An Efficient Library for Solving CSP with Local Search

Philippe Codognet* Daniel Diaz†

* University of Paris 6, LIP6/IA
8 rue du Capitaine Scott, 75015 Paris, FRANCE
Philippe.Codognet@lip6.fr

† University of Paris-I, CRI
C1407, 90 rue de Tolbiac, 75 634 Paris, FRANCE
Daniel.Diaz@univ-paris1.fr

1 Introduction

In the last years, the application of local search techniques for constraint solving started to raise some interest in the Constraint Programming community, We proposed some years ago a domain-independent local search method called Adaptive Search for solving Constraint Satisfaction Problems (CSP) [1]. This method has now been fully re-implemented as a C-based framework library available as freeware (both source code and several benchmark examples) at the URL : http://contraintes.inria.fr/~diaz/adaptive/. This new implementation is more generic and efficient than the previous version used in [1], but is dedicated to permutation problems, that is : all variables have a same initial domain and are subject to an implicit all-different constraint. Many classical problems fall into this category.

We have designed a new meta-heuristics that takes advantage of the structure of the problem in terms of constraints and variables and can guide the search more precisely than a global cost function to optimize (such as for instance the number of violated constraints). We also use an adaptive memory in the spirit of Tabu Search in order to prevent stagnation in local minima and loops. This method is generic, can apply to a large class of constraints (e.g. linear and non-linear arithmetic constraints, symbolic constraints, etc) and naturally copes with over-constrained problems. Preliminary results on some classical CSP problems (Nqueens, magic squares, all-interval series, number partitioning, etc) show very encouraging performances. The input of the method is a problem in CSP format, that is, a set of variables with their (finite) domains of possible values and a set of constraints over these variables. We also need, for each constraint, an *error function* that will give an indication on how much the constraint is violated. This is very similar to the notion of "penalty functions" used in (continuous) global optimization. For instance the error function associated to an arithmetic constraint $X - Y \leq C$ will be $\max(0, |X-Y|-C)$. Adaptive search relies on iterative repair, based on variables and constraint errors information,

seeking to reduce the error on the worse variable so far. The basic idea is to compute the error function of each constraint, then combine for each variable the errors of all constraints in which it appears, therefore projecting constraint errors on involved variables. Finally, the variable with the maximal error will be chosen as a "culprit" and thus its value will be modified. In this second step we use the well-known min-conflict heuristics and select the value in the variable domain that has the best temptative value, that is, the value for which the total error in the next configuration is minimal. In order to prevent being trapped in local minima, the Adaptive Search method also includes an adaptive memory *à la* Tabu Search (variables can be marked Tabu and "frozen" for a few iterations), but also integrates possible restart-based transitions to escape stagnation around local minima. Restarts are partial and are guided by the number of variables being marked Tabu.

In our new implementation stochastic moves can also occur to escape from plateaux[1], with a given probability. This last action has proved to be very effective on performances for benchmarks where many local minima with large plateaux occur, such as the magic square problem presented in section 5, for which a ten time speedup factor can be achieved with an adequate tuning (6%) of unconditional escape when a plateau is encountered.

2 The Adaptive Search Algorithm

The input of the method is a problem in CSP format, that is, a set of variables with their (finite) domains of possible values and a set of constraints over these variables; Constraint Solving and Programming has proved to be very successful for Problem Solving and Combinatorial Optimization applications, by combining the declarativity of a high-level language with the efficiency of specialized algorithms for constraint solving, borrowing sometimes techniques from Operations Research and Numerical Analysis [2].

Consider a n-ary constraint $c(X1, \ldots, Xn)$ and domains $D1, \ldots, Dn$ for variables $\{X1, \ldots, Xn\}$. An *error function* f_C associated to the constraint c is simply a real-valued function from $D1 \times \ldots \times Dn$ such that $f_C(X1, \ldots, Xn)$ has value zero if $c(X1, \ldots, Xn)$ is satisfied. The error function will in fact be used as a heuristic value to represent the degree of satisfaction of a constraint and will thus give an indication on how much the constraint is violated. This is very similar to the notion of "penalty functions" used in (continuous) global optimization. That is, this error function will be an (approximation of) the distance of the current configuration to the closest satisfiable region of the constraint. For instance the (exact) error function associated to an arithmetic constraint $X - Y \leq C$ will be $\max(0, |X-Y|-C)$. Observe that, as the error is only used to heuristically guide the search, we can use any approximation when the exact distance is difficult (or even impossible) to

[1] We will used the french spelling for the plural ("plateaux") instead of the english plural of the french word ("plateaus") …

compute, such as approximating a (set of) region(s) by a convex hull.

Adaptive search relies on iterative repair, based on variables and constraint errors information, seeking to reduce the error on the worse variable so far. The basic idea is to compute the error function of each constraint, then combine for each variable the errors of all constraints in which it appears (possibly normalized), therefore projecting constraint errors on involved variables. Finally, the variable with the maximal error will be chosen as a "culprit" and thus its value will be modified. The neighborhood consists in all possible modifications of the value of this variable, that is, in permutation problems, in all possible swaps with other variable/value pairs. The best neighbor (smallest value for the overall cost function) is then selected to compose the next configuration.
In order to prevent being trapped in local minima, the adaptive search method also includes an adaptive memory as in Tabu Search : each variable leading to a local minimum is marked and cannot be chosen for the few next iterations. It is worth noticing that conversely to most Tabu-based methods we mark variables and not couples <variable,value>, and we do not systematically mark variables when chosen in the current iteration but only when they lead to a local minimum.

