Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 580785, 12 pages
http://dx.doi.org/10.1155/2015/580785

Research Article

Top-k Based Adaptive Enumeration in Constraint Programming

Ricardo Soto,[1,2,3] Broderick Crawford,[1,4,5] Wenceslao Palma,[1] Eric Monfroy,[6] Rodrigo Olivares,[7] Carlos Castro,[8] and Fernando Paredes[9]

[1]*Pontificia Universidad Católica de Valparaíso, 2362807 Valparaíso, Chile*
[2]*Universidad Autónoma de Chile, 7500138 Santiago, Chile*
[3]*Universidad Central de Chile, 8370178 Santiago, Chile*
[4]*Universidad Finis Terrae, 7501015 Santiago, Chile*
[5]*Facultad de Ingeniería y Tecnología, Universidad San Sebastián, 8420524 Santiago, Chile*
[6]*CNRS, LINA, University of Nantes, 44322 Nantes, France*
[7]*Universidad de Valparaíso, 2362735 Valparaíso, Chile*
[8]*Universidad Técnica Federico Santa María, 2390123 Valparaíso, Chile*
[9]*Escuela de Ingeniería Industrial, Universidad Diego Portales, 8370109 Santiago, Chile*

Correspondence should be addressed to Ricardo Soto; ricardo.soto@ucv.cl

Received 23 October 2014; Accepted 3 February 2015

Academic Editor: Youqing Wang

Constraint programming effectively solves constraint satisfaction and optimization problems by basically building, pruning, and exploring a search tree of potential solutions. In this context, a main component is the enumeration strategy, which is responsible for selecting the order in which variables and values are selected to build a possible solution. This process is known to be quite important; indeed a correct selection can reach a solution without failed explorations. However, it is well known that selecting the right strategy is quite challenging as their performance is notably hard to predict. During the last years, adaptive enumeration appeared as a proper solution to this problem. Adaptive enumeration allows the solving algorithm being able to autonomously modifying its strategies in solving time depending on performance information. In this way, the most suitable order for variables and values is employed along the search. In this paper, we present a new and more lightweight approach for performing adaptive enumeration. We incorporate a powerful classification technique named Top-k in order to adaptively select strategies along the resolution. We report results on a set of well-known benchmarks where the proposed approach noticeably competes with classical and modern adaptive enumeration methods for constraint satisfaction.

1. Introduction

Constraint programming (CP) is a modern and efficient programming paradigm for solving complex constraint satisfaction and optimization problems. It is commonly employed for solving real-life problems in various application domains; practical examples can be seen in rostering [1], scheduling [2], manufacturing [3], engineering [4], and bioinformatics [5], among others. Under this paradigm, problems are seen as constraint networks, which consist in a sequence of variables owning a domain of values and a set of constraints imposing conditions that those variables must satisfy. A solution is then a set of values for the variables that satisfies the whole set of constraints. Constraint networks, also named constraint satisfaction problems (CSPs), are usually solved by a backtracking-based algorithm that explores a search tree where the potential solution is distributed. The exploration proceeds by combining two main phases: enumeration and propagation. Enumeration is the process of assigning values to variables in order to generate partial solutions, while the propagation attempts to remove from the constraint network the conflicting values that do not lead to any feasible solution.

The enumeration process involves two actions. Firstly, a variable from the constraint network is selected and then a value is assigned to this variable. These actions are handled by the variable and value ordering heuristics, which together

form the enumeration strategy. Enumeration is a crucial phase in constraint satisfaction that may conduct to very different solving processes depending on the strategy employed. Indeed, by selecting the correct strategy, a solution can be reached without performing backtracks and as a consequence requiring a minor solving time. However, a main concern in this context is the difficulty of a priori selecting the correct strategy as their performance is notably hard to predict.

During the last years, various approaches have been proposed to tackle this concern; most of them are based on performing adaptive enumeration during the search process. The idea is to enable the search algorithm to control and automatically choose its own ordering heuristics based on information collected during the solving process. A preliminary framework is proposed in [6] which allows the solver to choose and replace its heuristics during solving time by using priorities. The idea is to penalize bad performing strategies while the efficient ones receive more credits. The quality of strategies is evaluated by using performance indicators of the search process. A more modern and efficient approach is presented in [7], where a hyperheuristic is proposed to control the enumeration. A hyperheuristic [8, 9] can be seen as a method to choose heuristics [10]; in particular a meta-heuristic is used on the top of the hyperheuristic in order to select and adapt the ordering heuristics. Although this hyperheuristic approach has illustrated promising results [11–13], the top part is quite expensive adding a significant overhead to the whole resolution of the constraint satisfaction problem.

In this paper, we propose a novel and more lightweight framework for adaptive enumeration. We follow a different approach avoiding the use of hyperheuristics by incorporating a simple and efficient classification technique named Top-k (see Figure 1) in order to rapidly evaluate and rank the strategies along the resolution. We illustrate encouraging results where our approach noticeably competes with classical and modern adaptive enumeration methods for constraint satisfaction.

The rest of this paper is organized as follows. The related work is given in Section 2 followed by an overview of constraint programming and associated concepts. The new Top-k based approach for adaptive enumeration is illustrated in Section 4. In Section 5, we present the experiments and the analysis of results. Finally, we give conclusions and some future research directions.

