ belong to the same partition of the domain I_k can be removed from $D_{new}(X_k)$.

In [20], the concept of interchangeable values is extended to cycle of symmetries where values are symmetrical two by two. Therefore, if a value participates to no solution, all symmetrical values can be removed as well. Our approach applies also in this case, by defining cycle symmetries instead of simple interchangeable values.

Fifth Specialization (CUTS5)

One of the first general filtering algorithm for removing symmetries was given in [79]. The authors propose an algorithm, called SBDS, that, under the hypothesis that branching decisions are of type $Var = val$ as opposed to $Var \neq val$, is able to remove a given symmetry σ. We show that SBDS can be reinterpreted as a specialization of our filtering algorithm.

Let A be the set of already assigned variables at node N ($X_i = v_i \forall X_i \in A$); and $\{\sigma, \sigma_1, \dots, \sigma_n\}$ be the symmetry handled by SBDS. If X_k is the current branching variable at node N, the left branch imposes the constraint $X_k = val$. In [79], the authors show that the following constraints can be added in the right branch in order to remove symmetric solutions w.r.t $\{\sigma, \sigma_1, \dots, \sigma_n\}$:

$\forall X_i \in A(X_i = v_i, \sigma(X_i) = \sigma_i(\{v_i\}) \wedge X_k \neq val \Rightarrow \sigma(X_k) \neq \sigma_k(\{val\})$

This constraint can be interpreted as a special case of the filtering algorithm proposed in Section 4. To prove that, let $I_i, i = 1, \dots, n$ be the initial domain for variables X_i involved in the symmetry σ. By definition of symmetry, $I_i = \sigma_i(I_{idx(\sigma(X_i))}), i = 1, \dots, n$.

Let now consider the node configuration generated by the fail and backtrack of the left branch of node N. $\Delta = \{\delta_1, \dots, \delta_n\}$ where $\delta_i = \{v_i\}$ $\forall i | X_i \in A$, $\delta_k = val$, $\delta_j = (I_j \setminus RP_j)$ $\forall j | X_j \notin A, j \neq k$, where I_j is the initial domain of variable X_j. Positive branching constraints have been applied only to variables in A. Thus $RP_j = \emptyset$ $\forall j | X_j \notin A$. Then, $\delta_j = I_j$ $\forall j | X_j \notin A, j \neq k$.

We will show that when the left hand side of the implication is true, then the propagation algorithm in Section 4 imposes the right hand side to be true as well.

Let consider the left hand side $(X_i = v_i, \sigma(X_i) = \sigma_i(\{v_i\}) \forall X_i \in A) \wedge X_k \neq val$ to be true. Then the following matching of cardinality $n-1$ exists: $(D_{new}(X_l), \delta_i), \forall l | X_l = \sigma(X_i)$, with $X_i \in A$, in fact $D_{new}(X_l) = \sigma_i(\{v_i\})$, and $\delta_i = \{v_i\}$.

All $\delta_j, | X_j \notin A, j \neq k$ contain the whole initial domain of the corresponding variable $(\delta_j = I_j \forall j | X_j \notin A, j \neq k)$, and therefore they also find a match.

Finally, $\delta_k = \{val\}$, and $D_{new}(X_h) | \sigma(X_h) = X_k$ are unmatched. Therefore the algorithm will remove $\sigma_h(\delta_k)$ from X_h. But this corresponds exactly to the pruning performed by SBDS, i.e., $\sigma(X_k \neq val)$.

In the same way, it is easy to see that when the right hand side of the implication is false, then the left hand side must be false as well or a failure will be triggered.

A general way of coping with symmetries, similar to [79], is that of [8], which applies for any search strategy. When the branching strategy considers unary constraints on branching variables, our approach can again be seen as a generalization of that of [8]. However, when the hypothesys does not apply, their approach is more general provided that they are able to define the symmetrical counterpart for a general branching constraint.

6.1.7 Computational Results

Even if the main purpose of this paper is to provide a general framework for removing symmetries, we show some computational results. The approach was tested on three different symmetric problems: the pigeonhole problem, the Ramsey problem, and a crew allocation problem. Five tests were performed on every problem instance: test one (columns SYM) runs each problem without symmetry removal, test two (columns CST1) and three (columns CST2) runs it with a symmetry removal obtained by adding symmetry-breaking constraints, test four (columns CUTS1) and five (columns CUTS2) runs it with two approaches proposed in this paper. Two branching strategies were used to run the tests. The first, used in columns SYM, CST1 and CUTS1 chooses the first unbound variable and assigns the first possible value. The one used in CST2 and CUTS2 chooses the unbound variable having the largest domain, and for such a variable splits the domain in two, keeping the lower half of it first.

In general, we will see that all the techniques that remove symmetries perform equally well on all the problems. The advantage of the cut generation technique CUTS2, lies on the fact that it is not intrusive, i.e., it does not modify the structure of the problem letting the users making decisions on the heuristics without taking into consideration that the problem structure was changed to remove symmetries. Moreover, the results show that, despite the worst case complexity of the methods, CUTS2 performs, in practice, very well. In Table 6.1, results are reported in number of fails and CPU time in seconds. All tests were performed on a PC PentiumII 200Mh on Win98 using ILOG Solver.

In Table 6.1, P-9 and P-10 refer to the Pigeonhole problem. The symmetry-breaking constraint used in columns *CST1* and *CST2* where introduced by Puget, and impose $H_1 \leq H_1 \leq \dots H_N$.

The second problem considered is a crew allocation problem, defined by Nespoulous [122]. It consists in allocating a set of N hostesses to P crews that need between 1 and 3 hostesses. Hostesses speak English, Brazilian or both, and each crew requires a given language. Three sets of completely permutable variables can easily be recognized, being the set of variables representing the crew for the hostesses speaking the same set of languages. The symmetry-breaking constraint used in columns CUTS1 and CUTS2 where introduced by Nespoulous [122], and imposes an order within each set of completely permutable variables. Problem *CR1* is infeasible. Problem *CR2* has a very large number of symmetric solutions. Also in this example the different symmetries removing methods performs equally well; in $CR2$ all the algorithms that remove the symmetries find 103 non symmetrical solutions, while the algorithm that does not remove the symmetries finds 2216 solutions.

Problem	SYM		CST1		CST2		CUTS1		CUTS2	
	T	F	T	F	T	F	T	F	T	F
P-9	9.17	40k	0.1	128	0.2	706	0.1	128	0.2	706
P-10	77.8	362k	0.1	256	0.8	2080	0.1	256	0.7	1585
CR1	23.5	144k	0.2	820	0.5	1974	0.2	950	0.7	1974
CR2	12.4	0	0.6	25	0.6	34	0.6	109	0.6	34
R6-All	31	373	2.0	42	2.1	42	2.0	43	2.0	42
R14-100	19.2	26k	4.3	5648	4.3	5648	4.2	5659	4.3	5648
R16-100	183	417k	27.4	46k	27.4	46k	28.4	46k	27.5	45k
R17-I	>300	-	0.3	636	0.3	636	0.3	655	0.3	636

Table 6.1: Experimental Results.

The Ramsey problem is another well known symmetric problem. For its definition and CP model we refer to [137]. For $N < 17$ the problem is feasible and contains a large number of symmetrical solutions; for $N = 17$ the problem is infeasible. We have used the same model proposed by Puget in [137]. Problem *R6-All* finds all solution for a Ramsey problem with 6 nodes, algorithm *SYM* does not remove the variable permutation symmetries and finds 118212 solutions, while all other methods finds 7697 solutions. Problems *R14-100* and *R16-100* look for the first 100 solutions of Ramsey problem with 14 and 16 nodes respectively. Problems *R17-I* is infeasible and the results refer to the proof of infeasibility. In the Ramsey problem only variable permutation symmetries were removed using our method while other type of symmetries were handled as in [137]. We could have used our method to remove more symmetries, and the task of removing all symmetries in the Ramsey problem is subject of current work.

6.1.8 Conclusions and future work

In this paper we propose a method to collect information during search and a general filtering algorithm able to use the collected information to prune the search space in symmetric CSPs.

The method proposed does not interfere with problem dependent heuristics, and can be easily implemented in Constraint Programming when commonly verified hypothesis on branching constraint are respected. The general filtering algorithm can be, in practice, very easily specialized to obtain constraints able to remove a given set of symmetries, and we show that many relevant previous approaches can be reinterpreted as specialization of the filtering algorithm proposed. Finally, the separation between the symmetry independent Global Cut Seed data structure and the filtering algorithm allows to treat different symmetry removal constraints.

6.2 Scheduling with setup times: enhancing the edge finding propagation

In recent years Constraint Programming (CP) techniques have been successfully used to model and solve scheduling problems. Combining ideas from Artificial Intelligence (AI) and Operations Research (OR), constraint-based scheduling was able to maintain the best of both approaches, i.e., the modeling flexibility of AI scheduling systems and the efficiency of OR algorithms [124].

One of the most interesting and successful examples of integration of OR techniques in CP is the use of the techniques first presented by Carlier and Pinson [31] in constraint-based scheduling [125]. The subsequent research on improvements, additions, and generalizations of these techniques brought the two scheduling communities in AI and OR closer together, up to a point that many feel there is no longer a clear separation.

Roughly speaking, the techniques we discuss here deduce bounds for the start and end time of one activity by considering the relation between this activity and a subset of other activities requiring the same resource. In this paper an enhancement of these techniques is proposed which, taking into consideration sequence dependent setup times, is able to strengthen the bounds on the start and end time of activities. This work is based on two previous studies of Nuijten [125] and Brucker and Thiele [28]; the first introduces edge finding techniques presented by Carlier and Pinson [31] in constraint-based scheduling, while the second extends these techniques for scheduling problems with sequence dependent setup times. We maintain the constraint-based scheduling framework, and we enhance the algorithm proposed in [125] by following and extending the work proposed in [28].

6.2.1 Problem definition and Constraint Model

Although we are interested in general shop scheduling problems, for the description of the propagation algorithm we concentrate on the 1-machine scheduling problem where we have a machine μ and a set Ω of n activities $a_1, \ldots, a_n$ all to be processed on μ for $pt(a_i)$ time units ($i = 1 \ldots n$). The activities may have release dates and due dates. The set of all activities Ω is partitioned into k groups $g_1, g_2, \ldots, g_k$ such that the sequence dependent setup times existing among activities depend only on the groups the activities belong to. The group of an activity a is referred to as $gr(a)$; while the set of all groups $g_h, h = 1, \ldots, k$ is referred to as G. Given a setup time matrix s (square matrix of dimension k), s_{lm} represents the setup time between activities a_i, belonging to group g_l ($gr(a_i) = l$), and a_j, belonging to group g_m ($gr(a_j) = m$), if a_i and a_j are scheduled sequentially on machine μ with a_i first. In such a case, $start(a_j) \geq end(a_i) + s_{lm}$. For sake of convenience, we will refers to $I \subseteq G$ as both a set of groups, and the complete graph defined by the set of vertices I with costs on the arcs defined by the setup matrix s.

Each activity a is represented by using two variables being its start time $start(a) = [est(a)..lst(a)]$ and its end time $end(a) = [ect(a)..lct(a)]$, where $est(a)$ and $lst(a)$ are the earliest and latest start time of a, and $ect(a)$ and $lct(a)$ are the earliest and latest end (completion) time, respectively. These two variables are constrained by the relation $end(a) - start(a) = pt(a)$.

6.2.2 Preliminaries

We consider the relation between one activity a and a set of other activities $S \subset \Omega$. Defining $est(S) = \min_{a \in S}\{est(a)\}$, $lct(S) = \max_{a \in S}\{lct(a)\}$, and $pt(S) = \sum_{a \in S} pt(a)$ the following theorem represents the key element of the adjustment techniques:

Theorem 1 *Let μ be a machine, and $a \in \Omega$ and $S \subset \Omega$ activities, such that $a \notin S$ and all activities in $S \cup \{a\}$ are to be scheduled on μ. If*

$$est(S \cup \{a\}) + pt(S \cup \{a\}) > lct(S),$$

then $est(a)$ can be set to $\max_{S' \subseteq S}\{est(S') + pt(S')\}$.

Theorem 1 is justified by reasoning on the total sum of processing times required on machine μ by the set of activities $S \cup \{a\}$. Naturally there is also the symmetric theorem allowing to find bounds for the latest end time of activities. Using Theorem 1 it is possible to update $est(a)$ and $lct(a)$ for all $a \in \Omega$ in $O(nlog(n))$ time [32].

These theorems are valid both on problems with and without sequence dependent setup times. Brucker and Thiele in [28] introduce two restrictive hypotheses on the setup matrix, and, based on these hypotheses extend the previous theorem. The first hypothesis supposes that the setup between two activities belonging to the same group is equal to zero.

Hypothesis 1 (Group based setup)

$$\forall h = 1, \ldots, k \;\; s_{hh} = 0.$$

The second hypothesis supposes that the setup matrix s does not violate the triangular inequality:

Hypothesis 2 (Triangular inequality)

$$\forall i, j, l \in G \;\; s_{ij} \leq s_{il} + s_{lj}.$$

If the Hypothesis 2 holds, the Theorem 1 can be extended as follows:

Theorem 2 *Let μ be a machine, and $a \in \Omega$ and $S \subset \Omega$ activities subject to sequence dependent setup times, such that $a \notin S$ and all activities in $S \cup \{a\}$ are to be scheduled on μ. Let $Setup(S)$ be the minimal total setup time to schedule activities in S; and $Setup(S \rightarrow a)$ be the minimal total setup time to schedule all activities in S followed by activity a. If*

$$est(S \cup \{a\}) + pt(S \cup \{a\}) + Setup(S \cup \{a\}) > lct(S),$$

then $est(a)$ can be set to $\max_{S' \subseteq S}\{est(S') + pt(S') + Setup(S' \rightarrow a)\}$.

Again, there is a symmetric theorem providing bounds for the latest completion times of activities. Theorem 2 clearly provides better bounds for the earliest start time of activities than Theorem 1. On the other hand, the computation of $Setup(S)$ is itself an NP-Hard problem since it is equivalent to an Asymmetric Traveling Salesman Problem (ATSP). In order to effectively use Theorem 2 in place of Theorem 1 it is important to efficiently compute

$Setup(S)$, and $Setup(S \to a)$. We will discuss methods to compute $Setup(S)$; the methods proposed can easily be extended for $Setup(S \to a)$.

Let $G(S)$ be the set of groups of all activities in S ($G(S) = \{g | gr(a_i) = g,\ \forall a_i \in S\}$). If the hypotheses 2 and 1 hold, in [28] five different methods to calculate lower bounds on $Setup(S)$ (and $Setup(S \to a)$) are proposed. All of them reason on an Asymmetric Traveling Salesman Problem on the graph $G(S)$. In fact, it is easy to prove that if hypotheses 2 and 1 hold then $Setup(S) = Setup(G(S))$.

