Optimization Methods for Constraint Resource Problems

Daniel Sabin and **Eugene C. Freuder**
Department of Computer Science
University of New Hampshire
Kingsbury Hall
Durham, NH 03824
ds1, ecf@cs.unh.edu

Abstract

In this paper we present resource optimization methods for solving efficiently synthesis problems in a constraint-based framework. We obtain a tight lower bound of the problem optimum by adding redundant constraints that take into account the "wastage" in a partial solution. Abstraction through focusing on relevant features permits added interchangeability to deal with equivalent sets of partial solutions. Combining these two ideas allows us to discover fast the optimal solution, and also to prove very quickly its optimality.

Introduction

Many synthesis tasks can be reduced, on an abstract level, to the generic task of "assembling" some "artifact" from a set of "building blocks" (*e.g.* components in configuration and design, actions in planning, repair actions in therapy, qualitative models in model synthesis, *etc.*).

Central to synthesis is the notion of resource. An important part of the knowledge associated with a particular application domain is represented by *producer-consumer* relations between various parts of the artifact. They introduce cumulative restrictions on resource properties of a set of objects. All the resources in the model must be balanced, *i.e.* the amount of resource produced should be equal or greater than the amount used. In the majority of synthesis tasks, the optimization criterion implies the minimization or maximization of some resource and this is what eventually dictates the structure of the artifact.

In this paper we present resource optimization methods for solving efficiently synthesis problems in a constraint-based framework. Our original contribution is twofold:

- We show how to obtain a tighter lower bound of the problem optimum by adding redundant constraints that take into account the "wastage" in a partial solution.
- We show how abstraction through focusing on relevant features permits added interchangeability to deal with equivalent sets of partial solutions.

In Section 2 of the paper we describe a class of problems which is representative for most synthesis tasks. Section 3 describes briefly our algorithms. Each of the following two sections, on the lower bound computation and on the use of abstraction and interchangeability, have a subsection presenting a running example, demonstrating that these techniques can significantly reduce the search effort for finding the optimal solution and proving its optimality. Section 6 presents additional experimental evidence to support our claims.

Problem Definition

The problem we are interested in is very general. We are given a set of consumers, each characterized by the amount of resources it consumes. Available are several types of producers, each described by the amount of resources it can provide. A cumulative expression on some of the resources is designated as the *cost* of a solution. The task is to find the optimal set of producers such that:

- all the resources are balanced, and
- the cost of the solution is minimal.

Instances of this problem appear as subproblems in any synthesis task. Because the motivation of our work lies mainly in solving configuration tasks, the concrete examples used throughout the paper come from the configuration domain. Although we use simplified versions of real problems, the main aspects are preserved.

Example 1

Consider this problem, adapted from (ILOG 1998). A control system consists of a set of racks with electrical connectors in which one can plug different types of electronic cards. A rack has 3 connectors, and each connector can receive exactly one card. In addition to the number of connectors it provides, each rack is characterized by the maximal power it can supply. Cards are characterized only by the power they use. Available are two types of racks, capable of providing 90 and 110 units of power, and four types of cards, consuming 20, 45, 50, and, respectively, 65 units of power. The number and type of cards which can be connected to a rack is limited by two factors: the number of electrical connectors the rack has, and the maximal power the rack can provide.

The problem asks for the number and type of racks which can accept a particular set of cards, such that the overall power supplied by the racks is minimal.

Assume we are required to configure a control system that must accommodate four cards, $\{C_{20}, C_{45}, C_{50}, C_{65}\}$, one of each type available. We start by creating one instance of RACK, R_1, in which we plug cards C_{20} and C_{45}. None of the other two cards can use R_1 anymore, this would require more than the maximum 110 units of power a rack can provide. We add a new rack to the system, R_2 and plug in it C_{50}. The power limitation again prevents us from using the same rack for C_{65}, so we end up by using three racks. Since we are interested in minimizing the total maximal power, we choose for each of the three racks the lowest-power variant able to satisfy the request, thus obtaining a solution with cost 270. This gives us an upper bound for the optimal solution.

