International Journal of the Physical Sciences Vol. 6(14), pp. 3369-3376, 18 July, 2011
Available online at http://www.academicjournals.org/IJPS
DOI: 10.5897/IJPS11.530
ISSN 1992 - 1950 ©2011 Academic Journals

Full Length Research Paper

An extensible autonomous search framework for constraint programming

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

[1]Pontificia Universidad Católica de Valparaíso, Chile.
[2]Universidad Técnica Federico Santa María, 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 25 May, 2011

Constraint programming is a modern programming paradigm devoted to solve constraint-based problems, in particular combinatorial problems. In this paradigm, the efficiency on the solving process is the key, which generally depends on the selection of suitable search strategies. However, determining a good search strategy is quite difficult, as its effects on the solving process are hard to predict. A novel solution to handle this concern is called autonomous search, which is a special feature allowing an automatic reconfiguration of the solving process when a poor performance is detected. In this paper, we present an extensible architecture for performing autonomous search in a constraint programming context. The idea is to carry out an "on the fly" replacement of bad-performing strategies by more promising ones. We report encouraging results where the use of autonomous search in the resolution outperforms the use of individual strategies.

Key words: Constraint programming, autonomous search, heuristic search.

INTRODUCTION

Constraint programming (CP) is a modern and powerful programming paradigm devoted to the efficient resolution of constraint-based and optimization problems. It interbreeds ideas from different domains, e.g. from operational research, numerical analysis, artificial intelligence, and programming languages. Currently, CP is widely used in different application areas, for instance, in computer graphics to express geometric coherence, in engineering design for the conception of complex mechanical structures, in database systems to ensure and/or restore data consistency, in electrical engineering to locate faults, and even for sequencing the DNA in molecular biology (Rossi, 2006). Solving a problem in CP requires firstly modeling it as a constraint satisfaction problem (CSP). A CSP is a formal problem representation that mainly consists of a sequence of variables lying in a domain and a set of constraints. The goal is to find a complete variable-value assignment that satisfies the whole set of constraints. The common approach for solving CSPs is based on the generation of a tree data structure that holds the potential solutions by interleaving two main phases: 1) enumeration and 2) propagation.

In the enumeration phase, a variable is instantiated to create a branch of the tree, while the propagation phase is responsible for pruning the tree by filtering from domains the values that do not lead to any solution. In the enumeration phase, there are two major decisions to be made: the order in which the variables and values are selected. This selection is known as the variable and value ordering heuristics, and jointly constitutes the enumeration strategy. Such a pair of decisions is crucial in the performance of the resolution process, where a correct selection can dramatically reduce the computational cost of finding a solution. For instance, if the right value is chosen on the first try for each variable, a solution can be found without performing backtracks. The study of enumeration strategies has been the focus of research during many years. From the 70's, there exist different studies concerning strategies. For instance, preliminary studies were focused on defining general

*Corresponding author. E-mail: ricardo.soto@ucv.cl.

criteria, e.g. the smaller domain for variable selection, and its minimum, maximum, or a random value. There is also work focused on defining strategies for specific class of problems, e.g. for job shop scheduling (Smith and Cheng, 1993; Sadeh and Fox, 1996) and for configuration design (Chenouard et al., 2009). We can also find research focused on determining the best strategy based on some static criterion (Beck et al., 2004a, b; Sturdy, 2003).

However, it turns out that taking an a priori decision is quite difficult, as the effects on the solving process are hard to predict. During the last years there is a trend to analyze the state of progress of the solving process in order to automatically identify good-performing strategies (or a combination of them). For instance, the adaptive constraint engine (ACE) (Epstein et al., 2005) is a framework that learns ordering heuristics by gathering the experience from problem solving processes. The main idea is to manage a set of advisors that recommend in the form of comment a given action to perform e.g. “choose the variable with maximum domain size”. The reliability and utility of advisors is controlled by weights. Those weights are determined by a DWL (digression-based weight learning) algorithm, which learns by examining the solution’s trace of problems successfully solved.