Setup 1 *$setup_1(G(S))$ is calculated by solving an Asymmetric Traveling Salesman Problem in the setup matrix s. These values are calculated by enumerating all possible sequences, leading to a complexity of $O(|G(S)|!)$*

Setup 2 *$setup_2(G(S))$ is calculated by solving the Assignment Problem (AP) relaxation of the ATSP in the setup matrix s. The complexity of calculating the AP is $O(|G(S)|^3)$.*

Setup 3 *$setup_3(G(S))$ is calculated by solving the Minimum Spanning Tree (MST) relaxation of the Symmetric TSP on the matrix s' obtained by setting $s'_{lm} = \min\{s_{lm}, s_{ml}\}, \forall l, m = 1, \ldots, k$. The complexity of calculating the MST is $O(|G(S)|^2)$.*

Setup 4 *$setup_4(G(S))$ is calculated by taking the sum of the $|G(S)| - 1$ smallest setups of the set $\{s_{lm} | l, m \in G(S),\ l \neq m\}$.*

Setup 5 *$setup_5(G(S))$ is calculated by considering the minimum setup $s_{min}(G(S))$ in the set $\{s_{lm} | l, m \in G(S),\ l \neq m\}$. $setup_5(G(S))$ is defined as $setup_5(G(S)) = (|G(S)| - 1) s_{min}(G(S))$*

In order to maintain good performance of the algorithm that adjusts the bounds of the activities, [28] use $setup_4(G(S))$ or $setup_5(G(S))$, and update $est(a)$ and $lct(a)$ for all $a \in \Omega$ in $O(n \max\{nlog(k), k^2\})$ time.

6.2.3 Goals

In the remaining of the paper, we will enhance the approach proposed in [28] in three directions:

- we propose a more generic extension of Theorem 1, which is not subject of any restrictive hypotheses. We propose a weaker version of the Hypothesis 2, named *weak triangular inequality*, and we discuss the relative impact of the three hypotheses;
- one important drawback of the bounds proposed in [28] is the fact that, in theory, each set $G(S)$ corresponding to the set S of activities under consideration, has its own lower bound, calculated as $setup_x(G(S))$; in other words, the number of different values of $setup_x(G(S))$ is exponential on the size k of the setup matrix since the number of possible subsets of the set G is $2^{|G|} = 2^k$. It is therefore impractical to compute all bounds $setup_x(G(S))$ for each possible $G(S)$ and retrieve it when needed. We will propose different lower bounds with the characteristic that the number of different values $setup_x(G(S))$ is polynomial in the size k of the setup matrix. All the possible values of the lower bounds are computed once for all, and stored in a matrix. Using the precomputed values we could easily enhance the algorithm presented in [125] so that it considers sequence dependent setup times, while maintaining the same complexity $O(n^2)$.

- all the bounds proposed in [28] ignore the constraints related to the time dimension of the problem. We will propose a method able to consider also the time dimension leading to a better evaluation of the sum of setup times in the set of activities S.

6.2.4 Definitions

Given a complete digraph I identified by k vertices and a cost matrix s, we refer to I as the graph and its vertex set. For each $I_1 \subseteq I$ we define:

- $Setup(I_1)$ as the minimum cost path passing through all vertices in I_1 *exactly once.*
- $Setup(m_1, \dots, m_k)$ as the minimum cost path passing through each vertex $h \in I$ *exactly m_h times.*
- $Setup^m(I_1^m, I_2^\infty)$ as the minimum cost path passing through every vertex in $I_1^m \cup I_2^\infty$ *at least once*, and all vertices in I_1^m exactly m times (with $m \geq |I_1^m|$).
- $Setup^\infty(I_1)$ as the minimum cost path passing through all vertices in I_1 *at least once.*
- $Setup(m_1, \dots, m_i, I)$ as the minimum cost path passing through each vertex $h = 1, \dots, i$ *exactly m_h times*, and covering any number of times vertices in I (with $\{1, \dots, i\} \subseteq I$).
- $Setup^m(I_1^m, I_2^\infty, I)$ as the minimum cost path passing through every vertex in $I_1^m \cup I_2^\infty$ *at least once*, and all vertices in I_1^m exactly m times, and covering any number of times vertices in I (with $I_1^m \cup I_2^\infty \subseteq I$, and $m \geq |I_1^m|$).
- $Setup^\infty(I_1, I)$ as the minimum cost path passing through all vertices in I_1 *at least once*, and covering any number of times vertices in I (with $I_1 \subseteq I$).

It is easy to see that the following relations hold:

$$\begin{array}{ll} Setup^\infty(I_1, I) \leq Setup^\infty(I_1) \leq Setup(I_1) & \\ Setup^\infty(I_1) \leq Setup^m(I_1, \emptyset) \leq Setup(m_1, \dots, m_k) & I_1 = \{h \in I | m_h > 0\}, m = \sum_{h=1}^{k} m_h \end{array}$$

Moreover, it is easy to see that if $\exists h \in I | s_{hh} = 0$, then the following relations hold:

$$\begin{array}{ll} Setup^\infty(I_1) = Setup^m(I_1, \emptyset) & \forall I_1 \subseteq I, m \geq |I_1| \\ Setup^\infty(I_1, I) = Setup^m(I_1, \emptyset, I) & \forall I_1 \subseteq I, m \geq |I_1| \end{array}$$

It is useful to give an upper bound on the number of nodes in the optimal path of $Setup^\infty(I)$.

Theorem 3 *Let I be a complete graph of vertices $1, \dots, k$, and s be a cost matrix of size k associated to the graph I such that $s_{hh} = 0, h = 1, \dots, k$. Let $Setup^\infty(I)$ be value of the minimum cost path covering all vertices in I* at least *once.*

There exists a path covering all vertices in I at least once of minimum cost whose number of nodes is no more than k^2.

Proof. Let p be an optimal path for $Setup^\infty(I)$ of minimum length. Let $p_1, \dots, p_k$ be the list of the k different vertices in p in the order in which they appear in p. Let $Pred_h$ be the set of different vertices that appear in p before node p_h. For example, if p_h is in position n in the path p, and $p_h = g \in I$, then the vertex $g \notin Pred_h$. By definition of $p_1, \dots, p_k$, between any two nodes p_i, p_{i+1}, $i = 1, \dots, k-1$ only vertices $g \in Pred_{i+1}$ may appear in the fragment of *the optimal path from p_i to p_{i+1}. We want to show that any vertex g ($g \in Pred_{i+1}$) belonging to the fragment $p_i \rightarrow p_{i+1}$, appears at most once. If this is true, then the length of each fragment of p defined by $p_i \rightarrow p_{i+1}$ is bound by $|Pred_{i+1}| = i$, and therefore the length of the whole path is bound by $k(k-1)/2 < k^2$.*

Therefore, in order to complete the proof we need to show that any vertex g ($g \in Pred_{i+1}$) in the fragment of the optimal path from p_i to p_{i+1}, appears at most once. Suppose that there exists a vertex $g \in Pred_{i+1}$ appearing twice in the fragment $p_i \rightarrow p_{i+1}$ in position n, and m, with $n < m$. We can build a new fragment $p_i \rightarrow p_{i+1}$ by removing all vertices from $n+1$ to m. The new path represents a feasible solution for the problem since all vertices are still covered at least once, moreover the cost of the new path is not increased with respect to p since some arcs are removed, and it has a shorter length. But this is absurd since the path p was defined as the minimum length optimal path.

6.2.5 Considerations on the working hypotheses

Without any additional hypotheses on the structure of the problem defined in Section 6.2.1 the Theorem 1 can be extended to take into consideration sequence dependent setup times:

Theorem 4 *Let μ be a machine, and $a \in \Omega$ and $S \subset \Omega$ activities subject to sequence dependent setup times, such that $a \notin S$ and all activities in $S \cup \{a\}$ are to be scheduled on μ. Let $E = \Omega \setminus (S \cup \{a\})$. We define $PS(S)$ as the minimal sum of processing times and setup times of all operations in S and, possibly, operations in E, i.e., $PS(S) = \min_{P \supseteq S}\{pt(P) + Setup(P)\}$, where $Setup(P)$ is the minimal sum of setup times of operations in P. Similarly, we define $PS(S \rightarrow a) = \min_{P \supseteq S}\{pt(P) + Setup(P \rightarrow a)\}$. If*

$$est(S \cup \{a\}) + PS(S \cup \{a\}) > lct(S),$$

then $est(a)$ can be set to $\max_{S' \subseteq S}\{est(S') + PS(S' \rightarrow a)\}$.

Computing the value $PS(S)$ is clearly an NP-Hard problem; fortunately, Theorem 4 remain valid if $PS(S \cup \{a\})$ and $PS(S' \rightarrow a)$ are substituted with some lower bounds. For example, Theorem 1, ignoring the setup times, considers $pt(S \cup \{a\})$ as a lower bound of $PS(S \cup \{a\})$. Since the setup time depends only on the groups $gr(a_i)$ of activities $a_i \in PS(S)$, $PS(S)$ can be reformulated as follows:

$$PS(S) = \min_{P \supseteq S}\{pt(S) + pt(P \setminus S) + Setup(m_1(P), \ldots, m_k(P))\}$$

where $m_h(P)$ is the number of activities of type h in the set P (i.e., $m_h(P) = |\{a_i | a_i \in P \wedge gr(a_i) = h\}|$). If an activity $a_i \notin S$ belong to the set $P^* \supseteq S$ that minimizes $PS(S)$, the value $pt(a_i)$ is considered in $pt(P \setminus S)$. Let pt_f^{min} be the minimal processing time of any activity of group f, $pt_f^{min} = \min_{a_i \in \Omega,\, gr(a_i)=f}\{pt(a_i)\}$; a lower bound of $PS(S)$ can be defined by penalizing any activity $a_i \notin S$ by a value equal to $pt_{gr(a_i)}^{min}$ instead of $pt(a_i)$:

$$LB(PS(S)) = \min_{P \supseteq S}\{pt(S) + \sum_{h=1}^{k} m_h(P \setminus S) pt_h^{min} + Setup(m_1(S) + m_1(P \setminus S), \ldots, m_k(S) + m_k(P \setminus S))\}$$

Based on this last formulation, we change the setup matrix s in such a way that each (group dependent) penalty pt_h^{min} becomes part of the setup matrix. In order to do that, a new graph G' and a new setup matrix s' are considered. The graph G' duplicates each node in G, and assign additional penalty equal to pt_h^{min} to each arc outcoming from a duplicated node. The matrix s', of dimension $2k$, is defined as $s'_{f,g} = s_{f,g}$ for $f, g =\leq k$; $s'_{f,g} = s_{f,g-k}$ for $f \leq k, g > k$; $s'_{f,g} = s_{f-k,g} + pt_f^{min}$ for $f > k, g \leq k$; $s'_{f,g} = s_{f-k,g-k} + pt_f^{min}$ for $f > k, g > k$. G' is the complete graph defined by the cost matrix s'. It is easy to see that the previous formulation of $LB(PS(S))$ is equivalent to:

$$LB(PS(S)) = \min_{P \supseteq S}\{pt(S) + Setup(m_1(S) \ldots, m_k(S), m_{k+1}(P \setminus S), \ldots, m_{2k}(P \setminus S))\}$$

where $Setup(m_1(S) \ldots, m_k(S), m_{k+1}(P \setminus S), \ldots, m_{2k}(P \setminus S))$ is evaluated on the graph G' instead of G. Since P can be any superset of S, the previous formulation is equivalent to:

$$LB(PS(S)) = pt(S) + Setup(m_1(S) \ldots, m_k(S), G')$$

For every node $h \in G'$ such that $s_{hh} = 0 \wedge m_h(S) > 0$, we relax the constraint imposing that the optimal path will pass through node h exactly $m_h(S)$ times, and we only enforce that node h is covered. For every node $h \in G'$ such that $s_{hh} \neq 0 \wedge m_h(S) > 0$, we relax the constraint imposing that the optimal path will pass through node h exactly $m_h(S)$ times as follows. Let $G_1(S), G_2(S)$ such that $G(S) = G_1(S) \cup G_2(S)$ and $G_1(S) = \{h | s_{hh} \neq 0 \wedge m_h(S) > 0\}$, and $G_2(S) = \{g | s_{gg} = 0 \wedge m_h(S) > 0\}$; let $m(S) = \sum_{h \in G_1(S)} m_h(S)$. Nodes in $G_1(S)$ are globally covered $m(S)$ times, while nodes in $G2(S)$ are covered at least once. This leads to a lower bound on the previous formulation of LB(PS(S)):

$$LB(PS(S)) = pt(S) + Setup^{m(S)}(G_1(S), G_2(S), G')$$

This formulation does not use any additional hypotheses on the problem formulation, and it can be read as the sum of processing times in S plus the minimal sum of setup times of a path having the following characteristics: it passes at least once in every node in $G_1(S) \cup G_2(S)$, corresponding to the types of activities in S; it passes $m(S)$ times through all nodes in $G_1(S)$ where $G1(S)$ is the set of types g of activities in S having $s_{gg} \neq 0$, and $m(S)$ is the number of such activities; it may pass more than $m(S)$ through a type $g \in G_1(S)$ by paying a penalty equal to pt_g^{min}; it may pass through a type g not in $G(S)$ by paying a penalty equal to pt_g^{min}. Although finding this optimal path is still an NP-Hard problem, we will propose both an algorithm to solve it, and an algorithm to find a lower bound. In the rest of this section, we will see how this formulation can be simplified if some hypotheses hold. The simplification we aim at achieving may concerns, for example, the removal of the factor $m(S)$, leading to the simpler $Setup^{\infty}(G(S), G')$; or it may concern the removal of the penalty graph G', leading to $Setup^{m(S)}(G_1(S), G_2(S))$ and $Setup^{\infty}(G(S))$.

Hypotheses 2 and 1 have been introduced in [28] in order to simplify the definition and the computation of $PS(S)$. We will now define third hypothesis and will discuss the relative impact of the three hypotheses on the formulation of $PS(S)$.

Hypothesis 3 (Weak triangular inequality)

$$\forall a_i, a_j, a_l \in \Omega \;\; s_{f,g} \leq s_{f,h} + s_{h,g} + pt(a_l), \;\; f = gr(a_i), g = gr(a_j), h = gr(a_l).$$

We defined $PS(S)$ as the minimal sum of processing times and setup times of all operations in S and, possibly, operations in E; it is easy to see that if the Hypothesis 3 holds it is never convenient to insert operations from E to minimize $PS(S)$. Therefore, in this case:

$$PS(S) = \min_{P \supseteq S}\{pt(P) + Setup(P)\} = pt(S) + Setup(S)$$

or, equivalently,

$$PS(S) = pt(S) + Setup(m_1(S), \ldots, m_k(S))$$

Since Hypothesis 2 implies Hypothesis 3, clearly this last formulation of $PS(S)$ is also valid if Hypothesis 2 holds. Let $G_1(S), G_2(S)$ such that $G(S) = G_1(S) \cup G_2(S)$ and $G_1(S) = \{h | s_{hh} \neq 0 \wedge m_h(S) > 0\}$, and $G_2(S) = \{g | s_{gg} = 0 \wedge m_h(S) > 0\}$; let $m(S) = \sum_{h \in G_1(S)} m_h(S)$. A lower bound on the previous formulation of LB(PS(S)) is the following:

$$LB(PS(S)) = pt(S) + Setup^{m(S)}(G_1(S), G_2(S))$$

If the hypotheses 3 and 1 hold, the same lower bound can be written as:

$$LB(PS(S)) = pt(S) + Setup^{\infty}(G(S))$$

While, if the hypotheses 1 and 2 hold, it is easy to see that any optimal path for $Setup(m_1(S), \ldots, m_k(S))$ can always be transformed in an equivalent one covering all vertices in $G(S)$ exactly once. In fact, if hypotheses 1 and 2 hold, in the optimal schedule for $Setup(S)$ all activities are scheduled group by group. In this case, $Setup(m_1(S), \ldots, m_k(S)) = Setup^0(\emptyset, G(S)) = Setup^{\infty}(G(S)) = Setup(G(S))$, and therefore:

$$PS(S) = pt(S) + Setup(G(S))$$

In the following, several lower bounds will be proposed for calculating $Setup(G(S))$, $Setup^{\infty}(G(S))$, $Setup^{m(S)}(G_1(S), G_2(S))$, and $Setup^{m(S)}(G_1(S), G_2(S), G')$.

