Enhancing CLP branch and bound techniques for scheduling problems

F. Bosi and M. Milano*†

DEIS, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy

SUMMARY
In this paper, we propose a constraint logic programming (CLP) approach to the solution of a job shop scheduling problem in the field of production planning in orthopaedic hospital departments. A pure CLP on finite domain (CLP(FD)) approach to the problem has been developed, leading to disappointing results. In fact, although CLP(FD) has been recognized as a suitable tool for solving combinatorial problems, it presents some drawbacks for optimization problems. The main reason concerns the fact that CLP(FD) solvers do not effectively handle the objective function and cost-based reasoning through the simple branch and bound scheme they embed. Therefore, we have proposed an improvement of the standard CLP branch and bound algorithm by exploiting some well-known operations research results. The branch and bound we integrate in a CLP environment is based on the optimal solution of a relaxation of the original problem. In particular, the relaxation used for the job shop scheduling problem considered is the well-known shifted bottleneck procedure considering single machine problems. The idea is to decompose the original problem into subproblems and solve each of them independently. Clearly, the solutions of each subproblem may violate constraints among different subproblems which are not taken into account. However, these solutions can be exploited in order to improve the pruning of the search space and to guide the search by defining cost-based heuristics. The resulting algorithm achieves a significant improvement with respect to the pure CLP(FD) approach that enables the solution of problems which are one order of magnitude greater than those solved by a pure CLP(FD) algorithm. In addition, the resulting code is less dependent on the input data configuration.

KEY WORDS: integration of OR techniques in CLP; scheduling problems; hybrid solver

INTRODUCTION
Constraint logic programming (CLP) [1] on finite domains (CLP(FD)) is a powerful programming paradigm combining the advantages of logic programming and the efficiency of constraint solving.

*Correspondence to: M. Milano, DEIS, Università di Bologna, Viale Risorgimento 2, 40136 Bologna, Italy.
†E-mail: mmilano@deis.unibo.it

Contract/grant sponsor: CNR, Committee 12 on Information Technology (Project SCI*SIA)
CLP(FD) enables the model of a problem to be easily stated, and provides effective constraint propagation algorithms devoted to reducing the search space. Many real life applications, such as scheduling, planning, sequencing and assignment problems, have been effectively solved by using CLP(FD) techniques [2–5].

In this paper, we face a real life job shop scheduling application for Officine Rizzoli, an Italian company producing orthopaedic prostheses and girdles for a well-known Italian group of orthopaedic hospital departments. We have a set of jobs corresponding to customer orders, whose processing requires the execution of different production phases, called tasks. Tasks are linked by temporal constraints limiting their relative position along the time line. The execution of each task has a fixed duration and requires a certain amount of limited resources. These constraints determine the feasibility of the schedule. However, we are not interested in a feasible solution, but in an optimal one according to a given criterion. In the scheduling application discussed in this paper, we aim at minimizing the makespan, i.e. the duration of the overall schedule. Real life instances provided by Officine Rizzoli have the following average dimensions: 150 jobs each week should be allocated, with an average of 20 tasks each, over 20 productive resources, considering alternative processing units. Up to now, the problem has been solved by human experts and requires a considerable amount of time and expertise.

One of the more promising CLP areas of application to date probably concerns scheduling problems [6–8]. This is due to the exploitation of powerful propagation techniques deriving from operations research (OR), such as edge finding [6], in global constraints.

However, on solving the problem, we encountered and underlined some drawbacks of CLP(FD) that limit its use for optimization problems. In fact, while current CLP systems provide effective constraint propagation algorithms that allow the search space to be pruned by removing infeasible branches, they present some limitations in dealing with objective functions. In particular, optimization predicates provided by CLP(FD) languages impose, each time they find a solution, that further solutions will have a lower cost than the lowest one found so far. This mechanism triggers very little propagation (if any) on variable domains since the objective function is usually loosely connected with the decision variables. For example, in scheduling problems, one of the classical objective functions which should be minimized is the makespan, i.e. the completion of the whole schedule. This objective function is equal to the end time of the last job. Thus, the decision variable domain reduction loosely affects the bounds of the objective function and vice versa. In other cases, the objective function is almost unaffected by modification of the decision variables. In this setting, the CLP(FD) solver could benefit from the introduction of techniques aimed at reducing the search space on the basis of optimization criteria.

In addition, a pure CLP(FD) approach to the problem has been proved to be extremely dependent on the data configuration. Slight changes of the input data greatly affect system performances. Using pure CLP techniques, we could in general solve to optimality problems with 10–20 jobs with an average of 20 tasks each. Smaller instances, computed, for example, by removing some tasks from the 10–20 job instances, should in general be solved more efficiently. Instead, a pure CLP(FD) approach is not able to solve and prove optimality for these instances in a reasonable time.

For these reasons, we have argued that by integrating well-known OR techniques in CLP we could increase system performances and be less sensitive to input data variations. In fact, OR techniques provide a powerful way of coping with optimization predicates and the integration of CLP and OR techniques is an emerging research area [9–18] since it benefits from the advantages of both fields. From OR, we obtain a better way of exploring the search space by exploiting information deriving from optimization functions; from CLP we maintain the declarative semantics easing the problem
modelling, and its powerful constraint propagation algorithms. Moreover, we can further improve the effectiveness of lower bounding techniques coming from OR, by combining propagation techniques and domain reasoning with results coming from relaxations of the original problem.

The contribution of this paper is twofold: we propose (i) a way of achieving such a combination as a general CLP problem solving methodology and (ii) its application to a particular job shop scheduling problem. Concerning the first point, we have developed a methodology for designing hybrid solvers that exploit well-known branch and bound techniques [19] that allow the search space to be pruned on the basis of the lower bound calculation. A lower bound is an optimistic value of a solution of the original problem obtained by solving to optimality a relaxed one, i.e. a problem where some constraints have been relaxed. We have defined a propagation algorithm which prunes variable domains on the basis of the lower bound calculation at each node of the search tree. The lower-bound-based propagation interacts with other global problem constraints through shared variables.

Concerning the application, we first solved the problem via a pure CLP approach, achieving disappointing results. In fact, the resulting program failed to solve problems whose dimensions correspond to real world instances. For this reason, we have implemented the integration proposed on top of the finite domain solver of the CLP commercial language CHIP [20], and solved the job shop scheduling problem through the extension proposed. We have obtained a significant performance improvement of the resulting solver: we are able to solve and prove optimality of real world problems which are one order of magnitude greater than those solved by a pure CLP approach. In addition, the code is less dependent on the data configuration.

The paper is organized as follows: we first recall some concepts on optimization problems, on CLP and on OR lower bounding techniques. Then, we present the job shop scheduling problem. The specific problem is described, and the corresponding model is presented in terms of variables, constraints and objective function formalization. In addition, a description of the single machine lower bound for the specific problem is provided. After the problem description, we propose how to exploit information deriving from the optimal solution of the relaxed problem in a CLP algorithm. We also provide some methodological guidelines in order to build effective hybrid solvers. Implementation details and the user interface developed are then described. We present computational results, comparisons with a pure CLP approach and we underline the effectiveness of the heuristics used. Finally, some related approaches are compared with our proposal. A discussion and further work conclude the paper.

PRELIMINARIES

The aim of this section is to provide some preliminary notions on the concepts addressed in this paper. First, we will focus on optimization problems, we define their general structure and the basic steps for solving them. Then, we introduce CLP (FD) and we describe how to model an optimization problem by using such a programming language. Then, we discuss OR lower bounding techniques and how they can be exploited in order to reduce the search space. Readers already familiar with these concepts could skip this section.

