A Framework for expressing knowledge about constraint satisfaction problems, Roy, P. Pachet, F. Perrot, J.-F. FLAIRS 97, Daytona (Fl)

A FRAMEWORK FOR EXPRESSING KNOWLEDGE ABOUT CONSTRAINT SATISFACTION PROBLEMS

Pierre Roy, François Pachet, Jean-François Perrot

Laforia-IBP, Université Paris 6, Boîte 169, 4, place Jussieu,
75252 Paris Cedex, France.
E-mail: {roy|pachet|jfp}@laforia.ibp.fr

Abstract: This paper describes a framework for expressing and solving combinatorial problems. The framework is especially designed to allow the expression of various types of knowledge on problem domains, that can be exploited by the resolution mechanism to speed up the search. We illustrate the framework and the types of knowledge on a crossword example.

1 Introduction

Constraint satisfaction programming is a powerful paradigm for solving complex combinatorial problems, which has gained attention recently. The notion of constraint was initially seen as an algorithmic problem, e.g. by [Mackworth 77] and [Laurière 78] who see constraint graphs as networks of relations for finite domains. Complex combinatorial problems have been studied extensively in operation research, graph theory and artificial intelligence for over two decades, leading to the elaboration of a rich theoretical framework. The main notion that came up from these works is arc-consistency [Mackworth 77]. Most existing algorithms are based on the exploitation of arc-consistency: forward-checking [Haralick & Elliot 80], full lookahead, and various extensions (e.g. backjumping, [Prosser 93]). These mechanisms have been later incorporated into logic programming languages ([Colmerauer 1990], CHIP [Van Hentenryck 89], CLP (R) [Jaffar & Lassez 87]). More recently, these mechanisms have been integrated with object-oriented languages [Puget 94], [Caseau 94] or [Avesani et al. 90].

However, most difficult problems are still out of reach, even using state of the art CSP algorithms or languages. The main reason is well known in AI: general-purpose algorithms are, by definition, limited, because they do not have the knowledge specific to the problem instance. The idea of exploiting knowledge about problem instances has been explored already by J.-L. Laurière in the Alice system [Laurière 78]. Although Alice was able to adapt its reasoning to particular problem instances, it used only general-purpose heuristics and knowledge, and the user has no possibility of expressing domain specific knowledge to help the engine.

Following Laurière, we are convinced that knowledge is needed to improve the efficiency of enumeration algorithms. However, departing from his approach, we do not believe in general-purpose heuristics that are applicable to all domains. We claim that domain-specific knowledge is the key to improving the efficiency of CSPs, but that this knowledge must be carefully carved-up to fit the constraints imposed by the CSP technology. This paper describes a framework for expressing and solving combinatorial problems, in which domain specific knowledge can be expressed to increase the efficiency of the resolution.

2 The BackTalk Framework

In order to study the relevance of domain specific knowledge, we designed a framework called BackTalk (standing for Backtracking in Smalltalk) [Roy & Pachet 97] that achieves simultaneously two goals: 1) provide state of the art enumeration algorithms that are usable off-the-shelf to solve combinatorial problems on arbitrary domains, and 2) provide safe entry points to express specific knowledge, that will be exploited by the system to speed up the resolution.

There is a wealth of resolution algorithms for solving constraint satisfaction problems. The most widely used are all based on two main procedures: a *general backtracking* loop, which guaranties a complete exploration of the search space, and a *constraint propagation* procedure, which reduces the problem during backtracking. These algorithms instantiate progres-

sively the variables of the problem, and after each instantiation the constraints are considered to reduce the domains of the remaining variables. In case of failure, the system backtracks to a previous variable. These algorithms mainly differ in the amount of constraint propagation performed at each step.

This variation in the amount of constraint propagation depends on two factors: a global strategy concerning the whole problem, and a local strategy, applied to each constraint. The global strategy is determined by the resolution algorithm itself. The local strategy is determined by the type of constraint considered. The BackTalk framework is based on a reification of these two strategies. We will review each of them in the two following sections.

2.1 Local Strategies: Constraint Filtering

One of the key results of CSP is to show that, if one considers constraints individually, the maximum amount of reduction is given by the property of *arc-consistency* [Macworth 77]. Arc-consistency is a property of an individual constraint, that ensures that there is no value in the domains that violates the constraint, regardless of the other constraints of the problem. Achieving arc-consistency for one constraint allows to reduce the domains of its variables, without loosing any solution. The default method for achieving arc-consistency (usually called `revise`) requires the computation of the Cartesian product of the domains of the variables.