6.2.6 Lower bounds on the sum of setups

The propagation algorithm using Theorem 4 is called several times (at least once at each node of the search tree) during the execution of any branch and bound method for solving the scheduling problem. Whichever formulation of $PS(S)$ is used, it is clear that during the solving procedure of the scheduling problem, a bound on the setup times component of $PS(S)$ is called over and over with the same input. In other words, we keep calculating the same values a very large number of times. The only way to avoid the inefficiency due to the recomputation of the same bounds is to cache them and retrieve them when needed. Unfortunately, the number of different input of the bounding procedure is very large, and storing a value for each possible input is impractical. The idea is to define a mapping between the exponential number of possible inputs of the bounding procedure into a polynomial number of bounds. Once this mapping is established, the polynomial number of bounds can be computed once for all, stored in memory, and used when needed without increasing the complexity of the propagation algorithm. For example, a bounding procedure $setup_x$ could be defined as $setup_x(|G(S)|)$. Then, all the sets S such that $|G(S)| = h$ will share the same bound $setup_x(h)$. In this case, the mapping is based on the cardinality of the set $G(S)$.

This approach is inspired from Dynamic Programming State Space Relaxation (SSR) [42], and will make use of Dynamic Programming State Space Relaxation to calculate most bounds proposed. In Dynamic Programming (DP), a problem P is associated with a *state space graph* $SG = (S, T)$ where each element of the vertex set S is a state and each element of the arc set T represents a feasible transition between two states. The original problem is solved by solving a shortest path problem in the state space graph from an initial state to a final state (boundary condition)[6]. If the original problem is NP-Hard, the correspondent state space graph has an exponential number of nodes, and therefore the time and space complexity to solve the shortest path in the state space graph is also exponential. The idea of State Space Relaxation is to calculate a lower bound of the original problem by defining a new state space graph $RSG = (RS, RT)$ such that a function $w(s)$ exists mapping every state $s \in S$ into a state $w(s) \in RS$, and such that for every arc $(s_i, s_j) \in T$, $(w(s_i), w(s_j))$ is a feasible arc in RT. Moreover, the cost of $(w(s_i), w(s_j))$ is smaller or equal to the cost of (s_i, s_j). Then, if p is the shortest path from s_1 to s_i, and rp is the shortest path from $w(s_1)$ to $w(s_i)$, the cost of rp is less or equal to the cost of p. The relaxation represented by $RSG = (RS, RT)$ is useful if the number of vertices of the new graph is polynomial instead of exponential.

Bounds for $Setup(G(S))$ (hypotheses 1 and 2 hold)

If hypotheses 1 and 2 hold, $PS(S)$ can be formulated as $PS(S) = pt(S) + Setup(G(S))$ where $Setup(G(S))$ is the minimum cost path covering all nodes in $G(S)$ exactly once. We compute the value $Setup(G)$ by solving an Hamiltonian Path Problem using the following Dynamic Programming formulation: let 0 be a dummy node, and $s_{0h} = s_{h0} = 0$, for $h = 1, \dots, k$, and $s_{00} = \infty$. Let $setup_m(F, i)$ be the shortest path from node 0 to node i covering all nodes in the set F exactly once, with $m = |F|$.

$$\begin{array}{ll} setup_1(\{i\}, i) = s_{0i} & \forall i \in G \\ setup_m(F, j) = \min_{i \in F \setminus \{j\}}\{setup_{m-1}(F \setminus \{j\}, i) + s_{ij}\} & \forall F \subseteq G, |F| \geq 2, \forall i, j \in G \end{array}$$

Clearly, the optimal solution $Setup(G)$ is given by:

$$Setup(G) = \min_{i \in G}\{setup_k(G, i)\}$$

The value $Setup(G)$ can be computed by computing all values $setup_m(F, j), \forall F \subseteq G, |F| = m, \forall j \in G$, for increasing values of m. The complexity of calculating $Setup(G)$ is $O(k2^k)$.

We define $setup^c_{hi} = \infty$, $setup^e_{hi} = \infty$ representing the minimum cost path of length h containing (resp. not containing) node i. After each computation of $setup_h(F, j)$ we update $setup^c_{hi}$, and $setup^e_{hi}$ as follows:

$$\begin{array}{ll} \forall i \in F & setup^c_{|F|,i} = \min\{setup^c_{|F|,i}, setup_{|F|}(F, j)\} \\ \forall i \in G \setminus F & setup^e_{|F|,i} = \min\{setup^e_{|F|,i}, setup_{|F|}(F, j)\} \end{array}$$

During the execution of the propagation algorithm, we use as lower bound of $Setup(G(S))$ the value $lb = \max\{setup^c_{|G(S)|,g}, setup^e_{|G(S)|,f}\}$ for any $g \in G(S)$ (for example the smallest one), any $f \notin G(S)$ (for example the smallest one). Since all values $setup^c_{hi}$, and $setup^e_{hi}$ are computed once for all, the complexity of the propagation algorithm does not increase. In fact, the time $O(k2^k)$ is needed only once to set up the matrix $setup^c_{h,g}$, and $setup^e_{h,g}$ in a

[6]This definition of DP is somehow restrictive, but it is sufficient for the use of DP we make in this paper.

preprocessing step. The propagation algorithm used during the branch and bound uses the values stored in $setup^c_{h,g}$, and $setup^e_{h,g}$, and maintains a complexity of $O(n^2)$.

The second bounding procedure proposed is based on a state space relaxation of the previous formulation: we define $setup^c_m(i,j)$ as the shortest path from node 0 to node j passing through m nodes before j containing node i. We define $setup^e_m(i,j)$ as the shortest path from node 0 to node j passing through m nodes before j not containing node i. $setup^c_m(i,j)$, and $setup^e_m(i,j)$ can be computed using the following recursive relations:

$$\begin{aligned}
&setup^c_0(i,j) = \infty && \forall i,j \in G\\
&setup^e_0(i,j) = s_{0j} && \forall i,j \in G\\
&setup^c_m(i,i) = \infty && \forall m, i = 0,\dots,k\\
&setup^c_m(i,j) = \min\{\min_{l\in G\setminus\{i,j\}}\{setup^c_{m-1}(i,l) + s_{lj}\},\\
&\qquad\qquad setup^e_{m-1}(i,i) + s_{ij}\} && \forall i,j \in G, \forall m = 1,\dots,k-1\\
&setup^e_m(i,j) = \min_{l\in G\setminus\{i,j\}}\{setup^e_{m-1}(i,l) + s_{lj}\} && \forall i,j \in G, \forall m = 1,\dots,k-1
\end{aligned}$$

The values $setup^c_{hi}$, $setup^e_{hi}$ representing the minimum cost of a path of length h containing (resp. not containing) node i can be computed as:

$$\begin{aligned}
&setup^c_{hi} = \min\{\min_{j\in G\setminus\{i\}}\{setup^c_{h-1}(i,j)\}, setup^e_{h-1}(i,i)\} && \forall h, i = 1,\dots,k\\
&setup^e_{hi} = \min_{j\in G\setminus\{i\}}\{setup^e_{h-1}(i,j)\} && \forall h, i = 1,\dots,k
\end{aligned}$$

The State Space Relaxation and every value $setup^c_{hi}$, and $setup^e_{hi}$ can be computed in $O(k^4)$. Since all values $setup^c_{hi}$, and $setup^e_{hi}$ are computed once for all, the complexity of the propagation algorithm does not increase. Weaker bounds can also be defined by using a state space relaxation calculating the minimum cost path of length h from node 0 to node i, i.e., $setup_h(i)$. The computation of this bound would reduce the time complexity to $O(k^3)$.

Bounds for $Setup^\infty(G(S))$ (hypotheses 1 and 3 hold)

If hypotheses 1 and 3 hold, a lower bound on $PS(S)$ can be formulated as: $LB(PS(S)) = pt(S) + Setup^\infty(G(S))$ where $Setup^\infty(G(S))$ is the minimum cost path covering all nodes in $G(S)$ at least once. Again, we compute the value $Setup^\infty(G)$ by Dynamic Programming. The DP recursive formulation is quite similar to the previous one, the main difference consists in the fact that the index m representing the number of nodes in the path, does not necessarily corresponds to the cardinality of the set of nodes already covered since a node can be covered more than once. Note that, because of that, we cannot stop the DP recursion at $m = |G|$ as in the previous case, since the optimal path may have a length equal to $m > |G|$. On the other hand, we have shown (Theorem 3) that there exists an optimal path having length $m < |G|^2$. Therefore the recursion relation can safely be stopped at $m = |G|^2$. Let 0 be a dummy node, and $s_{0h} = s_{h0} = 0$, for $h = 1,\dots,k$, and $s_{00} = \infty$. Let $setup_m(F,i)$ be the shortest path from node 0 to node i covering all nodes in the set F (without loops) at least once through a path of length $m \geq |F|$.

$$\begin{aligned}
&setup_1(\{i\},i) = s_{0i} && i \in G\\
&setup_m(F,j) = \min_{i\in F\setminus\{j\}}\{\min\{setup_{m-1}(F\setminus\{j\},i) + s_{ij},\\
&\qquad\qquad setup_{m-1}(F,i) + s_{ij}\} && \forall F \subseteq G, \forall j \in G, \forall m = |F|,\dots,|F|^2\\
&setup_m(F,j) = \infty && \forall F \subseteq G, \forall j \in G, \forall m < |F|, m > |F|^2
\end{aligned}$$

Clearly, the optimal solution $Setup^\infty(G)$ is given by:

$$Setup^{\infty}(G) = \min_{i \in G, k \leq h \leq k^2} \{setup_h(G,i)\}$$

As in the previous bounds, the values $setup^c_{hi}$ and $setup^e_{hi}$ representing the minimum cost path covering a set of cardinality h containing (resp. not containing) node i can be defined. After each computation of $setup_m(F,j)$ with $|F| = h$, we update $setup^c_{hi}$, and $setup^e_{hi}$ as follows:

$$\begin{array}{ll} \forall i \in F & setup^c_{|F|,i} = \min\{setup^c_{|F|,i}, setup_m(F,j)\} \\ \forall i \in G \setminus F & setup^e_{|F|,i} = \min\{setup^e_{|F|,i}, setup_m(F,j)\} \end{array}$$

The complexity of calculating $Setup^{\infty}(G)$ is $O(k^2 2^k)$, although, in practice, the effort for computing it can be greatly reduced by considering dominance properties such as:

$$(setup_{m1}(F,j) \geq setup_{m2}(F,j) \wedge m1 > m2) \Rightarrow setup_{m1}(F,j) = \infty$$

The bound based on the state space relaxation is the following: we define $setup^c_m(i,j)$ as the shortest path from node 0 to node j passing through m nodes before j, without loops, and containing node i. We define $setup^e_m(i,j)$ as the shortest path from node 0 to node j passing through m nodes before j, without loops, and not containing node i.

$$\begin{array}{ll} setup^c_0(i,j) = \infty & \forall i,j \in G \\ setup^e_0(i,j) = s_{0j} & \forall i,j \in G \\ setup^c_m(i,j) = \min\{\min_{l \in G \setminus \{j\}}\{setup^c_{m-1}(i,l) + s_{lj}\}, & \\ \qquad\qquad setup^e_{m-1}(i,i) + s_{ij}\} & \forall m = 0,\ldots,k^2, \forall i,j \in G, i \neq j \\ setup^c_m(i,i) = \min_{l \in G \setminus \{i\}}\{setup^c_{m-1}(i,l) + s_{li}\} & \forall m = 0,\ldots,k^2, \forall i \in G \\ setup^e_m(i,j) = \min_{l \in G \setminus \{i,j\}}\{setup^e_{m-1}(i,l) + s_{lj}\} & \forall m = 0,\ldots,k^2, \forall i,j \in G \end{array}$$

A lower bound on $Setup^{\infty}(G)$ is:

$$LB(Setup^{\infty}(G)) = \max_{i \in G}\{\min_{k \leq m \leq k^2, j \in G}\{setup^c_m(i,j)\}\}$$

The values $setup^c_{hi}$, $setup^e_{hi}$ representing the minimum cost path of length h containing (resp. not containing) node i can be set as:

$$\begin{array}{ll} setup^c_{hi} = \min\{\min_{j \in G, h \leq m \leq h^2}\{setup^c_{m-1}(i,j)\}, setup^e_{m-1}(i,i)\} & \forall h, i \in G \\ setup^e_{hi} = \min_{j \in G \setminus \{i\}, h \leq m \leq h^2}\{setup^e_{m-1}(i,j)\} & \forall h, i \in G \end{array}$$

The State Space Relaxation and every value $setup^c_{hi}$, and $setup^e_{hi}$ can be computed in $O(k^5)$; nevertheless, the average complexity can be greatly reduced by consider dominance properties such as:

$$\begin{array}{l} (setup^c_{m1}(i,j) \geq setup^c_{m2}(i,j) \wedge m1 > m2 \geq k) \Rightarrow setup^c_{m1}(i,j) = \infty \\ (setup^e_{m1}(i,j) \geq setup^e_{m2}(i,j) \wedge m1 > m2 \geq k-1) \Rightarrow setup^e_{m1}(i,j) = \infty \end{array}$$

Since all values $setup^c_{hi}$, and $setup^e_{hi}$ are computed once for all, the complexity of the propagation algorithm does not increase.

Bounds for $Setup^{m(S)}(G1(S), G2(S))$ (Hypothesis 3 holds)

If Hypothesis 3 holds, a lower bound on $PS(S)$ can be formulated as: $LB(PS(S)) = pt(S) + Setup^{m(S)}(G1(S), G2(S))$. We compute the value $Setup^{n1}(G1, G2)$ by Dynamic Programming where $G_1 = \{h|s_{hh} \neq 0\}$, and $G_2 = \{h|s_{hh} = 0\}$, and $n1$ is the number of activities of setup group in $G1$, i.e., $n1 = |\{a_i|a_i \in \Omega \wedge gr(a_i) \in G1\}|$. $k1 = |G1|$, $k2 = |G2|$, with $k1 + k2 = k$.