Finally, a decision is computed as a weighted combination of the comments done by the advisors in a process called voting. Another interesting approach following a similar goal is the weighted degree heuristic (Boussemart et al., 2004). The idea is to associate weights to constraints, which are incremented during propagation whenever this causes a domain wipeout. The sum of weights is computed for each variable involved in constraints and the variable with the largest sum is selected. The principles that support the weighted degree procedure can be conceived in terms of an overall strategy that combines two heuristic principles, the fail-first and the contention principle. The fail-first principle says: to succeed, you must first search where you are most likely to fail; while contention principle says: those variables directly related to failure (domain wipeouts) are more likely to cause failure if they are chosen instead of other variables. The random probing method (Grimes and Wallace, 2007; Wallace and Grimes, 2009) address two drawbacks of the weighted degree heuristic. On one hand, the initial choices are made without information on edge weights, and on the other, the weighted degree is biased by the path of the search. This makes the approach too sensitive to local, instead of to global conditions of failure. The random probing method proposes to perform sampling during an initial gathering phase arguing that initial choices are often the most important. Preliminary results demonstrate that random probing performs better than weighted degree heuristic.

The aforementioned approaches are mainly focused on sampling and learning good strategies after solving a problem or a set of problems. In this paper, we focus on reacting as soon as possible instead of waiting the entire resolution process. To this end, we introduce a new framework that smartly combines CP with autonomous search (AS) (Hamadi et al., 2011). The CP component runs a classical solving process while the AS part is responsible for reacting as soon as a bad-performing strategy is detected. Reacting implies to replace “on the fly” the current strategy by another one looking more promising. Promising strategies are selected from a strategy rank which depends on a choice function. The choice function determines the performance of a given strategy in a given amount of time, and it is computed based upon a set of indicators and control parameters. Additionally, to guarantee the precision of the choice function, control parameters (Nannen, 2009) are smoothly adjusted by an optimizer. This framework has been implemented in the ECL^iPS^e Solver (Schimpf and Shen, 2010) and it is supported by a 4-component architecture described later in this paper. Another important capability of this new framework is the possibility of easily updating its components. This is useful for experimentation tasks. Developers are able to add new choice functions, new control parameter optimizers, and/or new ordering heuristics in order to test new CP-AS approaches. The experimental results demonstrate the effectiveness of the proposed framework, outperforming in several cases the use of individual strategies.

BACKGROUND

Here, we formally describe the CSP and we present the basic notions of CSP solving.

Constraint satisfaction problems

Formally, a CSP P is defined by a triple $P = \langle X,D,C \rangle$ where:

-X is an n-tuple of variables $X = \langle x_1, x_2,..., x_n \rangle$.
-D is a corresponding n-tuple of domains $D = \langle D_1, D_2,..., D_n \rangle$ such that $x_i \in D_i$, and D_i is a set of values, for $i = 1,...,n$.
-C is an m-tuple of constraints $C = C_1, C_2, ..., C_m$, and a constraint C_j is defined as a subset of the Cartesian product of domains $D_{j1} \times \cdots \times D_{jnj}$, for $j = 1,..., m$.
A solution to a CSP is an assignment $\{x_1 \rightarrow a_1,..., x_n \rightarrow a_n\}$ such that $a_i \in Di$ for $i = 1,...,n$ and $(a_{j1},...,a_{jnj}) \in C_j$, for $j = 1,...,m$.

CSP Solving

As previously mentioned, the CSP search phase is commonly tackled by building a tree structure by interleaving enumeration and propagation phases. In the enumeration phase, the branches of the tree are created by selecting variables and values from their domains. In the propagation phase, a consistency level is enforced to prune the tree in order to avoid useless tree inspections. Algorithm 1 depicts a general procedure for solving CSPs. The goal is to iteratively generate partial solutions, backtracking when an inconsistency is detected, until a result is reached. The algorithm begins by loading the CSP model.

