Enhancing CLP Branch and Bound Techniques for Scheduling Problems

Filippo Bosi, Michela Milano
DEIS, Univ. Bologna,
Viale Risorgimento 2 I-40136 Bologna, Italy.
E-mail: mmilano@deis.unibo.it
E-mail: bosi@programmers.net

Abstract

Constraint Logic Programming has been proven to be a suitable tool for solving combinatorial problems. However, it presents some limitations in dealing with optimization criteria, since the pruning of the search space on the basis of the objective function is very weak. On the other hand, Operation Research lower bounding techniques provide a powerful way of reducing the search space on the basis of *objective function reasoning*. They provide effective bounds for the original problem by computing the optimal solution of a relaxed problem. In this paper, we propose an integration of OR lower bounding techniques in CLP in order to achieve the benefits of both paradigms. We have implemented this technique in the CLP language CHIP and applied it, as a case study, to a Job Shop Scheduling application in the field of production planning. By integrating the OR lower bounding techniques in CLP, we are able to achieve promising results and to optimally solve problems which are one order of magnitude greater than those solved by a pure CLP approach.

Keywords: Integration of OR Techniques in CLP, Scheduling Problems, Hybrid solver configurations

1 INTRODUCTION

In this paper, we present an approach for combining Constraint Logic Programming (CLP) and Operations Research (OR) techniques. CLP [17] is a powerful programming paradigm combining the advantages of logic programming and the efficiency of constraint solving. Many real life applications, such as scheduling, planning, sequencing and assignment problems, have been effectively solved by using CLP techniques (see [8, 9, 13]).

While current CLP systems provide very effective constraint propagation mechanisms, they present some limitations in dealing with objective functions. In particular, optimization predicates provided by CLP languages impose, each time they find a solution, that further solutions will have a cost better than the best one found so far. This mechanism triggers very little propagation (if any) on variable domains since usually the objective function is loosely connected with decision variables. For example, in scheduling problems, the objective function which should be minimized is the *makespan*, i.e., the completion of the whole schedule. This objective function is equal to the ending time of the last job. Thus, the decision variables domain reduction loosely affects the bounds of the objective function and vice versa.

OR techniques provide a powerful way of coping with optimization predicates. In particular, Branch & Bound techniques [25] allow to prune the search space on the basis of the lower bound calculation. A lower bound is an optimistic value of a solution of the original problem obtained by optimally solving a relaxed one, i.e., a problem where some constraints have been relaxed. The better is the lower bound, the more accurate is the valuation of the optimal solution we can find in a given subtree.

In this paper, we argue that by combining Operations Research techniques in Constraint Logic Programming makes it possible to efficiently solve large size and hard problems. We obtain a performance improvement of the resulting solver and we are able to optimally solve problems which are one order of magnitude greater than those solved by a pure CLP approach.

For this reason, the integration of CLP and OR techniques is an emerging research area (see for example [3, 4, 7, 11, 12, 14, 19, 22, 23, 24]) since it gets the advantages from both fields. From OR, we get a better, effective way of exploring the search space, while retaining the CLP declarative semantics easing the problem modeling, and its effective constraint propagation. Moreover, we can further improve the effectiveness of lower bound coming from OR, by combining propagation techniques and domain reasoning with results coming from relaxations of the original problem. In fact, we define 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 problem global constraints through shared variables.

We have applied these techniques to a Job Shop Scheduling application for Officine Rizzoli, a company producing orthopedic prothesis and girdles for a well-known italian group of orthopedics hospital departments. The system, written in CHIP [10] 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. Another optimization parameter we are currently taking into account is the resource use balance. Resource balance should be obtained in order to avoid resorting to unnecessary overtime and to have some available work power for effectively dealing with emergencies.

Using only Constraint Logic Programming techniques we could optimally solve problems with 10-20 jobs with an average of 20 tasks each. In addition, CLP techniques suffer from heavy dependency of the solver from the data configuration. For some very loosely constrained problem instances, we could not cope with more than 5-7 jobs of 20 tasks each. Thanks to the integration proposed, we optimally solve real problems of 150 jobs and the code is less dependent on data configuration.