Let 0 be a dummy node, and $s_{0h} = s_{h0} = 0$, for $h = 1, \ldots, k$, and $s_{00} = \infty$. Let $setup_l(F1, F2, m, j)$ be the minimum cost path of length l from node 0 to node j covering all nodes in the set $F1$ exactly m times, and covering all nodes in the set F2 at least once.

$$
\begin{array}{ll}
setup_1(\{i\}, \emptyset, 1, i) = s_{0i} & \forall i \in G1 \\
setup_1(\emptyset, \{i\}, 0, i) = s_{0i} & \forall i \in G2 \\
setup_l(F1, F2, m, i) = \infty & \forall m < |F1|, \forall i \in G, \forall F1 \subseteq G1, \forall F2 \subseteq G2 \\
setup_l(\emptyset, F2, m, i) = \infty & \forall m > 0, \forall i \in F2, \forall F2 \subseteq G2 \\
setup_l(F1, F2, m, i) = \infty & \forall m > l, \forall i \in G, \forall F2 \subseteq G2, \forall F1 \subseteq G1 \\
setup_l(F1, F2, m, j) = \min\{ & \\
\qquad \min_{i \in F1 \cup F2}\{setup_{l-1}(F1, F2, m-1, i) + s_{ij}\}, & \forall j \in F1, \forall F1 \subseteq G1, \forall F2 \subseteq G2, \\
\qquad \min_{i \in F1 \cup F2}\{setup_{l-1}(F1 \setminus \{j\}, F2, m-1, i) + s_{ij}\}\} & \forall l = m + |F1|, \ldots, m + |F1|^2 \\
setup_l(F1, F2, m, j) = \min\{ & \\
\qquad \min_{i \in F1 \cup F2 \setminus \{j\}}\{setup_{l-1}(F1, F2 \setminus \{j\}, m, i) + s_{ij}\}, & \forall j \in F2, \forall F1 \subseteq G1, \forall F2 \subseteq G2, \\
\qquad \min_{i \in F1 \cup F2 \setminus \{j\}}\{setup_{l-1}(F1, F2, m, i) + s_{ij}\}\} & \forall l = m + |F1|, \ldots, m + |F1|^2
\end{array}
$$

The optimal solution $Setup^{n1}(G1, G2)$ is given by:

$$
Setup^{n1}(G1, G2) = \min_{i \in G, n1 + k2 \leq l \leq n1 + k2^2} \{setup_l(G1, G2, n1, i)\}
$$

In this case, instead of storing two matrices $setup^c_{hi}$ and $setup^e_{hi}$, we store $setup^c_{h1,h2,i}$ and $setup^e_{h1,h2,i}$ for all possible values of $m(S)$ in $Setup^{m(S)}(G1(S), G2(S))$. Since $m(S) = 0, \ldots, n1$, we need to store ($2\ k\ k1\ k2\ n1$) values $setup^c_{h1,h2,i,m}$ and $setup^e_{h1,h2,i,m}$. After each computation of $setup_l(F1, F2, m, j)$, with $|F1| = h1$, $|F2| = h2$, we update $setup^c_{h1,h2,i,m}$, and $setup^e_{h1,h2,i,m}$ as follows:

$$
\begin{array}{ll}
\forall i \in F1 \cup F2, & setup^c_{|F1|,|F2|,i,m} = \min\{setup^c_{|F1|,|F2|,i,m}, setup_l(F1, F2, m, j)\} \\
\forall i \in G \setminus (F1 \cup F2) & setup^e_{|F1|,|F2|,i,m} = \min\{setup^e_{|F1|,|F2|,i,m}, setup_l(F1, F2, m, j)\}
\end{array}
$$

The complexity of calculating $Setup^{n1}(G1, G2)$ is $O(\max\{n1, k2^2\}2^k)$, although, in practice, the effort for computing it can be reduced by considering dominance properties as in the previous cases.

The bound based on the state space relaxation is the following: we define $setup^c_l(m, i, j)$ as the shortest path from node 0 to node j passing through l nodes before j, m of which belongs to the set $G1$, $l - m$ belongs to $G2$, and containing node i; loops on nodes in $G2$ are not allowed. We define $setup^e_l(m, i, j)$ as the shortest path from node 0 to node j passing through l nodes before j, m of which belongs to the set $G1$, $l - m$ belongs to $G2$, and not containing node i; loops on nodes in $G2$ are not allowed.

$$\begin{array}{ll}
setup_0^c(0,i,j) = \infty & \forall i,j \in G \\
setup_0^e(0,i,j) = s_{0j} & \forall i,j \in G \\
setup_l^c(m,i,j) = \infty & \forall i,j \in G, l < m \\
setup_l^e(m,i,j) = \infty & \forall i,j \in G, l < m \\
setup_l^c(m,i,j) = \min\{\min_{q \in G}\{setup_{l-1}^c(m-1,i,q) + s_{q,j}\}, & \\
\qquad\qquad setup_{l-1}^e(m-1,i,i) + s_{ij}\} & \forall l = m,\dots,m+k^2, \forall i \in G, j \in G1 \\
setup_l^c(m,i,j) = \min\{\min_{q \in G\setminus\{j\}}\{setup_{l-1}^c(m,i,q) + s_{qj}\}, & \\
\qquad\qquad setup_{l-1}^e(m,i,i) + s_{ij}\} & \forall l = m,\dots,m+k^2, \forall i \in G, j \in G2, i \neq j \\
setup_l^c(m,i,i) = \min_{q \in G\setminus\{i\}}\{setup_{l-1}^c(m,i,q) + s_{qi}\} & \forall l = m,\dots,m+k^2, \forall i \in G2 \\
setup_l^e(m,i,j) = \min_{q \in G\setminus\{i\}}\{setup_{l-1}^e(m-1,i,q) + s_{qj}\} & \forall l = m,\dots,m+k^2, \forall i \in G, \forall j \in G1 \\
setup_l^e(m,i,j) = \min_{q \in G\setminus\{i,j\}}\{setup_{l-1}^e(m,i,q) + s_{qj}\} & \forall l = m,\dots,m+k^2, \forall i \in G, \forall j \in G2
\end{array}$$

A lower bound on $Setup^{n1}(G1,G2)$ is:

$$LB(Setup^{n1}(G1,G2)) = \max_{i \in G}\{\min_{k2+n1 \leq l \leq n1+k2^2, j \in G}\{setup_l^c(n1,i,j)\}\}$$

In this case we store $setup_{h2,i,m}^c$ and $setup_{h2,i,m}^e$ for $h2 \in G2$, $i \in G$, and $m = 0,\dots,n1$. The values $setup_{h2,i,m}^c$, and $setup_{h2,i,m}^e$ are calculated as:

$$\begin{array}{ll}
setup_{h2,i,m}^c = \min\{ & \\
\qquad \min_{j \in G1, m+h2 \leq l \leq m+h2^2}\{setup_{l-1}^c(m-1,i,j)\}, & \\
\qquad \min_{j \in G2, m+h2 \leq l \leq m+h2^2}\{setup_{l-1}^c(m,i,j)\}\} & \forall m = 0,\dots,n1, h2 \in G2, i \in G \\
setup_{h2,i,m}^c = \min\{setup_{h2,i,m}^c, setup_{l-1}^e(m-1,i,i)\} & \forall m = 0,\dots,n1, h2 \in G2, i \in G1 \\
setup_{h2,i,m}^c = \min\{setup_{h2,i,m}^c, setup_{l-1}^e(m,i,i)\} & \forall m = 0,\dots,n1, h2 \in G2, i \in G2 \\
setup_{h2,i,m}^e = \min\{ & \\
\qquad \min_{j \in G2\setminus\{i\}, m+h2 \leq l \leq m+h2^2}\{setup_{l-1}^e(m,i,j)\}, & \\
\qquad \min_{j \in G1\setminus\{i\}, m+h2 \leq l \leq m+h2^2}\{setup_{l-1}^e(m-1,i,j)\}\} & \forall m = 0,\dots,n1, h2 \in G2, i \in G
\end{array}$$

The State Space Relaxation and every value $setup_{h2,i,m}^c$ and $setup_{h2,i,m}^e$ can be computed in $O(\max\{k1^2, n2\}\ n2\ k^3)$; nevertheless, the average complexity can be reduced by considering dominance properties.

Bounds for $Setup^{\infty}(G(S), G')$, and $Setup^{m(S)}(G1(S), G2(S), G')$

Similar bounds can be found for $Setup^{\infty}(G(S), G')$, and $Setup^{m(S)}(G1(S), G2(S), G')$; the Dynamic Programming recursions for these two formulations will not be presented since they are similar to the previous ones and do not bring any added value to this study.

6.2.7 Including the time dimension

In case hypotheses 1 and 2 hold, the value of $PS(S)$ is calculated as $PS(S) = pt(S) + Setup(S)$. The optimal sequence of activities in S with respect to the minimization of sum of setup times, schedules all activities belonging to the same group one after the other. Therefore, if it is possible to prove that the set of all activities belonging to the same group cannot be sequenced one after the other, then the minimum cost of braking this sequence can be added to $Setup(S)$.

Let $S_h \subseteq S$ be the subset of S containing all activities a_i of type h, i.e., $S_h = \{a_i \in S | gr(a_i) = h\}$. If there exists an activity $a \in S \setminus S_h$ such that:

$$(est(S_h) + pt(S_h) > lct(a) - pt(a)) \wedge (est(a) + pt(a) > lct(S_h) - pt(S_h))$$

then we know that the set of all activities in S_h cannot be scheduled sequentially. In this case, since the hypothesis 2 holds, we can add a cost of breaking the chain equal to $s_{h,gr(a)}$. For each type h, we define $breakCost_h = 0$. For each activity a that enters the set S, we test if a breaks a set S_h, for each $h \neq gr(a)$, in this case we update $breakCost_h = \max\{breakCost_h, s_{h,gr(a)}\}$. In Theorem 2, the term $sum_{h=1}^{k} breakCost_h$ can be added to $Setup(G(S))$.

While all the bounds presented for evaluating the sum of setup times in $PS(S)$ can be computed in a preprocessing step, and do not increase the complexity of the propagation algorithm, the method presented here requires that each activity a entering the set S is tested against the current values of $est(S_h) + pt(S_h)$, $lct(S_h) - pt(S_h)$ for each $h \in G$. Therefore the complexity of the propagation algorithm is increased by a factor k.

6.2.8 Conclusion and Future work

Starting from the work of Nuijten [125], and Brucker and Thiele [28] we propose a more generic extension of the edge finding techniques presented by Carlier and Pinson [31]. The generalization proposed is applicable to the constraint-based scheduling framework described in [125], and aims at improving the propagation for scheduling problems with sequence dependent setup times. We discuss several hypotheses that enable to simplify these techniques, and propose different methods to calculate lower bounds on the sum of setup times used in the enhanced version of the algorithm proposed in [125].

The different lower bounds proposed have the characteristic that the number of different values of each bound is polynomial in the size k of the setup matrix and at most linear in the size of the problem. All the possible values of the lower bounds are computed once for all, and stored in a matrix. The use of the precomputed values leads to a propagation algorithm having the same complexity ($O(n^2)$) as the algorithm for scheduling problems without setup times proposed in [125].

For each formulation of the sum of setup times, we propose a bounds with exponential complexity, and a weaker one with polynomial complexity. Since in both cases the bounds are computed in a preprocessing step, the exponential bound appears very interesting especially in all cases in which the same setup matrix applies to several instances of scheduling problems. In these case, the bounds can be computed only once, and become input data for all the problem instances. Finally we propose a method able to consider also the time dimension leading to a better evaluation of the sum of setups in the set of activities S.

6.3 Position based scheduling

6.3.1 Introduction

In recent years Constraint Programming (CP) techniques have been successfully used to model and solve scheduling problems. Combining ideas from Artificial Intelligence (AI) and Operations Research (OR), constraint-based scheduling was able to maintain the best of both approaches, i.e., the modeling flexibility of AI scheduling systems and the efficiency of OR algorithms [124].

One of the most interesting and successful examples of integration of OR techniques in CP is the use of the techniques first presented by Carlier and Pinson [31] in constraint-based scheduling [125]. The subsequent research on improvements, additions, and generalizations of these techniques brought the two scheduling communities in AI and OR closer together, up to a point that many feel there is no longer a clear separation. In this paper we propose an integration of AI and OR techniques by making usage of methods developed within the OR community reinterpreting them in Constraint Programming.

Roughly speaking, the techniques we discuss here introduce a *position* variable for each activity on a given unary resource (resource of maximal capacity equal to one), and compute *conditional* bounds for the start and end time of one activity if it is scheduled in each possible position. Different methods to calculate the conditional bounds are proposed, taking into account release dates, deadlines, and sequence dependent setup times. Conditional bounds are extremely important information: they can be used to calculate lower bounds for a variety of objective functions such as sum of weighted tardiness and earliness, max weighted tardiness and earliness, sum of idle times, max of idle times, sum of setup times and costs, and makespan; these bounds can be used to reduce the search space, and to help in making good heuristic choices. Moreover, the position variables can be used as decision variables opening to new, and very promising branching techniques as well as local search moves.

Although we are interested in general shop scheduling problems, we concentrate on the 1-machine scheduling problem where we have a machine μ and a set A of n activities $a_1, \ldots, a_n$ all to be processed on μ for $pt(a_i)$ time units ($i = 1 \ldots n$). The activities may have release dates ($est(a_i)$) and deadlines ($lct(a_i)$). Sequence dependent setup times may exist among activities; given a setup time matrix s (square matrix of dimension n), s_{ij} represents the setup time between activities a_i and a_j, if a_i and a_j are scheduled sequentially on machine μ with a_i first. In such a case, $start(a_j) \geq end(a_i) + s_{ij}$.

6.3.2 Conditional Variables

This section reviews the concept of *Conditional Variable* introduced in Section 3.5.6, and used to formulate Dynamic Programming models for the traveling salesman problem.

A *Conditional Variable* $cVar$ is a variable depending on a constraint; it is defined by a the pair $cVar = (Dom, cst)$ where Dom is a domain of possible values, and cst is a constraint. The semantic of a conditional variable is the following:

$$cst \Leftrightarrow Dom \neq \emptyset$$

A conditional variable $cVar = (Dom, cst)$ is defined *true* if its conditioning constraint cst is true. It is defined *false* if its conditioning constraint cst is false. The constraint cst of $cVar$ is identified by $\mathtt{cst}(cVar)$.

The concept of conditional variables has been used in constraint programming for instance to implement constructive disjunctions [89]. In the following we will present some examples of constraints that could be defined using conditional variables.

The equality constraint can be defined between one conditional variable and a variable, and between two conditional variables: let $cVar = (Dom, cst)$ be a conditional variable, and var be a variable. The constraint $cVar = var$ holds if cst is false or if cst is true and the two variables assume the same value.

The equality constraint could propagate as follows:

$$\begin{aligned} i \notin \texttt{domain}(var) &\Rightarrow cVar \neq i,\ \forall i \in \texttt{domain}(cVar) \\ cst \Rightarrow (i \notin \texttt{domain}(cVar) &\Rightarrow var \neq i,\ \forall i \in \texttt{domain}(var)) \end{aligned}$$

Similarly, let $cVar_1 = (Dom_1, cst_1)$, and $cVar_2 = (Dom_2, cst_2)$ be two conditional variables. The constraint $cVar_1 = cVar_2$ holds if $cst_1 \wedge cst_2$ is false or if $cst_1 \wedge cst_2$ is true and the two variable assume the same value.

Conditional variables can be combined via arithmetical operators: for instance, two conditional variables $cVar_1 = (Dom_1, cst_1)$, and $cVar_2 = (Dom_2, cst_2)$ can be combined as $cVar_3 = cVar_1 + cVar_2$, where $cVar_3$ is defined as $cVar_3 = (Dom_3, cst_1 \wedge cst_2)$, where Dom_3 is defined by the usual propagation rule of the sum operator.

