Evolving Parameterised Policies for Stochastic Constraint Programming*

S. D. Prestwich[1], S. A. Tarim[2], R. Rossi[3], and B. Hnich[4]

[1]Cork Constraint Computation Centre, University College Cork, Ireland
[2]Operations Management Division, Nottingham University Business School, Nottingham, UK
[3]Logistics, Decision and Information Sciences Group, Wageningen UR, the Netherlands
[4]Faculty of Computer Science, Izmir University of Economics, Turkey
s.prestwich@cs.ucc.ie, armtar@yahoo.com,
roberto.rossi@wur.nl, brahim.hnich@ieu.edu.tr

Abstract. Stochastic Constraint Programming is an extension of Constraint Programming for modelling and solving combinatorial problems involving uncertainty. A solution to such a problem is a policy tree that specifies decision variable assignments in each scenario. Several solution methods have been proposed but none seems practical for large multi-stage problems. We propose an incomplete approach: specifying a policy tree indirectly by a parameterised function, whose parameter values are found by evolutionary search. On some problems this method is orders of magnitude faster than a state-of-the-art scenario-based approach, and it also provides a very compact representation of policy trees.

1 Introduction

Stochastic Constraint Programming (SCP) is a recently proposed extension of Constraint Programming (CP) designed to model and solve complex problems involving uncertainty and probability, a direction of research first proposed in [2]. Stochastic Constraint Satisfaction Problems (SCSPs) are in a higher complexity class than Constraint Satisfaction Problems (CSPs) and usually harder to solve.

An m-stage SCSP is defined as a tuple $(V, S, D, P, C, \Theta, L)$ where V is a set of decision variables, S a set of stochastic variables, D a function mapping each element of $V \cup S$ to a domain of values, P a function mapping each variable in S to a probability distribution, C a set of constraints on $V \cup S$, Θ a function mapping each constraint in C to a threshold value $\theta \in (0, 1]$, and $L = (\langle V_1, S_1\rangle, \ldots, \langle V_m, S_m\rangle)$ a list of *decision stages* such that the V_i partition V and the S_i partition S. Each constraint must contain at least one V variable, a constraint $h \in C$ containing only V variables is a *hard constraint* with threshold $\Theta(h) = 1$, and one containing at least one S variable is a *chance constraint*. To solve an m-stage SCSP an assignment to the variables in V_1 must be found such that, given random values for S_1, assignments can be found for V_2 such that, given random values for $S_2, \ldots$ assignments can be found for V_m so that, given

* B. Hnich is supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under Grant No. SOBAG-108K027. This material is based in part upon works supported by the Science Foundation Ireland under Grant No. 05/IN/I886.

random values for S_m, the hard constraints are each satisfied and the chance constraints (containing both decision and stochastic variables) are satisfied in the specified fraction of all possible *scenarios* (set of values for the stochastic variables). A useful concept is that of a *policy tree* of decisions, in which each node represents a value chosen for a decision variable, and each arc from a node represents the value assigned to a stochastic variable. Each path in the tree represents a different possible scenario and the values assigned to decision variables in that scenario. A *satisfying policy (tree)* is a policy tree in which each chance constraint is satisfied with respect to the tree. A chance constraint $h \in C$ is satisfied with respect to a policy tree if it is satisfied under some fraction $\phi \geq \Theta(h)$ of all possible paths in the tree.

As an example, consider a 2-stage SCSP with $V_1 = \{x_1\}$, $S_1 = \{s_1\}$, $V_2 = \{x_2\}$ and $S_2 = \{s_2\}$. Let $\text{dom}(x_1) = [1, 4]$, $\text{dom}(x_2) = [3, 6]$, $\text{dom}(s_1) = [4, 5]$ and $\text{dom}(s_2) = [3, 4]$ where $[a, b]$ represents the discrete interval $\{i \in \mathbf{Z} \,|\, a \leq i \leq b\}$, and the stochastic variable values each have probability 0.5. There are two chance constraints c_1: $(s_1 x_1 + s_2 x_2 \geq 30)$ and c_2: $(s_2 x_1 = 12)$ with $\theta_{c_1} = 0.75$ and $\theta_{c_2} = 0.5$. Decision variable x_1 must be set to a unique value while the value of x_2 depends on that of s_1. A policy for this problem is shown in Figure 1: notice that it is in the form of a tree. The 4 scenarios A, B, C and D each have probability 0.25. Constraint c_1 is satisfied in A, C and D therefore with probability 0.75. Constraint c_2 is satisfied in A and C therefore with probability 0.5. These probabilities satisfy the thresholds $\theta_{c_1}, \theta_{c_2}$ so this is a satisfying policy.