2. Related Work

In constraint solving, the enumeration strategy is composed of the pair variable and value ordering heuristics. The enumeration is adaptive when the solver is able to control and automatically choose its own ordering heuristics. We can distinguish two dimensions depending on the sources of feedback information: online and offline methods. The feedback, when used, corresponds to the information that is learned while solving (online) or while using a set of training instances (offline). In the context of offline methods, preliminary approaches proposed to sample and learn good

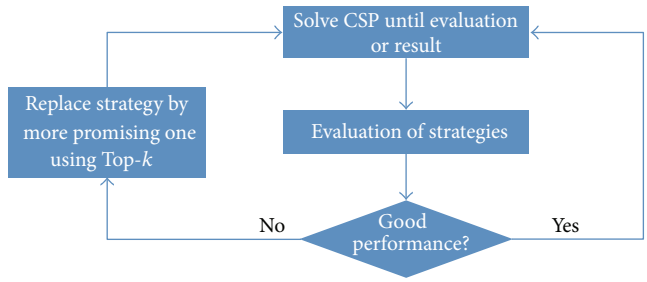


Figure 1: General flowchart of the adaptive enumeration based on Top-k.

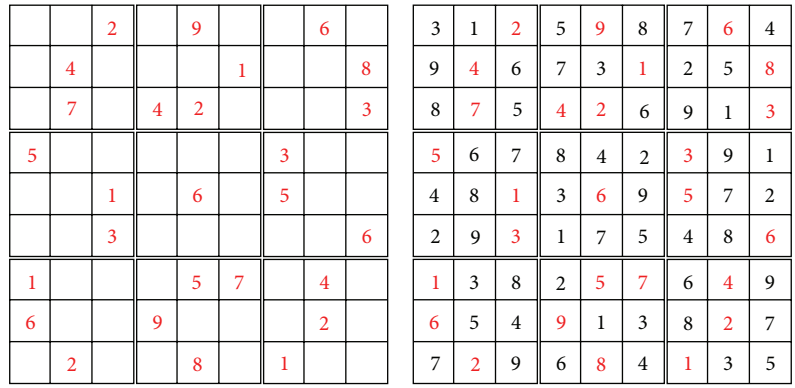
strategies after solving a problem. For instance in [14–16] learning algorithms are used to analyze the resolution process of problems successfully solved. This information is then used by advisors that are able to recommend good heuristics. Another interesting approach is suggested in [17], which proposes to select the variable with the largest weighted degree. The weighted degree is computed by associating weights to constraints which are incremented once they lead to the deletion of a variable's domain. Then, the weight degree of a variable corresponds to the sum of weights of constraints where they participate. This idea follows the contention principle, which states that variables directly related to conflicts are more likely to cause failures. A variation of the weighted degree strategy, known as random probing, is reported in [18, 19], which incorporates sampling during an initial gathering phase arguing that initial choices are often the most important.

Following an online process, a pioneer work is the one presented in [6]. This framework introduced a four-component architecture, allowing the dynamic replacement of enumeration strategies. The strategies are evaluated via performance indicators of the search process, and better evaluated strategies replace worse ones during solving time. Such a pioneer framework was used as basis of different related works. For instance, a more modern approach based on this idea is reported in [7]. This approach employs a two-layered framework where a hyperheuristic placed on the top-layer controls the dynamic selection of enumeration strategies of the solver placed on the lower-layer. A hyperheuristic can be regarded as a method to choose heuristics [10]. In this approach, two different top-layers have been proposed, one using a genetic algorithm [11, 12] and another using a particle swarm optimizer [13]. Similar approaches have also been implemented for solving optimization problems instead of pure CSPs [21].

3. Preliminaries

In this section, we briefly survey the basic concepts associated with constraint programming.

Definition 1 (constraint). A constraint c is a relation defined on a sequence of variables $X(c) = (x_{i_1}, \ldots, x_{i_{|X(c)|}})$, called the scheme of c. c is the subset of $\mathbb{Z}^{|X(c)|}$ that contains



		2		9			6	
	4				1			8
	7		4	2				3
5						3		
		1		6		5		
		3						6
1				5	7		4	
6			9				2	
	2			8		1		

3	1	2	5	9	8	7	6	4
9	4	6	7	3	1	2	5	8
8	7	5	4	2	6	9	1	3
5	6	7	8	4	2	3	9	1
4	8	1	3	6	9	5	7	2
2	9	3	1	7	5	4	8	6
1	3	8	2	5	7	6	4	9
6	5	4	9	1	3	8	2	7
7	2	9	6	8	4	1	3	5

Figure 2: An instance and the corresponding solution to a Sudoku puzzle.

the combinations of tuples $\tau \in \mathbb{Z}^{|X(c)|}$ that satisfy c. $|X(c)|$ is called the arity of c. A constraint c with scheme $X(c) = (x_1, \ldots, x_k)$ is also noted as $c(x_1, \ldots, x_k)$.

Definition 2 (constraint network). A constraint network also known as constraint satisfaction problem is defined by a triple $N = \langle X, D, C \rangle$, where we have the following.

(i) X is a n-tuple of variables of integer variables $X = (x_1, \ldots, x_n)$.

(ii) D is the corresponding n-tuple of domains for X, that is, $D = D(x_1) \times \cdots \times D(x_n)$, where $D(x_i) \subset \mathbb{Z}$ is the finite set of values that variable x_i can take.

(iii) C is an m-tuple of constraints $C = \{c_1, \ldots, c_e\}$, where variables in $X(c_j)$ are in X.

Definition 3 (instantiation). Given a network $N = \langle X, D, C \rangle$, we have the following.

(i) An instantiation A on $X' = (x_1, \ldots, x_k) \subseteq X$ is an assignment of values $(a_1, \ldots, a_k)$ to variables $x_1, \ldots, x_k$.

(ii) An instantiation B on X' is valid if $\forall x_i \in X'$, $A[x_i] \in D(x_i)$, where $A[x_i]$ denotes the value of A assigned to x_i.

Definition 4 (solution). Given a network $N = \langle X, D, C \rangle$, we have the following.

