Scientific Research and Essays Vol. 6(21), pp. 4587-4596, 30 September, 2011
Available online at http://www.academicjournals.org/SRE
DOI: 10.5897/SRE11.891
ISSN 1992-2248 ©2011 Academic Journals

Full Length Research Paper

Hybrid strategies of enumeration in constraint solving

Eric Monfroy[1,4]*, Broderick Crawford[1,2], Ricardo Soto[2,3], and Fernando Paredes[5]

[1]Departamento de Informática, Universidad Técnica Federico Santa María, Av. España 1680, Valparaíso, Chile.
[2]Pontificia Universidad Católica de Valparaíso, Chile.
[3]Universidad Autónoma de Chile, Chile.
[4]CNRS, LINA, Université de Nantes, France.
[5]Escuela de Ingeniería Industrial, Universidad Diego Portales, Santiago, Chile.

Accepted 16 August, 2011

Enumeration strategies (that is, selection of a variable and a value of its domain) are crucial components of Constraint Programming: they significantly influence the performances of the solving process, sometimes of several orders of magnitude. In this paper, we propose to use Local Search in order to help and guide enumeration: we extend the usual variable selection strategies of constraint programming and we perform the value selection with respect to the results of some local search. The experimental results obtained are rather promising.

Key words: Constraint programming, local search, hybridization methods, heuristic search.

INTRODUCTION

Constraint programming is a modern and powerful programming paradigm devoted to the efficient resolution of constraint-based combinatorial problems and optimization problems. Apt (2003) revealed in a study the formal description of constraint programming and Niederlinski (2011) reported a practical introduction of constraint logic programming. A constraint satisfaction problem (CSP) is a formal problem representation that mainly consists in a sequence of variables lying in a domain (the possible values of the variable) and a set of constraints linking these variables. The goal is to find a complete variable-value assignment that satisfies the whole set of constraints, and that optimizes an objective function in case of optimization.

Constraint programming is widely used in various application areas, such as scheduling, timetabling, travelling salesman problem (TSP), computer graphics for geometric coherence, conception of complex systems, database systems to ensure and/or restore data consistency.

*Corresponding author. E-mail: ericmonfroy@gmail.com.

Abbreviations: CSP, Constraint satisfaction problem; **LS,** local search; **TSP,** travelling salesman problem.

Systematic backtracking enhanced with pruning of the search space by local consistency enforcement has been successfully applied to combinatorial problems for decades. One of the main advantages of these techniques is their completeness: if the problem has a solution they find it, and they give a proof when there is no solution. However, they do not always scale well for large problems.

Incomplete methods, such as local search (LS) (Hoos and Stützle, 2004) or genetic algorithm (Michalewicz, 1998) explore some "promising" parts of the search space with respect to specific heuristics. These techniques are incomplete, but they scale better to large problems.

Constraint propagation based solvers (CP) (Apt, 2003) are complete methods designed to solve constraint satisfaction problems (CSP). They are based on a search tree structure and they proceed by interleaving enumeration phases and constraint propagation phases. Constraint propagation prunes the search tree by eliminating values that cannot participate in a solution: hence, they reduce the search space. Enumeration is in charge of branching: it creates one branch by instantiating a variable with a value of its domain ($x = v$) and another branch ($x \neq v$) which is used for backtracking when the first branch does not lead to any solution.

Every enumeration strategy which preserves solutions is valid (for example, first-fail, max-occurrence, brelaz, etc.). However, they have a significant impact on solving efficiency: there can be a factor of several magnitudes of execution time between two strategies. Moreover, it is well known that there is no universal strategy being the best (or one of the best) in terms of efficiency for all problems, that is, no free lunch theorems (Wolpert and Macreacdy, 1997). Numerous studies have been conducted about enumeration strategies (Beck et al., 2004), or sometimes for specific classes of problems, sometimes for genericity (Caseau and Laburthe, 1994).

A common idea to get more efficient and robust algorithms consists in combining several resolution paradigms in order to take advantage of their respective assets. Such combinations are now well recognized and they have been more and more studied during the last years by several communities, including the constraint programming community. Considering that very efficient constraint solvers are currently available, the challenge is to make them cooperate in order to:

1. Get better solution, either in terms of solution quality (a better optimum closer to the global optimum can be reach by combining various optimization systems) or in terms of types of solutions, for example, when there are numerous solutions.
2. Provide a solution that better suits the user (that is, when there are numerous solutions, provide a solution closer/similar to the solution that would give an expert of the domain).
3. Improve solving efficiency (power of the solver) and solving time (speed of the solver).
4. Tackle and solve more problem classes, for example, solve hybrid problems (with different types of domains for variables and different types of constraints) that could not be solved by a unique solver.

Hybridizations of CP and LS (Focacci et al., 2002; Wallace, 2005; Wallace and Schimpf, 2002) are now more and more studied in the constraint programming community. Our approach is influenced by the works described in previous studies (Mazure et al., 1998; Wallace and Schimpf, 2002). However, these works address SAT problems whereas we consider finite domains (that is, finite sets of integers), and thus algorithms and strategies are very different. Hulubei and O'Sullivan (2005) reported about the importance of the enumeration strategy for the heavy-tailed behaviour which characterizes problems very sensitive to the enumeration strategies.