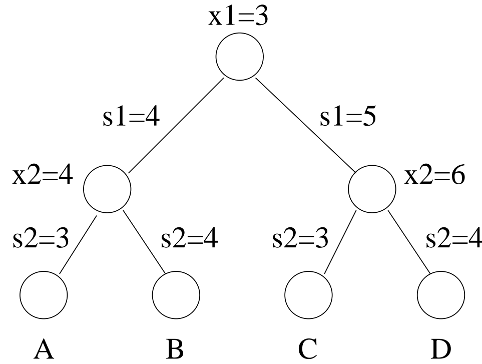

Fig. 1. Example of a satisfying policy tree

No practical way of solving large multi-stage SCSPs has yet been proposed. The design of local search algorithms for SCP has been suggested [20] in order to improve scalability but this idea does not seem to have been pursued, and it does not address the problem of representing large policy trees. We propose a novel approach: using an evolutionary algorithm to choose parameter values for a parameterised function that indirectly specifies a policy tree. The result is an incomplete SCP algorithm that is intended to scale well in two ways: a simple parameterised function can be used to

represent a large policy tree, and evolutionary search can handle problems with many decision variables.

The paper is organised as follows. Section 2 describes how to evolve parameterised functions that specify policies. Section 3 shows empirically that an evolutionary algorithm can find a function representing a satisfying policy. Section 4 discusses related work. Section 5 concludes the paper.

2 Evolving parameterised policies

Instead of explicitly representing a policy tree we use a parameterised function $\tau_{\underline{w}}$, whose input is the current stochastic variable assignments and a decision variable, and whose output is a domain value for that variable. Its parameters $\underline{w} = (w_1, w_2 \ldots)$ are real-valued numbers which we shall call *weights*. $\tau_{\underline{w}}$ completely defines the policy tree, and if it does not require an exponential number of weights then it avoids the memory problem associated with large trees. For any given function there exist policy trees that cannot be represented, and there is a risk that these are the only satisfying policy trees, but the hope is that relatively simple functions will suffice for most problems of interest.

To simplify the discussion we consider only SCSPs whose decision and stochastic variable domains are intervals $[L, U]$ of integer values, but the method easily generalises to variables with other domains. We assume a fixed ordering of the problem variables (any ordering that conforms to the stage structure will do). First we compute an affine function

$$\alpha_{\underline{w}}(S, x_j) = w_j + \sum_{i \in \sigma_j} w_i s_i$$

where σ_j denotes the set of indices of the stochastic variables S that precede decision variable x_j. This is the simplest possible function that involves all relevant stochastic variables; we do not claim that it will suffice for all SCSPs, but it requires only a linear number of weights and works well in experiments so far (see Section 3). The constant w_j is necessary because in the case of a decision variable x_j that is not preceded by any stochastic variable (so that $\sigma_j = \emptyset$) we require a default value: in the special cases of a deterministic CSP or a 1-stage SCSP no decision variable is preceded by a stochastic variable, so the policy is simply a weight w_j for each decision variable x_j. Note that the stochastic variables S_m (those in the final stage) do not precede any decision variables, and therefore do not appear in $\underline{w}$: thus they do not appear in the policy, though they are used to evaluate it.

However, the value of $\alpha_{\underline{w}}(S, x_j)$ is a real number and not a domain value, so to obtain an integer in $[L, U]$ it is discretised by truncation, then modular arithmetic is used to obtain an integer in the required range:

$$\tau_{\underline{w}}(S, x_j) = L + (\lfloor \alpha_{\underline{w}}(S, x_j) \rfloor \text{ mod } [U - L + 1])$$

Now $\underline{w}$ defines a policy: for each decision variable x_j we choose its value to be $\tau_{\underline{w}}(S, x_j)$.