Continuing the search in a backtracking manner, we eliminate C_{45} and plug C_{20} and C_{50} in R_1, which allows us to use the same rack for both C_{45} and C_{65}, making complete use of the power provided by R_2. In fact this new solution, of cost 200, is optimal. To prove its optimality we have to show that a solution of cost lower than 200 is not possible. Actually, we can restrict the new solution even more: the next possible combination of lower cost, two small racks, has a cost of 180.

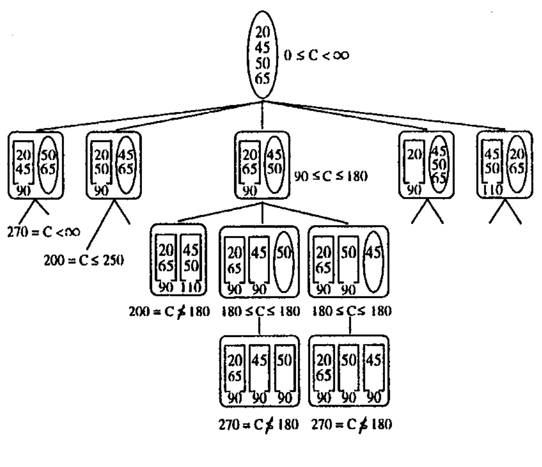

Figure 1: Snapshot of the search tree for an optimal solution.

We use *Branch and Bound* to reduce the amount of unnecessary work performed. The algorithm abandons a search path when the cost of the partial solution, *i.e.* the lower bound, exceeds the upper bound. On our example though, it turns out that the lower bound is not tight enough to really do any pruning. As we can see in Figure 1, it exceeds the upper bound too late, only after two racks have already been added to the system. This is because the lower bound computation is based solely on racks. To account for the amount of power left unused in each rack, the lower bound should consider both the maximal power of the existing racks, and the amount of power required by the cards which have not been plugged in yet. This would allow the algorithm to discover immediately after plugging C_{20} and C_{65} in R_1 that this partial solution actually incurs a minimum cost of 185 units, and therefore cannot lead to a solution of cost 180. In Section 4 we show how to achieve this by using specialized redundant constraints.

Example 2

Let us change the problem slightly. The two types of racks available provide 150 and 200 power units and the four types of cards require 20, 40, 50, and, respectively, 75 power units. The set of cards which must be plugged into racks is $\{ C^1_{20}, C^2_{20}, C^3_{20}, C^4_{20}, C^5_{20}, C^6_{20}, C^7_{20}, C^8_{20}, C^9_{45}, C^{10}_{45}, C^{11}_{45}, C^{12}_{45}, C^{13}_{50}, C^{14}_{50}, C^{15}_{75} \}$. The lower index represents the amount of power the instance requires. Instances requiring the same amount of power are of the same type. A snapshot of the search tree associated with this example is shown in Figure 2. We start again by creating an instance R_1 of rack, in which we plug cards C^1_{20} through C^9_{45}. The power provided by R_1, 200 power units, is consumed entirely. A new rack instance, R_2, receives cards C^{10}_{45} through C^{13}_{50}, which consume 170 power units of the maximum 200 it provides. Finally, the last two cards, C^{14}_{50}, C^{15}_{75}, are plugged in the third rack, R_3, and use 125 power units out of the maximum 150 the rack provides. The cost of this first solution is 550, and gives us an upper bound for the optimal solution. Since the increment for the cost is 50, the next solution will be better only if it has a cost of at most 500.