(i) A solution to a network N is an instantiation A on $X = (x_1, \ldots, x_n)$ iff it is valid and satisfies all the constraints. A solution is denoted as sol and SOL(N) corresponds to the set of solutions of N.

Example 5 (the Sudoku puzzle). As an example, let us consider the Sudoku problem as a constraint network. The problem consists in filling a 9×9 grid with prefilled cells, divided into nine 3×3 regions, so that each column, row, and region contain different digits from 1 to 9. Let $N = \langle X, D, C \rangle$ be the constraint network, which is composed of the following.

(i) $X = (x_{1,1}, \ldots, x_{n,m})$ is the sequence of variables, and $x_{i,j} \in X$ identifies the cell placed in the ith row and jth column of the Sudoku puzzle, for $i = 1, \ldots, n$ and $i = 1, \ldots, m$.

(ii) D is the corresponding set of domains, where $D(x_{ij}) \in D$ is the domain of the variable x_{ij};

(iii) C is the set of constraints defined as follows.

(a) To guarantee that values differ in rows and columns:

$$x_{k,i} \neq x_{k,j} \wedge x_{i,k} \neq x_{j,k}, \quad \forall \left(k \in [1,9],\ i \in [1,9],\ j \in [i+1,9]\right). \tag{1}$$

(b) To guarantee that values differ in each subregion:

$$x_{(k1-1)*3+k2,(j1-1)*3+j2} \neq x_{(k1-1)*3+k3,(j1-1)*3+j3}, \quad \forall \left(k1, j1, k2, j2, k3, j3 \in [1,3] \mid k2 \neq k3 \wedge j2 \neq j3\right). \tag{2}$$

An instance and the corresponding solution to a Sudoku puzzle can be found in Figure 2.

3.1. Constraint Solving. Constraint solving is usually handled by building a search tree composed of potential solutions. Those potential solutions are then verified if they can lead to a feasible solution. If not, the search tree is pruned by deleting from domain those unfeasible values. The search process is commonly divided in two phases: enumeration and propagation. The enumeration can be seen as a sorting process [22, 23] where the potential solutions are incrementally constructed by assigning values to variables. The propagation is the phase responsible for pruning the tree.

A general procedure for constraint solving is illustrated in Algorithm 1. As input, it receives the constraint network N and the enumeration strategy s to be employed. The output is a solution sol $\in$ SOL(N) to the constraint network or a failure; that is, no solution has been found. The process begins by initiating a while loop which iterates over a set of actions until a solution or a failure is found. The first action corresponds to selecting a variable from the set X to then associate it with a value from its domain according to s. The propagation

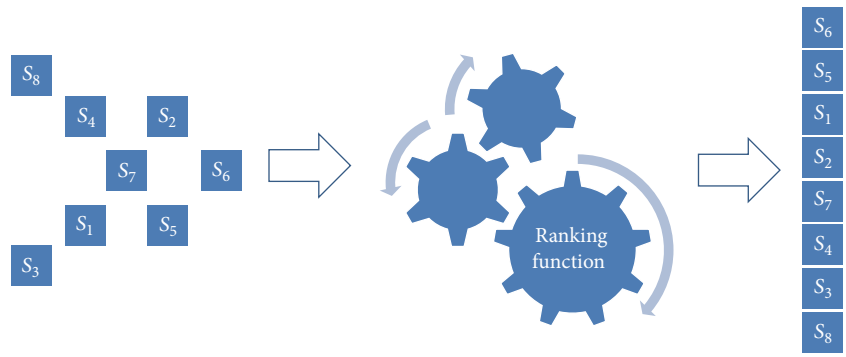


Figure 3: Ranking the portfolio of enumeration strategies (S_1 to S_8).

Input: $N = (X, D, C), s$
Output: $sol \in SOL(N)$ *or a failure*
(1) **While not** *success* **or** *failure* **do**
(2) *select a variable x from X according to s*
(3) *select a value from D(x) according to s*
(4) *propagate*
(5) **If** *DWO in future var* **then**
(6) *shallow backtrack*
(7) **End If**
(8) **If** *DWO in current var* **then**
(9) *backtrack*
(10) **End If**
(11) **End While**

Algorithm 1: Constraint solving.

phase is next called in order to delete from domains the values that will not lead to any solution considering the current instantiation. This phase may empty the domain of a given variable, which is known as a domain wipe-out (DWO). If a DWO happens in a future variable, a shallow backtrack assigns the following available value to the current variable. Finally, if a DWO occurs in the current variable, the backtrack jumps back to the previous instantiated variable that is still able to reach a solution.

4. Adaptive Enumeration via Top-k

In this section, we present the framework for adaptive enumeration via the Top-k approach. The idea is to evaluate, via performance indicators, and then to rank a portfolio of enumeration strategies in running time in order to use the most appropriate one at each step of the process (see Figure 3). Our goal here is to avoid the use of hyperheuristics to provide a faster process while maintaining the quality of solutions. In the following, we briefly describe the Top-k approach and its incorporation to the adaptive enumeration framework.

4.1. Top-k. In database systems, a Top-k query [24] aims at retrieving the k tuples that better satisfies a given set of preferences. This is a widely used technique as selecting data according to user requirements is largely required in different real-life domains such as the Web, multimedia search, distributed systems, and multicriteria decision making. A common way to identify the Top-k objects is to provide a score to each object according to its relevance within the tuple and then to compute a scoring function for all involved objects. There exist different techniques to processing Top-k queries which are mainly related to database access, design, and implementation aspects (see [24] for a detailed presentation). We adopt the monotone ranking function approach which is the most appropriate one for our purposes, as it can straightforwardly be mapped to a scoring function for evaluating performance. Another interesting feature is the ability to upper-bound objects scores. This allows one to early prune certain objects without exactly knowing their scores, alleviating the function computation, and as consequence accelerating the whole solving process. A monotone ranking function is defined as follows.