Constructive disjunction (in this setting we are only interested in exclusive disjunction) can be defined by means of the following global constraint: $xunion(cVars, y, x)$ where x, y are regular variables, and $cVars$ is an array of k conditional variables. Exactly one out of the k conditional variables $cVars[i]$ is a true variable (say $cVars[i^*]$), and the index i^* of the only true variable $cVars[i^*]$ is equal to x ($x = i^* \mid \texttt{cst}(cVars[i^*])$). Formally, the constraint $xunion(cVars, y, x)$ holds iff:

$$\begin{aligned} XOR_{i=1,\dots,k}\texttt{cst}(cVars[i]) \\ cVars[x] = y \end{aligned}$$

From the definition, it is easy to show that $\texttt{cst}(cVars[i])$ is equivalent to $x = i$, for all $i = 1, \dots, k$. The following propagation rules can be enforced on the bounds of the variables[7]:

$$\texttt{inf}(y) \geq \min_{i \in \texttt{domain}(x)} \{\texttt{inf}(cVars[i])\} \tag{6.1}$$

$$\texttt{sup}(y) \leq \max_{i \in \texttt{domain}(x)} \{\texttt{sup}(cVars[i])\} \tag{6.2}$$

$$\forall i \in \texttt{domain}(x) \quad \texttt{inf}(cVars[i]) \geq \texttt{inf}(y) \tag{6.3}$$

$$\forall i \in \texttt{domain}(x) \quad \texttt{sup}(cVars[i]) \leq \texttt{sup}(y) \tag{6.4}$$

$$\forall i \in \texttt{domain}(x) \quad \texttt{inf}(y) > \texttt{sup}(cVars[i]) \Rightarrow x \neq i \tag{6.5}$$

$$\forall i \in \texttt{domain}(x) \quad \texttt{sup}(y) < \texttt{inf}(cVars[i]) \Rightarrow x \neq i \tag{6.6}$$

$$\forall i = 1, \dots, k \quad i \notin \texttt{domain}(x) \Leftrightarrow \neg\texttt{cst}(cVars[i]) \tag{6.7}$$

$$\forall i = 1, \dots, k \quad i = \texttt{value}(x) \Leftrightarrow \texttt{cst}(cVars[i]) \tag{6.8}$$

[7]Similar rules can be enforced considering the entire domain of the variables, instead of the bounds.

In Section 3.5.6 we also proposed a generalization of the constraint $xunion(cVars, y, x)$ providing a constructive (exclusive) disjunction between a conditional variable and an array of conditional variables: x, y are conditional variables, and $cVars$ is an array of k conditional variables. This constraint has the same semantic as the previous one if $\texttt{cst}(x) \wedge \texttt{cst}(y)$ holds. Formally, the constraint $xunion(cVars, y, x)$ holds iff:

$$(\texttt{cst}(x) \wedge \texttt{cst}(y)) \Rightarrow (XOR_{i=1,\ldots,k}\texttt{cst}(cVars[i]) \wedge cVars[x] = y)$$

In this discussion we are interested in the special case where both x and y are conditioned by the same constraint, i.e., $\texttt{cst}(x) = \texttt{cst}(y)$. As shown in Section 6.3.4, this is the case of dynamic programming models; in general, Dynamic Programming formulations link a conditional variable y to arrays of conditional variables $vars$ through a transition function represented by a conditional variable x; the transition function conditional variable x, and the conditional variable y share the same conditioning constraint.

When the $xunion(cVars, y, x)$ constraint is stated on conditional variables x, y, and x and y are conditioned by the same constraint, the following propagation rules can be enforced on the bounds of the variables:

$$\texttt{inf}(y) \geq \min_{i \in \texttt{domain}(x)} \{\texttt{inf}(cVars[i])\} \tag{6.9}$$

$$\texttt{sup}(y) \leq \max_{i \in \texttt{domain}(x)} \{\texttt{sup}(cVars[i])\} \tag{6.10}$$

$$\forall i \in \texttt{domain}(x) \quad \texttt{cst}(y) \Rightarrow \texttt{inf}(cVars[i]) \geq \texttt{inf}(y) \tag{6.11}$$

$$\forall i \in \texttt{domain}(x) \quad \texttt{cst}(y) \Rightarrow \texttt{sup}(cVars[i]) \leq \texttt{sup}(y) \tag{6.12}$$

$$\forall i \in \texttt{domain}(x) \quad \texttt{inf}(y) > \texttt{sup}(cVars[i]) \Rightarrow x \neq i \tag{6.13}$$

$$\forall i \in \texttt{domain}(x) \quad \texttt{sup}(y) < \texttt{inf}(cVars[i]) \Rightarrow x \neq i \tag{6.14}$$

$$\forall i = 1, \ldots, k \quad \texttt{cst}(y) \Rightarrow (i \notin \texttt{domain}(x) \Rightarrow \neg\texttt{cst}(cVars[i])) \tag{6.15}$$

$$\forall i = 1, \ldots, k \quad \neg\texttt{cst}(cVars[i]) \Rightarrow x \neq i \tag{6.16}$$

$$\forall i = 1, \ldots, k \quad \texttt{cst}(y) \Rightarrow (i = \texttt{value}(x) \Rightarrow \texttt{cst}(cVars[i])) \tag{6.17}$$

$$\forall i = 1, \ldots, k \quad \texttt{cst}(cVars[i]) \Rightarrow x = i \tag{6.18}$$

6.3.3 Modeling activities using conditional variables

Each activity a in constraint based scheduling is represented by using two variables being its start time $start(a) := [est(a)..lst(a)]$ and its end time $end(a) := [ect(a)..lct(a)]$, where $est(a)$ and $lst(a)$ are the earliest and latest start time of a, and $ect(a)$ and $lct(a)$ are the earliest and latest end (completion) time, respectively. These two variables are constrained by the relation $end(a) - start(a) = pt(a)$. We propose the introduction of a variable $pos(a)$ representing the position of activity a on the resource μ; for instance, the activity a having $pos(a) = 1$ is the first activity scheduled in μ, an activity a having $pos(a) = [4..6]$ has at least three, at most five other activities preceding it. We will see that the newly introduced position variables play a key role in the proposed scheduling model.

For each activity a, $2n$ conditional variables $start(a,p)$, and $end(a,p)$ represent the start and the end time of a if a is scheduled in position p (with $end(a,p) - start(a,p) = pt(a)$). The conditioning constraint is clearly $(pos(a) = p)$. Variables $start(a,p)$, $start(a)$, and $pos(a)$

are linked by the constructive disjunctive constraint $xunion((start(a,p))_p, start(a), pos(a))$, which propagates as described in Section 6.3.2.

For each activity a, the domain variable $prev(a)$ identifies the activity directly preceding a in resource μ (see, e.g. [95] for details on the propagation rules linking $prev(a)$ and the variables $start(a)$, $end(a)$); the conditional variable $prev(a,p)$ (conditioned by $(pos(a) = p)$) identifies the activity directly preceding a in resource μ if a is in position p. Variables $prev(a,p)$, $prev(a)$ and $pos(a)$ are linked by the constructive disjunction $xunion((prev(a,p))_p, prev(a), pos(a))$.

For each activity a, the domain variable $idle(a)$ identifies the idle time[8] between activity a, and $prev(a)$, directly preceding a in resource μ; the conditional variable $idle(a,p)$ (conditioned by $(pos(a) = p)$) identifies the idle time between activities a, and $prev(a)$ if a is in position p. Variables $idle(a,p)$, $idle(a)$ and $pos(a)$ are linked by the constructive disjunction $xunion((idle(a,p))_p, idle(a), pos(a))$.

For each activity a, the domain variable $setupTime(a)$ (resp. $setupCost(a)$) identifies the sum of setup times (resp. costs) from the beginning of the schedule to activity a. The conditional variable $setupTime(a,p)$ (resp. $setupCost(a,p)$) is conditioned by $(pos(a) = p)$ and identifies the sum of setup times (resp. costs) from the beginning of the schedule to activity a if a is in position p. Variables $setupTime(a,p)$, $setupTime(a)$ and $pos(a)$ are linked by the constructive disjunction $xunion((setupTime(a,p))_p, setupTime(a), pos(a))$. Similarly, variables $setupTime^{-1}(a)$, $setupCost^{-1}(a)$, $setupTime^{-1}(a,p)$, and $setupCost^{-1}(a,p)$, identifying the sum of setup times and costs from activity a and the end of the schedule, can be defined.

While the variables $start(a)$, $end(a)$, $pos(a)$, and the conditional variables $start(a,p)$, and $end(a,p)$ are always defined, we are not forced to define all other variables proposed; only the subsets that are interesting for the problem at hand should, indeed, be defined.

The $start(a,p)$ and $end(a,p)$ variables are particularly important, and form the basis for the subsequent calculation of lower bounds for a variety of objective function. In the following we propose different methods to calculate bounds for the variables $start(a,p)$, and $end(a,p)$.

6.3.4 Calculating bounds on conditional activities

In the following only the bounds for the $start(a,p)$ will be considered. All the methods proposed can also be considered in their symmetric version and be used to update $end(a,p)$.

Bounds from the "real" variables

The constructive disjunction $xunion((start(a,p))_p, start(a), pos(a))$ propagates the bounds of $start(a)$ on every possible variable $start(a,p), \forall p \in \mathtt{domain}(pos(a))$ (see the propagation of $xunion$, from (6.1) to (6.8)).

$$\forall p \in \mathtt{domain}(pos(a)) \qquad \mathtt{inf}(start(a,p)) \geq \mathtt{inf}(start(a)) \tag{6.19}$$

$$\forall p \in \mathtt{domain}(pos(a)) \qquad \mathtt{sup}(start(a,p)) \leq \mathtt{sup}(start(a)) \tag{6.20}$$

The time complexity of updating the bounds of all the conditional start and end variables is $O(n^2)$; moreover the domain of variable $pos(a)$ is reduced as soon as the domain of a conditional variable becomes empty.

[8]We consider idle time between two activities a_i, a_j, with a_i directly preceding a_j, the value $idle(a_j) = start(a_j) - end(a_i) - s_{ij}$.

Bounds from subsets of activities

Given an activity a, and a set of activities $Pred(a)$ that are possible predecessors of a, we want to calculate the earliest start time of a if it is in position p. Let $ect_{Pred(a),p-1} = min_{S \subseteq Pred(a),|S|=p-1}\{ect(S)\}$, where $ect(S)$ is the optimal makespan of the activities in S sequenced in machine μ. Clearly, $start(a,p) \geq ect_{Pred(a),p-1}$. In the following we propose an algorithm to calculate a lower bound on $ect_{Pred(a),p}$ for all $p = 0, \dots, n-1$; the algorithm is based on the following idea: the deadline of every activity in $Pred(a)$ is relaxed; in this case we show that all values $ect_{Pred(a),p}$, for $p = 0, \dots, n-1$ can be incrementally calculated through a Dynamic Programming recursion.

Let $\mathcal{J} = \{1, 2, \cdots, m\}$ a set of m tasks that are to be processed on a single machine. Each task $o_j \in \mathcal{J}$ has a processing time $pt(o_j)$ and a release date $est(o_j)$. We can assume, without loss of generality, that the tasks are sorted in the order of their release dates. For any set $S \subseteq \mathcal{J}$, the earliest completion time of S, denoted by $ect(S)$, is given by the makespan of the tasks of S sequenced according to their release dates. Let $\mathcal{J}_j = \{1, 2, \cdots, j\}$ and T_{kj} be the earliest completion time of a set of k tasks included in $\mathcal{J}_j$, i.e., $T_{kj} = \min_{S \subseteq \mathcal{J}_j, |S|=k}\{ect(S)\}$. Conventionally, we set:

$$\forall 1 \leq j \leq n \qquad T_{0j} = 0 \tag{6.21}$$

$$\forall j < k \qquad T_{kj} = \infty \tag{6.22}$$

Theorem 1 *For any k, j such that $1 \leq k \leq j \leq m$, we have:*

$$T_{kj} = \min\left(T_{k,j-1}, \max\left(r_j, T_{i-1,j-1}\right) + p_j\right) \tag{6.23}$$

Proof. Let k and j such that $1 \leq k \leq j \leq m$, and let S_{kj} be a set of k tasks included in $\mathcal{J}_j$ such that $ect(S_{kj}) = T_{kj}$. We will separately discuss the cases $j \notin S_{kj}$, and $j \in S_{kj}$. If $j \notin S_{kj}$ (which means that $j > k$), then $S_{kj} \subseteq \mathcal{J}_{j-1}$ so that $T_{kj} = T_{k,j-1}$. If $j \in S_{kj}$, $S_{kj} = S^* \cup \{j\}$ with $S^* \subseteq \mathcal{J}_{j-1}$ and $|S^*| = k-1$. If $k = 1$, $S^* = \emptyset$ and $T_{1j} = r_j + p_j$. If $k > 1$, we need to show that:

$$\forall S \subset \mathcal{J}_{j-1}, |S| = k-1 \;\; ect(S \cup \{j\}) \geq ect(S_{k-1,j-1} \cup \{j\})$$

In fact, if the previous relation is true, then clearly $ect(S^* \cup \{j\}) = ect(S_{k-1,j-1} \cup \{j\})$, and therefore we can choose $S^* = S_{k-1,j-1}$. For any $S \subseteq \mathcal{J}_{j-1}$, $ect(S \cup \{j\}) = \max(ect(S), r_j) + p_j$ since the optimal makespan for $S \cup \{j\}$ is obtained by scheduling j last. Since S contains $k-1$ tasks, $ect(S) \geq ect(S_{k-1,j-1})$, which proves that $ect(S \cup \{j\}) \geq ect(S_{k-1,j-1} \cup \{j\})$. So, if $j \in S_{kj}$, $T_{kj} = \max(r_j, T_{i-1,j-1}) + p_j$. It is easy to see that with the limit conditions (6.21) and (6.22), the equation is valid even for $k = 1$ and for $k = j$.

Using this theorem, with the limit conditions (6.21) and (6.22) all the values T_{kj} can be computed by dynamic programing in $O(m^2)$ time and $O(m^2)$ space. Given a set $Pred(a)$ of possible predecessors of an activity a, we build the set $\mathcal{J} = \{1, \dots, m\} = Pred(a)$ sorted by increasing earliest start time; we compute $T_{p,m}$, for $p = 0, \dots, m$ which is clearly a lower bound for $start(a, p+1)$. Therefore we set:

$$\forall p \in \texttt{domain}(pos(a)) \qquad \texttt{inf}(start(a,p)) \geq T_{p-1,|Pred(a)|} \tag{6.24}$$

The time complexity of updating the bounds of all the conditional activity is $O(n^3)$ since the bounds $start(a,p)$, for $p = 1, \ldots, n$ of each activity a can be updated in $O(n^2)$ (which is the complexity of calculating $T_{p,m}$, for $p = 1, \ldots, n$). In practice, the average complexity can be reduced by calculating $T_{p,m}(a_2)$ corresponding to $Pred(a_2)$ using the information already calculated for $T_{p,m}(a_1)$ corresponding to $Pred(a_1)$. In fact, for many activities, the sets of possible predecessors are very similar. Note that the propagation rule (6.24) imposes $\texttt{inf}(start(a,p)) \geq \infty$ (i.e., $pos(a) \neq p$) for every p such that $p > |Pred(a)| + 1$ since $T_{p-1,|Pred(a)|} = \infty$ for $p - 1 > |Pred(a)|$.