Combinatorial optimization problems

Combinatorial optimization problems arise in many real life applications. These problems are, in general, difficult to solve since they are characterized by large search spaces and involve difficult
constraints and preferences. Most of them belong to the NP-complete class of problems. This means that a polynomial time algorithm for solving them does not exist.

A general model of these problems involves a set of variables representing problem basic entities, which 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 (the word combinatorial derives from that). 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 the so-called 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, for example, minimize the cost of the productive process, or maximization problems in the other case, for example maximize the profit derived by the productive process.

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

- a constraint satisfaction part aimed at finding a feasible solution which satisfies all the involved constraints;
- an optimization part aimed at selecting, among feasible solutions, the one minimizing or maximizing the objective function.

In general, the solution of this class of problem 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 in order to find a feasible and optimal solution. We introduce here the concept of the proof of optimality: when the optimal solution has been found, we have to prove that it is indeed optimal by exploring the remaining part of the search tree.

Formally, combinatorial optimization problems in general have the form \( \min f(x) : x \in X \), where \( X \) is a finite feasible solution set, and \( f(x) \) is a real-valued objective function defined over \( X \). Also maximization problems can be represented by this general form by inverting the sign of the \( f(x) \). Such problems cannot be solved by enumeration in a reasonable computing time since \( X \) may have a huge cardinality. Thus, techniques for pruning the search space should be defined. Conceptually, we can identify two kinds of pruning: pruning performed on the basis of feasibility reasoning, and pruning based on optimality reasoning. Feasibility pruning (also called constraint propagation) removes a priori (before exploration) combinations of assignments which do not lead to any feasible solution. Optimality pruning, instead, removes a priori combinations of assignments which do not lead to better solutions with respect to the best one found.

On one hand, CLP provides powerful methods for performing pruning on the basis of feasibility, but lacks good strategies for pruning on the basis of optimality. Since OR provides effective techniques for removing portions of the search space on the basis of optimality considerations, we argue that integrating these two complementary areas can lead to better results when solving combinatorial optimization problems.

**CLP**

CLP [1,21] is a class of programming languages combining the advantages of declarative and flexible models and the efficiency of constraint solving. CLP derives from logic programming [22], which is
a programming paradigm where computations can be seen as controlled deductions. Logic programs can be read in a declarative way and are therefore easy to understand, modify and extend. One of the most popular languages in this setting is Prolog, which is based on resolution as a sound and complete proof procedure. Resolution basically explores a search tree using a depth first search procedure with chronological backtracking.

The CLP techniques, thanks to the active use of constraints, enable the search space to be a priori pruned by removing combinations of assignments which cannot appear in any consistent solution. The main idea behind this approach is to prevent failures instead of recovering from failures that have already taken place, by means of expensive backtracking.

In this paper, we focus on CLP(FD) which has been successfully applied to several combinatorial optimization problems [2–5].

In CLP(FD) languages, variables range over finite domains of integers. Variable domains represent the possible values that variables can assume during the computation. For example, \( X \vdash [1 \ldots 10] \) states that variable \( X \) can assume one of the integer values from 1 to 10, \( Y \vdash [3, 5, 9] \) states that variable \( Y \) is either 3 or 5 or 9. Variables are linked by constraints that can be either mathematical constraints, such as \( X > Y \), \( X < Y \), \( X = Y \), \( X \leq Y \), \( X \geq Y \), \( X \neq Y \), or symbolic constraints. Symbolic constraints are global constraints allowing a declarative statement of the problem along with powerful filtering algorithms based on global reasoning. The general syntax of symbolic constraints is provided in terms of predicates, i.e. relations among entities appearing as parameters. For example, the constraint \( c(P_1, \ldots, P_n) \) is a constraint defining a relation among entities (variables or functions of variables) \( P_1, \ldots, P_n \). Each constraint has an associated declarative semantic defining when the constraint holds, i.e. is satisfied, and an operational semantic defining how the constraint is propagated.

CHIP [20] is a commercial CLP system providing a number of global constraints [23]. As an example, a constraint used in solving the application described in this paper is the cumulative constraint. Declaratively, the constraint

\[
\text{cumulative}([S_1, \ldots, S_n],[D_1, \ldots, D_n],[R_1, \ldots, R_n],\text{Val})
\]

holds if and only if the activities with starting times \( S_i \), duration \( D_i \) and requiring a certain amount of resource \( R_i \) do not exceed the limited resource capacity \( \text{Val} \) at any point in time. The constraint can have more complex forms, including other parameters [23]. The cumulative constraint provides a declarative and flexible abstraction of capacity constraints and can be used in many problems involving limitations on resource use. Operationally, it embeds powerful filtering algorithms aimed at effectively reducing the search space. We will see different uses of this constraint for modelling the scheduling application described in this paper.

In many cases, an optimization criterion must be satisfied, i.e. we have to find the best among the feasible solutions. Therefore, many CLP languages provide branch and bound procedures for minimization or maximization.

The structure of a general CLP(FD) program is the following:

- decision variable and domain creation;
- objective function definition;
- constraint posting;
- search.
The first step involves the definition of a mapping between problem entities and decision variables of the model. Domains contain all possible values that can be assigned to variables. The objective function is then defined on decision variables. In general, the objective function is represented by a variable, for example, a cost expression $C$, that provides a measure of the solution quality. During the third step, the symbolic and mathematical constraints among the variables can be posted. In this phase, constraint propagation is performed in order to remove combinations of assignments that are infeasible in the original problem definition.

An interesting feature of constraint programming is the interaction among constraint propagation algorithms embedded in constraints that cooperate and interact through shred variables. The propagation algorithm embedded in constraints 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 or the 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 other constraints involving $X$ are awakened, perform propagation on the basis of the current state of the variable domains and at the end are suspended waiting for another event. A constraint is solved when it is always true, i.e., $X1 < X2$ where $D1 = [1, 2, 4, 5]$ and $D2 = [7, 8, 9, 10]$ is always true for all pairs of values for $X$ and $Y$. Thus, after propagation, if the constraint is solved it is no longer suspended.

In order to explain the interaction among constraints, let us consider the example depicted in Figure 1. We have three variables (i.e. $X$, $Y$ and $Z$) ranging on the same domain $T1:::5$. The first propagation of $X = Y + 1$ yields the following domain reduction $X :: [2..5], Y :: [1..4], Z :: [1..5]$. Note that value 1 is deleted from the domain of $X$ since a value for $Y$ consistent with it does not exist. In a similar way, the value 5 is deleted from the domain of $Y$. At the end of the propagation of the first constraint, since $X = Y + 1$ is not solved, it is suspended. Then, $Y = Z + 1$ is considered, its propagation reduces domains as $X :: [2..5], Y :: [2..4], Z :: [1..3]$, and it is suspended. The domain of $Y$ has 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 awakened and propagated does not affect the result, but can possibly affect the performance of the propagation process.

Unfortunately, constraint propagation in CLP(FD) solvers is, in general, not complete. In fact, some values left in variable domains could not be part of any feasible solution. 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; or (iii) some domains contain more than one value. In this third case, a search is required (fourth step).

During search the problem is subsequently partitioned in subproblems by adding additional constraints, we call them branching constraints, that restrict the problem. Branching constraint propagation may remove values from variable domains thus triggering the constraint propagation process which is performed at each node of the search space. A general search strategy used in constraint programming is that of selecting a variable and tentatively assigning it a value belonging to its domain. Each variable instantiation raises an event awaking all the constraints involving that variable and a propagation process starts again. The performances of the overall algorithm are strongly influenced by the search strategy adopted. Thus, more informed search strategies possibly dependent on the problem characteristics can be defined in order to improve the efficiency of the approach.