Definition 6 (monotone ranking function). A function F, defined on predicates $p_1, \ldots, p_n$, is monotone if $F(p_1, \ldots, p_n) \leq F(p_1', \ldots, p_n')$ whenever $p_i \leq p_i'$ for every i.

Then, assuming that an indicator i of the search process can be seen as a predicate p_i, a scoring function F for evaluating the performance of enumeration strategies by using n indicators can be generically defined as

$$F = i_1 + \cdots + i_n, \tag{3}$$

where i_j with $j \in 1 \cdots n$ is an indicator that measures the performance of the strategy in a given amount of time. However, according to [12], indicators do not have the same relevance for evaluating a solving process; a weight is therefore associated with each indicator to balance their effect on the scoring function. Finally, the general scoring function for a given enumeration strategy $s_l \in S$ is defined as

$$F^{s_l} = w_1 i_1^{s_l} + \cdots + w_n i_n^{s_l}, \tag{4}$$

where w_j is the weight associated with indicator $i_j^{s_l}$, which in turn provides a score for the strategy s_l, with respect to a given performance criterion.

Input: $N = (X, D, C), S, I$
Output: $sol \in SOL(N)$ *or a failure*
(1) *set up portfolio*
(2) *select* $s_l \in S$ *using* Top-k
(3) **While not** *success* **or** *failure* **do**
(4) *select a variable* x *from* X *according to* s_l
(5) *select a value from* $D(x)$ *according to* s_l
(6) *propagate*
(7) **If** *DWO in future var* **then**
(8) *shallow backtrack*
(9) **End If**
(10) **If** *DWO in current var* **then**
(11) *backtrack*
(12) **End If**
(13) $\forall s_l \in S, \forall i \in I$ *get scores*
(14) $\forall s_l \in S$ *compute scoring function*
(15) *select* s_l *using* Top-k
(16) **End While**

Algorithm 2: Adaptive enumeration via Top-k.

Definition 7 (Top-k result set). Let S be the set of enumeration strategies used in the adaptive enumeration framework. The Top-k result denoted by $\text{Top}^k(S)$ is a sorted set on the score, in increasing order, such that

(1) $\text{Top}^k(S) \subseteq S$;

(2) if $\|S\| < k$, $\text{Top}^k(S) = S$; otherwise $\|\text{Top}^k(S)\| = k$;

(3) $\forall s \in \text{Top}^k(S), \forall s' \in S \setminus \text{Top}^k(S), F^s \leq F^{s'}$.

Definition 8 (result's rank). Given a Top-k result set R, the rank of result $r \in R$ is the position of r in the set R.

Because our approach does not deal with hard disks issues and the processing of large volumes of data in main memory we use the most straightforward way to compute the Top-k result set; that is, all the strategies are sequentially scanned and the score of each of them is calculated and placed in a sorted list. Thus, the adaptive enumeration framework selects the best enumeration strategy in order to continue with the solving process.

Algorithm 2 depicts from a high-level standpoint the integration of the Top-k approach in a classic constraint solving algorithm. The algorithm receives as input the constraint network N, the set of strategies denoted by S, and the set of indicators denoted by I; the output is a solution sol $\in$ SOL(N) to the constraint network or a failure. The process begins by setting up the portfolio and the indicators of the solving process and a cutoff value, which is usually a given number of steps (other stop cutoff values can be used such as the percentage or number of fixed variables, number of visited nodes, or number of backtracks). Then, the constraint network is attempted to be solved until the cutoff, and the best strategy is selected by using Top-k. Once the best s_l is selected, the constraint network is solved and at every step, the indicators provide the corresponding score, which are used to compute the scoring function. In this way, the best s_l is employed next along the resolution process. Finally, if a solution sol $\in$ SOL(N) exists, it is reported.

Table 1: Portfolio used.

Strategy	Variable ordering	Value ordering
S_1	First variable of the list	min. value in domain
S_2	The variable with the largest domain	min. value in domain
S_3	The variable with the smallest domain	min. value in domain
S_4	The variable with the largest number of attached constraints	min. value in domain
S_5	First variable of the list	max. value in domain
S_6	The variable with the largest domain	max. value in domain
S_7	The variable with the smallest domain	max. value in domain
S_8	The variable with the largest number of attached constraints	max. value in domain

5. Experimental Evaluation

This section illustrates the experimental results of the proposed approach. The Top-k adaptive enumeration has been implemented on the Ecl^{i}ps^e Constraint Logic Programming Solver v5.10 interfaced with Java. The experiments have been launched on a 3.3 GHz Intel Core i3 with 8 Gb RAM running Ubuntu. We test our approach using different instances of the following classic benchmarks:

(i) the n-queens problem with $n = \{8, 10, 12, 15, 20, 50\}$,

(ii) the magic squares with size $= \{3, 4, 5\}$,

(iii) the Sudoku puzzle,

(iv) the knight tour with size $= \{5, 6\}$.

We provide in the Appendix Section the encodings of the tested benchmarks. As in previous works [4], we use 65535 steps as the stop criterion; problems having no solution at this point are set to t.o. (time out). Once the problem is launched, the step begins from 0 and it is incremented by 1 each time a variable is instantiated by enumeration. The adaptive enumeration uses a portfolio of eight strategies, which are described in Table 1. Indicators and weights employed are described in Table 2. Weights have been collected from experience and tuning phases done in previous works [12, 13] in order to correctly represent the relevance that each one has on the scoring function.

We compare the adaptive enumeration based on Top-k with the two last reported online methods based on hyperheuristics, one controlled by a genetic algorithm (HH-GA) [25] and the other one by particle swarm optimization

Table 2: Search process indicators.