The paper is organized as follows: in section 2 we recall some concepts on CLP and OR techniques. In section 3, we propose the integration of the two techniques as a general problem solving methodology. Section 4 presents the case study in the field of Job Shop Scheduling. The specific problem is described, along with variable and constraint formalization. In addition, a lower bound description and its integration in a CLP framework is presented. Section 5 is devoted to implementation details. In section 6 we present some computational results. Section 7 describes some related approaches. Conclusion and further work follow.

2 PRELIMINARIES

2.1 Constraint Logic Programming

Constraint Logic Programming (CLP) [17, 16] is a class of programming languages combining the advantages of Logic Programming (LP) [20, 18] and the efficiency of constraint solving. CLP techniques, thanks to the active use of constraints, allow to *a-priori* prune the search space by removing combination 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 happened, by means of expensive backtracking.

In this paper, we focus on Constraint Logic Programming on Finite Domains CLP(FD) which has been successfully applied to several combinatorial optimization problems, see [8, 9, 13].

In CLP(FD) languages, variables range over finite domains of integers. The variable's domains represent possible values that variables can assume during the computation. For example, $X :: [1..10]$ states that variable X can assume one of the integer values from 1 to 10, $Y :: [3, 5, 9]$ 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 more powerful constraints with complex propagation mechanisms. A typical symbolic constraint is `atmost(Nb,[`$\mathtt{X}_1$`, ...`$\mathtt{X}_n$`],Val)`, available in most CLP(FD) solvers like CHIP [10]. Declaratively, the constraint `atmost(Nb,[`$\mathtt{X}_1$`, ...`$\mathtt{X}_n$`],Val)` holds if and only if at most `Nb` elements $\mathtt{X}_i$ are equal to `Val`. In the constraint, all the $\mathtt{X}_i$ are domain variables.

In many cases, an optimization criterion must be satisfied, i.e., we have to find the best of the feasible solutions. Therefore, many CLP languages provide *Branch & Bound* procedures for minimization or maximization.

The structure of a general CLP optimization program written in CHIP is the following:

```
optimize(Variables,C):-
     create_domains(Variables),
     create_cost(C),
     globalconstraints(Variables),
     min_max(labeling(Variables),C).

labeling([]).
labeling([Var|Rest]):-
     indomain(Var),
     labeling(Rest).
```

The predicate `create_domains(Variables)` creates variable domains, `create_cost(C)` defines the link between the cost variable `C` and decision variables `Variables`, the predicate `globalconstraints(Variables)` imposes symbolic and mathematical constraints on variables, while `min_max` searches for an optimal solution of the problem. The `labeling` procedure simply selects a variable and instantiates it to a value in its domain. More advanced labeling procedures can in general be defined.

The `min_max` optimization predicate provided by CHIP searches for a solution of a goal predicate that minimises a cost expression `C`. Every time a solution is found (with cost C^*) the search continues with the new constraint $C < C^*$. `C` is a domain variable which is linked to problem domain variables. By reasoning on domains, CHIP 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 greater value than the best solution found so far, `min_max` backtracks, searching for alternative partial solutions. The problem is that the lower bound obtained just by means of the original problem variable bounds is not usually a good lower bound for the problem itself. Therefore, pure Constraint Programming techniques are usually able to prune only the few levels of the search tree, when most decision variables have been instantiated.

On the other hand, by using a lower bound calculated exploiting a good relaxation of the original problem as in [11, 5, 6], it is possible to prune the search space more effectively. The global constraints of CHIP offer good lower bounds over generic problems categories, but they cannot be as good as special tailored relaxations exploiting extra knowledge of the problem domain. More precisely, the global constraints are based on generic relaxations over common problem categories. For instance, CHIP offers constraints like the cumulative that helps propagation for resource constrained problems like the job-shop. To mantain genericity, however, cumulative can not 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.