Then, a while loop encloses a set of actions to be performed until

Table 1. Search process indicators.

Name	Description
VFP	Number of variables fixed by propagation
n	Number of steps or decision points (n increments each time a variable is fixed enumeration)
Tn(S_j)	Number of steps since the last time that an enumeration strategy S_j was used until step nth
SB	Number of Shallow Backtracks
B	Number of Backtracks
In1	Represents a Variation of the Maximum Depth. It is calculated as: CurrentMaxDepth – PreviousMaxDepth
In2	Calculated as: CurrentDepth – PreviousDepth. A positive value means that the current node is deeper than the one explored at the previous step
B-real	Number of backtracks considering also the number of shallow backtracks
d	Current depth in the search tree
Thrash	The solving process alternates enumerations and backtracks on a few number of variables without succeeding in having a strong orientation. It is calculated as: $d_{t-1} - VFP_{t-1}$

Table 2. Variable ordering heuristics.

Name	Description
First (F)	The first variable of the list is selected
Minimum remaining values (MRV)	At each step, the variable with the smallest domain size is selected
Anti minimum remaining values (AMRV)	At each step, the variable with the largest domain size is selected
Occurrence (O)	The variable with the largest number of attached constraints is selected

Table 3. Value ordering heuristics.

Name	Description
In Domain (ID)	It starts with the smallest element and upon backtracking tries successive elements until the entire domain has been explored
In Domain Max (IDM)	It starts the enumeration from the largest value downwards

fixing all the variables (that is assigning a consistent value) or a failure is detected (that is no solution is found). The first two enclosed actions correspond to the variable and value ordering heuristics. The third action is a call to a propagation procedure, which is responsible for attempting to prune the tree. Finally, two conditions are included to perform backtracks. A shallow backtrack (Barták and Rudová, 2005) corresponds to try the next value available from the domain of the current variable, and the backtracking returns to the most recently instantiated variable that has still values to reach a solution.

ARCHITECTURE

Our framework is supported by four components: 1) SOLVE, 2) OBSERVATION, 3) ANALYSIS and 4) UPDATE.

1) The SOLVE component runs a generic CSP solving algorithm performing the aforementioned propagation and enumeration phases.

The enumeration strategies used are taken from the quality rank, which is controlled by the UPDATE component.

2) The OBSERVATION component aims at regarding and recording relevant information about the resolution process. These observations are called snapshots.

3) The ANALYSIS component studies the snapshots taken by the OBSERVATION. It evaluates the different strategies, and provides indicators to the UPDATE component. The indicators as well variable and value ordering heuristics used in this implementation are depicted in Tables 1, 2 and 3, respectively.

4) The UPDATE component makes decisions using the choice function. The choice function determines the performance of a given strategy in a given amount of time. It is calculated based on the indicators given by the ANALYSIS component and a set of control parameters computed by an optimizer.