For example, consider a 3-stage problem with $V_1 = \{x_1, x_2\}$, $S_1 = \{s_1, s_2\}$, $V_2 = \{x_3, x_4\}$, $S_2 = \{s_3, s_4\}$, $V_3 = \{x_5, x_6\}$ and $S_3 = \{s_5, s_6\}$. Suppose we wish to find

the value of x_3 given that $s_1 = 5$, $s_2 = 7$, $\text{dom}(x_3) = [5, 10]$ and our policy is specified by a weight vector

$$\underline{w} = (0.1, 5.3, 7.1, 9.9, 8.7, 4.1, -0.6, 5.5, -5.2, 2.9)$$

Notice that $\underline{w}$ has 10 components though there are 12 variables in the SCSP: this is because the S_3 variables do not precede any decision variables, as mentioned above. Then

$$\alpha_{\underline{w}}(S, x_3) = 8.7 + 7.1s_1 + 9.9s_2 = 113.5$$

and

$$\tau_{\underline{w}}(S, x_3) = 5 + (\lfloor 113.5 \rfloor \text{ mod } [10 - 5 + 1]) = 5 + (113 \text{ mod } 6) = 5 + 5 = 10$$

So under the policy defined by $\underline{w}$, variable x_3 is set to 10 under any scenario in which $s_1 = 5$ and $s_2 = 7$.

Now that we have defined the form of our policies we can describe how to search for them. The state space to be explored is the Cartesian product $\mathbf{R}^k$ representing the space of real-valued weight vectors $\underline{w}$, where k is the total number of SCP variables not counting those in S_m. To handle the SCP constraints we use *penalty functions* to obtain an unconstrained optimisation problem: this is a standard technique that penalises constraint violations, commonly used when applying genetic algorithms or local search to CSPs. Specifically, the objective function to be minimised is

$$\Phi(\underline{w}) = \sum_{h \in C} \phi(h, \underline{w})$$

where the penalty functions are

$$\phi(h, \underline{w}) = \begin{cases} 0 & \text{if } \pi_h(\underline{w}) \geq \Theta(h) \\ \Theta(h) - \pi_h(\underline{w}) & \text{if } \pi_h(\underline{w}) < \Theta(h) \end{cases}$$

and $\pi_h(\underline{w})$ is the probability that that h is satisfied under the policy defined by $\underline{w}$. Any policy defined by $\underline{w}$ such that $\Phi(\underline{w}) = 0$ is clearly a satisfying policy.

Given the search space and objective function we can apply an evolutionary (or local) search algorithm to solve the problem. In this paper we do not describe the algorithm we used in detail because our emphasis is on showing the feasibility of the approach. Briefly, it is a cellular evolution strategy with Cauchy mutation, plus some additional mutation heuristics designed for this application. Each chromosome is a weight vector $\underline{w}$, and for each chromosome we compute its fitness $\Phi(\underline{w})$ (fitness is conventionally maximised but we minimise Φ). To compute the $\pi_h(\underline{w})$ we check every leaf node in the implied policy tree. The probability associated with a leaf ℓ is the product of the probabilities associated with the stochastic variable assignments in the arcs of the path leading to ℓ. At each leaf a chance constraint $h \in C$ is either satisfied or violated, and by summing the probabilities of the leaves at which h is satisfied we obtain the probability that $\pi_h(\underline{w})$ that h is satisfied under the policy defined by $\underline{w}$. The $\pi_h(\underline{w})$ can also be estimated by sampling the leaves using any of the scenario reduction techniques used in [18], and this is important for problems with many stages. But we can sample

many more leaves than [18] because we do not use them to derive a deterministic CSP (in section 3 we use over 1000 scenarios). Chance and hard constraints are treated uniformly: the only difference between them is that a hard constraint h has $\Theta(h) = 1$ while a chance constraint has $\Theta(h) < 1$. We could compute $\pi_h(\underline{w})$ for every chromosome by using all leaves, but to be more efficient we use a number of leaves that depends on how promising the current fitness estimate is: only the fittest chromosomes (including the one representing the satisfying policy tree) sample all leaves. To do this we use the resampling scheme of [15].

We call our method EPP (Evolved Parameterised Policies). EPP transforms a multi-stage SCSP into a noisy numerical optimisation problem. The word "noisy" here refers to the fact that the objective function must be averaged over many scenarios. There are many evolutionary algorithms designed to handle noisy fitness functions: see [3, 9] for surveys.