2.2 Operations Research Lower Bounding Techniques

In this section, we describe the basic ideas behind Operations Research lower bounding techniques which have been widely used for solving hard combinatorial optimization problems. For a survey, refer to [21].

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. Such problems cannot be solved by enumeration in a reasonable computing time since X may have a huge cardinality.

One of the most successful approaches for the solution of general integer linear programs relies on Branch & Bound techniques that are based on the optimal solution of a relaxation of the original problem.

A relaxation of the problem is obtained by removing a set of constraints of the original problem in such a way that the resulting problem (the relaxed one) is more easily solvable.

It is easy to see that the optimal solution value of the relaxed problem provides an optimistic value (*lower bound*) of the optimal solution value of the original one. Indeed, the relaxed problem has a wider feasible solution set and its objective function is not greater than the original one for all the feasible solutions to the original problem.

In section 3, we will explain how to exploit the information provided by the relaxed problem solution (the lower bound) in a Constraint Logic Programming environment.

3 COMBINATION OF CLP AND OR

The lower bound information, inserted in a CLP scheme, serves mainly for three objectives:

1. pruning the search space by reducing the objective function variable bounds more effectively than pure CLP bounds based on domain reasoning;

2. pruning the search space by removing from variable domains those values which cannot improve the best solution found so far;

3. defining general, domain-independent heuristics for the selection of the value to assign on the basis of lower bound calculation, thus implementing a sort of Best Bound First search strategy.

The pruning of the search is thus due to both a direct domain reduction on the variable representing the cost of the solution (i.e., the parameter that should be minimized) and a decision variable domain reduction (points 1 and 2 in the above list). Let us see how these propagations work. The first propagation (point 1 in the above list) is simply an improvement of the classical CLP branch and bound technique. In CLP, the cost variable C is a domain variable ranging on $[C_{min}..C_{max}]$ and is a general function of 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_{min} represents the problem lower bound. In CLP, however, the bounds of the domain of C are computed starting from domain variables. Suppose, for example, that C represents the makespan of a schedule. Value C_{min} is set to the minimal ending time of each task. This usually does not lead to a good propagation, especially for variables related to the tasks that have to be executed at the beginning of the schedule. We argue that, by quickly solving a relaxed problem during the labeling procedure, we can obtain tighter lower bound information and thus, we have the possibility of achieving a better pruning on decision variable domains. If, at a certain node of the search tree, a lower bound LB is computed, it can be used for tighten the value of C_{min}. This is the OR classical propagation based on lower bound calculation. Obviously, if the computed lower bound is greater than C_{max}, the search process can be stopped and backtracking triggered.

The second propagation (point 2 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_{i_j}, we can compute a lower bound of a problem generated if v_{i_j} is assigned to X_i, i.e., if v_{i_j} is part of a solution. If the resulting lower bound is greater than C_{max}, obviously the value v_{i_j} cannot be part of a solution whose cost is better than the best one found so far. More formally, if $LB|_{X_i=v_{i_j}} > C_{max}$ then v_{i_j} is removed from the domain of X_i. Obviously, the lower bound calculation should be computed efficiently.

Using OR techniques over domain variables attached to CLP constraints allows to implement OR algorithms that can experience further gains in precision of the lower bound because of the concurrent exploitation of constraints declaratively stated on the variables during both search and domain definitions. Executing the search using the original variables of the main problem allows to take into account, without any explicit coding, the main global constraints on the problem (for example in the field

of scheduling applications, we have capacity constraints, precedence between tasks belonging to the same job).

Concerning point 3, 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 having chosen the next variable to instantiate X_i, we can order its domain values $v_{i_1}, \ldots, v_{i_n}$ on the basis of the lower bound $LB|_{X_i=v_{i_j}}$. In this way, we can explore first more promising branches, i.e., branches which can lead to a better solution. The strategy implemented cannot be viewed as a real OR Best Bound First Strategy since we have a depth-first component, while the BBF strategy inherently embeds a bread-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.

In the next section, we apply this general method to a Job Shop Scheduling problem.