In this paper, we propose a Master/Slave hybridization: Local search guides the search by helping and improving selections of variables and values for enumeration. Our goal is to design a solving process in which enumeration strategies (and more especially variable selection) impact less solving efficiency. In other terms, we want to avoid that a "bad" variable assignment and thus, a bad variable selection that drastically reduces efficiency, without penalizing a "good" selection strategy. Consequently, we also want to avoid heavy-tailed behavior phenomenon. The idea is thus to reduce the effect that could have a bad strategy selected by the user when this one does not know which strategy is the best adapted (or well adapted) to his problem.

The technique we use thus consists in improving standard enumeration strategies. Local search is performed before each enumeration of the solving process in order to improve standard variable selection strategies (for example, first-fail) by requesting some properties of variables (for example, not conflicting variables with respect to a local search) and to provide a value for the selected variable (for example, with respect to the evaluation function of the local search).

Thus, for each enumeration phase, LS is run on a different problem, that is, the problem that remains after 1) removing variables that have been instantiated during the constraint propagation phase (either by enumeration or propagation), 2) removing instantiated constraints (for example, constraint such that each variable of the constraint has been instantiated before) that are satisfied by these assignments, 3) and after reducing the domains of variables during the various propagation phases that is previously achieved.

Obviously, it can also happen that LS solves the remaining problem. Note that our algorithm is still complete. The experimental results we obtained with our prototype implementation are promising.

MOTIVATIONS AND BACKGROUND

A CSP is defined by:

- a set of variables $\{X_1 \cdots X_n\}$
- a set $D_i = \{v_{i,1} \cdots v_{i,i_n}\}$ of possible values for each variable X_i, that is, the domain of X_i,
- And a set of constraints. $\{C_1 \cdots C_k\}$.

A solution is an instantiation of all the variables satisfying all the constraints.

Constraint propagation based solvers

Systematic backtracking (based on a search tree structure) is used in many constraint solvers. This method is complete, and can be combined with constraint propagation, a technique for pruning the search space. However, due to the combinatorial aspect of the problems, and thus of the search tree, it often behaves poorly for large combinatorial problems. Constraint propagation based solvers can be described by the generic algorithm of Algorithm 1 (Apt, 2003). They