The optimal solution actually has a cost of 500 power units. It turns out that finding it and proving its optimality is more difficult then in the previous example. This is due to the fact that most of the search effort is spent on exploring sets of equivalent partial solutions, introduced in the search space by the use of multiple instances of the same type of card. In the figure we point out an example. Since the only restriction imposed on the cards is on their power consumption, although involving different values (card instances), the partial solutions in the three sets (1), (2) and (3) are equivalent. After the algorithm already investigated a solution assigning two 45 power unit cards to R_2 (set (1)), there is no point in trying other combinations of two similar cards (sets (2) and (3)). Therefore we can prune from the search tree the grayed regions without losing any problem solution. We show in Section 4 how to eliminate equivalent partial solutions efficiently using abstraction and interchangeability.

Problem representation

The constraint satisfaction problem (CSP) paradigm provides an elegant and natural framework for representing and solving a large variety of reasoning tasks, including synthesis, and for the past decade has been

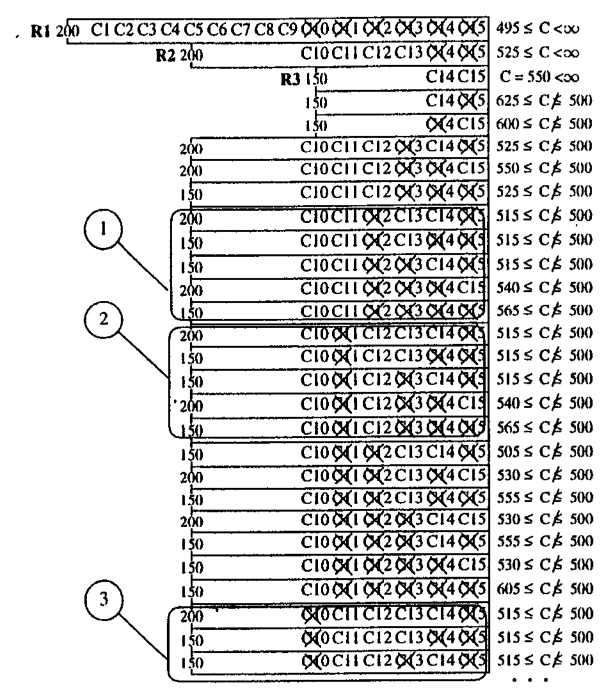

Figure 2: Snapshot of the search tree for an optimal solution.

widely accepted, both in academia and industry, as the formalism of choice for dealing with optimization problems as well.

We model application domain objects as *composite constraint satisfaction problems*, in a manner similar to the one presented in (Sabin & Freuder 1996). Due to space limitation, we just mention that object properties, which characterize its function and performance, are represented as variables, while various restrictions and requirements are expressed as constraints.

A producer-consumer relation implies a bidirectional connection between the objects involved in the relation. We capture this by adding a *port variable* to the model of each object that has resource properties. Ports are characterized by *base type* and *cardinality*. The domain of a port variable *P<T>[m..n]* is a set of objects of type $\mathcal{T}$, and the values the port can take are subsets of the domain, of cardinality at least m and at most n. We use the notation $|P|$ to refer to P's cardinality.

There are several types of constraints that can be posted on port variables, two of which are relevant in the context of our presentation:

- cardinality constraints, imposing a lower and/or upper bound on the number of objects that can be assigned to the port, *e.g.* $|P| \leq 2$, $|P| > 0$, *etc.*, and
- cumulative constraints on attributes of the objects assigned to the port, *e.g.* $\sum P.x \leq 100$, where x is a numeric attribute of instances of type $\mathcal{T}$.

The model for the two examples described in the previous section is the following:

- The model for *system* consists of two variables:
  - integer variable $power_{SYSTEM}$ with domain $\{0..\infty\}$, and
  - port variable $racks_{SYSTEM}$*< RACK>[1..∞]*.
- RACK is described by three variables:
  - integer variable $power_{RACK}$ with domain $\{90, 110\}$ and $\{150, 200\}$, respectively;
  - port variable $system_{RACK}$*< SYSTEM>[1..1]*;
  - port variable $cards\_{RACK}$*< CARD>[1..3]*.
- CARD instances are described by two variables:
  - integer variable $power_{CARD}$ with domain $\{20, 45, 50, 65\}$ and $\{20, 40, 50, 75\}$, respectively;
  - port variable $racks_{CARD}$*< RACK>[1..1]*.
- In addition, the model for objects of type SYSTEM and RACK contains constraints expressing producer-consumer relations:
  - $power_{SYSTEM} = \sum racks_{SYSTEM}.power$
  - $power_{RACK} \geq \sum cards_{RACK}.power$
- The cost of a solution is represented by the value of the variable $power_{SYSTEM}$.