3 Experiments

In this section we show empirically that it is possible to find a satisfying policy using EPP. We use a benchmark set of random SCSPs with 5 chance constraints over 4 decision variables $x_1 \dots x_4$ and 8 stochastic variables $s_1 \dots s_8$. The decision variable domains are the discrete intervals $\text{dom}(x_1) = [5, 10]$, $\text{dom}(x_2) = [4, 10]$, $\text{dom}(x_1) = [3, 10]$ and $\text{dom}(x_2) = [6, 10]$. The domains of stochastic variable s_1, s_3, s_5, s_7 contain 2 values while those of s_2, s_4, s_6, s_8 contain 3 values; in both bases the values are chosen randomly from the discrete interval $[1, 5]$ and have equal probabilities. The chance constraints are:

$$\begin{aligned} x_1s_1 + x_2s_2 + x_3s_3 + x_4s_4 &= 80 \quad (\theta = \alpha) \\ x_1s_5 + x_2s_6 + x_3s_7 + x_4s_8 &\leq 100 \quad (\theta = \beta) \\ x_1s_5 + x_2s_6 + x_3s_7 + x_4s_8 &\geq 60 \quad (\theta = \beta) \\ x_1s_2 + x_3s_6 &\geq 30 \quad (\theta = 0.7) \\ x_2s_4 + x_4s_8 &= 20 \quad (\theta = 0.05) \end{aligned}$$

where $\alpha \in \{0.005, 0.01, 0.03, 0.05, 0.07, 0.1\}$ and $\beta \in \{0.6, 0.7, 0.8\}$. The problems are 4-stage: $V_1 = \{x_1\}$, $S_1 = \{s_1, s_5\}$, $V_2 = \{x_2\}$, $S_2 = \{s_2, s_6\}$, $V_3 = \{x_3\}$, $S_3 = \{s_3, s_7\}$, $V_4 = \{x_4\}$ and $S_4 = \{s_4, s_8\}$. In total we have 6 α-values and 3 β-values, and we randomly generate 5 different sets of stochastic variable domains, giving 90 instances in total.

The table in Figure 2 compares the scenario-based approach (SBA) of [18] (see Section 4) with EPP. All figures are in seconds and "—" denotes that the time is greater than 200 seconds. All times were obtained on a 2.8 GHz Pentium (R) 4 with 512 MB RAM, or on another machine then normalised to this one. EPP figures are medians over 30 runs. Both methods used all $2^4 \times 3^4 = 1296$ scenarios. Though these are quite small SCSPs they turn out to be non-trivial for SBA, which transforms them into deterministic CSPs with 6739 variables and 6485 constraints. In contrast, EPP transforms them into unconstrained noisy optimisation problems with 10 real-valued variables.

A clear pattern emerges from the results: where SBA solved a problem it was up to 48 times faster than EPP, but EPP solved every problem that SBA solved plus many

problem set 1				problem set 2				problem set 3				problem set 4				problem set 5			
α	β	SBA	EPP	α	β	SBA	EPP	α	β	SBA	EPP	α	β	SBA	EPP	α	β	SBA	EPP
0.6	0.05	—	0.5	0.6	0.05	—	1.6	0.6	0.05	0.7	0.4	0.6	0.05	—	4.2	0.6	0.05	—	0.1
0.6	0.10	—	1.0	0.6	0.10	—	4.8	0.6	0.10	0.5	3.1	0.6	0.10	—	—	0.6	0.10	—	0.5
0.6	0.12	—	0.9	0.6	0.12	—	14	0.6	0.12	0.5	3.1	0.6	0.12	—	—	0.6	0.12	—	0.7
0.6	0.15	—	1.4	0.6	0.15	—	15	0.6	0.15	—	15	0.6	0.15	—	—	0.6	0.15	—	0.8
0.6	0.17	—	1.7	0.6	0.17	—	118	0.6	0.17	—	14	0.6	0.17	—	—	0.6	0.17	—	2.2
0.6	0.20	—	1.6	0.6	0.20	—	—	0.6	0.20	—	49	0.6	0.20	—	—	0.6	0.20	—	1.9
0.7	0.05	—	1.3	0.7	0.05	—	1.7	0.7	0.05	0.6	2.5	0.7	0.05	—	4.9	0.7	0.05	0.2	0.1
0.7	0.10	—	1.2	0.7	0.10	—	4.8	0.7	0.10	0.7	9.1	0.7	0.10	—	—	0.7	0.10	—	0.4
0.7	0.12	—	1.3	0.7	0.12	—	16	0.7	0.12	0.6	12	0.7	0.12	—	—	0.7	0.12	—	0.7
0.7	0.15	—	1.9	0.7	0.15	—	16	0.7	0.15	—	27	0.7	0.15	—	—	0.7	0.15	—	0.8
0.7	0.17	—	2.7	0.7	0.17	—	144	0.7	0.17	—	46	0.7	0.17	—	—	0.7	0.17	—	1.8
0.7	0.20	—	2.8	0.7	0.20	—	—	0.7	0.20	—	159	0.7	0.20	—	—	0.7	0.20	—	3.3
0.8	0.05	—	12	0.8	0.05	—	2.7	0.8	0.05	0.8	9.7	0.8	0.05	—	7.5	0.8	0.05	—	0.2
0.8	0.10	—	9.4	0.8	0.10	—	7.1	0.8	0.10	0.6	17	0.8	0.10	—	—	0.8	0.10	—	0.9
0.8	0.12	—	11	0.8	0.12	—	20	0.8	0.12	0.6	29	0.8	0.12	—	—	0.8	0.12	—	0.8
0.8	0.15	—	12	0.8	0.15	—	13	0.8	0.15	—	58	0.8	0.15	—	—	0.8	0.15	—	1.2
0.8	0.17	—	15	0.8	0.17	—	—	0.8	0.17	—	109	0.8	0.17	—	—	0.8	0.17	—	1.6
0.8	0.20	—	13	0.8	0.20	—	—	0.8	0.20	—	—	0.8	0.20	—	—	0.8	0.20	—	3.3