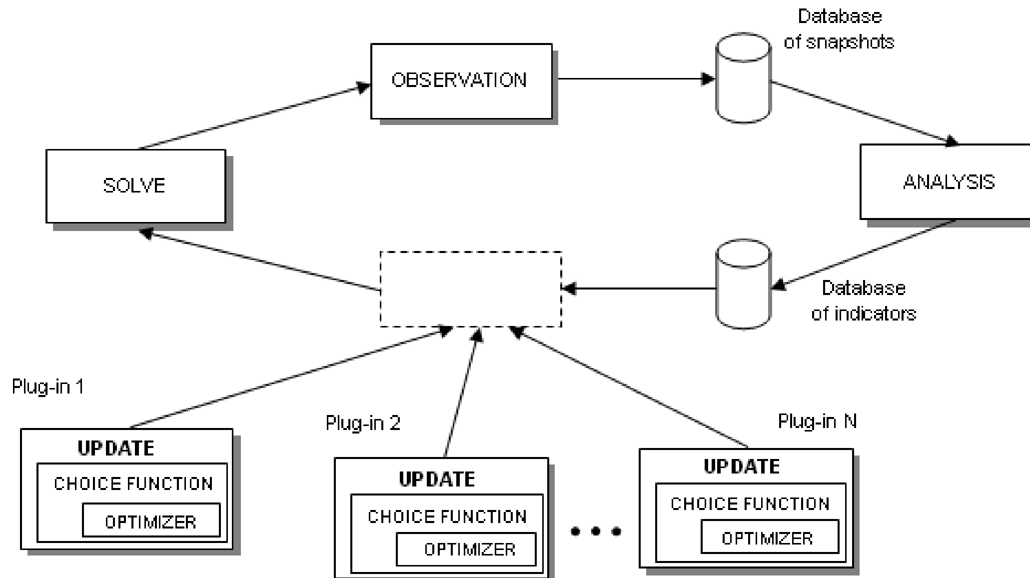

Figure 1. General schema of the architecture.

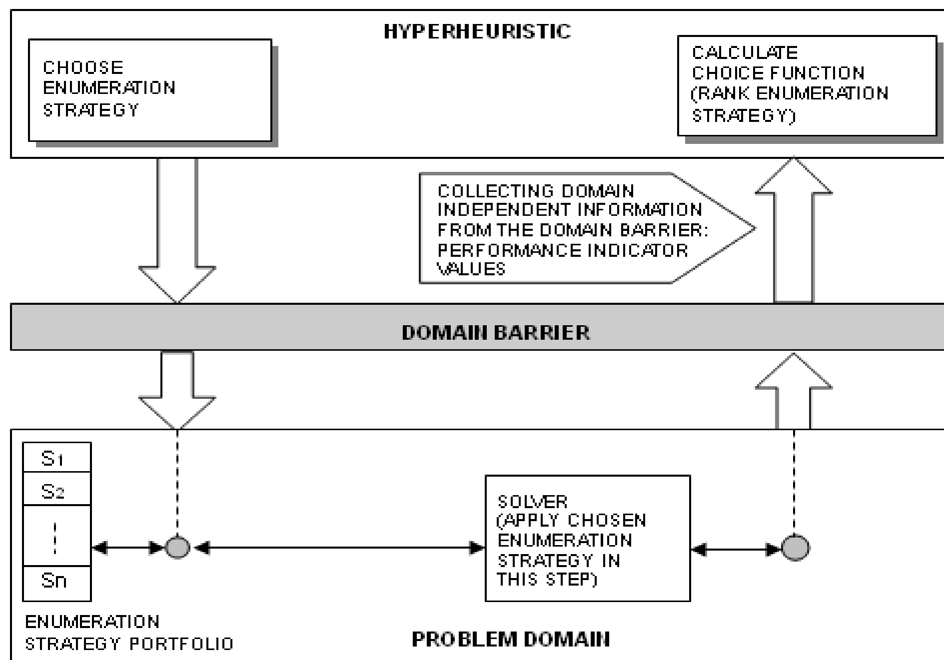

Figure 2. Hyper-heuristic approach for the UPDATE component.

The UPDATE component

The framework has been designed to allow easy modification of the UPDATE component. In fact, UPDATE is the most susceptible component to suffer modifications, since the most obvious experiment –in the context of AS– is to tune or replace the choice function or the optimizer. Figure 1 depicts a general schema of the architecture. The SOLVE, OBSERVATION, and ANALYSIS component have been implemented in ECL^{i}PS^{e}. The UPDATE component has been designed as a plug-in for the framework. Indeed, we have implemented a Java version of the UPDATE component which computes the choice function and optimizes its control parameters through a genetic algorithm. Another version of the UPDATE component, which is currently under implementation, uses a swarm optimizer.