Algorithms

For the purpose of this paper we consider only complete search methods because we are interested in proving the optimality of the solution. A lot of research effort has been invested lately in the study of branch and bound variants of CSP search algorithms (Freuder & Wallace 1992) (Cabon, Givry, & Verfaillie 1998). Branch and Bound keeps track of an upper and lower bound for the cost of the solution. The upper bound is the cost of the best solution, and can be updated when a new solution is discovered. The lower bound represents an estimate of the cost implied by the current (partial) solution, and gets monotonically updated as the algorithm advances on the solution path. These bounds are used for pruning entire branches from the search tree. At each step of the algorithm, the two bounds are compared against each other, and once the lower bound becomes at least as large as the upper bound[1], it is clear that the current search path cannot lead to a better solution, and is abandoned. Obviously, the better (tighter) the bounds are, the more pruning the algorithm achieves. Although it is fairly easy to come up with a good upper bound, in the majority of cases this is not true for the lower bound (Givry, Verfaillie, & Schiex 1997).

Port variables instantiation

One way to implement a port variable *V<T>[m..M]* is to maintain internally two sets of T instances, one representing the current value, the other one the domain. When the port is created, its domain consists of the

[1]We consider here that the objective is to minimize the cost variable, but the same principle applies when we try to maximize it

set of all $\mathcal{T}$ instances which exist at that time in the model plus a *wildcard* instance $*_{\mathcal{T}}$, accounting for any future instance of $\mathcal{T}$ that might be created. This representation is similar to the one presented in (Mailharro 1998), although the implementation details seem to be different.

The instantiation process we propose is fairly straightforward. Using the set of constraints posted on the port as a filter, inconsistent instances are eliminated from the domain. All instances which passed the filter, except for the wildcard, are moved to the current value set. When the filtering phase ends, there are two possibilities.

1. The cardinality of the current value is at least m. In this case the port has been successfully instantiated.
2. The cardinality does not satisfy the lower bound requirement. Again we are left with two possibilities.
   (a) The domain is empty, *i.e.* the wildcard has been rejected by the filter. In this case the port is considered to be *closed*. What this means is that no instance of type T can satisfy (anymore) the requirements imposed by the port, and therefore the instantiation fails.
   (b) The wildcard is still in the domain. The procedure will first create a new instance of T and add it to the domain of all ports with base type $\mathcal{T}$ which have not been closed yet[2]. Then, the instantiation process continues, with the new instance in the domain.

However, there is another aspect of the algorithm that we would like to point out. A connection established through ports is bidirectional. We capture this aspect in our model by using pairs of complementary ports. Assume that objects of type $\mathcal{U}$ have a port of type $\mathcal{T}$, say $P<\mathcal{T}>$. Objects of type $\mathcal{T}$ must then have a port of type $\mathcal{U}$, call it $Q<\mathcal{U}>$. Consider two instances, x and y, of type $\mathcal{U}$ and $\mathcal{T}$, respectively. Connecting y to the port P of x means adding y to the current value of P. This happens during the process of instantiating P. Due to bidirectionality, x then must be added to the current value of port Q of y as well. The implication of this step is twofold. First, if adding x to Q would lead to a constraint violation, then it is not possible to add y to P either. Second, instances can be added to a port's current value even after the port has been instantiated, as long as the port has not been closed yet.