Name	Weight	Description
VFP	50	Number of variables fixed by propagation.
$T_{st}(s_k)$	10	Number of steps since the last time that an enumeration strategy s_k was used until step stth.
SB	−10	Number of shallow backtracks [20]
B	−20	Number of backtracks.
In1	10	Represents a variation of the maximum depth. It is calculated as *Current Maximum Depth – Previous Maximum Depth*
In2	10	Calculated as *Current Depth – Previous Depth*. A positive value means that the current node is deeper than the one explored at the previous step.
Thrash	−20	The solving process alternates enumerations and backtracks on a small number of variables without succeeding in having a strong orientation. It is calculated as $d_{t-1} - \text{VFP}_{t-1}$, where d_t is the current depth in the search in a given time t.

Table 3: Runtime in ms 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 ($n = 50$)
S_1	12	13	20	86	21,027	t.o.
S_2	8	35	30	815	6,315	t.o.
S_3	7	10	25	4	17	962
S_4	15	11	23	54	21,945	t.o.
S_5	17	9	21	56	21,319	t.o.
S_6	12	12	17	778	5,561	t.o.
S_7	9	6	19	3	18	987
S_8	11	6	18	47	21,297	t.o.
HH-GA	151	124	285	215	2,358	21,009
HH-PSO	113	123	201	145	1,068	20,340
Top-k	5	4	7	18	110	2,845

Table 4: Runtime in ms for magic squares, Sudoku, and the knight problem with different strategies.

Strategy	MS ($n = 4$)	MS ($n = 5$)	Sudoku	Knight ($n = 5$)	Knight ($n = 6$)
S_1	21	1,216	8	1,321	t.o.
S_2	2,093	t.o.	3,644	t.o.	t.o.
S_3	5	318	7	1,352	t.o.
S_4	14	7,505	7	t.o.	t.o.
S_5	67	t.o.	11	1,326	t.o.
S_6	36	t.o.	2,127	t.o.	t.o.
S_7	88	t.o.	18	1,751	t.o.
S_8	31	t.o.	17	t.o.	t.o.
HH-GA	158	1,591	1,631	2,615	12,928
HH-PSO	327	1,868,453	234	85,574	11,547
Top-k	7	857	6,349	2,553	8,931

(HH-PSO) [13]. The portfolio used by hyperheuristics is the same as the one of the Top-k approach. We also include in the comparison the results of using a single strategy during the complete solving process (S_1 to S_8). Tables 3 and 4 report solving times needed to reach a solution. The results illustrate that Top-k is able to outperform both hyperheuristics and single strategies for small instances of the n-queens. Top-k is about 20 times faster than HH-PSO and about 30 times faster than HH-GA for $n = 8$; this difference is increased for $n = 10$ and $n = 12$, being about 7 times faster than

Table 5: 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 ($n = 50$)
S_1	10	6	15	73	10,026	>27,406
S_2	11	12	11	808	2,539	>39,232
S_3	10	4	16	1	11	177
S_4	10	6	15	73	10,026	>26,405
S_5	10	6	15	73	10,026	>27,406
S_6	11	12	11	808	2,539	>39,232
S_7	10	4	16	1	11	177
S_8	10	6	15	73	10,026	>26,405
HH-GA	4	6	4	73	0	7
HH-PSO	6	2	16	11	11	252
Top-k	5	2	2	13	18	212

Table 6: Number of backtracks solving magic squares, Sudoku, and the knight problem with different strategies.

Strategy	MS ($n = 4$)	MS ($n = 5$)	Sudoku	Knight ($n = 5$)	Knight ($n = 6$)
S_1	12	910	18	767	>19,818
S_2	1,191	>46,675	10439	>42,889	>43,098
S_3	3	185	4	767	>19,818
S_4	10	5,231	18	>18,838	>19,716
S_5	51	>46,299	2	767	>19,818
S_6	42	>44,157	6,541	>42,889	>43,098
S_7	3	>29,416	9	767	>19,818
S_8	29	>21,847	2	>18,840	>19,716
HH-GA	0	7	2	8,190	4,105
HH-PSO	4	53	826	667	29,608
Top-k	1	207	2,744	613	6,835

hyperheuristics for the biggest tested instance of n-queens. For the Sudoku puzzle, Top-k is not able to improve previous performance, but for magic squares and knight's tour, Top-k performs notably better than hyperheuristics, remaining competitive with respect to single strategies. Indeed, Top-k takes the first place for knight's tour ($n = 6$) when single strategies are not able to solve it before stop criterion.

These results evidence the ability of Top-k to dynamically adapt the search in order to use the best strategy for each part of the search tree, allowing producing better results than most of single strategies. Certainly, selecting the correct variable and value ordering for the different regions of the search space can have a dramatic effect on the performance of the backtracking algorithm with huge variances on solving performance [12, 13, 17, 26–28]. This is explained by the fact that selecting the correct strategy to the correct part of the search tree should lead to more promising instantiations (variable-value assignments conducting to solutions) than failures (variable-value assignments not conducting to any solution), reducing as a consequence the number of backtracks and runtime.

We also consider in the experimental evaluation the number of backtracks reported to reach a solution as a performance indicator (see Tables 5 and 6), since it allows one to measure how many times the process fails in instantiating a variable. Here, we illustrate that in adaptive enumeration the number of backtracks is not exactly related to solving time. Indeed, the Top-k approach can fail more to reach a result but obtain minor solving times than hyperheuristics (see, e.g., n-queens with $n = 20$ and $n = 50$; magic squares with size $= 5$). This is explained by the fact that the Top-k is a more lightweight component than the optimizer on top of hyperheuristics, which is able to find good heuristic configurations but add a noticeable overhead to the whole solving process. Indeed, the Top-k algorithm used runs in (n^2) while the optimizers used in HH-GA and HH-PSO run in exponential time. A graphical comparison of adaptive enumeration approaches in terms of runtime and backtracks can be seen in Figures 4 and 5, respectively.