4 THE SCHEDULING APPLICATION

4.1 The problem domain

We have applied our algorithm to a Job Shop Scheduling Problem in the field of production planning for an orthopedic 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 constituted by a set of tasks that have to be processed sequentially in order to complete the job. Each job J_k has a minimum starting time and a due date (i.e. maximum allowed ending time).

Each task has to be scheduled in time and assigned to a disjunctive resource with finite capacity. One resource 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 the processing time of one task is fixed and does not depend on the particular resource we choose for processing it. 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 due date for every job. One interesting additional constraint on the problem is a time limit on the maximum working time for each resource in a single day.

Our goal is to find a feasible solution for the problem that minimizes the makespan of the schedule. While scheduling tasks deciding start times with a granularity of 5 to 15 minutes, we only consider days in our cost expression. In fact, for our 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. This point is very important, since it allows to efficiently compute the lower bound of each value (delivery day) in the domain of decision variables.

4.2 Variables and Constraints

In order to define in a clean, readable way the constraints on the problem, we have decided to express the starting time of tasks with a redundant set of domain variables. For every task, in fact, we consider the variables $Time$, Day and $Start$. $Time$ contains the time interval in the day the task starts, Day contains information on the day the task starts being processed. Finally, $Start$ is defined by means of a constraint stated using $Time$ and Day. More precisely, we have imposed $Start\# = Day * 100 + Time$. In fact, in our problem a day is divided into 100 time intervals.

We can express in a very simple way the precedence constraints on the single tasks, simply stating $Start_i + Duration_i\# < Start_j$ if $Task_i$ has to precede $Task_j$.

Other constraints on the schedule can be expressed by using global constraints available in CHIP [2]. To express maximum resource capacity, for each resource we state a `cumulative` constraint 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. We also use a `cumulative` using Day and $Duration$ of the tasks in order to express the constraints on the maximum work per day and resource allowed for a feasible schedule.

In order to further enhance propagation, we state on the variables a redundant `precedence` constraints, that enforces the propagation, combining knowledge deriving from both resource assignment and precedence between tasks belonging to the same job. Finally, a `diffn` constraint is used in order to constrain the assignment of each task to a single resource instance.

4.3 Problem objective and lower bound

Job Shop Scheduling applications usually tend to optimize schedules using the *makespan* criterion. The makespan is the temporal extension of the entire set of jobs involved in the scheduling. So, given n jobs $j_1 ... j_n$, said End_{J_k} the ending time of job J_k, minimizing the makespan means finding a sequence of tasks that minimizes $max_{k=1..n}(End_{J_k})$.

A common OR relaxation for this problem is the single-machine lower bound, also called the shifted bottleneck procedure [1]. This lower bound is obtained as follows: in a Job Shop problem each task is bound to a single resource class, say R_i $i = 1..m$. For each resource, we consider the tasks using only that resource, we solve the scheduling subproblem, with the additional relaxation of considering the tasks interruptible as many times as we need, obtaining a value of the makespan that can be calculated in polynomial time with the number of tasks. Note that, begin the cost function computed in terms of days, the information on the makespan is also computed in terms of days needed to process all tasks sharing the same resource. If TR_i $i = 1..m$ 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 $max_{i=1..m}(TR_i)$. Note that the relaxation introduced by the single machine bound to split into several parts the task processing has been chosen in order to keep the lower bound algorithm simple, and to avoid the need to implement a branch and bound techniques to calculate the lower bound itself. While this choice leads to an undeniable computation gain, it achieves a worse lower bound value, especially for highly constrained problems.

The lower bound calculation has been implemented in the CLP solver itself by

solving the relaxed problem obtained by removing both the non-preemptivity and the precedence constraints between tasks that share the same resource and thus solving a preemptive scheduling problem considering only tasks that use a single resource. The only constraints that are relaxed are the one involved in the subproblem to be solved. This allows to take into account all the problem constraints during the lower bound calculation since the `min_max` predicate used to calculate the lower bound still works considering constraints stated both the problem decision variables and on the new subproblem to be solved. On the contrary, 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. Neither does it consider additional constraints on the specific domain, such as maximum working time per period limitations (day, week...) or other restrictions required by the single problem to be solved.