Fig. 2. Experimental results

more, and in some cases EPP was at least 2000 times faster; EPP is on average much faster than SBA. Where SBA and EPP both failed to solve an instance, the instance might be infeasible. However, we do know that in a few cases both SBA and EPP failed to solve a feasible instance (verified by further experiments) so there is room for improvement. It might be that our parameterised policy space does not contain satisfying policies for these problems, and that a more complex parameterised function is required.

4 Related work

Several SCSP solution methods have been proposed. [20] presented two complete algorithms based on backtracking and forward checking and suggested some approximation procedures, while [1] described an arc-consistency algorithm. In the method of [18] an SCSP is transformed into a *deterministic equivalent* Constraint Satisfaction Problem (CSP) and solved by standard CP methods. It is also extended to handle multiple chance constraints and multiple objective functions. This method gives much better performance on the book production planning problem of [20] compared to the tree search methods. To reduce the size of the CSP *scenario reduction* methods are proposed, as used in Stochastic Programming. These choose a small but representative set of scenarios. However, it might not always be possible to find a small representative set of scenarios, and in some cases choosing an inappropriate set of scenarios can yield an unsolvable CSP. Moreover, using even a modest number of scenarios leads to a CSP that is several times larger than the original SCSP. [4] modify a standard backtracking algorithm to one that can handle multiple chance constraints and uses polynomial space,

but is inefficient in time. [16] proposed a cost-based filtering technique for SCP. For the special case of SCP with linear recourse, [19] propose a Bender's decomposition algorithm.

Stochastic Boolean Satisfiability (SSAT) is related to SCP. A recent survey of the SSAT field is given in [13], on which we base this discussion. An SSAT problem can be regarded as an SCSP in which all variable domains are Boolean, all constraints are extensional and may be non-binary, and all constraints are treated as a single chance constraint (there are also restricted and extended versions). Our method therefore applies immediately to SSAT problems. SSAT algorithms fall into three classes: systematic, approximation, and non-systematic. Systematic algorithms are based on the standard SAT backtracking algorithm and correspond roughly to some current SCP algorithms. Approximation algorithms work well on restricted forms of SSAT but less well on general SSAT problems. For example the APPSSAT algorithm [12] considers scenarios in decreasing order of probability to construct a partial tree, but does not work well when all scenarios have similar probability. A non-systematic algorithm for SSAT is randevalssat [10], which applies local search to the decision (existential) variables in a random set of scenarios. This algorithm also suffers from memory problems because it must build a partial tree.

5 Conclusion

We have proposed a method for SCP called EPP, based on the evolution of a parameterised function that indirectly specifies a policy tree. EPP does not suffer from the memory problems of most methods and does not introduce a large number of new variables. It is also the first incomplete algorithm for SCP, and experiments show that on some problems EPP is several orders of magnitude faster than the current best (complete) method. It does not exploit constraint filtering techniques but these could perhaps be used to handle hard constraints. EPP will also require slight modification for handling variable domains that contain arbitrary integers or real numbers, and for handling problems with objective functions. We will explore these issues in future work and test EPP on more interesting SCP problems, and also on SSAT, QBF and QCSP problems which can all be modelled as SCSPs.