Let us note that the UPDATE component is carefully supported by a hyperheuristic approach (Figure 2). The hyperheuristic is a heuristic that operates at a higher level of abstraction than the CSP

solver (problem domain). The hyperheuristic is responsible for deciding which enumeration strategy to apply at each decision step during the search. It manages the portfolio of enumeration strategies having no prior problem specific knowledge. The hyper-heuristic and the choice function work in conjunction. The choice function provides guidance to the hyperheuristic by indicating which enumeration strategy should be applied next based upon the information of the search process. The choice function is defined as a weighted sum of indicators expressing the recent improvement produced by the enumeration strategy that had been called.

The choice function

The choice function (Soubeiga, 2009) attempts to capture the correspondence between the historical performance of each enumeration strategy and the decision point currently being investigated. Here, a decision point or step is every time the solver is invoked to fix a variable by enumeration. The choice function is used to rank and choose between different enumeration strategies at each step. For any enumeration strategy S_j, the choice function f in step n for S_j is defined by Equation 1, where l is the number of indicators considered and α is a parameter to control the relevance of the indicator within the choice function.

$$f_n(S_j) = \sum_{i=1}^{l} \alpha_i f_{i_n}(S_j) \quad (1)$$

Additionally, to control the relevance of an indicator i for a strategy S_j in a period of time, we use a popular statistical technique for producing smoothed time series called exponential smoothing. The idea is to associate, for some indicators, greater importance to recent performance by exponentially decreasing weights of older observations. In this way, recent observations give relatively more weight than older ones. The exponential smoothing is applied to the computation of f_{i_1} (S_j), which is defined by Equations 2 and 3, where v_1 is the value of the indicator i for the strategy S_j in time 1, n is a given step of the process, β is the smoothing factor, and $0 < \beta < 1$.

$$f_{i_1}(S_j) = v_1 \quad (2)$$

$$f_{i_n}(S_j) = v_n + \beta_i f_{i_{n-1}}(S_j) \quad (3)$$

Let us note that the speed at which the older observations are smoothed (dampened) depends on β. When β is close to 0, dampening is quick and when it is close to 1, dampening is slow. The general solving procedure including AS can be seen in Algorithm 2. Three new function calls have been included at the end: for calculating the indicators, the choice function, and for choosing promising strategies, that is, the ones with highest choice function. They are called after constraint propagation to compute the real effects of the strategy (some indicators may be impacted by the propagation).

The parameter optimizer

As previously mentioned, an optimizer determines the most appropriate set of parameters α_i for the choice function. The parameters are fine-tuned by a genetic algorithm (GA) which trains the choice function carrying out a sampling phase. Sampling occurs during an initial information gathering phase where the search is run repeatedly to a fix cutoff (that is until a fixed number of variables instantiated, visited nodes or backtracks). After sampling, the problem is solved with the most promising set of parameter values for the choice function. The GA evaluates and evolves different combinations of parameters, relieving the task of manual parameterization. Each member of the population encodes the parameters of a choice function. Then, these individuals are used in order to create a choice function instance. Each choice function instantiated (each chromosome) is evaluated in a sampling phase trying to solve partially the problem until a fixed cutoff. As an evaluation value for the chromosome, an indicator of performance process is used (number of backtracks). After each chromosome of the population is evaluated, selection, crossover and mutation are used to breed a new population of choice functions. As noted above, the GA is used to tune the choice function.

A population size of 10 is used and the domain of parameters α_i is [-100, 100]. The crossover operator randomly selects two chromosomes from the population and mates them by randomly picking a gene and then swapping that gene and all subsequent genes between the two chromosomes. The two modified chromosomes are then added to the list of candidate chromosomes. The crossover operator uses a fixed crossover rate; this operation is performed 0.5 as many times as there are chromosomes in the population. The mutation operator runs through the genes in each of the chromosomes in the population and mutates them in statistical accordance to the given mutation rate (0.1). Mutated chromosomes are then added to the list of candidate chromosomes destined for the natural selection process.