In order to achieve good performance with this lower bound calculation technique we need a good value selection heuristics for the relaxed subproblem solution. In this case, the standard labeling strategy of CHIP that is selecting the minimum value in the variable domain, is a good heuristics for the makespan optimization. Otherwise the calculation of the lower bound could be too heavy, computationally speaking, even if selecting only tasks over single resource categories produces, under realistic conditions, a problem that is smaller than the original one by an order of magnitude.

5 IMPLEMENTATION

In this section, we explain how to use the lower bound information in the search strategy and provide some implementation details. Translated into CLP code, the lower bound technique proposed in section 4.3 is simply a matter of extracting, filtered on a single resource, a subset of the original problem's status list, then executing a branch and bound (`min_max` predicate) on the new problem obtained. Of course, using the original problem's status list for solving relaxations of the problem itself leads to wrong instantiations of the variables, from the main problem point of view. Using the undoing capabilities of the backtracking mechanism of CLP languages, keeping the code clean is just a matter of forcing a backtracking after having calculated the bound. Translated into CHIP code, this means

```
% +StatusList is the list of the tasks exploiting the same resource
% -LowerH     is the bound for that resource
calc_SMLowerBound(StatusList,LowerH):-
   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(StatusList,CostList), % builds a cost list for the bound
   min_max(labeling(StatusList),CostList), % branch and bound
   maximum(LowerB,CostList),
   setval(last_lb,LowerB), % saves the LB value
   setval(last_lb_ok,1),   % flag of forced backtracking
```

```
   fail. % forces a backtracking

calc_SMLowerBound(_,LowerH):-
   getval(last_lb_ok,LastLbOk),
   LastLbOk > 0,
   getval(last_lb,LowerH).
```

The predicate `calc_SMLowerBound` returns in `LowerH` the value of the bound (number of days) obtained for tasks using a single machine. The global variable `last_lb` saves the value of the bound when the backtracking is forced by the fail predicate. The other, `last_lb_ok`, is needed to distinguish between a forced backtracking (if `last_lb_ok` has value 1) or a backtracking induced by some failure during the search process. This additional information, beside ensuring the correct calculation of the bound, could be exploited for another important part of the scheduling process, that is 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 resource is one of the main responsible for a failure of the entire search process.

In order to enhance the search process to exploit a lower bound function, we have to modify the labeling procedure as follows

```
labeling([],CostList):-
     sum_cost(CostList,Value),
     newUB(Value).
labeling([Task|Rest],CostList):-
     indomain(Task),
     calc_lowerBound([Task|Rest],LowerBound),
     satisfyBound(LowerBound),
     labeling(Rest).
```

The predicate `calc_lowerBound/2` partitions the task status list (i.e., `[Task|Rest]`) separating tasks that must be processed by different resources. Then, `calc_SMLowerBound/2` schedules every set of tasks using the same resource, and `calc_lowerBound/2` returns in `LowerBound` the maximum value obtained. This is a lower bound for the original problem.

The predicate `satisfyBound/1` checks whether current upper bound is strictly greater than the parameter passed, i.e., the lower bound calculated by `calc_lowerBound/2`. The predicate `newUB/1`, called at the end of the labeling process, that is when a feasible solution has just been found, sets the new upper bound to the value of the cost of the solution itself. This value, as enforced by `satisfyBound/1`, must be strictly smaller than the last upper bound. The definition of `satisfyBound/1` is straightforward

```
satisfyBound(LowerBound):-
      upperbound(UpperBound), % gets the current upper bound
      LowerBound < UpperBound, !.
satisfyBound(_):-
      write('pruning...'),nl,fail.
```