In practice, this general `revise` procedure can be drastically improved by taking the nature of the constraint into account, either to specialize the general `revise` procedure into an *equivalent* and more efficient procedure (e.g. the all-different constraint, [Régin 94]), or by implementing a weaker form of revision that reaches only an approximation of full arc-consistency, but with a better complexity. This procedure is usually called *filtering*.

Since the filtering procedure depends on the nature of the constraint, we chose to represent constraints as classes organized in a class hierarchy, and represent filtering as a method (`filter`) of these classes (see Figure 1). The root of the hierarchy (class `Constraint`) implements `filter` with the default `revise` procedure of Mackworth. Each subclass redefines the `filter` method with a specialized filtering algorithm adapted to the constraint. For instance, the all-different constraint is represented by class `AllDiffCt`, in which method `filter` basically implements the procedure of [Régin 94].

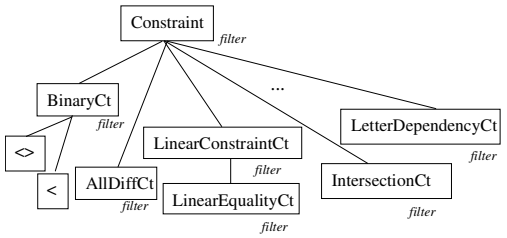


Figure 1. The constraint hierarchy in BackTalk. Method `filter` is redefined in subclasses.

2.2 Global Strategies: Resolution Algorithms

Filtering as described in the preceding section concerns constraints taken individually. This filtering procedure is controlled by resolution algorithms using different global strategies.

The most radical strategy consists in filtering systematically all the constraints of the problem until a fixed point is reached, after each instantiation. This algorithm is called *full lookahead* [Nadel 88]. Various algorithms were designed to reach efficiently this fixed point: AC-3 [Macworth 77], AC-5 [Deville & Van Hentenryck 91], etc. Full lookahead reduces domains as much as possible, but it may be costly in general.

Another widely used algorithm is *forward-checking* [Haralick & Elliot 80]. In forward-checking, only constraints involving the *last instantiated variable* are filtered. This algorithm reduces less than full lookahead, but also spends much less time in the reduction phase. Other algorithms, such as back-jumping and combinations of these methods are described in [Prosser 93] and propose yet different compromises between domain reductions and search.

Each of these algorithms is adapted to specific situations, and none of them is always better than the others. Full lookahead is especially efficient when 1) the filtering of a constraint is cheap and 2) when there are strong dependencies between variables not directly linked by a constraint. Forward-checking is interesting in the case of "weakly constrained" problems, and where constraint filtering is expensive. Unfortunately these criteria (less constrained, strong dependencies) are hard to specify formally, and thus to automate.

In order to let the user chose the most appropriate algorithm for his problem, we represent algorithms also as classes in BackTalk. These classes are also organized into an inheritance hierarchy. An abstract class (`Solver`) implements the default enumeration mechanism. Each enumeration algorithm is represented by a subclass which redefines parts of this

mechanism (cf. Figure 2). BackTalk is designed in such a way that constraint objects and algorithm objects are independent: any algorithm may use any constraint. The only requirement of constraint objects is that they should respond to the filtering messages.

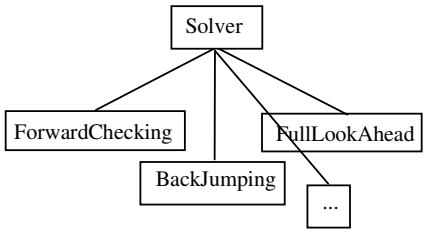


Figure 2. The hierarchy of algorithms in BackTalk.

2.3 Heuristics

The framework allows to specify heuristics for the important steps of the main loop: choice of the next variable, choice of the next value, and choice of the next constraint to filter. These heuristics are represented as methods, called by the solver, so that each problem instance can specify its own set of heuristics.

3 Example: Crosswords

Let us illustrate how domain-specific knowledge can be expressed in BackTalk on a crossword problem. The problem consists in finding a crossword, given an initial grid with black and white squares and a list of words. The problem is a typical complex combinatorial problem: a reasonable list of words contains about 150,000 words, a standard grid contains about 30 words of size 2 to 12, leading to a search space of about 10^{100} combinations. CSP is therefore particularly well adapted to solve it. We consider a formulation of this problem in which variables are the words to find, and constraints are intersections between two words.

However, the CSP formalism is not enough to cope with the complexity of this problem. We will show here how specific knowledge can be expressed in the BackTalk framework to speed up the resolution. This knowledge is three-fold: topologic knowledge, lexical knowledge, and knowledge on letter distribution.