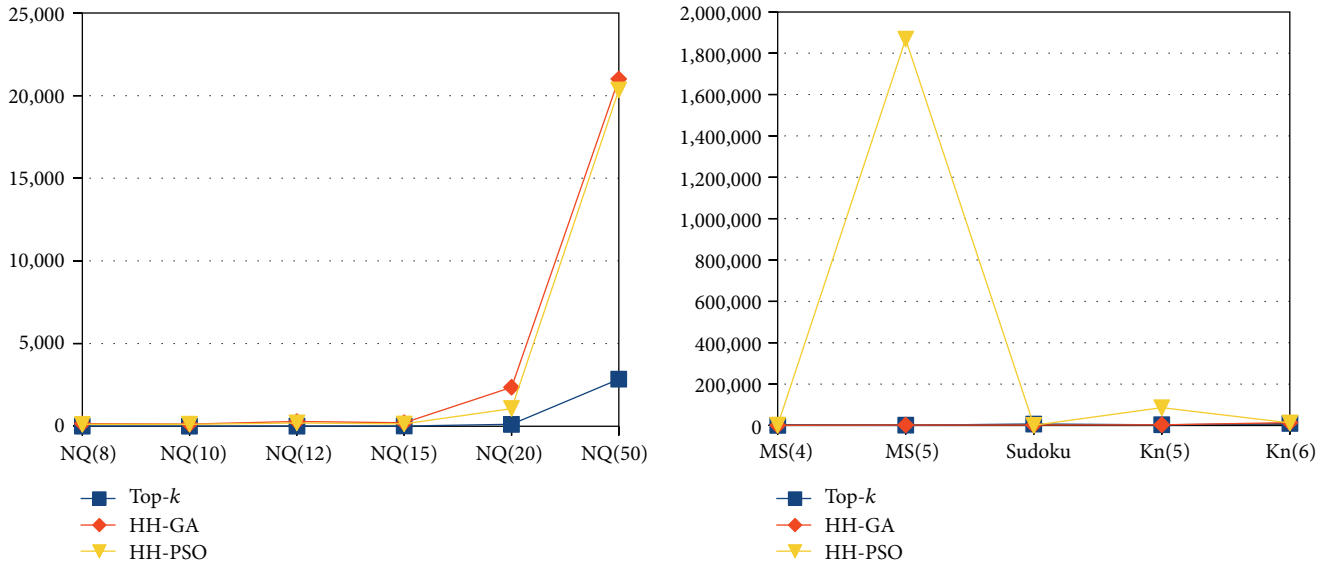


Figure 4: Comparing runtimes of adaptive approaches.

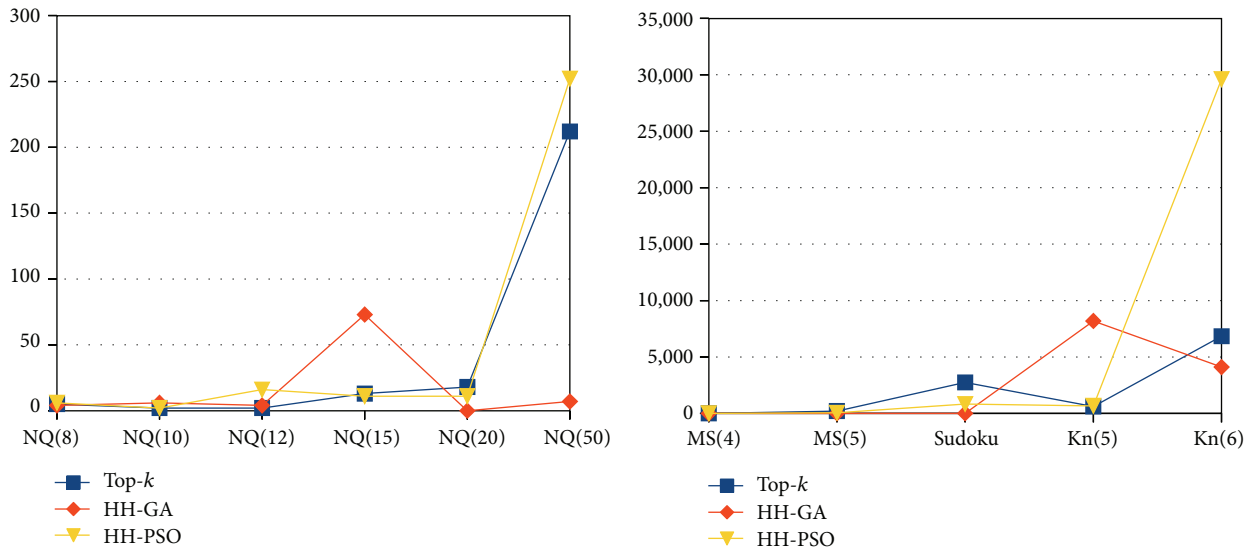


Figure 5: Comparing backtracks of adaptive approaches.

6. Conclusions

In this paper, we have presented a new and more lightweight framework for adaptive enumeration. The idea is to reduce the cost of associated hyperheuristics presented in previous approaches so as to alleviate the whole work obtaining faster solving processes. To this end, we have incorporated a simple and efficient classification technique named Top-k in order to rapidly evaluate and rank the strategies along the resolution. We have performed a set of experiments on different instances of classic benchmarks where the Top-k based adaptive enumeration is able to outperform in several cases the last adaptive enumeration methods reported in the literature while keeping the quality of solutions. Indeed, considerable runtime reductions are achieved in several instances of the tested problems.