When solving optimization problems, the search is aimed not only at finding a feasible solution, but also at finding an optimal one. Commercial CLP(FD) solvers, like CHIP, provide predicates that minimize (respectively maximize) the problem objective function \( f(x) \). Every time a solution is found with cost \( f(x^*) \), the search continues with the new constraint \( f(x) < f(x^*) \) (respectively \( f(x) > f(x^*) \)) imposing that further solutions should have a lower (respectively higher) cost with respect to \( f(x^*) \). As mentioned, a domain variable, say \( C \), is associated with the objective function. \( C \) is linked to problem domain variables. By reasoning on domains, a CLP(FD) solver can recognize if, at any level of the search tree, a good solution can be found by means of further in-depth exploration. So, if the lower bound of the domain of \( C \) has a larger value than the best solution found so far, the search backtracks, in order to find alternative partial solutions. Note that the lower bound obtained only by means of the original problem variable bounds is not usually a tight lower bound for the problem itself. Therefore, pure constraint programming techniques are usually able to prune only few levels of the search tree, when most decision variables have been instantiated.

The global constraints of CHIP [23] offer better bounds over generic problem categories [24], but in order to compute the lower bound they do not solve to optimality a relaxed problem. For instance, CHIP offers constraints like cumulative that helps the propagation for resource constrained problems like job shop. To maintain generality, however, the cumulative cannot be specially tailored for every kind of job shop problem we face. This is the main reason why, using both global constraints together with other more specialized relaxations, we can further enhance the performance of the solver.

**OR lower bounding techniques**

In this section, we describe the basic ideas behind OR lower bounding techniques which have been widely used for solving hard combinatorial optimization problems. Interested readers can refer to [25] for a detailed explanation of this technique.

One of the most successful approaches for the solution of optimization problems relies on the branch and bound techniques that are based on the optimal solution of a relaxation of the original problem.

A relaxation of a problem \( P \) is obtained by removing a set of constraints of \( P \) in such a way that the resulting problem (the relaxed one) can be more easily (thus, more efficiently) solved. In addition, we are guaranteed that the optimal solution value of the relaxed problem \( f(x^*) \) provides an optimistic evaluation of the optimal solution value of the original one. \( f(x^*) \) is called lower bound for minimization problems, and upper bound for maximization ones. Indeed, the relaxed problem has
a wider feasible solution set with respect to the original problem. Thus, if the optimal solution of the
relaxed problem is feasible for the original one, it is also optimal for the original problem.

In branch and bound, the lower-bound computation is performed at each node of the search tree. A
node is characterized by the set of branching decisions, for example, variable instantiations, performed
in order to reach the node itself. These branching decisions are taken into account when computing the
relaxation at the node. In minimization problems, if the optimal solution of the relaxed problem at a
given node of the search tree has a value greater than the best solution found so far, we can prune the
whole sub-tree starting from that node since it does not contain any solution that improves the objective
function. If it is not the case, the search goes on in order to further refine the relaxation by branching
on one selected variable. The process interleaving bound computation and search goes on until the
original problem is solved to optimality or is proven infeasible.

Note the difference between this kind of pruning of the search space and that performed by
traditional CLP systems. The pruning achieved thanks to lower-bound computation is a pruning based
on optimality reasoning since it prunes portions of the search space that will not lead to better solution
with respect to the best one found so far.

On solving to optimality, the relaxed problem provides, in general, two important pieces of
information: the optimal solution of the relaxed problem \( x^* \), i.e. an assignment of values to variables,
and the value of the objective function of such an optimal solution \( f(x^*) \). These pieces of information
can be used in order to achieve a better pruning of the search space and to define more informed search
strategies. We will explain how to exploit the information provided by the solution of the relaxed
problem in a CLP environment.

THE SCHEDULING APPLICATION

The problem domain

In this section, we describe the job shop scheduling problem in the field of production planning for
an orthopaedic company. The system schedules about 150 jobs each week with an average of 20 tasks
each over 20 productive resources, considering alternative processing units and trying to minimize the
schedule makespan.

Formally, we have a set of jobs \( J_1, \ldots, J_n \). Each job \( J_i \) is composed of a set of tasks \( T_{i1}, \ldots, T_{im} \)
that have to be processed sequentially in order to complete the job. Each job is identified by:

- a unique identifier, i.e. name;
- a minimum starting day, i.e. release_date;
- a maximum allowed ending day, i.e. due_date;
- the list of tasks composing the job, i.e. taskList.

Tasks in the taskList appear in the order they should be executed. Each task belonging to the job
jobName is identified by:

- a unique name, i.e. taskName;
- a duration, i.e. duration, given in hours;
- a resource class, i.e. resource, with finite capacity;
the minimal number of days that the task should remain in the task queue before being processed, i.e. queue;

- the minimal number of days that the tasks belonging to the same job (i.e. following taskName in the taskList) should wait after the end of the execution of taskName, i.e. wait_after.

Once started, the processing of a task cannot be interrupted and must be completed within the same day. The tasks belonging to a job have to be processed in a strict order, with no alternative processing sequences. Of course, we have to schedule the tasks in time in order to satisfy the release date and the due date of the job they belong to.

Each resource class is identified by:

- a resource class name, i.e. resource;
- a number instanceN of equivalent alternative resource instances;
- the maximum working time for each resource in a single day, i.e. timeLimit;
- a preferred limit of the working time within a single day, i.e. timePreferred.

Each resource instance may work on one task at a time, or may be waiting between the processing of two tasks. The resources are available over the whole scheduling span, and resource instances belonging to the same resource class are equivalent and interchangeable. Thus, the processing time of one task is fixed and does not depend on the particular resource we choose to process it.

Our goal is to find a feasible solution for the problem that minimizes the makespan of the schedule. It is worth noting that, while a solution of our problem is an assignment of the starting time of each task whose granularity goes from 5 to 15 min, we only consider days in our cost expression. In fact, for a customer, it is equivalent if a job gets done in the early morning or late in the afternoon. What is important is the delivery day. On the contrary, the finer grained schedule should be provided to production managers who allocate resources to tasks by respecting the timings computed by the scheduler.

**Variables and constraints**

In order to define the problem model in a clean and readable way, we associate three domain variables to each task: Time, Day and Start. The time line is divided into a set of days, and each day is divided into a set of 96 time slots. In fact, we have 96 slots of 5 min contained in an 8 h working day. Thus, Time contains the temporal location of a task within the day the task starts and Day contains information on the day the task starts being processed. Finally, Start represents the absolute temporal location of the task along the time line. More precisely, we have imposed: Start = Day * 96 + Time. Note that, since each task starting on a given day must finish within that day, variables Start and Time domain do not contain forbidden starting times that lead a task being spread over two days.

Clearly, using three problem variables for each task is redundant (variable Start embeds all the information on the task allocation), but allows us to model problem constraints in the most natural way. This redundancy introduces an overhead due to the propagation of the above constraints. However, it simplifies the problem representation and the constraint handling.

As previously mentioned, we have temporal constraints between couples of tasks. Precedence constraints can be expressed in a very simple way: if task T_i and task T_j belong to different jobs and T_i has to precede T_j, we impose Start_i + duration_i ≤ Start_j. If the two tasks belong to the
same job, the second task has to wait \( \text{wait} \_\text{after} \) units of time after the end of the first one before being processed. Thus, we impose \( \text{Start}_i + \text{duration}_i + \text{wait} \_\text{after}_i * 96 \leq \text{Start}_j \). In order to further enhance propagation, we state on the variables belonging to the same job a redundant precedence constraint \([23]\), available in CHIP, that enforces the propagation, combining knowledge deriving from both resource assignment and precedence between tasks.