Achieving Optimality through Constraint Propagation

The search algorithm we use is not a Branch and Bound algorithm, but achieves the same effect through constraint propagation on redundant specialized constraints.

[2]We will show in Section 5 that this step can fail as well due to global limitations on the total number of instances of a given type.

Our algorithm is based on a powerful CSP algorithm, *MAC* (Sabin & Freuder 1994)(Sabin & Freuder 1997). MAC uses constraint propagation for maintaining *arc-consistency* during search. Every time the domain of a variable is modified, the constraints in which the variable is involved are responsible for propagating the change to related variables. For more details on how this can be done efficiently see the original papers.

MAC is a general-purpose CSP search algorithm. In particular, it has no provision for finding optimal solutions. However, we do not need to change the algorithm for making it search for the optimal solution, we update the problem instead. Each time a solution of cost C is found, the constraint $cost < C$ is added to the problem to reflect the new upper bound, and then simulate a failure, thus forcing the algorithm to look for a better solution. A similar technique can be found in (ILOG 1998). It is obvious that the value C is the upper bound of the solution and that by adding the new constraint the updated upper bound becomes actively involved in the search.

The lower bound is integrated in the model through the use of resource constraints. In our problem the value of the cost variable $power_{SYSTEM}$ is controlled by the equality constraint with $\sum racks_{system}.power$. Internally, the lower bound of the $\sum$ is updated incrementally, as new elements are added to $racks_{SYSTEM}$. Through the equality constraint, the change propagates and updates the lower bound of the $power_{SYSTEM}$ variable.

Before moving further, we want to mention briefly that when deciding which variable to instantiate next, port variables are always preferred, and among several port variables, we choose first the ones belonging to a producer.

Improved lower bound computation

Let us get back to Example 1. We can observe from the beginning that the amount of power the racks have to provide must be at least 180 power units, the amount of power required by the four cards. The current model does not provide any way of directly relating this information to the cost variable. We will add a redundant constraint which, through propagation, will provide the connection.

To be able to keep track of the power requirement for all the cards in the system, we need a global point of view. We associate with each type $\mathcal{U}$ a special type of port variable, called *metaport*. A metaport variable associated with type $\mathcal{T}$, $M<\mathcal{T}>$, contains all the instances of $\mathcal{T}$ that have been created and are currently part of the model.

Cardinality constraints on metaports allow us to put a limitation on the total number of instances of a given type that can be created. In addition to the usual constraints that can be posted on regular port variables (resource, cardinality, *etc.*), metaports offer a special type of resource constraint, called *balancing* constraint.

The constraint is described by a 4-tuple <P, C, x, y>, where

- P is a metaport variable associated with type $\mathcal{T}$,
- C is a metaport variable associated with type $\mathcal{U}$,
- x and y are attributes representing the amount of resource r produced by an instance of $\mathcal{T}$ and used, respectively, by an instance of $\mathcal{U}$.

A balancing constraint implies the existence of a producer-consumer relation between instances of the two types, $\mathcal{T}$ and $\mathcal{U}$, on resource r, *i.e.* any instance t of $\mathcal{T}$ has a port U<$\mathcal{U}$> and $t.x \geq \sum t.U.y$. Its semantics is the following:

- The initial lower bound for $\sum$P.x is the lower bound of $\sum$C.y
- The lower bound of $\sum P.x$ is updated incrementally as the result of:
  - Creating a new instance u of type U: the value of $\sum P.x$ is increased by $u.y$
  - Closing an instantiated port $t.U$ on attribute y: the lower bound of $\sum P.x$ is increased by the difference $t.x$ - $\sum t.U.y$.

We extend now the model for SYSTEM. We add two metaports, P< *RACK*> and C< *CARD*>, as well as two constraints: *Balance(P, C, $power_{RACK}$, $power_{CARD}$)* and, since all instances of RACK must be part of the system, $power_{SYSTEM} = \sum P.power_{RACK}$. The results of this change are presented in Figure 3.