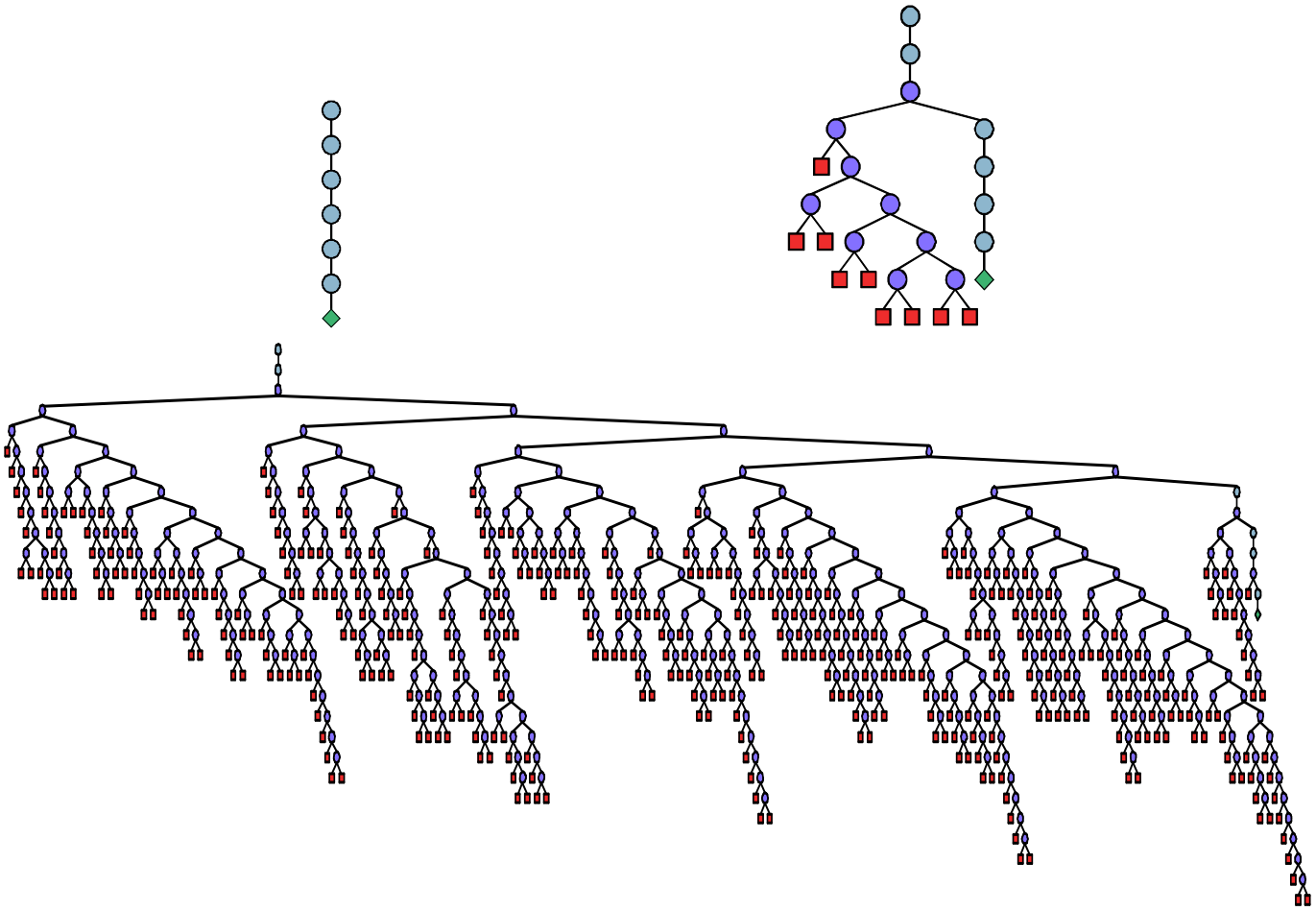

Figure 1. 10-Queen problem solved with 3 standard enumeration strategies.

interleaved propagation phases with split/enumeration steps. Propagation reduces the search space by removing values of variables that cannot participate to a solution of the CSP.

A split cuts a CSP P into several CSPs P_i, such that the union of solutions of the P_i is equal to the solutions of P. In other words, a split preserves the solutions of the initial CSP. Each P_i differs from P in that the split domain is replaced by a smaller domain. The two main classes of split strategies are:

1. Segmentation: split a domain into 2 (leading to a binary tree) or several (leading to an n-ary tree).
2. And enumeration: split a domain into one value (assignment of the variable to this value) and the rest of the domain (the variable must be different from this value).

In the following, we focus on enumeration. The search function of Algorithm 1 manages the choice points (that is, sub-CSPs created by split) by doing recursive calls to the SOLVE algorithm: it may define searches such as depth first, or breadth first search; it can enforce finding one solution or all solutions; or can manage optimization or satisfaction. Finished is a Boolean set to true when the problem is either solved or not satisfiable.

The impact of enumeration

Figure 1 show 3 search trees for the resolution of the 10-queen problem (for example, http://en.wikipedia.org/wiki/Eight_queens_puzzle for a description of the problem) with 3 different enumeration strategies. The first strategy (Figure 1, up-left) directly goes to a solution (6 enumerations, no backtrack): thus the tree is only composed of one branch. The second one (Figure 1, up-right), after a bad choice for the second enumeration (generating 17 backtracks), reaches a solution. The last strategy (Figure 1, down) performs numerous wrong choices (807 backtracks) before reaching a solution. Obviously strategies have drastically different efficiencies, often several orders of magnitude, and thus it is crucial to select a good one that

unfortunately cannot be predicted in the general case.

Local search (LS)

While complete methods can prove that a problem is not satisfiable (exhaustive exploration of the search space), incomplete methods will be ineffective in that case. However, they scale very well to large applications since they mainly rely on heuristics providing a more efficient exploration of interesting areas of the search space. This class of methods (called metaheuristics) covers a wide panel of paradigms from evolutionary algorithms to local search techniques (Michalewicz, 1998; Hoos and Stützle, 2004) on which we focus in the following.

LS techniques usually aim at solving optimization problems as shown in Hoos and Stützle (2004). In the context of constraint satisfaction, these methods minimize the number of violated constraints to find a solution of the CSP. A local search algorithm (Algorithm 2), starting from an initial configuration s, explores the search space by a sequence of moves. At each iteration, the next move corresponds to the choice of one of the so-called neighbors s' of s. This neighborhood often corresponds to small changes of the current configuration. Moves are guided by a fitness function (f on Algorithm 2) which evaluates their benefit from the optimization point of view, in order to reach a local optimum. The algorithm stops (Boolean finished) when a solution is found or when a maximum number of iterations is reached. The current optimum is stored in the variable opt which is updated when the new configuration is better than the previous one with respect to the fitness function.

Hybridizing CP and local search (LS)

A common idea to get more efficient algorithms consists in combining resolution paradigms (Focacci et al., 2002; Wallace and Schimpf, 2002; Wallace, 2005). Such combinations are more and more studied in the constraint programming community, mainly for combining CP with LS or genetic algorithms. Hybridizations have often been tackled through a Master/Slave like management, and are often related to specific problems or class of problems. Among Master/Slave approaches in which CP is the master are, local probing (Kamarainen and Sakkout, 2004), selecting variables by LS (Mazure et al., 1998), S for improving partial assignment (Caseau and Laburthe, 1994; Prestwich, 2000), LS for guiding the backtrack (Prestwich, 2001).

Generally, when LS is considered as the master, CP is used to improve the quality or the size of the neighborhood (Jussien and Lhomme, 2002). Other techniques sacrifice completeness. Numerous works also consider sequential or parallel hybridization of black-box solvers Castro C (Monfroy, 2004). Few works focus on generic hybrid solvers or hybrid frameworks (Monfroy et al., 2005).