Finally, concerning the temporal constraint, for each task \( T_i \) we impose \( \text{Day}_i > \text{queue}_i \), and we have to respect the release date and due date of the job \( J_k \) the task belongs to, i.e. \( \text{Day}_i \geq \text{release}_k \) and \( \text{Day}_i \leq \text{due}_k \).

To express the maximum resource capacity, for each resource we state a cumulative constraint \([23]\), available in CHIP, on the tasks that must be processed by that resource. These constraints impose a limit on the maximum number of tasks, requiring the same resource category, that can be processed at the same time. The form of these constraints is:

\[
\text{cumulative}([\text{Start}_1 \ldots \text{Start}_k], [\text{Duration}_1 \ldots \text{Duration}_k], [1 \ldots 1], \text{instanceN})
\]

We also use a cumulative on variables \( \text{Day} \) in order to express the constraints on the maximum work per day and resource allowed for a feasible schedule. The form of these constraints is:

\[
\text{cumulative}([\text{Day}_1 \ldots \text{Day}_h], [1 \ldots 1], [\text{Duration}_1 \ldots \text{Duration}_h], \text{timeLimit} * \text{instanceN})
\]

The graphical representation of the two constraints is given in Figure 2. On the left-hand side, we have the first cumulative constraint. The grey boxes represent tasks: the \( x \)-coordinates represent the duration of the tasks, while the \( y \)-coordinates represent the resource use which is limited by the number of resources available in a given class, i.e. \( \text{instanceN} \). On the right-hand side, we have the second cumulative constraint. The grey boxes again represent tasks: the \( x \)-coordinates represent the days where tasks are scheduled, while the \( y \)-coordinates represent the resource use (intended as the duration of the task using a particular resource), which is limited by the number of resources available in a given class multiplied by the \( \text{TimeLimit} \) imposed for a given resource, i.e. \( \text{instanceN} * \text{TimeLimit} \). Note that \( \text{TimeLimit} \) is global for a resource class and it is not related to the single instance.

The parameter \( \text{TimePreferred} \) is used in order to balance the use of the resources. It is a soft constraint and the search heuristic used takes into account this balance. However, it is not imposed as a hard constraint in the problem formulation, as done for \( \text{TimeLimit} \).

---

Table I. Description of the tasks belonging to the first job, j01.

<table>
<thead>
<tr>
<th>Job</th>
<th>taskName</th>
<th>resource</th>
<th>duration</th>
<th>queue</th>
<th>wait after</th>
</tr>
</thead>
<tbody>
<tr>
<td>j01</td>
<td>t0101</td>
<td>cad</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j01</td>
<td>t0102</td>
<td>cam</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j01</td>
<td>t0103</td>
<td>cad</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j01</td>
<td>t0104</td>
<td>termoformatore</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>j01</td>
<td>t0105</td>
<td>termoformatore</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>j02</td>
<td>t0201</td>
<td>cad</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j02</td>
<td>t0202</td>
<td>termoformatore</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j02</td>
<td>t0203</td>
<td>cad</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Problem objective and lower bound

The job shop scheduling application we have faced requires optimal schedules with respect to the makespan criterion. The makespan is the temporal extension of the entire set of jobs involved in the scheduling. So, given \( n \) jobs \( j_1 \ldots j_n \), and \( End_{j_k} \) the ending time of job \( j_k \), minimizing the makespan means finding a sequence of tasks that minimizes \( \max_{k=1\ldots n}(End_{j_k}) \).

A common relaxation for this problem is the single-machine lower bound. We have implemented a CLP adaptation of the shifted bottleneck procedure proposed by Adams et al. [26]. The lower bound is obtained as follows: in a job shop problem each task is bound to a single resource class, say \( R_i, i = 1 \ldots m \). For each resource, we consider only the tasks using that resource, and we solve the scheduling subproblem by optimally sequencing the selected tasks. Although the single-machine problem is NP-complete in the strong sense [27], it is known that branch and bound algorithms are able to efficiently solve fairly large problems with data drawn from a realistic range. Note that, since the cost function is computed in terms of the days, the information on the makespan is also computed in terms of days needed to process all the tasks sharing the same resource. If \( TR_i, i = 1 \ldots m \) is the makespan bound calculated for each resource in the original problem, we have a bound for the makespan of the original problem by calculating \( \max_{i=1\ldots m}(TR_i) \).

We implemented the single-machine relaxation by adapting the algorithm proposed in [26]. The approach they used is to sequence the machines one at a time. For each machine not yet sequenced, they solve to optimality a one-machine scheduling problem. Every time a new machine is sequenced, they re-optimize previously sequenced machines by again solving a one-machine scheduling problem. The re-optimization is needed because the procedure defined in [26] considers and solves all machines independently. By using CLP, we are able to take into account problem constraints which enable one to solve single-machine scheduling problems which are tightly connected. In fact, in the lower bound computation, the constraints imposed on the original problem variables are still active and help in reducing variable domains. In contrast, the single-machine lower bound computation in the original OR implementation does not take into account the global constraints stated on the original problem. That is, when the lower bound is calculated at some point of the search process, the OR lower bound algorithm does not consider, for example, precedence constraints stated between couples.
of tasks of the same job that do not share the same resource. In addition, it does not consider additional constraints on the specific domain, such as maximum working time per period limitations (day, week, etc.) or other restrictions required by the single problem to be solved.

More details on the implementation will be given in the rest of the paper.

Example

We present here, as a simple example, a small instance of the job shop scheduling problem with the corresponding solution. We consider two jobs $j_01$ and $j_02$. The first one has a release date equal to 0, the second one equal to 1, while both have a due date equal to 16.

Each job has an associated a task list. For the first job, $j_01$, the task list is composed by five tasks $[t_0101, t_0102, t_0103, t_0104, t_0105]$ which should be sequenced in the order they appear in the list. The second job, $j_02$, is composed by three tasks: $[t_0201, t_0202, t_0203]$.

We have three resource classes each containing two resource instances: two cad operators, two cam operators and two termoformatori, whose maximal availability per day is 50 and the preferred availability is 5. The description of tasks belonging to the first job is reported in Table I.
The solution of the first day of the schedule is shown in Figure 3 through a user friendly interface which will be described later. Note that all tasks which could be scheduled on the first day (i.e., whose queue days are set to 0) are in fact scheduled for the first day. Note also that t0105 is not scheduled for the first day since the parameter wait after of task t0104 is set to 1. Thus, tasks following t0104 on the same job should wait one day before being processed.

COMBINATION OF CLP AND OR

Motivation

According to the problem modelling previously defined, we have first implemented a pure CLP algorithm for solving the problem without computing any lower bound on the problem.

We encountered two main problems: the first concerns the low efficiency of the resulting algorithm, and the second concerns the high dependence of the system performances on the input data configuration. We now explain the main reasons for the two problems and in the next section define an approach that overcomes both limitations.

Concerning efficiency, CLP has effective domain reduction algorithms, particularly suited for scheduling problems [6–8]. Filtering algorithms enable the search space to be effectively reduced and feasible solutions to be found rather quickly. When coping with optimization problems, we look for an optimal solution among the feasibility set. As mentioned, CLP systems usually provide built-in predicates for optimization problems, which enable the best solution to be found by implementing a very simple branch and bound algorithm. The idea is to solve a set of satisfiability problems (i.e., a feasible solution is found if it exists), leading to successively better solutions. 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 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) only the upper bound of the domain of the objective function is reduced; and (ii) in general, the link between the variable representing the objective function and problem decision variables is quite loose and does not produce effective domain filtering.