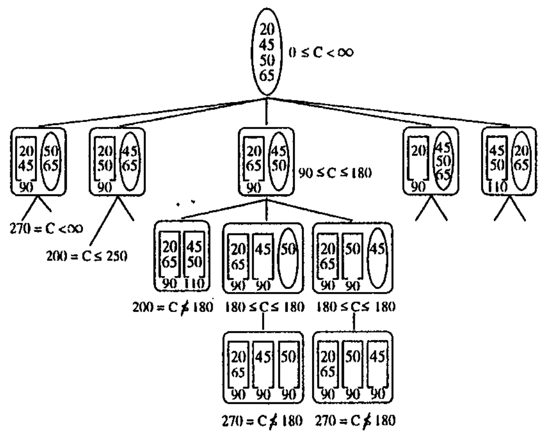

Figure 3: Snapshot of the search tree for an optimal solution.

Equivalent Partial Solutions

Let us get back to Example 2. Although we did not mention this before, the instantiation algorithm considers an implicit ordering among the elements in the domain, thus avoiding symmetries introduced by permutation of values. For example, once the algorithm discovers the solution of cost 550 which assigns the value { C_{45}^{10}, C_{45}^{11}, C_{45}^{12}, C_{50}^{13} } to rack R_2, it will never consider trying permutations of this set as values for R_2.

This cuts down some of the search effort, but we are still left with partial solutions that are equivalent in the sense that they all participate exactly with the same amount to the final solution cost.

Eliminating equivalent partial solutions through interchangeability

The simplest type of equivalence is introduced by multiple instances of the same type. Take a look at Figure 2. Exchanging C_{45}^{12} for C_{45}^{11} in the partial solution that includes two instances of 45 power unit cards in the value of R_2, C_{45}^{10} and C_{45}^{11}, will lead to a solution of equal cost. This is because in our model any two card instances of the same type are identical in all respects.

By analogy with (Freuder 1991), we say that two instances are *interchangeable* if replacing one by the other in any solution produces another solution of equal cost. According to this definition, two card instances of the same type are interchangeable.

The process we propose for eliminating equivalent partial solutions is the following. Once an instance is rejected from a domain during port variable instantiation, we look for all the other instances of components of the same type and reject them as well. The effect of doing this on problem in Example 2 is shown in Figure 4.

Although true for cards, it is not always the case that instances of the same type are interchangeable. Here is a simple example. We have two racks of the same type, R_{150}^{1} and R_{150}^{2}. Due to the different sets of cards already connected to the two racks, R_1 has 30 units of power still available, while R_2 has 50 left. Suppose the two racks are in the domain of card C_{45}^{11} which must be instantiated next. R_1 is rejected because of the power requirement, but rejecting R_2 based on the fact that the two instances have the same type would be wrong, since R_2 satisfies the power requirement.

The question is then how to decide when two instances are interchangeable. Remember that they are modeled as composite CSPs. Since all instances from the domain of a port have the same type, the corresponding composite CSPs have the same sets of variables and internal constraints. Then a sufficient condition, but not necessary, for two instances to be interchangeable is that pairs of corresponding variables have the same domain in both problems. In case the domains are the same, the two instances are clearly interchangeable.

Abstraction and context-dependent interchangeability

But this method might prove to be too restrictive. Assume that type CARD can be refined to several specialized types, each with additional features and providing non-identical functionality. Some cards requiring equal amounts of power are not instances of the same type anymore. Their models may differ, both in structure (*i.e.* number and type of variables and constraints) and in the domain of the variables. According to the above definition, these instances are not interchangeable anymore. However, because the only relevant aspect for deciding whether a card can be connected to a rack or not is the amount of power it requires, solutions involving the same number of cards with equal power requirements are still equivalent.

We abstract the model for CARD and RACK through focusing on relevant common features only. Considering only the abstracted model permits added interchangeability. The decision on what features are relevant is made based on the set of constraints imposed on the port variable.