Before detailing the basic iteration of the adaptive search algorithm, we need to add some extra control parameters to tune the search process, in particular to handle (partial) restarts. In order to avoid being trapped with a large number of Tabu variables and therefore no possible diversification, we decide to randomly reset a certain amount of variables when a given number of variables are Tabu at the same time. Thus the *reset limit* is the number of simultaneous Tabu variables to reach in order to perform a (partial restart). On restart, depending on the *reset percentage*, we randomly reset a certain ratio of variables to random values.

Input :

Problem given in CSP form :

- a set of variables V={V1, V2,…, Vn} with associated domains of values
- a set of constraints C={C1, C2,…, Ck} with associated error function
- a combination function to project constraint errors on variables
- a (positive) cost function to minimize

Some tuning parameters :

- T : Tabu tenure (number of iterations a variable will be frozen on local minima)
- RL : reset limit
- RP : reset percentage
- Max_I : maximal number of iterations before total restart

- Max_R : maximal number of total restarts

Output :

a sequence of moves (modification of the value of one of the variables) that will lead to a solution of the CSP (configuration where all constraints are satisfied) if the CSP is satisfied or to a quasi-solution of minimal cost otherwise.

Algorithm :

Iteration = 1
Restart = 1
Tabu_Nb = 0
repeat
Start from a random assignment A of variables in V
Opt_Sol = A
Opt_Cost = cost(A)
Repeat
1. Compute errors of all constraints in C and combine errors on each variable (by considering for a given variable only the constraints on which it appears)
2. select the variable X (not marked Tabu) with highest error
3. evaluate costs of possible moves from X
4. **if** no improving move exists
then mark X as Tabu until iteration number : Iteration + T
Tabu_Nb = Tabu_Nb + 1
if Tabu_Nb ≥ RL
then randomly reset RP variables in V (and unmark those Tabu)
else select the best move and change the value of X accordingly
to produce next configuration A'
if cost(A') ≤ Opt_Cost
then Opt_Sol = A'
Opt_Cost = cost(A')

until a solution is found or Iteration ≥ Max_I
until a solution is found or Restart ≥ Max_R
output (Opt_Sol, Opt_Cost)

3 Escaping Plateaux

The above algorithm does not perform any special action to in case of plateaux, that is, when the

selected variable has no strictly improving moves in the neighborhood but has some neighbor(s) with equal cost value. In some of our benchmark, we have found experimentally that the number of neighbors with cost values equal to that of the selected variable is around N, for a problem with N variables and thus neighborhoods of size N^2. We have thus decided to add a stochastic component that will perform as follows when a plateau is encountered :

- with probability $1\text{-}p$: escape from plateau (mark current variable as Tabu and randomly choose another variable, then change its value)
- with probability p : choose a neighbor on the plateau

We have experimentaly found that the best results are achieved witha value of p around 95%.

4 A Sample Benchmark : Magic Squares

The magic square puzzle consists in placing on a NxN square all the numbers in $\{1,2,..., N^2\}$ such as the sum of the numbers in all rows, columns and the two diagonal are the same. It can therefore be modeled in CSP by considering N^2 variables with initial domains $\{1,2,..., N^2\}$ together with linear equation constraints and a global *all_different* constraint stating that all variables should have a different value. The constant value that should be the sum of all rows, columns and the two diagonals can be easily computed to be $N(N^2+1)/2$.

The instance of adaptive search for this problem is defined as follows. The error function of an equation $X1 + X2 + ... + Xk = b$ is defined as the value of $X1 + X2 + ... + Xk - b$. The combination operation is the absolute value of the sum of errors (and not the sum of the absolute values, which would be less informative : errors with the same sign should add up as they lead to compatible modifications of the variable, but not errors of opposite signs). The overall cost function is the addition of absolute values of the errors of all constraints The method will start by a random assignment of all N^2 numbers in $\{1,2,..., N^2\}$ on the cells of the NxN square and consider as possible moves all swaps between two values. We thus have at each iteration a dynamic neighborhood consisting of N^2 configurations.

We report in the following the performances results on several NxN instances (remark that a NxN instance means N^2 variables with domains of size N^2) for :

- the Localizer++ C-based local search library [2],
- the basic adaptive search algorithm (with no stochastic escape on plateaux)
- the improved adaptive algorithm with 0.94 probability of staying on the plateau (if any)

Results are given in seconds, for a PC with pentium-III 800 Mhz processor.

Size	Localizer++	Basic Adaptive (100%)	Adaptive (94%)
20 x 20	313	3.4	0.2
30 x 30	1969	18	1.2
40 x 40	8553	58	3.4
50 x 50	23158	203	5.6
100 x 100			131

5 Conclusion

We have presented the basic features of the Adaptive Search method, a new metaheuristics for solving Constraint Satisfaction Problems (CSP). This method has now been fully re-implemented as a C-based framework library available as freeware (both source code and several benchmark examples) at the URL : http://contraintes.inria.fr/~diaz/adaptive/. Preliminary results are encouraging and a recent extension consists in considering controlled stochastic moves when a plateau is reached around a local minima. On the Magic Square Problem, this mechanism gives a ten time speedup factor.

We are currently investigating multi-point extensions (*à la* genetic algorithms) and more flexibility and stochasticity in the choice of the variable to modify.

6 References

[1] P. Codognet and D. Diaz. Yet Another Local Search Method for Constraint Solving. In : *Proceedings of SAGA01, 1st International Symposim on Stochastic Algorithms : Foundations and Applications*, LNCS 2246, Springer Verlag 2001.

[2] L. Michel and P. Van Hentenryck. Localizer++ : an open library for local search. *Research Report*, Brown University 2001.

[3] V. Saraswat, P. Van Hentenryck et al. Constraint Programming, *ACM Computing Surveys*, vol. 28 no. 4, December 1996.