In CLP, the objective function is represented by a domain variable $C$ ranging over $[C_{\text{min}}, \ldots, C_{\text{max}}]$ which is a function of the problem decision variables, i.e., $C = f(X_1, \ldots, X_n)$. Thus, the domain reduction of $C$ can be reflected on decision variable domains and vice versa. $C_{\text{min}}$ represents the problem lower bound, and $C_{\text{max}}$ the problem upper bound. In CLP the bounds of the domain of $C$ are computed starting from domain variables. Suppose, for example, that $C$ represents the makespan of a schedule. $C_{\text{max}}$ at the beginning of the constraint solving process represents the schedule horizon, and it is updated each time a solution is found. The value $C_{\text{min}}$ is set to the minimal ending time of each task. If the earliest ending time of the tasks changes, $C_{\text{min}}$ is updated accordingly. The fact that the $C$ bounds are updated during search is important since they provide information on the range where the optimal solution can be found. If the connection between $C$ and the problem decision variables were tight, the information provided by the upper bound (the best solution found) would be enough to find the optimal solution and prove optimality rather efficiently. Unfortunately, this is, in general, not the case in CLP. Thus, good information on the problem lower bound could be useful in order to speed up
the search. In addition, it may be less computationally expensive to compute a good lower bound $C_{\text{min}}$ than it is to perform effective constraint propagation from a given upper bound $C_{\text{max}}$.

As an example, consider three tasks sharing the same resource, whose durations are 9, 30 and 40, and whose ending time variables have the following domains: $E_1 :: [1\ldots20]$, $E_2 :: [10\ldots100]$, $E_3 :: [1\ldots100]$. We consider in this example that the queue and wait after parameters are equal to 0 for all tasks. The corresponding makespan variable domain is $C :: [10\ldots100]$. Now, suppose that the first task is scheduled to start at time 1. The end of task 1 is set to 10, i.e. $E_1 = 10$. This assignment does not produce any propagation on variable $C$. Moreover, suppose it is discovered that task 2 should end before time 50: the corresponding variable domain changes to $E_2 :: [10\ldots49]$. Again, this reduction does not affect the bounds of the variable makespan. Instead, by computing the lower bound on the problem gives a result which is at least 79. This value can be used as a lower bound for variable $C$.

The second problem is related to the first and concerns the variations of system performances to slightly different input data. In some problem instances, we have found that a pure CLP approach achieves good results, and we have been able to solve to optimality problems with 10–20 jobs with an average of 20 tasks each (still far from the size of real problems). However, by changing these data configurations by removing tasks, we obtain smaller (loosely constrained) instances which in general should be solved more efficiently. This does not happen and in some cases we could not even solve instances with five jobs of 20 tasks each. These instances are loosely connected and present a large number of feasible solutions which should be explored in order to find the optimal one: thus, on the one hand, classical CLP propagation algorithms are not very effective and, on the other, the loose connection between the objective function and problem decision variables greatly affects the overall system performance.

To achieve more independence from data configuration and improve the system performance, we inserted a lower bound and information on the costs in the CLP scheme and used this information both for pruning purposes and to guide the search.

**Problem solving method**

In this Section, we provide a high level description of the proposed technique, aimed at identifying the basic solving components and their interaction during the solving process. The aim of this description is to provide general guidelines from a software engineering perspective that enable the integration of OR techniques in CLP solvers.

CLP(FD) provides declarative, expressive and flexible modelling abstractions, i.e. global constraints that interact with each other through shared variables, as depicted in Figure 1. CLP(FD) enables rapid prototyping, a compact representation of the problem and an easy definition of new search strategies.

Concerning the solving part, the core component of the application is the CLP(FD) solver exploiting domain reduction thanks to constraint propagation and problem dependent search strategies. The pruning performed is on the basis of feasibility reasoning, even if the constraint linking the objective function with decision variables is considered for propagation. However, as previously mentioned, this constraint triggers very little propagation, if any.

Concerning the optimization side, we propose to embed in the solver architecture an additional component, called the optimization component in charge of computing the optimal solution of a relaxation of the problem. The optimization component behaves similarly to global constraints in the
sense that it provides information that can be used in order to prune the search space. This time, however, pruning is based on optimality reasoning.

The optimization component and the CLP(FD) solver exchange results during the search process. The CLP(FD) solver produces the relaxed problem instance $P_{rel}$ in terms of variables, constraints and objective function. The first instance of $P_{rel}$ is computed before starting the search strategy. During search, at each node of the search tree, $P_{rel}$ is updated according to the current variable and domain configuration. For example, if at a certain node, a set of variables are instantiated, the relaxation can be simplified by considering these variables already fixed. The relaxation provides two pieces of information: the relaxed problem optimal solution $x^*$ and its objective function value $f(x^*)$. The optimal solution $f(x^*)$ is linked to the objective function of the original problem, representing its domain lower bound. The $x^*$ can be used in order to guide the search.

The resulting architecture (Figure 4) is an hybrid solver exploiting both pruning for feasibility and pruning for optimality. The integration proposed is very tight since the two solvers interact and exchange results during the whole search process. They are triggered each time new useful information is provided. When one of them detects infeasibility, the overall process fails and backtracking is performed.

The optimization component can be any kind of solver able to efficiently compute the optimal solution of the problem it receives as input. Examples of similar architectures have been widely studied [9,17,28]. The relaxation considered in these cases is a linear problem where continuous variables are linked by linear constraints. As an optimization component, a linear solver embedding a simplex algorithm was used.

In this paper, since the relaxed problem can be solved with CLP(FD) techniques in reasonable time, we have chosen to merge the two components into a single one, but still have a conceptual separation of the tasks performed by the two software blocks. In the next section, we provide details on how the information exchanged can be exploited by two components.
The integration

We now describe how information deriving from each solver in Figure 4 can be exploited by the other one. Let us start from the information provided as input to the optimization component. The CLP(FD) solver decomposes the model of the scheduling problem into a set of instances that independently consider single machines. Basically, we relax the constraints linking couples of tasks which should be executed on different machines. All these single-machine problems are provided to the optimization component, which schedules tasks in each problem separately and takes the maximum value of the solutions. During the search, constraints are propagated and branching decisions are taken in the CLP(FD) solver. Thus, new information such as variable bounds or variable fixing can be passed to the optimization component in order to refine the relaxation.

The exploitation of results, coming from the optimization component, is the most interesting part of the hybrid solver. Solving relaxed problems to optimality, in a CLP scheme, mainly serves three objectives:

(i) pruning the search space by reducing the objective function variable bounds more effectively than pure CLP bounds based on domain reasoning;
(ii) pruning the search space by removing from variable domains those values which cannot improve upon the best solution found so far;
(iii) defining cost-based heuristics for the selection of the value to assign on the basis of lower bound calculation, thus implementing a sort of best bound first (BBF) search strategy.

The first propagation (point (i) in the above list) is straightforward. It is a simple improvement of the classical CLP branch and bound technique. By quickly solving a relaxed problem, we can obtain tighter lower bound information and, thus, we have the possibility of achieving a better pruning of the search space. If, at a certain node of the search tree, a lower bound \(LB\) is computed, it can be used to tighten the value of \(C_{\text{min}}\). This is the OR classical propagation based on lower bound calculation. Obviously, if the lower bound computed is greater than the current \(C_{\text{max}}\), the search process can be stopped and backtracking triggered.