3.1 Heuristics

A good heuristic for the choice of the next variable to instantiate is to chose the variable with the smallest domain (so-called "first fail" principle). In the case of crosswords, we can use the intuitive knowledge on crosswords that it is better to proceed region by region, rather than exploring several areas at the same time. This corresponds to the "intensification principle" used in Tabu search for instance [Glover 89]. The min-size heuristic is a short-sighted strategy that has to be somehow compensated. In this respect, intensification may be seen as a min-size heuristic augmented with a rudimentary anticipation capability.

This knowledge is faithfully represented by a special heuristic requiring that the next variable will be the variable with the smallest domain, within the set of variables connected to the current one.

3.2 Choosing the Right Algorithm

The crossword problem is a typical example of a "weakly constrained" problem. Intuitively, the idea is that instantiating a variable with a given word will have a limited impact on the variables not directly crossing it. This is explained by the fact that the distribution of letters in words is quite uniform, except for special letters (such as "q"). Therefore, the instantiation of one letter does not allow to deduce much information on the other letters of the word. Since the instantiation of a word variable determines only one character for every crossing word, it has little influence on the crossing words, and, in general, very little influence on other, non-crossing words. Therefore we chose naturally the forward-checking resolution algorithm which performs the optimal amount of constraint propagation. The exceptional cases due to non uniform distribution of letters is treated in section 3.4.

3.3 A Filtering Method for the Intersection Constraint

The crossword problem, in its basic form, contains only intersection constraints. Recall that by default, filtering constraints consists in computing approximately the Cartesian product of the domains, and retaining only the consistent tuples.

Knowledge on intersection can be used to improve the filtering of these constraints. Indeed, checking that two sets of words are consistent can be computed much more faster, by noticing that the set of possible intersections is small: there are only 26 letters in the alphabet. The refined procedure is the following:

filter intersection between X and Y:

- (i, j) := intersection of X and Y.
- Compute *possibleLetters* (X,i) = the set of possible letters at position i for X.
- Remove from domain(Y) all words which do not contain one of *possibleLetters* (X,i) at position j.
- Compute *possibleLetters* (Y,j) = the set of possible letters at position j for Y.
- If *possibleLetters*(Y,j) $\not\subset$ *possibleLetters* (X,i) then remove from domain(X) all words which do not contain one of *possibleLetters* (Y,j) at position i.

It is easy to show that this procedure achieves arc-consistency for the intersection constraint. The complexity is linear, to be compared to the quadratic complexity of the default filtering method !

3.4 Exploiting Knowledge on Letter Distribution to Combine Intersection Constraints

A last type of knowledge is that all letters are not distributed uniformly in words. A typical example of regularity is the fact that the letter "q" is almost always followed by letter "u" (at least, in English and French). There are numerous examples of this kind of rule, such as: "no word starts by the same consonant twice", or "j is never repeated twice", "letters are rarely repeated three times", and so forth. These regularities are not always true, but only give strong indications on letters not yet found.

One simple way of representing this type of knowledge is with heuristics on the choice of values of variables. The problem with this solution is that it would be difficult to combine several of these rules. Moreover these heuristics would have to be dynamically created and removed during backtracking.

These rules basically give information on letters which belong to words not directly intersected with the current word. In the case of a variable v instantiated with a word containing a "q", the high probability for the crossing word to have a "u" following the "q" may be used to reduce the domain of the variable v' that is just parallel to variable v (see Figure 3).

It is possible to express this piece of knowledge in terms of constraints and variables, by considering a *virtual constraint* between two parallel words v and v', only when certain conditions are satisfied (here, letter "q" appears). Of course, it would be awkward to actually add dynamically this virtual constraint to the problem, because this virtual constraint is already represented by constraint u between v' and w.

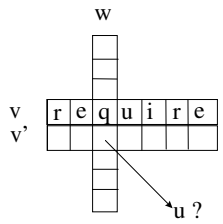


Figure 3. The letter "q" implicitly creates a relation between v and v'. This relation corresponds exactly to the intersection between v' and w.