A straightforward generalization of the Theorem proposed could consider the fact that some activities in $Pred(a)$ are *necessary* predecessors of a. In a future work, we plan to generalize the approach to take into consideration sequence dependent setup times.

Bounds from State Space Relaxation

The bounds based on subsets of activities consider the fact that if activity a is in position p, then $p-1$ *different* activities precede a. Nevertheless, the relative position of the $p-1$ activities preceding a is ignored, as well as their deadlines, and the sequence dependent setup times. Here we propose an adaptation of a method already used in [42, 116] for Traveling Salesman Problems with Time Window, and based on Dynamic Programming State Space Relaxation (see also Sections 3.5.6).

The conditional variable $prev(a_i,p)$ represents a possible transition between the two activities $a_j = prev(a_i,p)$, and a_i. The relation between the (start and end) conditional variables of a_j, and a_i is:

$$start(a_i,p) - idle(a_i,p) = end(a_j,p-1) + s_{ji}$$

this relation[9] leads to a CP based state space relaxation using the constructive disjunction constraint on conditional variables.

$$\forall i, p = 2, \ldots, n-1 \;\; xunion((end(a_j,p-1) + s_{ji})_j, (start(a_i,p) - idle(a_i,p)), prev(a_i,p))$$

In particular, the bounds of $start(a_i,p)$ are updated by the $xunion$ constraint as follows (see Section 6.3.2 for the complete set of propagation rules of the $xunion$ constraint):

$$(6.25) \;\; \texttt{inf}(start(a_i,p)) \geq \min_{a_j \in \texttt{domain}(prev(a_i,p))} \{\texttt{inf}(end(a_j,p-1) + s_{ji})\} + \texttt{inf}(idle(a_i,p))$$

$$(6.26) \;\; \texttt{sup}(start(a_i,p)) \leq \max_{a_j \in \texttt{domain}(prev(a_i,p))} \{\texttt{sup}(end(a_j,p-1) + s_{ji})\} + \texttt{sup}(idle(a_i,p))$$

The bound of all $start(a,p)$ conditional variables can be updated by triggering the propagation of all $xunion$ constraints in the correct order in $O(n^3)$. Moreover, the use of Constraint Programming enables to easily incrementally recalculate the bounds after modification of some $start(a,p)$.

[9]The idle time could be omitted if it is uninteresting by adapting the propagation of the constructive disjunction constraint.

Bounds from extended State Space Relaxation

The bounds based on State Space Relaxation proposed use the "prev" relation to represent the transition between two states. We could imagine to use a different transition function instead. Consider the following DP formulation:

$$start(a_i, p) \geq \min_{a_j \in \texttt{domain}(prev_k(a_i,p))} \{end(a_j, p-k) + \Delta_k^*\}$$

stating that the start of activity a_i in position p is greater or equal to the end of the activity $a_j = prev_k(a_i, p)$ in position $p - k$ plus the time Δ_k^* used by all other activities scheduled between a_j and a_i. Clearly, if $k = 1$, this DP formulation becomes the one proposed in the previous section. The variable $prev_k(a_i, p)$ identifies the activity a_j that precedes a_i with a distance of k activities.

In the following we propose a method to update the bounds of all $start(a, p)$ conditional variables that is based on a relaxation of this DP formulation for $k = 1, \dots, n - 1$.

Before describing the method we need to introduce the definitions of ranked and unranked activities. An activity a is said to be *ranked first* if its position is known, and there is a known sequence of $pos(a) - 1$ activities a_i having known position $pos(a_i) < pos(a)$. An activity a is said to be *ranked last* if its position is known, and there is a known sequence of $n - pos(a)$ activities a_i having known position $pos(a_i) > pos(a)$. Let Urk be the set of activities in A that are still unranked (i.e. the set of activities that are neither ranked first nor last). Let p_{min}, (resp. p_{max}) be the minimal (resp. maximal) position available for activities in Urk.

We define $eMin_p = \min_{a_i \in Urk}\{\texttt{inf}(end(a_i, p))\}$ the minimal end time among all unranked activities if scheduled in position p (for $p = p_{min}, \dots, p_{max}$); let $(actPt_h)_h$ be the array of activities $a_i \in Urk$ sorted by increasing processing time; we define $\Delta_0 = 0$ and $\Delta_k = \sum_{h=1,\dots,k} pt(actPt_h)$ for all $k = 1, \dots, |Urk|$; Δ_k represents the sum of the processing times of the k smallest unranked activities.

It is easy to see that for every activity $a_i \in Urk$, and for every $p, p' = p_{min}, \dots, p_{max}$, with $p' < p$,

$$start(a_i, p) \geq eMin_{p'} + \Delta_{p-p'-1}$$

since it is clearly a relaxation of the DP formulation based on $prev_k(a_i, p)$. Since this relation is valid for every p', we calculate $sMin_p = \max_{p'=p_{min},\dots,p-1}\{eMin_{p'} + \Delta_{p-p'-1}\}$, which we use as a lower bound for $start(a_i, p)$. Therefore we can set:

$$\texttt{inf}(start(a_i, p)) \geq sMin_p \tag{6.27}$$

Since the arrays $(eMin_p)_p$, $(\Delta_k)_k$, and $(sMin_p)_p$ can all be calculated in $O(n^2)$, the complexity of updating all the bounds $start(a, p)$ is $O(n^2)$.

6.3.5 Calculating bounds on various objective functions

Based on the conditional variables $start(a, p)$ and $end(a, p)$, we can easily define a conditional variable representing the component of the cost function for activity a if it is scheduled in position p. For example, in a min sum tardiness problem, the conditional variable $tard(a_i, p)$ is defined as $tard(a_i, p) = w_i \max\{end(a_i, p) - dd_i, 0\}$, where dd_i, and w_i are the due date and the tardiness weight for activity a_i. The conditional variable $early(a_i, p)$ can be defined in a similar way. The idle time $idle(a,p)$ has been already defined in Section 6.3.4. The conditional variable $setupTime(a,p)$ (resp $setupCost(a,p)$) can be defined using the constructive disjunction constraint on conditional variables.

$$\forall i, p = 1, \dots, n \;\; xunion((setupTime(a_j, p-1) + s_{j,i})_j, setupTime(a_i, p), prev(a_i, p))$$

In general, let $cost(a_i, p)$ be the conditional variable representing the component of the cost function for activity a_i if it is scheduled in position p. Depending on the problem, $cost(a_i, p)$ can be $tard(a_i, p)$, $setupTime(a_i, p)$, etc. or even a linear combination of them. We consider here two types of objective functions: a min sum cost objective

$$sumCost = \sum_{i=1,\dots,n} cost(a_i, pos(a_i))$$

and a min max cost objective

$$maxCost = \max_{i=1,\dots,n} \{cost(a_i, pos(a_i))\}$$

In every feasible solution each activity will be scheduled in a different position. This consideration allows us to use an $allDiff$ constraint among all position variables, and, more important, allows us to use global optimization constraints (see Section 3, and [67]) to calculate bound on the objective functions and to perform cost based filtering.

The optimization constraint on variables $pos(a)$, and $sumCost$ will be based on a Min Sum Assignment Problem (AP) (as in Section 3, and [67]). The Min Sum Assignment Problem finds the assignment of positions to activities (each activity in a different position) such that the sum of assignment costs is minimal. The assignment cost c_{ip} considered in the computation of the AP are the lower bounds of the conditional cost variables $cost(a_i, p)$, i.e., $c_{ip} = \texttt{inf}(cost(a_i, p))$. The AP computation gives (i) a lower bound LB on the $sumCost$ variable, (ii) the optimal relaxed solution, (iii) the reduced costs rc_{ip} representing the additional cost to be paid over the lower bound LB if activity a_i is scheduled in position p. The reduced costs rc_{ip} can be used to define a gradient function $grad(i,p) = rc_{ip}$ as in [67].

The optimization constraint on variables $pos(a)$, and $maxCost$ will be based on a Bottleneck Assignment Problem (B-AP) [34] (as in [42, 116]). The Bottleneck Assignment Problem finds the assignment of position to activities (each activity in a different position) such that the maximum among the assignment costs is minimal. The assignment cost c_{ip} considered in the computation of the B-AP are the lower bounds of the conditional cost variables $cost(a_i, p)$, i.e., $c_{ip} = \texttt{inf}(cost(a_i, p))$. The B-AP computation gives (i) a lower bound LB on the $maxCost$ variable, (ii) the optimal relaxed solution, (iii) the reduced costs rc_{ip} representing the additional cost to be paid over the lower bound LB if activity a_i is scheduled in position p. The reduced costs rc_{ip} can be used to define a gradient function $grad(i,p) = rc_{ip}$ as in [67]. The complexity of solving a Bottleneck Assignment Problem is $O(n^3)$, while its average complexity is almost linear.

In the following we will only consider min sum problems using the AP as optimization component of the $allDiffCost$ constraint described in Section 3, but similar considerations hold for min max problems using the B-AP as optimization component of an $allDiffCostMax$. An $allDiffCost$ constraint calculates a lower bound LB on the total cost, and perform the gradient based propagation from the objective variable $sumCost$ to the position variables pos_i as shown in the following:

$$\text{(6.28)} \qquad \mathtt{inf}(sumCost) \geq LB$$
$$\text{(6.29)} \qquad \forall i, p = 1, \ldots, n \qquad LB + grad(i,p) \geq \mathtt{sup}(sumCost) \Rightarrow pos(a_i) \neq p$$

The gradient function can also be used to perform propagation from the objective variable $sumCost$ to the conditional variables $cost(a_i, p)$, which, in turn, propagate on the conditional variables $start(a_i, p)$, $end(a_i, p)$.

In fact, the upper bound of $cost(a_i, p)$ can be updated as follows:

$$\text{(6.30)} \forall i, p = 1, \ldots, n \qquad \mathtt{sup}(cost(a_i, p)) \leq \mathtt{inf}(cost(a_i, p)) + \mathtt{sup}(sumCost) - LB - grad(i,p)$$

The computation of the AP is performed in $O(n^3)$; while any recomputation of the AP due to modifications of the costs can be performed incrementally in $O(n^2)$ for each modified row or column in the cost matrix; the reduced costs are computed during the computation of the solution. Once the AP solution and the reduced costs are available, the propagation rules (6.28 - 6.30) update the domain of the variables in $O(n^2)$.

Note that the propagation rule (6.30) enable to update the $start(a)$, $end(a)$ bounds of activities by back-propagating the objective function variable to the conditional variables $cost(a, p)$, which propagate to the $start(a, p)$, $end(a, p)$, and finally to $start(a)$, $end(a)$. The propagation rule (6.30) could be improved by considering the fact that if the cost $cost(a, p)$ is a monotonic function of $start(a, p)$ (or of $end(a, p)$), an increase in the lower bound of the cost $cost(a_i, p)$ may generate an increase of the lower bound of the cost $cost(a_j, p')$ for every $a_j, j = 1, \ldots, n$, and $p' = 1, \ldots, n$ with $p' > p$. Consider, for example, a min sum tardiness problem where $cost(a_i, p) = w_i \max\{end(a_i, p) - dd_i, 0\}$; the goal it to find the maximal value of $end(a_i, p)$ such that the lower bound is still smaller or equal to the upper bound. In order to do that we want to evaluate the impact on the cost function of scheduling a_i in position p, with an increase in the lower bound of $cost(a_i, p)$ equal to $\Delta cost(a_i, p)$. The increase $\Delta cost(a_i, p)$ can only be generate by an increase, say of $\Delta end(a_i, p)$ of the conditional variable $end(a_i, p)$. But the increase of the lower bound of $end(a_i, p)$ implies also an increase in the lower bound of every conditional variable $end(a_j, p')$ for $j = 1, \ldots, n$ and $p' = 1, \ldots, n$, with $p' > p$. A method to effectively and efficiently exploiting these considerations is subject of current study.

6.3.6 Improving the bounds using the additive bounding technique

A further improvement on the lower bound can be obtained by using the additive bounding technique (see Section 1.4.2, and [61]). Suppose we are able to calculate $grad(i, p, j)$, as the additional cost to be paid over LB if a_i is scheduled in position p immediately before a_j (in position $p + 1$) for all $p = 1, \ldots n - 1$; and $grad(i, n - 1, j)$, as the additional cost to be paid over LB if a_i is scheduled as last activity and a_j as first one.

If $grad(i, p, j)$ is a *residual cost* (see Section 1.4.2, and [61]) and we can apply the following additive bounding procedure.

Knowing $grad(i, p, j)$ for all $i, j, p = 1, \ldots, n$, we define a cost matrix add as:

$$add_{ij} = \min_{p=1,\ldots,n} \{grad(i, p, j)\}$$

if $grad(i, p, j)$ is a *residual cost*, add_{ij} is also a *residual cost* representing the cost to be paid if activities a_i, and a_j are scheduled sequentially. Since each activity is followed by another activity (the last one is conventionally followed by the first one), the min sum scheduling problem can be relaxed into an Asymmetric Traveling Salesman Problem (ATSP). Any relaxation of the ATSP is a valid relaxation of the scheduling problem; in particular, a good relaxation of an ATSP is, again, a min sum Assignment Problem.

By solving an AP on the variables $prev(a_j)$ with cost matrix add_{ij} we obtain an optimal AP solution of value $LB1$. This value can be added to the value LB, and $LB + LB1$ is a valid lower bound for the original problem. Moreover, the reduced cost $addRc_{ij}$ can be used to filter suboptimal assignments of value for the variables $prev(a_j)$ as in [67].

A definition of $grad(i, p, j)$ such that $grad(i, p, j)$ is a residual cost is the following:

$$\begin{array}{rl} \forall i, j, p = 1, \ldots, n, p \neq n & grad(i, p, j) = (rc_{ip} + rc_{j,p+1})/2 + \delta(ipj) \\ \forall i, j = 1, \ldots, n & grad(i, n, j) = (rc_{in} + rc_{j0})/2 \end{array}$$

where rc_{ip} are the reduced cost of the activity/position assignment problem, and $\delta(ipj)$ is the increment over $\texttt{inf}(cost(a_i, p))$ if j directly follows i.

The additive bounding procedure can be computed in $O(n^3)$. Preliminary computational experience shows that the value $LB1$ does not increase the value LB by more than 5%. For this reason, the extra computation could be triggered only when $(UB - LB)/LB$ is smaller than a threshold that could be set equal to 5%.

6.3.7 Branching on position variables: a CSP approach to scheduling

One advantage of the proposed model is the fact that the position variables $pos(a)$ can be used as decision variables for the scheduling problem opening the doors for new branching strategies in the style of branching strategies for Constraint Satisfaction Problems. It is, for example, extremely easy to implement a max-regret branching strategy on the position variables since for every value in the domain of every variable we have a good evaluation of the regret based on the gradient function $grad(a, p)$ of the $allDiffCost$ optimization constraint.

Ideas that are quite intuitive can very easily be implemented using the position variables; for example, we may want to decompose the scheduling problem in two "almost" independent areas: half of the n activities will be scheduled in a position $p \geq n/2$, and the other half in a position $p > n/2$. Another idea could be to do a dichotomic search on the position variables: at each node of the search tree the activity a having $pos(a)$ with the largest domain is selected, and its domain is split in two parts: being p^* the "middle" position of the domain, we could branch on $pos(a) \leq p^*$ OR $pos(a) > p^*$. The position belonging to the optimal solution of the assignment problem can be used as tentative value in the branch and bound.