The second propagation (point (ii) in the above list), instead, is much more powerful since it is a direct propagation on decision variables. Consider a variable \(X_i\) whose domain contains values \(v_{i_1}, \ldots, v_{i_n}\). Intuitively, for each domain value \(v_{ij}\), we can compute a lower bound of a problem generated if \(v_{ij}\) is assigned to \(X_i\), i.e. if \(v_{ij}\) is part of a solution. If the resulting lower bound is greater than \(C_{\text{max}}\), obviously the value \(v_{ij}\) cannot be part of a solution whose cost is better than the best one found so far. More formally, if \(LB|_{X_i=v_{ij}} > C_{\text{max}}\) then \(v_{ij}\) is removed from the domain of \(X_i\).

This propagation, called cost based domain filtering, was proposed in [28,29]. It was applied to all variables not yet instantiated. However, performing this propagation is convenient when the algorithm used in order to solve the relaxed problem to optimality is polynomial. In this paper, even if very efficient, the solution of the single-machine lower bound is an NP-complete problem. Thus, the computation of a lower bound for each value in the domain of uninstantiated variables has been found to be too heavy in terms of a trade-off between the propagation achieved and the computational effort. Thus, it was restricted during the labelling phase. After the variable selection, say \(X\), we trigger this propagation only on values belonging to the domain of \(X\). Even if restricted to the labelling phase, this propagation significantly reduces the search space to be explored since it performed a deep look-ahead on each value instantiation.
Concerning point (iii), we can exploit pieces of information deriving from the solution of the relaxed problem both for choosing the next variable to be assigned and for choosing the value to be assigned next.

The solution of the shifted bottleneck procedure provides important information on the criticality level of a resource. In particular, the criticality level of a given machine is measured in terms of bottleneck quality, i.e. the value of the optimal solution of the single-machine subproblem. Thus, we first select resources with high criticality, i.e. those resources representing a bottleneck for the overall problem. Within tasks using that resource, we use a chronological order selection, widely used in scheduling applications. Thus, we select the task which can be scheduled first.

Concerning value selection, evaluating lower bounds for each variable at each choice point serves as a general, domain-independent heuristics for the value selection strategy, thus implementing a sort of best bound first (BBF) search strategy (or frontier search). In fact, after choosing the next variable to instantiate \( X_i \), we use the solution of the relaxed problem as a tentative value.

In this way, we can first explore more promising branches, i.e. branches which can lead to a better solution. The strategy implemented cannot be viewed as a real OR BBF strategy since we have a depth-first component, while the BBF strategy inherently embeds a breadth-first component. However, even with a depth-first strategy, the lower bound on domain values computed at each node provides important information on the best branch to explore.

A similar technique was recently proposed also in [30] in the field of dynamic scheduling where a solution should be changed according to new problem specifications and requirements. They develop the so-called probe backtracking algorithm, i.e. a search strategy based on the values proposed by optimally solving a subproblem representing a relaxation of the original problem.

IMPLEMENTATION DETAILS

In this section, we explain how the lower bound is computed and how the information deriving from the optimal solution of the relaxed problem is exploited in the search strategy. We describe the main steps performed by different components of the hybrid solver. We then provide some code fragments and their relative explanation. However, the basic concepts are explained in the present section independently from the code.

In this paper, we have described an hybrid solver architecture which is composed of two parts: a CLP(FD) solver in charge of solving the problem by constraint propagation and search, and an optimization component which optimally solves a relaxation of the original problem. Then, we have argued that in the specific application described in this paper, the relaxation can be efficiently solved again by a CLP(FD) solver. Thus, we have implemented the optimization component by using the CLP(FD) language.

As depicted in Figure 4, the CLP(FD) solver provides to the optimization component the relaxed problem instance. Thus, it should first extract the single-machine subproblems from the original problem: we partition the status list of the original problem in order to isolate tasks which should be executed on the same machine.

Once the relaxation input data have been provided, the optimization component is triggered. In order to find the optimal solution of the relaxed problem, we have exploited the CLP(FD) solver itself and, in particular, the CLP(FD) branch and bound technique. Thus, the optimal solution of the
The relaxed problem was implemented on top of the CHIP finite domain solver. In fact, CHIP is not easily extensible with user-defined constraints and the corresponding propagation. Clearly, this choice affects the performance of the overall system. In fact, if the solution of the relaxed problem were embedded in the solver and if its computation could be triggered by domain reduction events (as proposed for example in [28]), the general algorithm would be much more efficient. However, relying on the implementation proposed, the resulting code is considerably more efficient than the pure CLP(FD) version.

The optimization component returns the optimal solution of the relaxed problem, i.e. the number of days needed for allocating tasks on each machine separately. Since we use the original problem variables in the lower bound calculation, we have that all the problem constraints are propagated during the lower bound calculation. The relaxed problem solution instantiates original problem decision variables. This instantiation should be undone. Thus, each time a solution is computed, we need to store the result in a global variable and undo all the instantiation performed. This can be achieved by forcing a backtracking after the computation of the optimal solution of the relaxation on each machine.

In order to achieve good performance with the lower bound calculation technique, we need a good variable and value selection heuristics for the relaxed subproblem. We have used a heuristic that selects the variables in a chronologically sorted order and assigns the minimum value in the variable domain. It is in general a good heuristic for the makespan optimization.

The optimal solution of the relaxed problem, besides ensuring the correct calculation of the bound, could be exploited for another important part of the scheduling process, that is, the explanation of failures. In fact, if a bound calculation fails on a particular resource in the highest levels of the search tree, it is a clear indication of the fact that such a resource represents a bottleneck in the overall schedule. In general, the solution of the shifted bottleneck procedure provides the criticality level of a resource.

Considering propagation explained in point (i), each time the optimal solution of the relaxed problem is computed, we update the objective function domain lower bound with the value computed.

Concerning points (ii) and (iii), we worked on the exploitation of the lower bound at each choice point, both for the selection of the value to be assigned at each choice point and for the pruning of variable domains. We first select the most critical machine, and among tasks using that resource the one which can be assigned first. Then, for every value in the selected variable domain directly involved in the cost expression (that is the day the task has to be executed), we calculate the lower bound corresponding to that assignment, obtaining a list of terms \( \text{bound}(\text{Value}, \text{Bound}) \).

The values associated with a bound which is greater than the best solution found so far can be deleted from the variable domain. In addition, sorting this list on an ascending value of the \text{Bound} parameter, we obtain a way to select, with respect to the cost value, the most promising \text{Value}. In this way, we achieve a sort of BBF (or frontier) search process with a depth-first component since we first select the variable (depth first) and then we choose the value on the basis of the lower bound computation. This leads to obtaining better cost values of the first solution found, thus a better upper bound and higher effectiveness in pruning the search tree at the beginning of the search process. Acting during search, the reduction of the variable domains is performed by excluding from the selected variable domain (prior to instantiation) those values for which we have

\[
\text{bound}(\text{Value}, \text{LB}) \text{ with } \text{LB} \geq \text{UpperBound}
\]
This technique solves one of the worst problem of constraint programming systems when coping with optimization problems, especially when dealing with loosely constrained problems, where a very large number of feasible solutions exist, often with little variation with respect to the cost value.

This sort of propagation has been proved to be useful not only when applied locally to the variable involved in a choice point of the search process, but also as a way of reducing the domains of the whole set of uninstantiated variables before actually starting the search. In fact, after obtaining a good upper bound \( C_{\text{max}} \) exploiting a good heuristic on the problem, we effectively prune the search space before actually starting the search phase, using the reduction algorithm based on the lower bound.

