Continuous Search in Constraint Programming: An Initial Investigation

Alejandro Arbelaez, Youssef Hamadi

Continuous Search in Constraint Programming: An Initial Investigation

Alejandro Arbelaez[1] (student)
Youssef Hamadi[1,2]

[1] Microsoft-INRIA joint-lab, Orsay France,
alejandro.arbelaez@inria.fr
[2] Microsoft Research, Cambridge United Kingdom,
youssefh@microsoft.com

Abstract. In this work, we present the concept of Continuous Search, the objective of which is to allow any user to eventually get top performance from their constraint solver. Unlike previous approaches (see [9] for a recent survey), Continuous Search does not require the disposal of a large set of representative instances to properly train and learn parameters. It only assumes that once the solver runs in a real situation (often called production mode), instances will come over time, and allow for proper offline continuous training. The objective is therefore not to instantly provide good parameters for top performance, but to take advantage of the *real* situation to train in the background and improve the performances of the system in an incremental manner.

1 Introduction

In Constraint Programming, properly crafting a constraint model which captures all the constraints of a particular problem is often not enough to ensure acceptable runtime performance. One way to improve efficiency is to use well known tricks like redundant and channeling constraints or to be aware that the constraint solver has a particular global constraint that can do part of the job more efficiently. The problem with these improvements (or tricks) is that they are far from obvious. Indeed, they do not change the solution space of the original model, and for a normal user (with a classical mathematical background), it is difficult to understand why adding redundancy helps.

Because of that, normal users are often left with the tedious task of tuning the search parameters of their constraint solver, and this again, is both time consuming and not necessarily straightforward. Indeed, even if tuning is conceptually simple (try different parameters, pick the best), it requires a set of representative instances in order to properly work. This might be obvious for a constraint programmer, but not for a normal user which could train on instances far different from the ones faced by his application.

In this work, we present the concept of Continuous Search (CS), the objective of which is to allow any user to eventually get top performance from their constraint solver. Unlike previous approaches (see [9] for a recent survey), Continuous Search does not require the disposal of a large set of representative

instances to properly train and learn parameters. It only assumes that once the solver runs in a real situation (often called production mode), instances will come over time, and allow for proper offline continuous training. The objective is therefore not to instantly provide good parameters for top performance, but to take advantage of the 'real' situation to train in the background and improve the performances of the system in an incremental manner.

The Continuous Search paradigm, uses an online learning algorithm to update a prediction function which matches instances features to the most efficient set of parameters for a given instance (e.g. Variable/Value selection algorithms). Since CS can start without offline training, this prediction function might be initially undefined. If this is the case, an instance is solved by running the solver with its default parameters. Once the instance is solved and the solution is given back to the application[1], we start our Continuous Search training. At that point, the instance is reused and the goal is to refine the strategy used to tackle it. This is done by a specific repair-like algorithm which perturbs the strategy to find a new strategy able to solve the problem more efficiently. If such a strategy is found, its parameters are stored with the instance features, and therefore, could be reused to solve similar instances more efficiently.

Technically, this means that there are two different search efforts. The one done to solve the real problem, and the one related to the long term improvement of the constraint solver. We believe that this extra use of computational resources is realistic, since nowadays systems (especially production ones) are almost always on. Moreover, this has to be balanced against the huge computational cost of offline training [10]. Last, this late adaptation is the only way to face the 'real' instances and even, to adapt to changes on the modeling or to the arrival of a completely new class of problem.

The paper is organized as follows. Background material is presented in Section 2. Section 3 introduces the continuous search paradigm. Section 4 presents experimental results. Finally, before our general conclusion, Section 5 presents related work.

2 Background

In this section, we briefly introduce definitions and notations used hereafter.

2.1 Constraint Satisfaction Problems

Definition 1 *A Constraint Satisfaction Problem (CSP) is a triple* (X, D, C) *where,*

- $X = \{X_1, X_2, \ldots, X_n\}$ *represents a set of* n *variables.*
- $D = \{D_1, D_2, \ldots, D_n\}$ *represents the set of associated domains, i.e., possible values for the variables.*
- $C = \{C_1, C_2, \ldots, C_m\}$ *represents a finite set of constraints.*

[1] Since our standpoint is real settings, we consider the full application stack where solvers are not isolated pieces of software called from a command line, but are critically embedded in large and complex applications.

Each constraint C_i is associated to a set of variables $vars(C_i)$, and is used to restrict the combinations of values between these variables. Similarly, the degree $deg(X_i)$ of a variable is the number of constraints associated to X_i and $dom(X_i)$ corresponds to the current domain of X_i.

Solving a CSP involves finding a solution, i.e., an assignment of values to variables such as all constraints are satisfied. If a solution exists the problem is stated as satisfiable and unsatisfiable otherwise.