A method that, in the preliminary computational results, demonstrated to be very effective is the following: in the model proposed the evaluation of the additional cost $grad(i,p)$ to be paid if activity a_i is scheduled in position p is available for every $i, p = 1, \ldots, n$. The idea is to use this information to try "exclude" bad solutions. In other words, we want to add a constraint to the model that will heuristically remove bad solutions. Let $MaxGrad$ be the maximum of $grad(i,p)$, i.e., $MaxGrad = \max_{i,p=1,\ldots,n}\{grad(i,p)\}$ calculated at the root node. Let B_i a set of positions p of activity a_i with "high" value of $grad(i,p)$

$$B_i = \{p | grad(i,p) \geq \alpha MaxGrad\}$$

for some $0 < \alpha < 1$. The constraint HC excluding bad solutions can be defined as follows:

$$pos_i \neq p, \ \forall i = 1, \ldots, n, \forall p \in B_i$$

Note that this constraint enforces heuristic decisions that could be incorrect, and could even exclude every optimal solution. On the other hand, the search space obtained after the addition of this constraint is a lot smaller than the search space of the original problem P, and the resulting problem $P \uplus HC$ can be quickly solved up to optimality by branch and bound. The optimal solution of $P \uplus HC$ is obviously a feasible solution of P and, in general, it has a very good quality.

Discrepancy based filtering

The heuristic method described can be used in a Discrepancy based tree search leading to a very interesting form of filtering. For every activity a_i, we have defined a set B_i of position p that are heuristically considered "bad" positions for a_i. Instead of removing the values in B_i from the domain of $pos(a_i)$ for all $i = 1, \ldots, n$, the following tree search is defined:

```
ILCGOAL2(preSearchGoal, ActivityArray, activities, IlcInt, index) {
    if (index = 0)
        return 0;
    Activity act = activities[index];
    IlcIntVar posVar = getPositionVar(act);
    IlcIntSet badPosSet = getBadPositionSet(act);
    return IlcAnd(IlcOr(remove(posVar,badPosSet),
                            set(posVar,badPosSet)),
                    preSearchGoal(activities,index-1));
}
```

Activities are statically selected one after the other, and the set B_i of bad position for a_i are removed and set on backtracking (`badPosSet` represents B_i, and the goals `remove` and `set` are used to remove (resp. set) B_i from `domain`$(pos(a_i))$). Each leaf of the tree search defined by `preSearchGoal` represents a partial solution that can be extended by tree search. Suppose that the tree search `preSearchGoal` is explored in Limited Discrepancy Search. Let d be the current discrepancy. We will first reach the leaf corresponding to the situation where the positions B_i have been removed from the domain of every $pos(a_i)$ (for $d = 0$); then we will reach all the leaves (at most n) where exactly one activity a_j has `domain`$(pos(a_j)) \subseteq B_j$ and all the other activities $a_i, i \neq j$ have `domain`$(pos(a_i)) \cap B_i = \emptyset$ (for $d = 1$). In general, when $d = k$, in all the leaves of the `preSearchGoal` tree search exactly k activities a_j have `domain`$(pos(a_j)) \subseteq B_j$ and all the other $n - k$ activities a_i have `domain`$(pos(a_i)) \cap B_i = \emptyset$. Knowing the current discrepancy d, we can use this information to compute a better lower bound and to remove some values from the domain of $pos(a)$. The problem P explored with discrepancy on `preSearchGoal` equal to d defines a subproblem $P' = P \uplus ($`card`$(pos(a_i) = p \in B_i) = d)$.

Consider a LDS iteration exploring nodes of discrepancy equal to d; suppose that there exist $k < d$ position variables such that `domain`$(pos(a_j)) \subseteq B_j$. Let S be the set of $n - k$ activities a_i such that $\neg($`domain`$(pos(a_i)) \subseteq B_i)$. k out of the d available discrepancy have been used, and $d - k$ still need to be used and will concern $d - k$ activities in the set S. We calculate, for each activity $a_i \in S$, the minimum reduced cost of the activity/position

Assignment Problem $rc_{ip^*} = \min_{p \in B_i}\{rc_{ip}\}$. Let ΔC be the sum of the $d-k$ smallest rc_{ip^*}. Since we know that for $d-k$ activities $a_i \in S$ their positions will be chosen within B_i, then $LB + \Delta C$ is a valid lower bound for the problem P'.

In practice, using this methods we are able to demonstrate that only subtrees corresponding to small values of discrepancy must be explored.

6.3.8 Preliminary computational results

The techniques described have been tested on the Min-Sum Tardiness Scheduling Problem and the results have been detailed in Section 4.5. This problem considers a set of n activities $a_1, ..., a_n$ that have to be scheduled on a single unary resource (resource with maximal capacity equal to one) μ. Each activity a_i has to be processed on μ for p_i time units, has a release date r_i, and a due date d_i. Activities cannot start before their release date, but can end before or after their due dates. If activity a_i ends after its due date d_i a penalty cost $tard_i$ has to be considered, with $tard_i = w_i * (end(a_i) - d_i)$ where w_i is given, and $end(a_i)$ is the end time of activity a_i. If a_i ends before its due date d_i, $tard_i = 0$. The problem consists in finding a feasible schedule which minimizes

$$totTard = \sum_{i=1}^{n} tard_i = \sum_{i=1}^{n} w_i * max\{(end(a_i) - d_i), 0\}$$

The current implementation is still in a prototype version where all the algorithms described are implemented in a non incremental way, and results in terms of run time are therefore very inefficient.

Nevertheless, we could evaluate the quality of the lower bounds obtained on randomly generated problem instances of 10 tasks described in [43], and the gain in terms on number of fails with respect to a pure CP approach.

On average, the number of fails of the pure CP approach is 60 times more than the approach that uses the filtering algorithms proposed, maintaining the same search method. The gain with respect to pure CP approach becomes even more important when the branching strategy uses the information calculated by the conditional variable constraints and by the optimization constraint. In this case, the number of fails of the pure CP approach is 700 times more than the approach that uses the optimization constraint both for filtering and in the heuristic.

The calculated lower bound appears to be very good: on the average of the 120 instances, the ratio LB/OPT results equal to $94,5\%$; on 53 instances the calculated lower bound is equal to the optimal solution, while the minimal ratio LB/OPT is equal to 61%.

These very promising preliminary results encourage us to implement the incremental version of all methods proposed, and to continue working on this subject.

Bibliography

[1] T.S. Abdul-Razaq, C.N. Potts, and L.N. Van Wassenhove. A survey of algorithms for the single machine total weighted tardiness scheduling problem. *Discrete Applied Mathematics*, 26:235–253, 1990.

[2] Ahuja, Magnanti, and Orlin. *Network Flows.* Prentice Hall, 1993.

[3] A. Allahverdi, J.N.D. Gupta, and T. Aldowaisan. A review of scheduling research involving setup consideration. *Omega*, forthcoming, 1998.

[4] D. Applegate, R.E. Bixby, V. Chvátal, and W. Cook. Finding cuts in tsp. Unpublished, 1995.

[5] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3:149–156, 1991.

[6] N. Ascheuer, M. Fischetti, and M. Grötschel. Solving ATSP with time windows by branch-and-cut. To appear in *Mathematical Programming*, 2000.

[7] E. Backer. An exact algorithm for the timed constrained travelling salesman problem. *Operations Research*, 31:938–945, 1983.

[8] R. Backofen and S. Will. Excluding symmetries in constraint based search. In *Proceedings CP'99*, pages 400–405, 1999.

[9] E. Balas, M. Fischetti, and W. Pulleyblank. The precedence constrained asymmetric travelling salesman problem. *Mathematical Programming*, 68:241–265, 1995.

[10] E. Balas and N. Simonetti. Linear time dynamic programming algorithms for some classes of restricted tsp's. unpublished, 1996.

[11] E. Balas and P. Toth. Branch and bound methods. In E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, editors, *The Travelling Salesman Problem.* John Wiley and Sons, 1985.

[12] P. Baptiste, C. Le Pape, and W. Nuijten. Incorporating efficient operations research algorithms in constraint-based scheduling. In *Proc. 1st International Joint Workshop on Artificial Intelligence and Operations Research*, 1995.

[13] P. Baptiste, C. Le Pape, and W. Nuijten. Efficient operations research algorithms in constraint-based scheduling. In *Proceedings of IJCAI'95*, 1995.

[14] P. Barth and A. Bockmayr. Finite domain and cutting plane techniques in CLP(PB). In L. Sterling, editor, *Logic Programming - Proceedings of the Twelfth International Conference on Logic Programming*, pages 133–147, Cambridge, Massachusetts, 1995. The MIT-press.

[15] P. Barth and A. Bockmayr. Modelling mixed-integer optimisation problems in constraint logic programming. Technical Report MPI-I-95-2-011, Max-Planck-Institut für Informatik, 1995.

[16] J. C. Beck. *Texture measurements as a basis for heuristic commitment techniques in constraint-directed scheduling.* PhD thesis, University of Toronto, 1999.

[17] N. Beldiceanu, E. Bourreau, H. Simonis, and D. Rivrau. Introducing metaheuristics in CHIP. In *Proceedings of the 3rd Metaheuristics International Conference*, Angra do Reis, Brazil, 1999.

[18] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathematical Computer Modelling*, 20(12):97–123, 1994.

[19] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[20] B. Benhamou. Study of symmetries in constraint satisfaction problems. In *Proceedings of PPCP'94*, LNCS - Springer Verlag, 1994.

[21] H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle and L. Plümer, editors, *Logic Programming: formal Methods and Practical Applications*, pages 245–272. North Holland, 1995.

[22] C. Bèssiere, E. Freuder, and J.C. Régin. Using constraint metaknowledge to reduce arc-consistency computation. *Artificial Intelligence*, 107:125–148, 1999.

[23] C. Bèssiere, P. Meseguer, E. Freuder, and J. Larrosa. On forward checking for non binary constraint satisfaction. In J. Jaffar, editor, *Principle and Practice of Constraint Programming - CP99, LNCS 1713*, Berlin Heidelberg, 1999. Springer-Verlag.

[24] C. Bèssiere and J.C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, 1997.

[25] C. Bèssiere and J.C. Régin. Enforcing arc consistency for global constraints by solving subproblems on the fly. In J. Jaffar, editor, *Principle and Practice of Constraint Programming - CP99, LNCS 1713*, Berlin Heidelberg, 1999. Springer-Verlag.

[26] A. Bockmayr and T. Kasper. Pseudo-boolean and finite domain constraint programming: A case study. In U. Geske and H. Simonis, editors, *Deklarative Constraint Programmierung, Dresden*, GMD-Studien Nr. 297, 1996.

[27] A. Bockmayr and T. Kasper. Branch-and-infer: A unifying framework for integer and finite domain constraint programming. *INFORMS J. Computing*, 10:287 – 300, 1998.

[28] P. Brucker and O.Thiele. A branch and bound method for the general shop problem with sequence dependent setup-times. *OR Spektrum*, 18:145–161, 1996.

[29] A. Caprara, M. Fischetti, P. Toth, D. Vigo, and P.-L. Guida. Algorithms for railway crew management. *Mathematical Programming*, 79:125–141, 1997.

[30] A. Caprara, F. Focacci, E.Lamma, P.Mello, M.Milano, P.Toth, and D.Vigo. Integrating constraint logic programming and operations research techniques for the crew rostering problem. *Software Practice & Experience*, 28(1):49–76, 1998.

[31] J. Carlier and E. Pinson. An algorithm for solving job shop scheduling. *Management Science*, 35:164–176, 1989.

[32] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.

[33] G. Carpaneto, S. Martello, and P. Toth. Algorithms and codes for the assignment problem. *Annals of Operations Research*, 13:193–223, 1988.

[34] G. Carpaneto and P. Toth. Algorithms for the solution of the bottleneck assignment problem. *Computing*, 27:179–187, 1981.

[35] Y. Caseau and P. Koppstein. A Rule-based approach to a Time-Constrainted Traveling Salesman Problem. In *Proceedings of Symposium of Artificial Intelligence and Mathematics*, 1992.

[36] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *Proceedings of ICLP'94*, 1994.

[37] Y. Caseau and F. Laburthe. Improving branch and bound for job-shop scheduling with constraint propagation. In M. Deza, R. Euler, and Y. Manoussakis, editors, *Proceedings of Combinatorics and Computer Science, CCS'95*, LNCS 1120, pages 129–149. Springer-Verlag, Berlin Heidelberg, 1995.

[38] Y. Caseau and F. Laburthe. Solving small TSPs with constraints. In L. Maish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming - ICLP'97*, pages 316–330, Cambridge, Massachusetts, 1997. The MIT-press.

[39] Y. Caseau and F. Laburthe. Solving various weighted matching problems with constraints. In G. Smolka, editor, *Principle and Practice of Constraint Programming - CP97, LNCS 1330*, pages 17–31, Berlin Heidelberg, 1997. Springer-Verlag.

[40] Y. Caseau and F. Laburthe. Heuristics for large constrained routing problems. *Journal of Heuristics*, 5:281–303, 1999.

[41] A. Cesta, A. Oddi, and S. Smith. A constraint-based method for project scheduling with time windows. *Journal of Heuristics*, 2001. To appear.

[42] N. Christofides, A. Mingozzi, and P. Toth. State space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11:145–164, 1981.

[43] C. Chu. A branch-and-bound algorithm to minimize total tardiness with different release dates. *Naval Research Logistics*, 39:265–283, 1992.

[44] G.B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.

[45] K. Darbi-Dowman and J. Little. Properties of some combinatorial optimization problems and their effect on the performance of integer programming and constraint logic programming. *INFORMS Journal of Computing*, 10:276–286, 1998.

[46] B. De Backer, V. Furnon, and P. Shaw. An object model for meta-heuristic search in constraint programming. In *Proceedings of the First International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - CP-AI-OR-99*, 1999.

[47] M. Dell'Amico and S. Martello. Linear assignment. In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimization*. John Wiley & Sons, Chichester, 1997.

[48] M. Dell'Amico and M. Trubian. Applying tabu-search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1993.

[49] M. Desrochers, J. Desrosiers, and M.M. Solomon. A new Optimization Algorithm for the Vehicle Routing problem with Time Windows. *Operations Research*, 40:342–354, 1992.

[50] J. Desrosiers, Y. Dumas, M.M. Solomon, and F. Soumis. Time constrained routing and scheduling. In M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser, editors, *Network Routing*. Elsevier, 1995.

[51] M. Dincbas, P. Van Hentenryck, and H. Simonis. Solving the car sequencing problems in Constraint Logic Programming. In *Proceedings of European Conference on Artificial Intelligence ECAI88*, 1988.

[52] M. Dincbas, P. Van Hentenryck, and H. Simonis. Solving large combinatorial problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[53] M. Dincbas, P.Van Hentenryck, M.Simonis, A.Aggoun, T.Graf, and F.Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer System*, pages 693–702, 1988.

[54] S.B. Dreyfus and A.M. Law. *The Art and Theory of Dynamic Programming*. Academic Press, 1989.

[55] Y. Dumas, J. Desrosiers, E. Gelinas, and M.M. Solomon. An optimal algorithm for the travelling salesman problem with the time windows. *Operations Research*, 43:367–371, 1995.