We visualize different directions for future work; a straightforward one is the incorporation of new combinations of strategies to provide a bigger portfolio, while finding the balance to avoid increasing the cost of the rank computation. Another interesting idea is to experiment with new and possibly more lightweight optimizers in order to alleviate the work of the hyperheuristic approach. Finally, the use of a similar adaptive framework could be used to interleave different propagation techniques.

```
:-lib(ic).
queens(N, Board):-
  length(Board, N),
  Board:: 1..N,
  (fromto(Board, [Q1|Cols], Cols, []) do
    (foreach(Q2, Cols), param(Q1), count(Dist,1,_) do
      noattack(Q1, Q2, Dist)
    )
  ),
%call to search predicate
noattack(Q1,Q2,Dist):-
  Q2 #\= Q1,
  Q2 - Q1 #\= Dist,
  Q1 - Q2 #\= Dist.
```

Algorithm 3

```
:-lib(ic).
magic(N):-
  NN is N*N,
  Sum is N*(NN+1)//2,
  dim(Square, [N,N]),
  Square:: 1..NN,
  alldifferent(Square),
  (for(I,1,N),
    foreach(U,UpDiag),
    foreach(D,DownDiag),
    param(N,Square,Sum)
  do
    Sum #= sum(Square[I,1..N]),
    Sum #= sum(Square[1..N,I]),
    U is Square[I,I],
    D is Square[I,N+1-I]
  ),
  Sum #= sum(UpDiag),
  Sum #= sum(DownDiag),
  Square[1,1] #< Square[1,N],
  Square[1,1] #< Square[N,N],
  Square[1,1] #< Square[N,1],
  Square[1,N] #< Square[N,1],
  % call to search predicate
  write(Square).
```

Algorithm 4

Appendix

In this appendix we provide the encodings of all models employed in the experiments.

A. Ecl^{i}ps^e Models

A.1. N−Queens. See Algorithm 3.

A.2. Magic Squares. See Algorithm 4.

A.3. Knight's Tour. See Algorithm 5.

A.4. Sudoku. See Algorithm 6.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

```
:-lib(ic).
knight_tour(N, Board):-
  N2 is N*N,
  length(Board, N2),
  C = [-2,-1,1,2],
  alldifferent(Board),
  ( fromto(Board, [Q1|Cols], Cols, []), param(N,C) do
    t_Q2(Cols,Q2),
    I:: 1..N,
    J:: 1..N,
    A:: C,
    B:: C,
    IA #= I+A,
    JB #= J+B,
    IA #>= 1,
    JB #>= 1,
    IA #=< N,
    JB #=< N,
    abs(A)+abs(B) #= 3,
    (I-1)*N + J #= Q1,
    Q2 #= (I-1+A)*N + J+B
  ),
  % call to search predicate
t_Q2([A|B],A).
t_Q2([],A).
```

Algorithm 5

```
sudokusolve(ProblemName):-
  problem(ProblemName, Board),
  sudoku(3, Board).
sudoku(N, Board):-
  N2 is N*N,
  dim(Board, [N2,N2]),
  Board[1..N2,1..N2] :: 1..N2,
  ( for(I,1,N2), param(Board,N2) do
    Row is Board[I,1..N2],
    alldifferent(Row),
    Col is Board[1..N2,I],
    alldifferent(Col)
  ),
  ( multifor([I,J],1,N2,N), param(Board,N) do
    ( multifor([K,L],0,N-1), param(Board,I,J),
      foreach(X,SubSquare) do
        X is Board[I+K,J+L]
    ),
    alldifferent(SubSquare)
  ),
  term_variables(Board, Vars),
  % call to search predicate
  write(Board).
problem(1, []([]([]_, _, 2, _, _, 5, _, 7, 9),
  []([]1, _, 5, _, _, 3, _, _, _),
  []([]_, _, _, _, _, _, 6, _, _),
  []([]_, 1, _, 4, _, _, 9, _, _),
  []([]_, 9, _, _, _, _, _, 8, _),
  []([]_, _, 4, _, _, 9, _, 1, _),
  []([]_, _, 9, _, _, _, _, _, _),
  []([]_, _, _, 1, _, _, 3, _, 6),
  []([]6, 8, _, 3, _, _, 4, _, _))).
```

Algorithm 6

Acknowledgments

Ricardo Soto is supported by Grant CONICYT/FONDECYT/INICIACION/11130459, Broderick Crawford is supported by Grant CONICYT/FONDECYT/1140897, and Fernando Paredes is supported by Grant CONICYT/FONDECYT/1130455.

References

[1] J.-P. Métivier, P. Boizumault, and S. Loudni, "Solving nurse rostering problems using soft global constraints," in *Principles and Practice of Constraint Programming—CP 2009: Proceedings of the 15th International Conference, CP 2009 Lisbon, Portugal, September 20–24, 2009*, vol. 5732 of *Lecture Notes in Computer Science*, pp. 73–87, Springer, Berlin, Germany, 2009.

[2] P. Baptiste and C. L. Pape, "Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems," in *Principles and Practice of Constraint Programming-CP97: Third International Conference, CP97 Linz, Austria, October 29–November 1, 1997 Proceedings*, vol. 1330 of *Lecture Notes in Computer Science*, pp. 375–389, Springer, Berlin, Germany, 1997.

[3] R. Soto, H. Kjellerstrand, O. Durán, B. Crawford, E. Monfroy, and F. Paredes, "Cell formation in group technology using constraint programming and Boolean satisfiability," *Expert Systems with Applications*, vol. 39, no. 13, pp. 11423–11427, 2012.