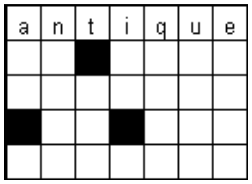
A reasonable way to implement this "virtual constraint" is to represent each of these rules as a global constraint, involving all the variables of the problem, called LetterDependencyCt, which is given statically at the problem formulation phase. The statement of this constraint consists in 1) a condition on values of variables, and 2) a set of intersection constraints to filter when the condition is satisfied. For instance, our 'q' → 'u' rule would be expressed as:

```
AS SOON AS
 there exists a variable v, whose value
 contains a "q"
THEN
 filter the intersection constraint between
 v', parallel to v, and w (perpendicular to
 v at position of letter "q").
```

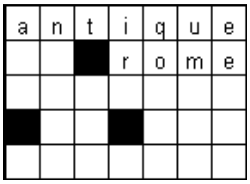
This constraint is endowed with a particular filtering method, which triggers the filtering of the adequate intersection constraints (u in our example), as soon as the condition is satisfied. Therefore there is no modification of the forward-checking algorithm.

3.5 Results

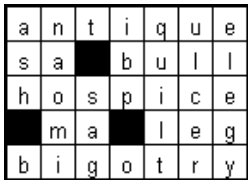
We conducted a series of experiments on crosswords, with and without these three kinds of knowledge. Figure 4 illustrates the effect of exploiting knowledge on letter distribution.



A. Word 'antique' is fixed before the resolution.



B. Without rule 'q' precedes 'u', word 'rome' is chosen, which leads to the development of a useless search tree.



C. When the rule 'q' precedes 'u', the solver only instantiates the variable parallel to 'antique' with words having a 'u' at second position.

Figure 4. The resolution of a crossword.

These experiments show clearly that our approach allows to reduce the domains of word variables, thereby reducing the number of backtracks.

The following table gives execution times and number of failures when combining the various knowledge representations described in this paper. Expectedly, the best strategy is achieved when combining all three types of knowledge with forward-checking.

Specialized filtering	First fail heuristic	Knowing 'q' → 'u'	CPU (in sec)	fails
NO	**NO**	**NO**	> 3,600	> 5,000
YES	**NO**	**NO**	545	10,546
YES	**NO**	**YES**	166	3,396
YES	**YES**	**NO**	23	268
YES	**YES**	**YES**	6	36

Experimental comparisons between full lookahead and our strategy led to the following conclusions: with full lookahead, the system filters a lot of constraints. However, due to the uniform distribution of letters in words, there are only few additional domain reductions, compared to our strategy. Full lookahead and our method explore similar search spaces, but the time spent after each instantiation is dramatically smaller using the latter.

4 Conclusion

We have described a framework for expressing and solving combinatorial problems, that is designed to accommodate various kinds of domain-specific knowledge. We illustrated the framework and three types of knowledge on a crossword solver.

References

Avesani, P. Perini, A. Ricci, F. (1990) COOL: An Object System with Constraints. Proc. of TOOLS'2, Paris (France), Angkor, pp. 221-228.

Caseau, Y. (1994) Constraint Satisfaction with an Object-Oriented Knowledge Representation Language. *Journal of Applied Artificial Intelligence*, Vol. 4, pp. 157-184.

Colmerauer, A. (1990) An introduction to Prolog-III. *CACM*, 33 (7): 69.

Deville, Y. Van Hentenryck, P. (1991) An efficient arc-consistency algorithm for a class of CSP problems. Proc. of IJCAI '91, Chambéry (France), pp. 325-330.

Glover, F. (1989) Tabu search - Part I. *ORSA Journal on Computing*, 1 (3), pp. 190-206.

Haralick, R.M. Elliot, G.L. (1980) Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, vol. 14, pp. 263-313.

Jaffard, J. Lassez, J.-L. (1987). Constraint logic programming. Proc. of 14th POPL'87, Munich (Germany).

Laurière, J.L. (1978). A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10, pp. 29-127.

Mackworth, A. (1977). Consistency on networks of relations. *Artificial Intelligence*, (8) pp. 99-118.

Nadel, B. (1988) Tree search and arc-consistency in constraint satisfaction algorithms. *Search in Artificial Intelligence*, Springer-Verlag, pp. 287-340.

Roy, P. Pachet, F. (1997) Reifying Constraint Satisfaction in Smalltalk. *Journal of Object-Oriented Programming*, to appear.

Prosser, P. (1993) Domain filtering can degrade intelligent backtracking search. Proc. of IJCAI'93, Chambéry (France), pp. 262-267.

Puget, J.-F. (1994) A C++ implantation of CLP. Ilog Solver collected papers. ILOG technical report.

Régin, J-Ch. (1994) A filtering algorithm for constraint of difference in CSPs. Proc. of 12th AAAI'94, pp. 362-267, Seattle (Washington).

Van Hentenryck, P. (1989) Constraint satisfaction in Logic programming. Logic Programming Series, MIT Press, Cambridge, MA.