LOCAL SEARCH (LS) GUIDED ENUMERATION IN A CP SOLVER

We are interested in making good choices for enumeration, i.e., selection of a variable and a value. Our idea is the hybridization LS helping CP for enumerations. There exist numerous enumeration strategies for CP, and some are known to be efficient for some problems. We thus focus on improving these strategies, keeping their own qualities. Let us illustrate this on an example. One standard strategy, the first-fail, selects the variable with the smallest domain, and instantiates it with the first value of its domain. In our hybrid solver, we run a LS before enumeration. A possible modification of the first-fail is to select the smallest variable which is not in conflict in LS, and to assign it the tentative value given by the LS.

The hybrid algorithm

We designed a Master/Slave hybridization (Algorithm 3) in which a LS guides enumeration in CP to quickly find a solution: a LS is performed before each enumeration to find information about variables and their tentative values. The exploration of the search tree is a depth-first left-first search. The "while loop" of the generic algorithm does not exist anymore since we are looking for one solution. The selec_enumeration function is the key point: not only it is based on criteria for variable and value selection, but also on the result of the LS, that is, the tentative values Tentative_Val of the variables. Tentative values are the result of the search made by LS that maximized the fitness function that is, in this case, the number of violated constraints. A LS finds tentative values for the non instantiated variables of the CSP considering that values assigned by previous enumeration and propagation are correct: a LS does not reconsider previous work (propagation and instantiation); this task is left to the CP solver that will backtrack when it detects insatisfiability of the CSP.

Let us fix first some notions. A variable can only be instantiated by propagation or enumeration, but never by LS itself. A tentative value is a value given by LS to a variable which has not yet been instantiated; this value must be an element of the domain of the variable; we denote it $X \equiv v$. A tentative *CSP* is a CSP in which variables which do not have yet a value are given a tentative value. A conflicting constraint is a constraint which cannot be satisfied with the values and tentative values of the variables it contains. Consider the following CSP: $X = 3, Y \equiv 4, X > Y$.

- 3 is the value of X given by enumeration or

propagation.

- 4 is the tentative value given to Y by the last LS.
- The constraint $X > Y$ is a conflicting constraint.

A possibly conflicting variable (conflicting variable in short) is a not yet instantiated variable that appears in a conflicting constraint. Note that the tentative value of this variable may participate in a solution. Consider the CSP given by $X \in [1..5]$, $Y \in [1..5]$, $X > Y$ and the tentative values, $X \equiv 3$, $Y \equiv 4$. Y is a conflicting variable since it appears in $X > Y$ which is a conflicting constraint. However, a solution of the CSP is $X = 5$, $Y = 4$.

The local search (LS) algorithm

We use a descent algorithm for LS: we only consider improving (in terms of the evaluation function) neighbors when moving. Moreover, we do not consider all the neighbors to select the best improving neighbor, but we selected the first improving neighbor. Diversification is achieved by restarting the algorithm (that is, a loop around the algorithm of Algorithm 2). The result of the LS is thus a local optimum, the best configuration w.r.t. the evaluation function.

A configuration consists in giving a tentative value to each non instantiated variable of the CSP. The size of configurations thus changes at each LS since the number of non instantiated variables varies w.r.t. to enumerations, propagations, and backtrackings. The algorithm is as much generic as can be. It is parameterized by:

1. $fMaxIter$, a function to compute the maximum number of iterations in a search;
2. $fMaxRestart$ a function to compute the maximum number of restarts, that is, the maximum number of local searches to be performed;
3. $feval$, a fitness/evaluation function to estimate configurations,
4. $fneighbor$, a neighborhood function to compute neighbors of configurations.

We now present some of our functions. This list is not exhaustive, and some more functions can be designed. $fMaxIter$ and $fMaxRestart$ are functions based on the number of variables and constraints. For example, functions such as $p.N\_Var$ where p is a given integer and N_Var is the number of non instantiated variables can implement $fMaxIter$ and $fMaxRestart$.

The evaluation function is based on the constraints, the variables, their current domains, and/or their number of occurrences in the CSP. Thus, possible functions are $fConfC$ (the number of conflicting constraints), $fConfV$ (the number of conflicting variables), or functions such as $fCWeightff$ defined by $fCWeightff(V) = \sum_{i=1}^{n} 1/size(V_i)$ where V is a set of conflicting variables, and $size$ is a function that returns the number of values in the domain of a variable. Functions such as $fCWeightff$ give more importance to some variables for evaluation (the ones with small domains for $fCWeightff$) in order to be combined with some variable selection strategies of the CP algorithm (for example, a first-fail variable selection strategy in the case of $fCWeightff$). We thus have a set of evaluation functions to privilege (or un-privilege) the variables with the smallest domain, with the largest domain, the variables having more occurrences, etc. We do not detail these other functions, their principle being similar to $fCWeightff$. These functions return 0 when the tentative CSP is solved.