EPP is closely related to a machine learning method that has been used for many optimisation problems involving uncertainty: *neuroevolution*. In neuroevolution the parameterised function is an artificial neural network whose parameters are the network weights, which are found by evolutionary search. Unlike our simple function, neural networks are universal function approximators which can in principle approximate any policy. They might turn out to be necessary for harder SCP problems, but on our benchmark set they had no effect other than to make the problem harder to solve, because they must learn more weights. Neuroevolution has been applied to very challenging control problems with good results: see for example [7, 8, 17]. It has also been used for learning to play games such as Backgammon [14], Go [11], Checkers [5] and Chess [6]. These successes indicate that EPP might work well on real-world SCP problems that are too large to solve by complete methods.

References

1. T. Balafoutis, K. Stergiou. Algorithms for Stochastic CSPs. *12th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 4204, Springer, 2006, pp. 44–58.
2. T. Benoist, E. Bourreau, Y. Caseau, B. Rottembourg. Towards Stochastic Constraint Programming: A Study of On-Line Multi-Choice Knapsack with Deadlines. *7th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* 2239, Springer, 2001, pp. 61–76.
3. H.-G. Beyer. Evolutionary Algorithms in Noisy Environments: Theoretical Issues and Guidelines for Practice. *Computer Methods in Applied Mechanics and Engineering* 186(2–4):239–267, 2000.
4. L. Bordeaux, H. Samulowitz. On the Stochastic Constraint Satisfaction Framework. *ACM Symposium on Applied Computing*, 2007, pp. 316–320.
5. D. B. Fogel, K. Chellapilla. Verifying Anaconda's Expert Rating by Competing Against Chinook: Experiments in Co-Evolving a Neural Checkers Player. *Neurocomputing* 42(1-4):69–86, 2002.
6. D. B. Fogel, T. J. Hays, S. L. Hahn, J. Quon. A Self-Learning Evolutionary Chess Program. *Proceedings of the IEEE* 92(12):1947–1954, 2004.
7. F. Gomez, J. Schmidhuber, R. Miikkulainen. Efficient Non-Linear Control Through Neuroevolution. *Journal of Machine Learning Research* 9:937–965, 2008.
8. N. M. Hewahi. Engineering Industry Controllers Using Neuroevolution. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 19(1):49–57, 2005.
9. Y. Jin, J. Branke. Evolutionary Optimization in Uncertain Environments — a Survey. *IEEE Transactions on Evolutionary Computation* 9(3):303–317, 2005.
10. M. L. Littman, S. M. Majercik, T. Pitassi. Stochastic Boolean Satisfiability. *Journal of Automated Reasoning* 27(3):251–296, 2001.
11. A. Lubberts, R. Miikkulainen. Co-Evolving a Go-Playing Neural Network. *Genetic and Evolutionary Computation Conference*, Kaufmann, 2001, pp. 14–19.
12. S. M. Majercik. APPSSAT: Approximate Probabilistic Planning Using Stochastic Satisfiability. *International Journal of Approximate Reasoning* 45(2):402–419, 2007.
13. S. M. Majercik. Stochastic Boolean Satisfiability. Handbook of Satisfiability, Chapter 27, IOS Press, 2009, pp. 887–925.
14. J. B. Pollack, A. D. Blair. Co-Evolution in the Successful Learning of Backgammon Strategy. *Machine Learning* 32(3):225–240, 1998.
15. S. D. Prestwich, S. A. Tarim, R. Rossi, B. Hnich. A Steady-State Genetic Algorithm With Resampling for Noisy Inventory Control. *10th International Conference on Parallel Problem Solving From Nature, Lecture Notes in Computer Science* vol. 5199, Springer, 2008, pp. 559–568.
16. R. Rossi, S. A. Tarim, B. Hnich, S. D. Prestwich. Cost-Based Domain Filtering for Stochastic Constraint Programming. *14th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* 5202, Springer, 2008, pp. 235–250.
17. K. O. Stanley, R. Miikkulainen. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation* 10(2):99–127, 2002.
18. S. A. Tarim, S. Manandhar, T. Walsh. Stochastic Constraint Programming: A Scenario-Based Approach. *Constraints* 11(1):1383–7133, 2006.
19. S. A. Tarim, I. Miguel. A Hybrid Bender's Decomposition Method for Solving Stochastic Constraint Programs with Linear Recourse. *Lecture Notes in Computer Science* vol. 3978, Springer, 2006, pp. 133–148.
20. T. Walsh. Stochastic Constraint Programming. *15th European Conference on Artificial Intelligence*, 2002.