satisfyBound/1 is checked after every assignment at a choice point. This ensures that the current partial solution could lead to a better solution than the best one found so far in the search process. The predicate newUB/1 sets the new upper bound for the search process by adding to the database the new term upperbound(UBValue), after having retracted older versions, i.e., older values of the upper bound.

```
newUB(UBValue):-
    retractall(upperbound(LastUB)),
    assert(upperbound(UBValue)).
```

newUB/1 is called at the end of the labeling phase, when all the variables have been instantiated, that is when a feasible solution has been found. The need to call retract and assert to save current upper bound value has to be found in the particular implementation of the min_max predicate of CHIP. When the search is restarted after a solution has been found, all the changes to global variables' values get undone, so global variables are not the correct way to save information about the search process progress.

The bounding strategy implemented using the improved lower bound helps speeding up the search process. In particular, we achieve the ability to prove optimality for bigger problems. However, there is still much room for improvement in the search process, exploiting other techniques derived from Operations Research.

We have been working on the selection of the value to assign at each choice point, exploiting the information deriving from the calculation of the lower bound. At each choice point, for every value in the selected variable domain directly involved in the cost expression (that is, in our problem, the day the task has to be executed), we calculate the lower bound corresponding to that assignment, obtaining a list of terms bound(Value, Bound). Sorting this list on ascending value of the Bound parameter, we get a way to select, with respect to the cost value, the most promising Value to choose in the search process.

Implementing this new search strategy in the program, is just a matter of substituting, in the labeling clause the call to indomain/1 predicate with the following code:

```
dom(Task,ChoiceList),   % extracts domain values for Task
make_lblist(Task, [Task|Rest], CostList, 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
Task = Choice.
```

The predicate make_lblist/2 builds LBList, a list of terms bound(Value, Bound) solving for every value in ChoiceList the lower bound information obtained by solving the relaxed subproblem shown in section 4.3. In order to complete its task it has to exploit the information coming from the status list composed by the variables still to be instantiated [Task|Rest] and, of course, the cost list CostList of the optimization problem.

What we achieve is a sort of Best Bound First (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 LB computation. This leads to much better cost values of the first solution found, obtaining better upper bound and higher effectiveness in pruning the search tree at the beginning of the search process. What is more, while integrating this new search method into CLP, we have seen the way to actively use the lower bound information in pruning the search space doing domain reduction, as explained in section 3 (point 2). Acting in the `labeling` predicate, reduction of the Task variable is performed by excluding from the selected variable domain (prior to instantiation) those values in `LBList` for which we have

`bound(Value, LB)` with `LB` $\geq$ `UpperBound`

This technique remedies one of the worst problem of Constraint Programming optimization systems when coping with optimization problems, especially when dealing with loosely constrained problems like Job Shop scheduling, 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 proven to be useful not only when applied locally to the variable involved in a choice point of the search process, but even as a way of reducing the domains of the whole set of uninstantiated variables before actually starting the search. In fact, having obtained a good upper bound C_{max} exploiting a good heuristics on the problem, we effectively prune the search space before actually starting the search phase, using the reduction algorithm based on lower bound.

6 RESULTS

By integrating OR techniques in CLP we have achieved some major benefits over pure CLP: more data independence of the search process, ability to solve bigger and harder problems, greater ability to solve in a complete way loosely constrained problems with (relative) high efficiency.

In Table 1, we report some results of the code implemented in CHIP [10] for solving instances of the scheduling problem described in section 4. The results have been obtained on a Workstation SUN Sparc 10.

In Table 1 we consider three cases:

- the pure CLP code using the CHIP `min_max` optimization predicate, (column CLP);
- the CLP code exploiting the single machine lower bound described in section 4.3 used in order to prune the cost variable domain only, i.e., the propagation described in point 1 in section 3, (column CLP+LB);
- the CLP code exploiting all pruning described in points 1 and 2 of section 3 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 Opt). For some problems, the optimal solution has not been found after 60 minutes.