Note that this propagation mechanism was implemented on top of the CHIP solver. A more efficient implementation relies on the possibility of triggering this propagation each time the domain of variable \( \text{Day} \) changes. In fact, if we remove a value which is part of the optimal solution of the relaxed problem, we can trigger a new lower bound computation and update the objective function domain accordingly. However, this kind of propagation would be too expensive and a trade-off between the number of times we compute the optimal solution of the relaxed problem and the pruning performed should be found. This is the subject of current investigation.

**Comments on the code**

In this section we will comment on some pieces of code. Readers not familiar with CLP(FD) languages can skip this Section.

In Figure 5 we provide the main structure of the code for the optimal solution of the relaxation. The predicate \( \text{calc\_SMLowerBound} \) returns in \( \text{LowerB} \) the value of the bound (number of days) obtained for tasks using a single machine. It works on a list \( \text{RelaxationList} \) containing only tasks sharing the same resource, and returns in \( \text{LowerB} \) the optimal solution of the relaxation. The global variable \( \text{last\_lb} \) saves the value of the bound and the optimal solution of the relaxed problem when the backtracking is forced by the fail predicate. The variable \( \text{last\_lb\_ok} \) is needed to distinguish between a forced backtracking (if \( \text{last\_lb\_ok} \) has value 1) or a backtracking induced by some failure during the search process. The CHIP built in predicate \( \text{min\_max} \) implements the branch and bound procedure by imposing, each time a solution is found, that new solution should have a makespan shorter than the best one found. The heuristic used, i.e. \( \text{labeling\_rel} \), implements a chronological instantiation of variables.

Another interesting piece of code concerns the search strategy for solving the original problem (Figure 6). The search strategy is based on information derived by the bound computation. In particular, variable selection is performed by focussing on more critical resources first (i.e. \( \text{select\_critical\_resource} \)) and among the tasks using it, it selects that which can be executed first (i.e. \( \text{select\_first\_task} \)). Concerning value selection, the predicate \( \text{make\_lblist/2} \) builds \( \text{LBLList} \), a list of terms \( \text{bound(Value, Bound)} \) solving for every value in \( \text{ChoiceList} \) the relaxed subproblem. If the corresponding bound is greater than the current upper bound of variable \( \text{MakeSpan} \), the value is removed by the current domain of the variable \( \text{Task} \). On the contrary, the value is left in the domain.

Finally, after instantiation, a new computation of the relaxed problem is performed (indeed, only if the bound can change) and the variable corresponding to makespan updated. The predicate \( \text{calc\_lowerBound/2} \) partitions the task status list (\( \text{TaskList} \)) separating tasks that must be processed by different resources. Then, it creates the relaxation variable lists and solves the
calc_SMLowerBound(RelaxationList, LowerB):-
    schedule(MaxDay, _), % gets the temporal extension of the schedule
    LowerB :: 0..MaxDay, % the bound is a value between 0 and MaxDay
    setval(last_lb.ok, 0),
    make_lbcostlist(RelaxationList, CostList), % builds a cost list for the bound
    min_max(labeling, rel(RelaxationList), CostList), % optimal solution of the relaxed problem
    setval(last_lb, LowerB, RelaxationList), % saves the LB and the relaxed problem solution
    setval(last_lb.ok, 1), % flag of forced backtracking
    fail. % forces a backtracking

calc_SMLowerBound(RelaxationList, LowerB):-
    getval(last_lb.ok, LastLbOk)
    LastLbOk > 0,
    getval(last_lb, LowerB, RelaxationList).

Figure 5. Computation of the optimal solution of the relaxed problem.

labeling([], []).  % extracts domain values for Task
labeling(TaskList, MakeSpan):-
    select_critical_resource(TaskList, Resource),
    select_first_task(Resource, Task1),
    dom(Task1, ChoiceList),
    make_lblist(Task1, TaskList, MakeSpan, ChoiceList, LBList), % creates LBList, List of terms bound(Value, bound)
    sort(LBList, LBList1, 2), % sorts LBList on ascending Bound value
    member(bound(Choice, LBValue), LBList1), % choice point on LBList1
    Task1 = Choice,
    calc_lowerBound(TaskList, LowerBound),
    LowerBound #< MakeSpan,
    delete(Task1, TaskList, TaskListNew),
    labeling(TaskListNew, MakeSpan).

Figure 6. Search strategy based on bound computation.

relaxed problems, as previously explained, via the predicate calc_SMLowerBound/2. Finally, calc_lowerBound/2 returns in LowerBound the maximum value obtained. This is a lower bound for the original problem. The constraint LowerBound #< MakeSpan updates the lower bound of the Makespan domain and checks whether the current upper bound (the best solution found so far) is strictly greater than the lower bound calculated by calc_lowerBound/2. If that is not the case, the computation fails and backtracking is forced. This propagation ensures that the current partial solution could lead to a better solution than the best one found so far in the search process.
Interface

The scheduler has been interfaced both with the Officine Rizzoli data bases and with the user. The data bases contain information on jobs, the corresponding tasks, their durations and resource requirements and on the available resources. The user interface provides the user a friendly environment for input and output data. In particular, the user selects jobs to be scheduled, the resources available and the schedule time limit. The corresponding information is retrieved from the data base and is provided as input to the scheduler. As a result, the user obtains the Gantt chart (as shown in Figure 3) and a textual description of the optimal schedule.

The interfaces with the data base and the user have been implemented in Delphi 4, a rapid application development (RAD) tool for Windows with drag-and-drop visual programming and a visual component library.

An interesting extension that we are currently developing for the interface is the possibility of guiding and helping the user in selecting jobs to be scheduled, in defining the temporal horizon for the schedule through a pre-processing step. In fact, by exploiting the single machine lower bound, or a simpler lower bound calculation, users can be a priori informed on the infeasibility of a given instance they are composing, on the minimum time limit that allows potential feasibility and an explanation of failures due to the limited availability of a given resource.

RESULTS

By integrating OR techniques in CLP we have achieved some major benefits over a pure CLP approach: more independence of the algorithm from input data, the ability to solve larger and harder problems, the ability to solve in a complete way loosely constrained problems with (relatively) high efficiency.

In Table II, we report some results of the code implemented in CHIP [20] for solving instances of the scheduling problem described in this paper. The results have been obtained on a workstation SUN Sparc 10. For each row, we report average results obtained on 20 instances for each number of tasks.

In Table II we consider three cases:

- the pure CLP code using the CHIP min\text{-}max optimization predicate, (column CLP);
- the CLP code exploiting the single machine lower bound used in order to prune the cost variable domain only, i.e. the propagation described in point (i), in the section ‘COMBINATION OF CLP AND OR’, (column CLP+LB);
- the CLP code exploiting all the pruning described in points (i) and (ii) of the section ‘COMBINATION OF CLP AND OR’, and the lower bound based heuristic, (column CLP+LB+BBF).

For each technique, we report the computational time in seconds spent in order to find the first solution (columns First) and the time spent in order to find the optimal solution (columns Optimol). For some problem instances we have omitted the result (‘—’ in Table II) since the optimal solution was not found after 60 min.

As can be noted, while the pure CLP approach outperforms both approaches which use information provided by the lower bounds in finding the first solution, it fails to obtain the optimal solution in a reasonable time. This can be explained as follows: the CLP-OR approaches exploiting the lower
Table II. Computational results. The times are reported in seconds.