EXPERIMENTAL RESULTS

Our implementation has been written in the ECL^{i}PS^e Solver version 5.10. Tests have been performed on a 2.33 GHZ Intel Core2 Duo with 2GB RAM running Windows XP. The stop criterion is 65535 steps for each experiment and the problems used were the following:

-N-queens (NQ) -10 linear equations (10-Equation)
-Magic Squares (MS) -20 linear equations (20-Equation)
-Sudoku -Knight's tour (Knight)

Tables 4 and 5 present the results measured in terms of number of backtracks, Tables 6 and 7 present the results in terms of number of visited nodes, and Tables 8 and 9 show the runtimes. For the evaluations, we consider 8 enumeration strategies (F+ID, AMRV + ID, MRV + ID, O + ID, F + IDM, AMRV + IDM, MRV + IDM, and O + IDM), a random selection, and our autonomous search (AS) approach. Let us note that the portfolio of AS is composed of the same eight strategies mentioned earlier. Results show that the AS approach gain very good position in the global ranking. It is the best in several cases, for instance in almost all instances of the N-Queens problem (n=8, 20, 50, 75), in both instances of the magic squares, in Equation 10, and in the Knight problem (n=6). Considering only backtracks, the AS approach is also the best one for the Sudoku, and considering visited nodes, it takes the second place by a short difference w.r.t. F+IDM. Tables 8 and 9 depict the runtimes for the benchmarks. We include them in order to illustrate the expected overhead of using the choice function.

Table 4. Number of backtracks solving different instances of the N-Queens problem with different strategies.

Strategy	NQ (n=8)	NQ (n=10)	NQ (n=12)	NQ (n=15)	NQ (n=20)	NQ (n=50)	NQ (n=75)
F + ID	10	6	15	73	10026	>27406	>26979
AMRV + ID	11	12	11	808	2539	>39232	>36672
MRV + ID	10	4	16	1	11	177	818
O+ID	10	6	15	73	10026	>26405	>26323
F+IDM	10	6	15	73	10026	>27406	>26979
AMRV + IDM	11	12	11	808	2539	>39232	>36672
MRV+ IDM	10	4	16	1	11	177	818
O+IDM	10	6	15	73	10026	>26405	>26323
Random	5	8	18	98	32	>32718	>32973
AS	4	6	4	73	0	7	74

Table 5. Number of backtracks solving Eq-10, Eq-20, Magic Squares, Sudoku, and the Knight problem with different strategies.

Strategy	Eq-10	Eq-20	MS (n=4)	MS (n=5)	Sudoku	Knight (n=5)	Knight (n=6)
F + ID	3	3	12	910	18	767	>19818
AMRV + ID	5	1	1191	>46675	10439	>42889	>43098
MRV + ID	4	3	3	185	4	767	>19818
O+ID	3	3	10	5231	18	>18838	>19716
F+IDM	10	5	51	>46299	2	767	>19818
AMRV +IDM	8	3	42	>44157	6541	>42889	>43098
MRV+ IDM	3	8	97	>29416	9	767	>19818
O + IDM	10	5	29	>21847	2	>18840	>19716
Random	4	5	17	>39742	250	>40022	>35336
AS	3	3	0	7	2	8190	4105

Table 6. Number of visited nodes solving different instances of the N-Queens problem with different strategies.

Strategy	NQ (n=8)	NQ (n=10)	NQ (n=12)	NQ (n=15)	NQ (n=20)	NQ (n=50)	NQ (n=75)
F + ID	24	19	43	166	23893	>65535	>65535
AMRV + ID	21	25	30	1395	4331	>65535	>65535
MRV + ID	25	16	45	17	51	591	2345
O+ID	25	19	46	169	24308	>65535	>65535
F+IDM	24	19	43	166	23893	>65535	>65535
AMRV + IDM	21	25	30	1395	4331	>65535	>65535
MRV+ IDM	25	16	45	17	51	591	2345
O+IDM	25	19	46	169	24308	>65535	>65535
Random	15	23	41	205	78	>65535	>65535
AS	14	22	18	169	20	66	296