The neighborhood functions aim at finding an improving neighbor. A neighbor is computed by changing the tentative value of k variables. k is either a given integer, or the result of a function based on the number of variables (such as $log(Number\ Variable)$). There are several criteria to select the variables to be changed: randomly, a randomly selected conflicting variable, and a family of functions based on the domain size and/or occurrences of variables (criteria similar to the criteria of the evaluation function). For these last functions, a higher probability is given to the variables we want to privilege. The new tentative value is a randomly selected value of the domain of the variable.

Let us illustrate it for the $Nweightff2$ neighborhood function which gives more probability to change the value of a variable with a small domain. To this end, we construct a list of pairs $((V_1, I_1), (V_2, I_2), \ldots, (V_n, I_n))$ where V_i are variables, and I_i are consecutive intervals of width $\frac{1}{size(V_i)^2}$ defined by:

$$I_1 = \left[0..\left(\frac{1}{size(V_1)^2}\right)\right]$$

And

$$I_i = \left[\sum_{j=1}^{i-1}\left(\frac{1}{size(V_j)^2}\right)..\sum_{j=1}^{i}\left(\frac{1}{size(V_j)^2}\right)\right]$$

Then, a random real number r between 0 and $\sum_{j=1}^{n} 1/size(V_j)^2$ is generated, and the selected variable is the one which interval contains, r. Note that $NCweightff2$ is defined the same way, but it only considers conflicting variables. We do not detail the other neighborhood functions based on smallest/largest domain, number of occurrences since they are built the same way as $Nweightff2$. A LS strategy is thus a combination of the 4 parameters (that is, functions) described above.

Strategies of enumeration

The enumeration strategies are a combination of a

Table 1. n-queens problem with various strategies and hybrid strategies.

	10 - queens		20 - queens		25 - queens		50 - queens	
	t	e	t	e	t	e	t	e
first	0.02	16	44.69	22221	13.05	4508	-	-
S1	0.051	15.01	0.473	51.57	0.937	73.31	12.17	702
S4	0.055	15.39	0.519	52.07	1.121	87.33	8.68	252.07
ff	0.02	13	0.09	40	0.20	68	2.78	506
S6	0.039	13.301	0.384	33.43	0.662	41.58	4.81	84.91
S14	0.043	13.60	0.385	33.23	0.639	37.51	5.41	114.14
S15	0.043	13.48	0.383	31.20	0.648	35.31	5.22	100.93
S16	0.04	13.00	0.401	33.44	0.624	32.64	4.39	65.76
S23	0.073	12.80	0.578	29.79	1.082	40.095	7.584	67.38
S35	0.084	12.56	0.576	25.66	1.045	33.56	7.822	77.08
S36	0.082	11.25	0.620	27.30	1.022	29.73	7.025	50.48
S42	0.101	7.85	0.790	15.51	1.427	20.26	10.627	33.78

variable selection criterion and a value selection criterion. We consider standard variable selection criteria of CP that we can refine using the result of the LS, for example, selecting variables that are not conflicting in the LS. Here are some of the variable selection criteria:

1. $first$ selects the first variable (resp. the first non conflicting variable) occurring in the CSP;
2. $first\_nc$: the first non conflicting variable occurring in the CSP;
3. $first - fail$ the variable with the smallest domain;
4. $first - fail\_nc$ the non conflicting variable with the smallest domain;
5. $occurrence$: the variable with the largest number of occurrences in constraints;
6. $occurrence\_nc$: The non conflicting variable with the largest number of occurrences in constraints.

The value selection totally relies on the result of the LS: the tentative value (in the result of LS) of the variable is selected. A hybrid strategy is thus the combination of 6 parameters: the value selection criterion, the variable selection criterion, the function to compute the numbers of iterations, the function to compute the number of restarts, the evaluation function and, and the neighborhood function.

RESULTS

Our prototype implementation has been written with the ECLiPSe Constraint Programming System (Schimpf and Shen, 2010) using finite domain libraries for constraint propagation and the repair library which significantly eased the handling of tentative values in LS.

We did not try to optimize the code. Instead, we kept it as open and parameterized as possible to allow various hybrid strategies. In the same way, we did not specialize the LS for specific problems, although it is well known that generic LS generally behaves poorly. The tests were run on a Athlo 3000+ with 1Go of RAM. For each test we performed 1000 runs and we present the average result.

Experimentations

Table 1 shows the results of several strategies for some instances of the n-queen problem. We use a standard model of the problem, as can be found in the examples of ECLiPSe. The all different global constraint is duplicated into the pair-wise differences of all variables ($\forall i,\ \forall j \neq i, X_i \neq X_j$). Hence, the pruning is more powerful using the global constraint, and the LS can better count conflicting constraints using the difference constraints ($X_i \neq X_j$).