N. Tasks	CLP		CLP+LB		CLP+LB+BBF	
	First	Opt	First	Opt	First	Opt
75	3	15	4	8	13	20
150	7	-	9	75	18	80
300	10	-	15	-	30	70
1500	30	-	50	-	120	450
2250	45	-	120	-	180	900

Table 1: Computational Results

As we can note, while the pure CLP approach outperforms both approaches which use information provided by the lower bounds in finding the first solution, it fails in obtaining the optimal solution in reasonable time. This can be explained as follows: CLP-OR approaches exploiting the lower bound spend more time (due to the overhead introduced by the lower bound propagation) in order to find the first solution, which is usually a good one, while the pure CLP code finds quickly the first (generally worse) solution. The proof of optimality is obviously more efficient if we find a good first solution and if information on the lower bound is exploited.

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

For some simple data configurations, however, 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 seconds. These results however are not robust, since by changing one data configuration parameter, performances decreases significantly. With the enhanced lower bound based approach, we obtain an increased robustness of the algorithm which does not depend on parameter changing.

7 RELATED WORK

The study of hybrid algorithms for solving combinatorial optimisation 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 optimisation problems arises from the fact that different techniques are suited to solving different aspects of the problem.

A recent ESPRIT Project, CHIC-2 [7], started in 1996, is aimed at developing hybrid algorithms for large scale optimization problems (LSCO), to provide a methodology for efficient exploitation of these algorithms, and to build a platform to support the creation of applications to solve LSCO problems. It will incorporate several solution methods and will be designed to be extensible. This is because, as new solution algorithms are discovered they will have to be incorporated.

The integration of existing techniques in hybrid algorithms starts from a comparison of different methodologies on the same problem. A number of works compare the performances of CP-based solvers toward OR-based solvers (in particular, integer linear programming techniques) when applied to particular problems, see [14, 15, 19, 22, 24].

As concerns specific hybrid solutions to particular problems, we concentrate on those integrating Operations Research lower bounding techniques in Constraint Programming. The resulting algorithms can have different level of coupling among different

techniques. In particular, in [12] a loose integration of CP and OR techniques is performed for solving the fleet assignment problem. In this case, the integration takes the form of one technique that feeds its result to a second one. In fact, a Constraint Programming (CP) solver provides a good solution used as input to the Mathematical Programming (MP) system to "warm-start" the Simplex algorithm. In our approach, the CP solver and the OR branch and bound technique are more tightly coupled and intertwine their execution for finding an optimal solution.

A tighter integration is performed in [23] where the authors define an integration of Mixed Integer Programming (MIP) and Constraint Logic Programming (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 splitted in two subproblems containing respectively the constraints $v_i \geq \lceil s_{v_i} \rceil$ and $v_i < \lfloor s_{v_i} \rfloor$. This approach has some similarities with ours since we both use a lower bounding procedure for achieving additional pruning. However, they use the relaxed problem solution in order to define an effective branching rule, but they do not remove from variable domains any feasible value for the original problem. The constraint propagation here proposed possibly removes feasible values which cannot lead to any improvement of the best solution found so far.

In [3] an 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 [5] 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 [4], a heuristic method for the *Crew Rostering Problem* (CRP) is described. In order to find a good solution, the algorithm exploits the information given by the lower bounding procedure, which provides an optimistic evaluation of the number of weeks W needed to find a solution. If a feasible W weeks long solution is found, it is an optimal one. An additional constraint on the number of weeks to be used is imposed, thus transforming an optimization problem into a more difficult constraint satisfaction problem whose solution, if it exists, is usually a very good one. While generating the solution, every time a roster is completed, the evaluation of the number of weeks needed is updated by recalculating the lower bound.

In [11] and [6] an even closer integration of Constraint Programming and Lower bounding techniques is performed and applied to the Travelling Salesman Problem. The lower bound calculation is used in order to prune the objective function domain variable, and directly the decision variables domains as we perform in this paper. The difference with the present work is that in [11] the propagation is inserted in an *optimization oriented constraint*, while in this paper it is implemented on top of the constraint solver. In addition, while in [11] a specific OR algorithm has been implemented and encapsulated in the constraint, in this paper, we have used a CLP `min_max` optimization predicate in order to compute the lower bound. In [6], 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.