As shown before, constraints on ports involve attributes of the instances in the domain of the port, which in our model are represented by variables. It is this restricted set of variables which will be checked for domain identity in deciding whether two instances are interchangeable or not. In our example, the set of variables contains only the variable $power_{CARD}$.

According to the new definition of interchangeability, cards with equal power requirements are interchangeable. Applying the algorithm presented earlier on the problem instance in Example 2 produces the results presented in Figure 4.

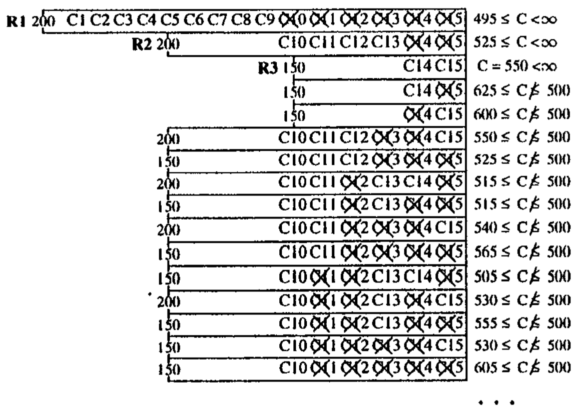
R1 200 C1 C2 C3 C4 C5 C6 C7 C8 C9 ~~C10~~ ~~C11~~ ~~C12~~ ~~C13~~ ~~C14~~ ~~C15~~ 495 ≤ C < ∞
R2 200 C10 C11 C12 C13 ~~C14~~ ~~C15~~ 525 ≤ C < ∞
R3 150 C14 C15 C = 550 < ∞
150 C14 ~~C15~~ 625 ≤ C ≰ 500
150 ~~C14~~ C15 600 ≤ C ≰ 500
200 C10 C11 C12 ~~C13~~ ~~C14~~ C15 550 ≤ C ≰ 500
150 C10 C11 C12 ~~C13~~ ~~C14~~ ~~C15~~ 525 ≤ C ≰ 500
200 C10 C11 ~~C12~~ C13 C14 ~~C15~~ 515 ≤ C ≰ 500
150 C10 C11 ~~C12~~ C13 ~~C14~~ ~~C15~~ 515 ≤ C ≰ 500
200 C10 C11 ~~C12~~ ~~C13~~ ~~C14~~ C15 540 ≤ C ≰ 500
150 C10 C11 ~~C12~~ ~~C13~~ ~~C14~~ ~~C15~~ 565 ≤ C ≰ 500
150 C10 ~~C11~~ ~~C12~~ C13 C14 ~~C15~~ 505 ≤ C ≰ 500
200 C10 ~~C11~~ ~~C12~~ C13 ~~C14~~ C15 530 ≤ C ≰ 500
150 C10 ~~C11~~ ~~C12~~ C13 ~~C14~~ ~~C15~~ 555 ≤ C ≰ 500
150 C10 ~~C11~~ ~~C12~~ ~~C13~~ ~~C14~~ C15 530 ≤ C ≰ 500
150 C10 ~~C11~~ ~~C12~~ ~~C13~~ ~~C14~~ ~~C15~~ 605 ≤ C ≰ 500
. . .

Figure 4: Snapshot of the search tree for an optimal solution.

Experimental Evaluation

In order to test the performance of our approach, we used a set of randomly generated test problem instances similar to the one presented in Example 2. Each instance is characterized by the cardinality of the set of cards. We generated problems having between 10 and 200 cards, with an increment of 10. For each number of cards we generated 50 problem instances. The types of the cards were assigned randomly among the four types.

We conducted two sets of experiments in which we addressed the problem of finding the optimal solution and proving its optimality. First, we compared our algorithm with a program implemented specifically for solving this problem, presented in (ILOG 1998). The results, in terms of CPU time, are presented in Figure 5. The advantage of our method is obvious. For example for problems with 30 cards, we limited the running time for the Solver code to two hours, while our algorithm completed on average in 0.5 seconds.