A column represents a problem instance; a row is the performance of a strategy for the various instances; *t* is the average CPU time (over 1000 runs) in seconds and *e* the average (due to the random aspects of the LS) number of enumerations (both "good" enumerations that lead to a solution, and "bad" ones that enforced backtrack). We only consider one LS, that is there is no restart, and the neighborhood function changes the value of one variable. The enumeration strategies appearing in Table 1 are defined as follows:

first: pure CP strategy,

- selection of variable: $first$, that is, the first variable in the order of appearance in the CSP
- selection of value: smallest value of the domain

S1: hybrid strategy,

- selection of variable: $first$

- selection of value: tentative value of this variable computed by the LS defined by:
  - evaluation function: $fConfC$, that is, number of conflicting constraints
  - neighborhood function: $fNConf$, that is, change of a conflicting variable
  - size of the LS: $2.number\_variables$, that is, twice the number of variables in the CSP

S4: hybrid strategy,

- selection of variable: $first\_nc$, that is, the first non conflicting variable
- selection of value: tentative value of the LS:
  - evaluation function: $fConfV$, that is, number of conflicting variables
  - neighborhood function: $fNConf$
  - size of the LS: $2.number\_variables$

ff: pure CP strategy,

- selection of variable: $first\_fail$, that is, the first variable with the smallest domain
- selection of value: smallest value of the domain

S6: hybrid strategy,

- selection of variable: $first\_fail$
- selection of value: tentative value of the LS:
  - evaluation function: $fConfV$
  - neighborhood function: ran, that is, randomly selects a conflicting variable and gives it a value randomly chosen in its domain to create a neighbor
  - size of the LS: $2.number\_variables$

S14: hybrid strategy,

- selection of variable: $first\_fail\_nc$, that is, the first non conflicting variable with the smallest domain
- selection of value: tentative value of the LS:
  - evaluation function: $fConfC$
  - neighborhood function: ran
  - size of the LS: $2.number\_variables$

S15: hybrid strategy,

- selection of variable: $first\_fail\_nc$
- selection of value: tentative value of the LS:
  - evaluation function: $fConfC$
  - neighborhood function: ran
  - size of the LS: $2.number\_variables$

S16: hybrid strategy,

- selection of variable: $first\_fail\_nc$
- selection of value: tentative value of the LS:
  - evaluation function: $fConfC$
  - neighborhood function: $NCWeightff2$, that is, gives more probability to change the value of a conflicting variable with a small domain
  - size of the LS: $2.number\_variables$

S23: hybrid strategy,

- selection of variable: $first\_fail$
- selection of value: tentative value of the LS:
  - evaluation function: $fConfC$
  - neighborhood function: ran
  - size of the LS: $5.number\_variables$

S35: hybrid strategy,

- selection of variable: $first\_fail\_nc$
- selection of value: tentative value of the LS:
  - evaluation function: $fConfC$
  - neighborhood function: ran
  - size of the LS: $5.number\_variables$

S36: hybrid strategy,

- selection of variable: $first\_fail\_nc$
- selection of value; tentative value of the LS:
  - evaluation function: $fConfC$
  - neighborhood function: ran
  - size of the LS: $5.number\_variables$

S42: hybrid strategy,

- selection of variable: $first\_fail\_nc$
- selection of value: tentative value of the LS:
  - evaluation function: $fConfC$
  - neighborhood function: ran
  - size of the LS: $10.number\_variables$

We first compare strategies based on the $first$ criterion: variable to enumerate are selected in the order of appearance in the CSP. $first$ is the pure CP strategy (without LS): the smallest value of the domain of the selected variable is chosen for enumeration. *S1* and *S4* are hybrid strategies. *S1* is also based on a $first$ criterion for variable selection; the selected value is the tentative value of the result of the LS like for each of our hybrid strategies. The evaluation function counts the number of conflicting constraints ($fConfC$). The neighborhood change a conflicting variable ($fNConf$). The size of the LS is twice the number of variables in the problem. *S4* differs in that the first non conflicting variable is selected for enumeration ($first\_nc$), and the evaluation function is the number of conflicting variables ($fConfV$). Both *S1* and *S4* show very good results compared to $first$, both in CPU time and number of enumerations. We don't have the results for 50-queens for the pure CP $first$ strategy:

for 30-queens, more than 4.000.000 enumerations are already required.

All the other strategies (*ff* and *S6* to *S42*) are based on a $first\ fail$ variable criterion. *ff* is the pure CP strategy: enumeration with the smallest value of the domain of the variable with the smallest domain. *S6* and *S23* also select the variable with smallest domain, whereas *S14, 15, 16, 35, 36, and 42* select the non conflicting variable with the smallest domain. We can see with these strategies that it tends to be better to select a non conflicting variable.

The evaluation function for each *S14, 15, 16, 35, 36, 42* is the number of violated constraints. With respect to other tests we performed for n-queens, it seems that using an evaluation function based on the size of the domain (such as $fCWeightff$) slightly reduces the number of enumerations but also slightly increases the CPU time.

For *S14,15,16,35,36,42*, the number of iterations is $p.number\_variables$, where $p = 2$ for *S14,15,16*, $p = 5$ for *S23,35,36*, and $p = 10$ for *S42*. We can see that when p increases, the number of enumerations decreases, and the CPU time increases (due to the overhead of the longer LS). Increasing p is more profitable for larger problems, such as 100-queens.