[56] ECRC. *ECL^iPS^e User Manual Release 3.5*, 1992.

[57] H.H. El Sakkout and M. Wallace. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints*, 5:359–388, 2000.

[58] T. Fahle, U. Junker, S.E. Karisch, N. Kohl, M. Sellmann, and B. Vaaben. Constraint programming based column generation for crew assignment. *Journal of Heuristics*, 2001. to appear.

[59] J. Feldman and D. Vergamini. Practical patterns for constraint programming. In *Proceedings of the International Conference on the Practical Applications of Constraint Technology, PACT98*, 1998.

[60] T. Feo and M. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[61] M. Fischetti and P. Toth. An additive bounding procedure for combinatorial optimization problems. *Operations Research*, 37:319–328, 1989.

[62] M. Fischetti and P. Toth. An additive bounding procedure for the asymmetric travelling salesman problem. *Mathematical Programming*, 53:173–197, 1992.

[63] M. Fischetti and P. Toth. An efficient algorithm for the min-sum arborescence problem on complete digraphs. *ORSA Journal of Computing*, 5(4):426–434, 1993.

[64] F. Focacci, P. Laborie, and W. Nuijten. Solving scheduling problems with setup times and alternative resources. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling, AIPS'00.* AAAI Press, 2000.

[65] F. Focacci, F. Laburthe, and A. Lodi. Local search and constraint programming. Technical report, University of Bologna, 2000.

[66] F. Focacci, E. Lamma, P. Mello, and M. Milano. Constraint logic programming for the crew rostering problem. In *Proceedings of the third International Conference on Practical Applications of Constraint Technology*, 1997.

[67] F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In J. Jaffar, editor, *Principle and Practice of Constraint Programming - CP99, LNCS 1713*, pages 189–203. Springer-Verlag, Berlin Heidelberg, 1999.

[68] F. Focacci, A. Lodi, and M. Milano. Solving TSP with time windows with constraints. In D. De Schreye, editor, *Logic Programming - Proceedings of the 1999 International Conference on Logic Programming*, pages 515–529. The MIT-press, Cambridge, Massachusetts, 1999.

[69] F. Focacci, A. Lodi, and M. Milano. Cutting planes in constraint programming: an hybrid approach. In R. Dechter, editor, *Principle and Practice of Constraint Programming - CP 2000, LNCS 1894*, pages 187–201. Springer-Verlag, Berlin Heidelberg, 2000.

[70] F. Focacci, A. Lodi, and M. Milano. Mathematical programming techniques in constraint programming: a short overview. *Journal of Heuristics*, 2001. To appear.

[71] F. Focacci, A. Lodi, M. Milano, and D. Vigo. Solving TSP through the integration of OR and CP techniques. *Electronic Notes in Discrete Mathematics*, 1, 1999. http://www.elsevier.com/cas/tree/store/disc/free/endm/.

[72] F. Focacci, A. Lodi, M. Milano, and D. Vigo. An introduction to constraint programming. *Ricerca Operativa*, 91:5–20, 2000.

[73] E. Freuder. Eliminating interchangeble values in constraint satisfaction problems. In *Proceedings AAAI'91*, pages 227–233, 1991.

[74] E.C. Freuder. Synthesizing constraint expressions. *Communication of the ACM*, 21(11):958–966, 1978.

[75] E.C. Freuder. A sufficient condition for backtrack free search. *Communication of the ACM*, 29(1):24–32, 1982.

[76] D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings of AAAI'94*, 1994.

[77] M.R. Garey and D.S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness.* Victor Klee, 1979.

[78] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie Mellon University, 1979.

[79] I.P. Gent and B. Smith. Symmetry breaking during search in constraint programming. In *TR 99.02 School of Computer Studies*, 1999.

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

[81] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bull. Amer. Math. Soc.*, 64:275–278, 1958.

[82] M.T. Hajian, H. El-Sakkout, M. Wallace, J.M. Lever, and E.B. Richards. Towards a closer integration of finite domain propagation and simplex-based algorithms. Technical report, IC-Parc, 1995.

[83] R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[84] W. Harvey. *Nonsystematic Backtracking Search.* PhD thesis, Stanford University, 1995.

[85] W. Harvey and M. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th IJCAI*, pages 607–615. Morgan Kaufmann, 1995.

[86] S. Heipcke. Comparing constraint programming and mathematical programming approaches to discrete optimisation - the change problem. *Annals of Operations Research*, 50:581–595, 1999.

[87] S. Heipcke. An example of integrating constraint programming and mathematical programming. *Electronic Notes in Discrete Mathematics*, 1, 1999. http://www.elsevier.com/cas/tree/store/disc/free/endm/.

[88] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[89] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). Technical Report CS-93-02, Brown University, 1993.

[90] J.H. Holland. *Adaption in Natural and Artificial Systems: An Introductory Anal ysis with Applications to Biology, Control, and Artificial Intelligence.* MIT press, Cambridge, MA, 1992. 2nd edition.

[91] J.N. Hooker. Constraint satisfaction methods for generating valid cuts. In D.L. Woodruff, editor, *Advances in Computational and Stochastic Optimization, Logic Programming and heuristic Search*, pages 1–30. Kluwer, 1997.

[92] J.N. Hooker and M.A. Osorio. Mixed logical/linear programming. *Discrete Applied Mathematics*, 96:395–442, 1997.

[93] ILOG. *ILOG Concert 1.0 User's Manual and Reference Manual.*

[94] ILOG. *ILOG Planner 3.2 User's Manual and Reference Manual.*

[95] ILOG. *ILOG Scheduler 5.0 User's Manual and Reference Manual.*

[96] ILOG. *ILOG Solver 5.0 User's Manual and Reference Manual.*

[97] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the Conference on Principle of Programming Languages*, 1987.

[98] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[99] M. Jünger, G. Reinelt, and G. Rinaldi. The travelling salesman problem. In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimization.* John Wiley & Sons, Chichester, 1997.

[100] M. Jünger, G. Rinaldi, and S. Thienel. *MINCUT Software package.* Universität zu Köln, 1996.

[101] T. Kasper. *A unifying logical framework for Integer Linear Programming and Finite Domain Constraint Programming.* PhD thesis, Fachbereich Informatick, University of Saarlandes, Saarbrüken, Germany, 1998.

[102] G. Kindervater and M. Savelsbergh. Vehicle routing: Handling edges exchanges. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 337–360. J. Wiley & Sons, Chichester, 1997.

[103] S. Kirkpatrick, C. Gellat, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[104] V. Kumar. Algorithms for constraint-satisfaction problems: a survey. *AI Magazine*, 13:32–44, 1992.

[105] P. Laborie. Modal precedence graphs and their usage in ILOG Scheduler. Technical Report OIR-1999-1, ILOG, 1999.

[106] A. Langevin, M. Desrochers, J. Desrosiers, and F. Soumis. A two-commodity flow formulation for the Traveling Salesman and Makespan Problem with Time Windows. *Networks*, 23:631–640, 1993.

[107] J.L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[108] C. Le Pape and P. Baptiste. Constraint propagation techniques for disjunctive scheduling: The preemptive case. In *Proc. 12th European Conference on Artificial Intelligence*, 1996.

[109] S. Lin and B.W. Kernighan. An effective heuristic for the traveling salesman problem. *Operations Research*, 21:498–516, 1973.

[110] J.W. Lloyd. *Foundation of Logic Programming - Second Extended Edition.* Springer-Verlag, 1987.

[111] Grötschel M., Lovász L., and Schrijver A.J. *Geometric Algorithms and Combinatorial Optimization.* John Wiley & Sons, New York, 1988.

[112] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[113] K. Marriott and P.J. Stuckey. *Programming with Constraints.* The MIT Press, 1998.

[114] T. Mautor and P. Michelon. MIMAUSA: A new hybrid method combining exact solution and local search. In *Proceedings of the 2nd International Conference on Meta-Heuristics*, Sophia-Antipolis, France, 1997.

[115] P. Meseguer and C. Torras. Solving strategies for highly symmetric csps. In *Proceedings IJCAI'99*, pages 400–405, 1999.

[116] A. Mingozzi, L. Bianco, and S. Ricciardelli. Dynamic programming strategies for the travelling salesman problem with time windows and precedence constraints. *Operations Research*, 45:365–377, 1997.

[117] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24:1097–1100, 1997.

[118] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[119] U. Montanari. Networks of Constraints: Foundamental Properties and Applications to Pcture Processing. *Information Sciences*, 7:95–132, 1974.

[120] B. Nadel. *Tree Search and Arc Consistency in Constraint-Satisfaction Algorithms*, pages 287–342. Springer Verlag, NY, 1990. L.Kanal, V.Kumar eds.

[121] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley and Sons - New York, 1988.

[122] E. Nespoulous. *Methodes de detection et d'exploitation des proprietes de symetries sur les CSP entiers.* PhD thesis, Universite' de Bourgogne, 1999.

[123] N. Nilsson. *Problem Solving Methods in Artificial Intelligence.* McGraw-Hill, 1971.

[124] W. Nuijten and C. Le Pape. Constraint-based job shop scheduling with ILOG SCHEDULER. *Journal of Heuristics*, 3:271–286, 1998.

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

[126] G. Ottosson, J.N. Hooker, H.J. Kim, and E.S. Thorsteinsson. On integrating constraint propagation and linear programming for combinatorial optimization. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on on Innovative Applications of Artificial Intelligence - AAAI99*, pages 136–141, 1999.

[127] G. Ottosson and E.S. Thorsteinsson. Linear relaxation and reduced cost based propagation of continuous variable subscript. In *Proceedings of the Second International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - CP-AI-OR-00*, pages 129–138, 2000.

[128] Greger Ottosson, Erlendur S. Thorsteinsson, and John N. Hooker. Mixed global constraints and inference in hybrid CLP–IP solvers. In Susanne Heipcke and Mark Wallace, editors, *CP99 Post-Conference Workshop on Large Scale Combinatorial Optimisation and Constraints*, volume 4 of *Electronic Notes in Discrete Mathematics*. Elsevier Science, October 1999.

[129] M Padberg and G Rinaldi. Optimization of a 532-symmetric traveling salesman problem. *Operations Research Letters*, 6:1–8, 1987.

[130] M. Padberg and G. Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47:19–36, 1990.

[131] L. Perron. Search procedures and parallelism in constraint programming. In J. Jaffar, editor, *Principle and Practice of Constraint Programming - CP99, LNCS 1713*, pages 346–360. Springer-Verlag, Berlin Heidelberg, 1999.

[132] G. Pesant and M. Gendreau. A view of local search in constraint programming. In E.C. Freuder, editor, *Principle and Practice of Constraint Programming - CP96, LNCS 1118*, pages 353–366. Springer-Verlag, Berlin Heidelberg, 1996.

[133] G. Pesant, M. Gendreau, J.Y. Potvin, and J.M. Rousseau. An exact constraint logic programming algorithm for the travelling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998.

[134] G. Pesant, M. Gendreau, J.Y. Potvin, and J.M. Rousseau. On the flexibility of Constraint Programming models: From Single to Multiple Time Windows for the Travelling Salesman Problem. *European Journal of Operational Research*, 117(2):253–263, 1999.

[135] M. Prais and C. Ribeiro. Reactive grasp: An application to a matrix decomposition problem in TDMA traffic assignment. Technical report, Catholic University of Rio de Janeiro, Department of Computer Science, 1998.

[136] L. Proll and B. Smith. Integer linear programming and constraint programming approaches to a template design problem. *INFORMS Journal of Computing*, 10:265–275, 1998.

[137] J. F. Puget. On the satisfiability of symmetrical constraint satisfaction problems. In *Proceedings ISMIS'93*, pages 350–361, 1993.

[138] J.-F. Puget and M. Leconte. Beyond the glass-box: Constraints as objects. In *Proceedings of ILPS '95*. MIT Press, 1995.

[139] J.F. Puget. A C++ implementation of CLP. Technical Report 94-01, ILOG Headquarters, 1994.

[140] P. Refalo. Tight cooperation and its application in piecewise linear optimization. In J. Jaffar, editor, *Principle and Practice of Constraint Programming - CP99, LNCS 1713*, Berlin Heidelberg, 1999. Springer-Verlag.

[141] P. Refalo. Linear formulation of constraint programming models and hybrid solvers. In R. Dechter, editor, *Principle and Practice of Constraint Programming - CP 2000, LNCS 1894*. Springer-Verlag, Berlin Heidelberg, 2000.

[142] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the Twelfth National Conference on Artificial Intelligence - AAAI94*, pages 362–367, 1994.

[143] J.C. Régin. Arc consistency for global cardinality constraints with costs. In J. Jaffar, editor, *Principle and Practice of Constraint Programming - CP99, LNCS 1713.* Springer-Verlag, Berlin Heidelberg, 1999.

[144] J.C. Régin. The symmetric Alldiff constraint. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence - IJCAI99*, pages 420–425, 1999.

[145] Y. Rochat and E.D. Taillard. Probabilistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics*, 1:147–167, 1995.

[146] R. Rodosek, M. Wallace, and M.T. Hajian. A new approach to integrating mixed integer programming and constraint logic programming. *Annals of Operational Research*, 86:63–87, 1999.

[147] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. *Proceedings of ECAI'90*, 1990.

[148] P. Roy and F. Pachet. using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In *Proceedings of ECAI98 Workshop on Non Binary Constraints*, pages 27–33, 1998.

[149] R. Russell. Hybrid heuristics for the vehicle routing problem with time windows. *Transportation Science*, 29:156–166, 1995.

[150] M.W.P. Savelsberg. Local search in Routing Poblem with Time Windows. *Annals of Operations Research*, 4:285–305, 1985.

[151] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In M. Maher and Puget J.-F., editors, *Principle and Practice of Constraint Programming - CP98, LNCS 1520*, pages 417–431. Springer-Verlag, Berlin Heidelberg, 1998.

[152] P. Shaw, V. Furnon, and B. De Backer. A lightweight addition to CP frameworks for improved local search. In *Proceedings of CP-AI-OR'00*, Padderborn, Germany, 2000.

[153] H. Simonis. Calculating lower bounds on a resource scheduling problem. Technical report, Cosytec, 1995.

[154] B.M. Smith, S.C. Brailford, P.M. Hubbard, and H.P. Williams. The progressive party problem: Integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1996.

[155] M.M. Solomon. Algorithms for the vehicle routing and scheduling problem with time window constraints. *Operations Research*, 35:254–265, 1987.

[156] G.J. Sussman and G.L. Steele. Constraints - a language for expressing almost hierarchical descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[157] E.D. Taillard, P. Badeau, M. Gendreau, F. Guertin, and J.-Y. Potvin. A new neighborhood structure for the vehicle routing problems with time windows. unpublished, 1995.

[158] E.P.K. Tsang. *Foundation of Constraint Satisfaction.* Academic Press, 1993.

[159] P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, 1999.

[160] P. Van Hentenryck and J.P. Carillon. Generality versus specificity: an experience with AI and OR techniques. In *Proceedings of the Seventh National Conference on Artificial Intelligence - AAAI88*, pages 660–664, 1988.

[161] M. Wallace. Practical applications of consraint programming. *Constraints*, 1:139–168, 1996.

[162] T. Walsh. Depth-bounded discrepancy search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence - IJCAI.* Morgan Kaufmann, 1997.