For instance, for smaller instances of the N-queens problem (n=8, 10, 12, 15) as well as for the Sudoku and Magic Squares (n=4) the overhead is nearly 2 s w.r.t. the average runtime, which is around 0 s. We estimate that such an overhead is reasonable, considering the strong work done by the choice function. For harder problems, the overhead begins to be less relevant, for instance for 20-Queens, the AS runtime is 7 s slower than the best runtime, but about 15 s faster than four strategies (F+ID, O+ID, F+IDM, and O+IDM). For the Knight problem (n=5), five of ten strategies solve the problem before the stop criterion, AS is one of them, being only about 5 s slower than the best runtime. For the Magic Squares (n=5), only four strategies are able to solve the problem,

Table 7. Number of visited nodes solving eq-10, eq-20, Magic Squares, Sudoku, and the Knight problem with different strategies.

Strategy	Eq-10	Eq-20	MS (n=4)	MS (n=5)	Sudoku	Knight (n=5)	Knight (n=6)
F + ID	12	11	37	1901	158	3113	>65535
AMRV + ID	14	8	1826	>65535	30139	>65535	>65535
MRV + ID	12	11	22	546	76	3113	>65535
O+ID	12	11	31	13364	196	>65535	>65535
F+IDM	19	13	110	>65535	58	3113	>65535
AMRV + IDM	17	11	69	>65535	19550	>65535	>65535
MRV+ IDM	10	17	230	>65535	153	3113	>65535
O+IDM	19	13	61	>65535	62	>65535	>65535
Random	13	13	47	>65535	1019	>65535	>65535
AS	10	11	16	40	61	28643	16840

Table 8. Runtimes in seconds for different instances of the N-Queens problem with different strategies.

Strategy	NQ (n=8)	NQ (n=10)	NQ (n=12)	NQ (n=15)	NQ (n=20)	NQ (n=50)	NQ (n=75)
F + ID	0	0	0.031	0.109	23.468	t.o.	t.o.
AMRV + ID	0	0	0.015	1.625	8.391	t.o.	t.o.
MRV + ID	0.016	0	0.016	0.015	0.031	1.031	8.562
O+ID	0.016	0	0.015	0.109	23.109	t.o.	t.o.
F+IDM	0	0.015	0.016	0.109	22.922	t.o.	t.o.
AMRV + IDM	0	0.015	0.015	1.609	8.328	t.o.	t.o.
MRV+ IDM	0.016	0.015	0.015	0	0.031	1.031	8.579
O+IDM	0	0	0.016	0.094	22.875	t.o.	t.o.
Random	0.0063	0.083	0.0306	0.3739	5.1619	t.o.	t.o.
AS	1.89	1.89	1.89	2.485	7.875	24.343	49.859

Table 9. Runtimes in seconds for eq-10, eq-20, Magic Squares, Sudoku, and the Knight problem with different strategies.

Strategy	Eq-10	Eq-20	MS (n=4)	MS (n=5)	Sudoku	Knight (n=5)	Knight (n=6)
F + ID	0.219	0.016	0.015	2.437	0.063	2.985	t.o.
AMRV + ID	0.016	0.016	3.781	t.o.	34.735	t.o.	t.o.
MRV + ID	0.016	0.015	0.015	0.516	0.016	2.61	t.o.
O+ID	0.016	0.031	0.046	9.344	0.11	t.o.	t.o.
F+IDM	0.031	0.031	0.141	t.o.	0.015	2.578	t.o.
AMRV + IDM	0.015	0.032	0.062	t.o.	22.953	t.o.	t.o.
MRV+ IDM	0	0.031	0.156	t.o.	0.063	2.594	t.o.
O+IDM	0.031	0.031	0.063	t.o.	0.015	t.o.	t.o.
Random	0.021	0.021	0.0568	t.o.	0.3041	t.o.	t.o.
AS	3.687	5.375	2.203	2.875	18.328	7.422	114.906

being the AS approach the third best runtime. For 50-Queens and 75-Queens,

AS is one of the only three strategies that solve the problem before the stop criterion. Finally, AS is the unique strategy that solves the Knight problem with n=6, and as a consequence the only one that solves the complete set of problems.