In *S14,23,35*, the neighborhood function selects randomly a variable to change its value to create a neighbor; in *S6,15,36* it is a random conflicting variable whose value is changed; in *S16*, $NCWeightff2$ gives more probability to change the value of a variable with a small domain (this is coherent with the $first - fail$ selection of variable for enumeration). The difference between *S14,15,16* is only the neighborhood function: similarly to the other tests we made, it tends that it is slightly worth (in terms of enumerations, and CPU time) having more clever neighborhood functions based on conflicting variables and probabilities with respect to the size of the domains of the variables.

The advantage of the hybrid strategies over the pure CP $first - fail$ strategy is in terms of enumerations, lesser enumerations are required to solve the problems. But this is at the cost of CPU time, however, the larger the instances and the smaller the difference of time. Moreover, the overhead could be significantly reduced (and erased in most of the case) with a better implementation of the LS algorithm: recall that we did not specialized it, and let it open and parameterized with numerous functions in order to be able to perform some tests on other types of problems, with other evaluation functions, neighborhood functions, etc.

DISCUSSIONS

We tested the same strategies on several instances of the Latin square problem (for example, http://en.wikipedia.org/wiki/Latin).

The comments are the same as for n-queens: all the hybrid strategies improve the number of enumerations. The differences between the hybrid strategies are a bit more pronounced, but the main differences are coming from the length of the search.

Only *S42* behaves differently: it is from far the best strategy in terms of enumerations (for example, less than half the number of enumerations for Latin-10), but also the slowest one (2 to 3 times slower than other strategies for Latin-15). *S42* is the strategy that performs the longest LS: this explains the good enumeration. But the overhead is not sufficient to explain the very slow run and we could not find any explanation for it.

The magic sequence problem (http://delphiforfun.org/programs/MagicSequence.htm) does not show the interest of hybrid strategies: on average, the hybrid strategies do not improve the number of enumerations for the magic sequence problem. However, the overhead is negligible. This is due to the solution $(n-4, 2, 1, 0, \cdots, 0, 1, 0, 0, 0)$ for sequences of size $n > 6$ and the pruning of the occurrence global constraint: quickly, most of the domains are reduced to $[0,1]$ and whatever enumeration strategy performs well. Thus, guiding the search with, a local search does not help much.

We were surprised that on average, all the hybrid strategies show quite similar performances. Only the length of the LS has a significant impact: the longer the LS, the lesser enumerations, but at the cost of CPU time. However, it seems that integrating criteria (such as, domain size) in the neighborhood and evaluation functions pays in term of enumeration and is not too costly in time.

As expected the hybrid strategies reduce the number of enumeration. This is due to the fact that better enumerations are found using a kind of probing (the LS), and that from times to times, it also happens that the probing (the LS) finds a solution; most of the time, this phenomenon happens at the bottom of the search tree, when there remain few variables to be instantiated.

We observed the standard deviation and variance of the number of enumerations and of the CPU time: the shorter is the LS, the larger is the variance. It could be interesting to study more deeply this deviation to classify hybrid strategies for some practical use. Indeed, when one has to solve few problems, it can be more useful to have a strategy which is maybe a bit worse in average but that has a smaller deviation/variance; this way, runs would be more homogeneous without bad runs. When one has to solve numerous problems, we think a strategy with a smaller average (even with a large variance) is well suited:

1. First, the time lost in the long runs is partially erased by the average on numerous problems.
2. But also, one could give a time out; a run exceeding this time out would be stopped and tried again with another strategy (preferably one having a worse average, but having a smaller variance).

In our experimental results, the difference (in terms of

enumeration and time) between our hybrid strategies is less noticeable than between the pure CP strategies. For example, there are several orders of magnitude between the first and first-fail strategies in the n-queens. Moreover, our hybrid strategies are better (enumeration and time) than the pure first one, and close (better for enumerations, worse in time due to the non specialized implementation of LS, but less than an order of magnitude) to the $first - fail$ one.

Thus, if one does not know which strategy is adapted to his problem, it is convenient to select an hybrid one: it will either gives good performance, or acceptable performance with respect to the best strategy. Moreover, the LS implementation can significantly be improved to reduce and even erase the overhead of the hybrid strategies.

CONCLUSION

In this paper, a hybrid (and complete) solver that uses a hybrid enumeration strategy, that is, LS to guide the enumeration process was presented. Our technique is rather simple to integrate in a constraint programming system, and the first experimental results are rather promising. In a study by Mazure et al. (1998) a GSAT-like procedure guides the branching of logically-complete algorithms based on Davis and Putnam's like techniques. Although, this could be seen as similar to our technique, this happen in the context of SAT, and the heuristics and algorithms (both the complete and incomplete ones) are quite different.

Wallace and Schimpf (2002) also reports about a repair-based algorithm, GSAT, combined with a constructive algorithm using propagation. At each node of the search tree, GSAT is run on all variables in order to choose which variable to enumerate. Our work is close to this algorithm since the systematic backtracking techniques are both enhanced with propagation. However, Wallace and Schimpf (2002) addresses SAT problems, and thus the enumeration strategies, and the incomplete techniques are also quite different. From some aspects, in the local probing of Kamarainen and Sakkout (2004), LS helps in selecting variables. However, the key idea is that LS creates assignments, and CP modifies the sub-problem that LS is solving in order to guide it to search method where solutions can be found.