<table>
<thead>
<tr>
<th>Number of tasks</th>
<th>CLP</th>
<th>CLP+LB</th>
<th>CLP+LB+BBF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>First</td>
<td>Optimal</td>
<td>First</td>
</tr>
<tr>
<td>75</td>
<td>3</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>150</td>
<td>7</td>
<td>—</td>
<td>9</td>
</tr>
<tr>
<td>300</td>
<td>10</td>
<td>—</td>
<td>15</td>
</tr>
<tr>
<td>1500</td>
<td>30</td>
<td>—</td>
<td>50</td>
</tr>
<tr>
<td>2250</td>
<td>45</td>
<td>—</td>
<td>120</td>
</tr>
</tbody>
</table>

bound spend more time (due to the overhead introduced by the lower bound computation) in order to find the first solution, which is usually a good one, while the pure CLP code quickly finds the first (generally worse) solution. The convergence to optimality is obviously more efficient if we find a good first solution and if the information on the lower bound is exploited.

Note that by using the integration proposed, we can solve to optimality problems which are one order of magnitude greater than those solved by a pure CLP approach.

For some simple data configurations, however, the pure CLP outperforms our method based on branch and bound and obtains the proof of optimality even for large instances. In fact, for some instances of 200 tasks we have obtained the proof of optimality after 12 s. These results, however, are not robust, since by slightly changing one data configuration parameter, performances decrease significantly. With the enhanced lower bound-based approach, we obtain an increased robustness of the algorithm which does not depend on a parameter changing.

RELATED WORK

The study of hybrid algorithms for solving combinatorial optimization problems is an emerging research area. Hybrid algorithms are based on the integration of different approaches, such as constraint programming, linear programming, branch and bound techniques and local search. The strength of such algorithms in solving optimization problems arises from the fact that different techniques are suited to solving different aspects of the problem.

A recent ESPRIT project, CHIC-2 [11], started in 1996, is aimed at developing hybrid algorithms for large scale optimization problems (LSCO), to provide a methodology for the efficient exploitation of these algorithms and to build a platform to support the creation of applications to solve LSCO problems. It incorporates several solution methods and is designed to be extensible, since new solution algorithms can be easily incorporated as they are discovered.

The integration of existing techniques in hybrid algorithms starts from a comparison of different methodologies used for the same problem. A number of papers compare the performances of constraint programming-based solvers with those OR-based solvers (in particular, integer linear programming techniques) when applied to particular problems [14–16, 18, 31]. A recent paper that clearly compares
CP and mathematical programming approaches to discrete optimization problems is that by Heipcke [32].

In this paper we focus on those integrating OR lower bounding techniques in constraint programming. The resulting algorithms can have different levels of coupling among different techniques. In particular, in [13] a loose integration of constraint programming and OR techniques is performed to solve the fleet assignment problem. In this case, the integration takes the form of one technique that feeds its result to the second one. In fact, a constraint programming solver provides a good solution that is used as input to the mathematical programming system to ‘warm-start’ the simplex algorithm. In our approach, the constraint programming solver and the OR branch and bound technique are more tightly coupled and intertwine their execution to find an optimal solution.

A tight integration is performed in [17] where the authors define an integration of mixed integer programming (MIP) and CLP. A CLP solver performs local propagation on domain variables, while a MIP solver performs a global propagation based on the optimal solution of the relaxed problem. This solution assigns, in general, non-integer values to variables. A non-integer variable \( v_i \) whose value is \( s_{v_i} \) is selected and the problem is split into two subproblems containing, respectively, the constraints \( v_i \geq \lfloor s_{v_i} \rfloor \) and \( v_i < \lfloor s_{v_i} \rfloor \). This approach has some similarities with ours since we both use a lower bounding procedure to achieve additional pruning. They also use the relaxed problem solution in order to define an effective branching rule, but they do not remove from the variable domains any feasible value for the original problem. The constraint propagation proposed here possibly removes feasible values which cannot lead to any improvement of the best solution found so far.

In [9] the integration of a finite domain constraint solver and a simplex-based solver on real numbers is presented. The two solvers cooperate through variable sharing, thus achieving an effective value propagation. A similar propagation on the basis of the lower bound is presented in [33] where the branch and bound classical algorithm is improved by means of a powerful propagation on decision variables subject to the so-called task interval constraints.

In [12] and [34] an even closer integration of constraint programming and lower bounding techniques is performed and applied to the travelling salesman problem (TSP). In [29] the same technique was applied to a time-constrained variant of the TSP and in [28] the technique has been generalized. The lower bound calculation is used in order to prune both the objective function domain variable and the decision variable domains, as we perform in this paper. The difference with the present work is that in [12,29] the propagation is embedded in an optimization oriented constraint, while in this paper it is implemented on top of the constraint solver. Thus, our approach is less efficient, but more flexible since the user can implement and control the lower bound computation. In addition, in [12] a specific OR algorithm has been implemented and encapsulated in the constraint, in this paper, we have used the CLP(FD) solver itself in order to compute the lower bound. In [34], while exploiting the same propagation techniques as we do in this paper, they start over a new tree search each time a solution is found. This may lead to a redundant exploration, but ensures that the tree is ‘well built’ for the problem to be solved.

Concerning scheduling problems, we should mention the approaches exploiting information on bottlenecks on resources [35]. Minton et al. proposed an approach based on using an initial, partially feasible, complete assignment in conjunction with the min-conflict heuristics [36]. Information coming from the optimal solution of subproblems was exploited in the field of dynamic scheduling by El Sakkout and Wallace [30], who proposed the so-called probe backtrack algorithm where a backtrack search is supported by lookahead procedures which generate potentially good assignments (probes).
Quite recently, the integration of local search and meta-heuristics in constraint programming has been investigated (see, for example, [37–39]). Among incomplete methods to scheduling problems we should mention, for example, [40,41]. However, as the methods fairly different to ours, we can hardly compare the two. Local search and meta-heuristics in constraint programming lead to development of an incomplete search method, while we propose to use an exact algorithm which optimally solves (and proves the optimality of) the problem.

CONCLUSION AND FUTURE WORK

In this paper, we have proposed an integration of OR lower bounding techniques in CLP. We have used the lower bound information in order to achieve a more effective pruning of the search space and in order to define general domain-independent heuristics on value ordering. The advantages of CLP, such as rapid prototyping, clear and easily modifiable code and flexibility in dealing with new constraints, are maintained in the integrated approach.

We have implemented a working prototype by using the CLP language CHIP which provides very effective global constraints performing powerful propagations.

The results obtained are very encouraging since the integration enables us to outperform pure CLP approaches in finding the optimal solutions and is able to solve problems which are one order of magnitude greater than those solved by a pure CLP solver.

We are currently studying how to further improve the propagation from the objective function towards decision variables by applying it at each node of the search tree to the whole set of the uninstantiated variables. This, however, is a computationally expensive task, and we need to tune the amount of propagation, the search speed and memory requirements. We could reduce the cardinality of the set of the variables involved in propagation, by identifying a subset of the variables that could lead to a more effective reduction of the search space reasoning on the problem domain characteristics.

Further works are aimed at considering other objective functions. In fact, we believe that the makespan criterion, under certain real world conditions, does not represent an effective optimization condition. We are currently working on the application of our search scheme to a problem with a cost function that allows us to balance the use of resources throughout the whole schedule, minimizing the global use of resources over a certain limit, thus minimizing the need for overtime and always trying to keep a certain spare workpower for optimally dealing with emergencies.

We are currently studying improvements to the user interface which can (i) guide the user in the definition of (probably feasible) problem instances and (ii) explain failures due to limited resource availability.

ACKNOWLEDGEMENTS

We would like to thank Paola Mello and Evelina Lamma for useful discussion and suggestions. We also thank Officine Rizzoli for providing data for real problem instances. The authors’ work has been partially supported by CNR, Committee 12 on Information Technology (Project SCI*SIA). We thank also anonymous referees for interesting and detailed comments on an earlier version of this paper.
REFERENCES