[4] R. Chenouard, L. Granvilliers, and P. Sebastian, "Search heuristics for constraint-aided embodiment design," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, vol. 23, no. 2, pp. 175–195, 2009.

[5] P. Barahona and L. Krippahl, "Constraint programming in structural bioinformatics," *Constraints*, vol. 13, no. 1-2, pp. 3–20, 2008.

[6] C. Castro, E. Monfroy, C. Figueroa, and R. Meneses, "An approach for dynamic split strategies in constraint solving," in *Proceedings of the 4th Mexican International Conference on Artificial Intelligence (MI-CAI '05)*, vol. 3789 of *Lecture Notes in Computer Science*, pp. 162–174, Springer, 2005.

[7] B. Crawford, R. Soto, C. Castro, and E. Monfroy, "A hyper-heuristic approach for dynamic enumeration strategy selection in constraint satisfaction," in *Proceedings of the 4th International Work-Conference on the Interplay between Natural and Artificial Computation (IWINAC '11)*, vol. 6687 of *Lecture Notes in Computer Science*, pp. 295–304, Springer, 2011.

[8] E. K. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg, "Hyper-heuristics: an emerging direction in modern search technology," in *Handbook of Metaheuristics*, vol. 57 of *International Series in Operations Research & Management Science*, pp. 457–474, Springer, New York, NY, USA, 2003.

[9] P. Ross, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, Springer, Berlin, Germany, 2005.

[10] Y. Hamadi, E. Monfroy, and F. Saubion, *Autonomous Search*, Springer, Berlin, Germany, 2012.

[11] R. Soto, B. Crawford, E. Monfroy, and V. Bustos, "Using autonomous search for generating good enumeration strategy blends in constraint programming," in *Computational Science and Its Applications—ICCSA 2012: Proceedings of the 12th International Conference, Salvador de Bahia, Brazil, June 18-21, 2012, Proceedings, Part III*, vol. 7335 of *Lecture Notes in Computer Science*, pp. 607–617, Springer, Berlin, Germany, 2012.

[12] B. Crawford, C. Castro, E. Monfroy, R. Soto, W. Palma, and F. Paredes, "Dynamic selection of enumeration strategies for solving constraint satisfaction problems," *Romanian Journal of Information Science and Technology*, vol. 15, no. 2, pp. 106–128, 2012.

[13] B. Crawford, R. Soto, E. Monfroy, W. Palma, C. Castro, and F. Paredes, "Parameter tuning of a choice-function based hyper-heuristic using Particle Swarm Optimization," *Expert Systems with Applications*, vol. 40, no. 5, pp. 1690–1695, 2013.

[14] S. L. Epstein, E. C. Freuder, R. J. Wallace, A. Morozov, and B. Samuels, "The adaptive constraint engine," in *Principles and Practice of Constraint Programming—CP 2002: 8th International Conference, CP 2002 Ithaca, NY, USA, September 9–13, 2002 Proceedings*, vol. 2470 of *Lecture Notes in Computer Science*, pp. 525–540, Springer, Berlin, Germany, 2002.

[15] S. Epstein and S. Petrovic, "Learning to solve constraint problems," in *Proceedings of the Workshop on Planning and Learning (ICAPS '07)*, 2007.

[16] Y. Xu, D. Stern, and H. Samulowitz, "Learning adaptation to solve constraint satisfaction problems," in *Proceedings of the 3rd International Conference on Learning and Intelligent Optimization (LION '09)*, pp. 507–523, 2009.

[17] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI '04)*, pp. 146–150, IOS Press, 2004.

[18] D. Grimes and R. J. Wallace, "Learning to identify global bottlenecks in constraint satisfaction search," in *Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference (FLAIRS '07)*, pp. 592–597, AAAI Press, May 2007.

[19] R. J. Wallace and D. Grimes, "Experimental studies of variable selection strategies based on constraint weights," *Journal of Algorithms*, vol. 63, no. 1–3, pp. 114–129, 2008.

[20] R. Barták and H. Rudová, "Limited assignments: a new cutoff strategy for incomplete depth-first search," in *Proceedings of the ACM Symposium on Applied Computing (SAC '05)*, pp. 388–392, 2005.

[21] E. Monfroy, C. Castro, B. Crawford, R. Soto, F. Paredes, and C. Figueroa, "A reactive and hybrid constraint solver," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 25, no. 1, pp. 1–22, 2013.

[22] Y. Han, "Deterministic sorting in $O(n \log\log n)$ time and linear space," *Journal of Algorithms*, vol. 50, no. 1, pp. 96–105, 2004.

[23] M. Thorup, "Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise Boolean operations," *Journal of Algorithms*, vol. 42, no. 2, pp. 205–230, 2002.

[24] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys*, vol. 40, no. 4, article 11, 2008.

[25] B. Crawford, R. Soto, C. Castro, E. Monfroy, and F. Paredes, "An extensible autonomous search framework for constraint programming," *International Journal of Physical Sciences*, vol. 6, no. 14, pp. 3369–3376, 2011.

[26] T. Balafoutis and K. Stergiou, "Evaluating and improving modern variable and revision ordering strategies in CSPs," *Fundamenta Informaticae*, vol. 102, no. 3-4, pp. 229–261, 2010.

[27] P. van Beek, "Backtracking search algorithms," in *Handbook of Constraint Programming*, chapter 4, pp. 85–134, Elsevier, 2006.

[28] I. P. Gent, E. MacIntyre, P. Presser, B. M. Smith, and T. Walsh, "An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem," in *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming (CP '96)*, vol. 1118 of *Lecture Notes in Computer Science*, pp. 179–193, Springer, Berlin, Germany, 1996.