Although, we did not see the heavy-tailed behavior (Hulubei and O'Sullivan, 2005) in our runs and tests, more observations are needed to determine whether we definitively avoided it. We plan to improve the implementation of our LS algorithm in order to reduce or even erase the overhead given by the hybrid strategies. We think of refining the notions of neighborhood and evaluation to integrate some constraints of the model in these functions, that is, to obtain a generic LS which can adapt itself to some specific constraints of the problem.

We plan to extend our hybridization to optimization and perform some tests with other enumeration strategies.

REFERENCES

Apt KR (2003). Principles of Constraint Programming. Cambridge University Press.

Beck JC, Prosser P, Wallace R (2004). Variable Ordering Heuristics Show Promise. In Proceedings of CP'2004, LNCS, 3258: 711-715.

Caseau Y, Laburthe F (1994). Improved clp scheduling with task intervals. In Proceedings of ICLP'1994, pages, MIT Press, pp. 369-383.

Castro C, Monfroy E (2004). Designing hybrid cooperations with a component language for solving optimization problems. In Proceedings of AIMSA'2004, volume LNSC, Springer, 3192: 447-458

Focacci F, Laburthe F, Lodi A (2002). Local search and constraint programming. In Handbook of Metaheuristics, volume 57 of International Series in Operations Research and Management Science. Kluwer Academic Publishers.

Hoos H, Stützle T (2004). Stochastic Local Search: Foundations and Applications. Morgan Kaufmann, San Francisco (CA), USA.

Hulubei T, O'Sullivan B (2005). Search heuristics and heavy-tailed behaviour. In Proceedings of CP'2005, LNCS, Springer, 3709: 328-342.

Jussien N, Lhomme O (2002). Local search with constraint propagation and conflict-based heuristics. Artif. Intell., 139(1): 21-45.

Kamarainen O, Sakkout H (2004). Local probing applied to network routing. In Proceedings of CPAIOR'2004, LNCS, Springer, 3011: 173-189.

Mazure B, Sais L, Grégoire E (1998). Boosting complete techniques thanks to local search methods. Annals of Mathematical Artif. Intell., 22(3-4): 319-331.

Michalewicz Z (1998). Genetic Algorithms + Data Structures = Evolution Programs. Springer.

Monfroy E, Saubion F, Lambert T (2005). Hybrid csp solving. In Proceedings of FroCos'2005, LNCS, Springer. Invited, 3717: 138-167.

Niederlinski A (2011). A Quick and Gentle Guide to Constraint Logic Programming via ECLiPSe (also available through http://www.anclp.pl/).

Prestwich S (2000). A hybrid search architecture applied to hard random 3-sat and low-autocorrelation binary sequences. In Proceedings of CP'2000, LNCS, Springer, 1894: 337-352.

Prestwich S (2001). Local search and backtracking vs non-systematic backtracking. In Proceedings of AAAI'2001, Fall Symposium on Using Uncertainty within Computation, pp. 109-115. AAAI Press. Also Technical Report FS-01-04.

Schimpf J, Shen K (2010). ECLiPSe - from LP to CLP. To appear in Theory and Practice of Logic Programming - Special issue on Prolog systems, Preprint ar 14: 1012.4240v1.

Wallace M (2005). Hybrid algorithms, local search, and Eclipse. CP Summer School 05. http://www.math.unipd.it/~frossi/cp-school/wallace-lec_notes.pdf.

Wallace M, Schimpf J (2002). Finding the right hybrid algorithm - a combinatorial meta-problem. Ann. Math. Artif. Intell., 34(4): 259-269.

Wolpert DH, Macreacdy WG (1997). No Free Lunch Theorems for Optimization. IEEE Trans. Evol. Comput. 1(1): 67-87.

Algorithm 1: A Generic SOLVE algorithm.

WHILE not finished
constraint propagation
IF not finished THEN
select an enumeration E
apply E
search

Algorithm 2. A generic local search algorithm.

choose $s \in S$ (an initial configuration)
$opt \leftarrow s$ (record the best configuration)
WHILE not finished DO
choose s' a neighbour of s
$s \leftarrow s'$ (move to s')
$opt \leftarrow s$ if $f(s) \prec f(opt)$

Algorithm 3. A Depth-First Left-First hybrid SOLVE algorithm for finding ONE solution.

```
solve(CSP)
CSP' ← constraint_propagation(CSP)
IF solution(CSP') OR failed(CSP')
THEN RETURN(CSP')
ELSE
Tentative_Val ← LS(CSP')
(Var,Val)← select_enumeration(CSP',VarCrit,ValCrit, Tenta-
tive_Val)
RES ← solve(CSP' ∧ Var=Val)
IF failed(RES)
THEN solve(CSP' ∧ Var≠Val)
ELSE RETURN(RES)
```