CONCLUSION

In this work, we have presented an extensible AS

framework for CP. Based on a set of indicators; our approach measures the resolution process state to allow the replacement of strategies exhibiting poor performances. A main element of the architecture is the choice function, which is responsible for determining the quality of strategies. The choice function is calculated based upon a set of indicators and control parameters, while the adjustment of parameters is handled by a genetic algorithm.

We have applied our approach to solve different CSPs, the results demonstrate that in several cases the dynamic selection outperforms the use of classic enumeration strategies. The framework introduced here is ongoing work, and we believe there is a considerable scope for future work, for instance, the addition of new combination of enumeration strategies, analysis of the control parameters, as well as the study of new statistical methods for improving the choice function.

REFERENCES

Barták R, Rudová H (2005). Limited assignments: A new cutoff strategy for incomplete depth-first search. In Proceedings of the 20th ACM Symposium on Appl. Comput., (SAC), pp. 388–392.

Beck JC, Prosser P, Wallace RJ (2004a). Trying again to fail-first. In Workshop on Constraint Solving and Constraint Logic Programming (CSCLP), volume 3419 of Lecture Notes in Comput. Sci., pp. 41–55. Springer.

Beck JC, Prosser P, Wallace RJ (2004b). Variable ordering heuristics show promise. In Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP), volume 3258 of Lecture Notes in Comput. Sci., pp. 711–715.

Boussemart F, Hemery F, Lecoutre C, Sais L (2004). Boosting systematic search by weighting constraints. In Proceedings of the 16th Eureopean Conf. Artif. Intell., (ECAI), 146–150. IOS Press, 2004.

Chenouard R, Granvilliers L, Sebastian P (2009). Search heuristics for constraint- aided embodiment design. AI EDAM. 23(2):175–195.

Epstein SL, Freuder E, Wallace RJ (2005). Learning to support Constraint Programmers. Computa. Intell., 21(4): 336-371.

Grimes D, Wallace RJ (2007). Learning to identify global bottlenecks in constraint satisfaction search. In Proceedings of the Twentieth Int. Florida Artif. Intell. Res. Society (FLAIRS) Conference, pp. 592–597. AAAI Press.

Hamadi Y, Monfroy E, Saubion F (2011). Autonomous Search. Springer, 2011. To appear.

Nannen V, Smit SK, Eiben AE (2008), Costs and Benefits of Tuning Parameters of Evolutionary Algorithms. In Proceedings of the 10th Conference on Parallel Problem Solving from Nature (PPSN), volume 5199 of Lecture Notes in Comput. Sci., pp. 528-538. Springer.

Rossi F (2006). Handbook of Constraint Programming. Elsevier, 2006.

Sadeh NM, Fox MS (1996). Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. Artif. Intell., 86(1):1–41.

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 arXiv:1012.4240v1.

Smith SF, Cheng C (1993). Slack-based heuristics for constraint satisfaction scheduling. In Proceedings of the Eleventh National Conference on Artif. Intell. (AAAI), pp. 139–144.

Soubeiga E (2009). Development and Application of Hyperheuristics to Personnel Scheduling. PhD thesis, University of Nottingham Sch. Comput. Sci.,

Sturdy P (2003). Learning good variable orderings. In Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming, volume 2833 of Lecture Notes in Comput. Sci., 997. Springer.

Wallace RJ, Grimes D (2008). Experimental studies of variable selection strategies based on constraint weights. J. Algorithms, 63(1-3):114–129.