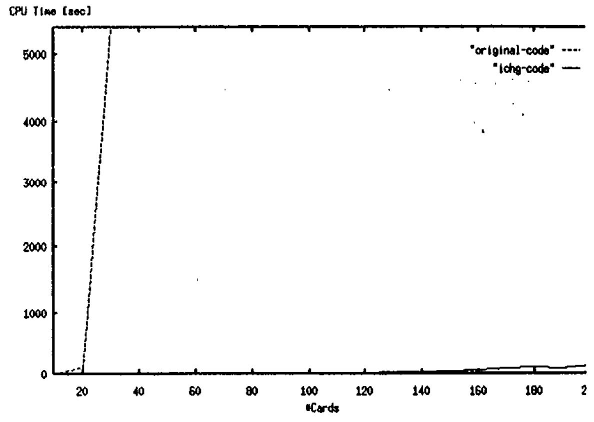

Figure 5: Comparison with the original Solver code.

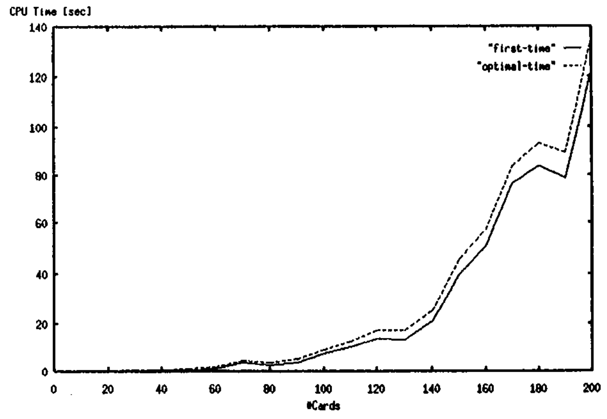

Figure 6: Search effort in terms of CPU time.

For the second set of experiments we used only our algorithm and compared the search effort spent for finding the first solution with the search effort required for finding the optimal solution and proving its optimality. The results are presented in Figure 6. The figure consist of two plots, one for the first solution (the plot name is prefixed by *first*), the other for finding and proving the optimality (the plot name is prefixed by *optimal*). Each point of the plot was computed as the average over the 50 problem instances generated for each value of the number of cards.

As we can observe, the two plots are very close to each other, which proves the advantages of our method: not only we discover fast the optimal solution, but we are also able to prove very quickly its optimality.

Acknowledgment

This material is based on work supported by ILOG S.A. and the National Science Foundation under Grant No. IRI-9504316.

References

Cabon, B.; Givry, S. D.; and Verfaillie, G. 1998. Anytime Lower Bounds for Constraint Violation Minimization Problems. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, number 1520 in Lecture Notes in Computer Science. Springer.

Freuder, E. C., and Wallace, R. 1992. Partial Constraint Satisfaction. *Artificial Intelligence* (58).

Freuder, E. C. 1991. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proceedings of the Ninth National Conference on Artificial Intelligence.* AAAI Press.

Givry, S. D.; Verfaillie, G.; and Schiex, T. 1997. Bounding the Optimum of Constraint Optimization Problems. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, number 1330 in Lecture Notes in Computer Science. Springer.

ILOG, S. A. 1998. *ILOG Solver User's Manual.*

Mailharro, D. 1998. A Classification and Constraint based framework for configuration. *AIEDAM.* Special issue on Configuration.

Sabin, D., and Freuder, E. C. 1994. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence.* Wiley.

Sabin, D., and Freuder, E. C. 1996. Configuration as Composite Constraint Satisfaction. In *Artificial Intelligence and Manufacturing Research Planning Workshop.* AAAI Press.

Sabin, D., and Freuder, E. C. 1997. Understanding and Improving the MAC Algorithm. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, number 1330 in Lecture Notes in Computer Science. Springer.