8 CONCLUSION

In this paper, we have proposed an integration of OR lower bounding techniques in Constraint Logic Programming. 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 Constraint Logic Programming language CHIP which provides very effective global constraints performing powerful propagations.

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

We are currently trying to further improve propagation from the objective function toward decision variables by applying it, at each choice point during the labeling phase, on the whole set of uninstantiated variables. This, however, is a computationally expensive task, and we need to further investigate on balancing the amount of propagation with 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 variables that could lead to a more effective reduction of the search space reasoning on the problem domain characteristics. An alternative approach could be reducing domains not at every choice point.

Further works are aimed to investigating the application of other objective functions and lower bounds to Job Shop Scheduling problems. In fact, we believe that the makespan criterion, under certain real world conditions, does not represent as 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 trying to keep always a certain spare workpower for optimally dealing with emergencies.

9 Acknowledgments

We would like to thank Paola Mello and Evelina Lamma for useful discussion and suggestions. We thank also Officine Rizzoli for providing data for real problem instances. Authors' work has been partially supported by CNR, Committee 12 on Information Technology (Project SCI*SIA).

References

[1] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job-shop scheduling. *Management Science*, 3:391–401, 1998.

[2] N. Beldiceanu and E. Contejean. Introducing global constraints in chip. *Mathematical Computer Modelling*, 20(12):97–123, 1994.

[3] H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In C. Beierle and L.Plumer, editors, *Logic Programming: formal Methods and Practical Applications*. North Holland, 1995.

[4] 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.

[5] Y. Caseau and F. Laburte. Improving Branch and Bound for Jobshop Scheduling with constraint propagation. In M. Deza, R. Euler, and Y. Manoussakis, editors, *Combinatorics and Computer Science*, LNCS 1120, pages 129–149. Springer Verlag, 1995.

[6] Y. Caseau and F. Laburte. Solving small TSPs with constraints. In *Proceedings of the Fourteenth International Conference on Logic Programming - ICLP'97*, pages 316–330, 1997.

[7] CHIC-2. Creating hybrid solutions for industry and commerce. Technical report, IC-PARC, 1996. Deliverable 3.1.

[8] 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.

[9] M. Dincbas, P.Van Hentenryck, and M.Simonis. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[10] 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.

[11] F. Focacci, A. Lodi, M. Milano, and D. Vigo. Solving TSP through the integration of OR and CP techniques. In *Proceedings of the CP98 Workshop on Large Scale Combinatorial Optimization and Constraints*, 1998. `http://www.icparc.ic.ac.uk/ mgw/chic2_workshop.html`.

[12] 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.

[13] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[14] P. Van Hentenryck and J.P. Carillon. Generality versus specificity: an experience with AI and OR techniques. In *Proceedings of AAAI'88*, volume 2, 1988.

[15] K.L. Hoffman and M. Padberg. Solving airline crew-scheduling problems by Branch and Cut. Technical report, George Mason University and New York University, 1992.

[16] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the Conference on Principle of Programming Languages*, 1987.

[17] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.

[18] R. Kowalski. *Logic for Problem Solving*. North-Holland, 1979.

[19] J. Little and K. Darby-Dowman. The significance of Constraint Logic Programming to Operational Research. Technical report, Brunel University - Dept. of Mathematics and Statistics, 1995.

[20] J.W. Lloyd. *Foundation of Logic Programming - Second Extended Edition*. Springer-Verlag, 1987.

[21] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons - New York, 1988.

[22] J.F. Puget and B. De Backer. Comparing constraint programming and MILP. In *Proceedings of APMOD'95*, 1995.

[23] R. Rodosek, M. Wallace, and M.T.Hajian. A new approach to integrating mixed integer programming and constraint logic programming. *Annals of Operational Research*, 1997.

[24] 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.

[25] H.P. Williams. *Model Solving in Mathematical Programming*. John Wiley and Sons, 1993.