In this paper, we consider four well known variable selection heuristics. *min-dom* selects the variable with the smallest domain [4], *wdeg* [2] selects the variable which is involved in more failed constraints, *dom/wdeg* [2] which selects the variable which minimizes the ratio $\frac{dom}{wdeg}$ and *impacts* [7] whose objective is to select the variable-value pair that maximizes the reduction of the remaining search space.

2.2 Support Vector Machines

Among the prominent Machine Learning (ML) algorithms are *Support Vector Machines* (SVM) [3]. This algorithm is highly used in binary classification due to its statistical learning properties, it determines the separating hyperplane with maximum distance or margin to the closest examples (so-called support vectors) of the training set.

Learning a high quality model depends on the quality of the training set. On the one hand, the description of the examples must enable to discriminate among positive and negative examples; on the other hand, the available examples must enable to accurately localize the frontier between the two classes.

3 Continuous Search in Constraint Programming

The goal of this paper is to take advantage of real world situations as shown in Figure 1, where instances are presented one at a time. Therefore, in Continuous Search settings we consider two different phases, exploitation (or solving time) and exploration (or learning time). The former tries to solve new instances using the acquired knowledge, and the latter is focused on improving the classification accuracy by means of re-using previously seen instances.

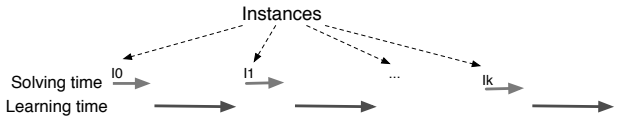

Fig. 1. Continuous Search scenario

3.1 Online Learning

To deal with the continuous search scenario, we follow the same approach as in [1], where the authors propose to use a supervised Machine Learning algorithm to select the most appropriate heuristic at different states (so-called checkpoints) of the search tree. To this end, we characterize CSP instances by means of features

(i.e., general information common to all CSPs). Those features are the input of an Online SVM algorithm which selects the best heuristic within the checkpoint window.

The features set is divided into two main categories, *static* and *dynamic*. The former intends to distinguish instances from each other (e.g., number of variables, constraints, etc.), while the latter is used to monitor the progress of the search process (e.g., max. number of failures, variable's weight, etc.). For more details about these features see [1].

Our choice of an Online SVM algorithm is motivated by the fact that it does not need to re-train the classifier once a new example arrives. Note that training a classical SVM (so-called batch learning) requires the solution of an optimization problem which is not an ideal situation in Continuous Search.

4 Experiments

This section describes the experimental validation of the proposed approach. In these experiments we included a collection of 100 *nurse-scheduling problems* from the MiniZinc[3] repository.

4.1 Experimental setting

The learning algorithm used in the experimental validation of the proposed approach is a Support Vector Machine with Gaussian kernel; we used the libSVM implementation. All our CSP heuristics (see Section 2.1) are home-made implementations integrated in the Gecode-2.1.1 constraint solver.

ID	Variable sel	Value Sel	ID	Variable sel	Value sel
1	**dom/wdeg**	**min**	5	dom/deg	min
2	dom/wdeg	max	6	dom/deg	max
3	wdeg	min	7	min-dom	min
4	wdeg	max	8	impacts	—

Table 1. Candidate heuristics; the default heuristic is the first one.

Two CSP adaptive strategies have been experimented, respectively considering the first 4 and 8 strategies in Table 1. In all cases, the default heuristic is the first one: *dom/wdeg* for variable selection and *min-value* for value selection.

The exploration examples are generated by adding minor perturbations to the default execution of heuristics (i.e., executing the default heuristic at each checkpoint). Thus, during the learning time, we ran the last seen instance replacing the default heuristic by another candidate in exactly one checkpoint, this process is repeated for each heuristic in Table 1 for a limited number of checkpoints (10 in this paper). Currently, we are working on a more informative way of selecting the exploration points considering the distance of unlabeled examples to the decision boundary.

All experiments were performed on a 8-machine cluster running Linux Mandriva 2009, all machines have 64 bits and two quad-core 2.33 Ghz with 8 Gb of RAM. A time out of 10 minutes was used for each experiment.

[3] Available at http://www.g12.cs.mu.oz.au/minizinc/download.html

It can be observed in Figure 2 that the dynamic approach is able to solve more instances that its competitors (i.e., the default strategy and a random heuristic selection). However the performance goes down as the number of candidate heuristics increases. The main explanation for this phenomenon relies on the fact that we are not using any sophisticated strategy for breaking ties (i.e., if several heuristics are predicted to outperform the default one we pick one at random). We are currently studying different approaches to breaking ties, selecting the best algorithm using the decision value, exploiting the fact that examples that are far from the classification boundaries are more likely to be correctly predicted.

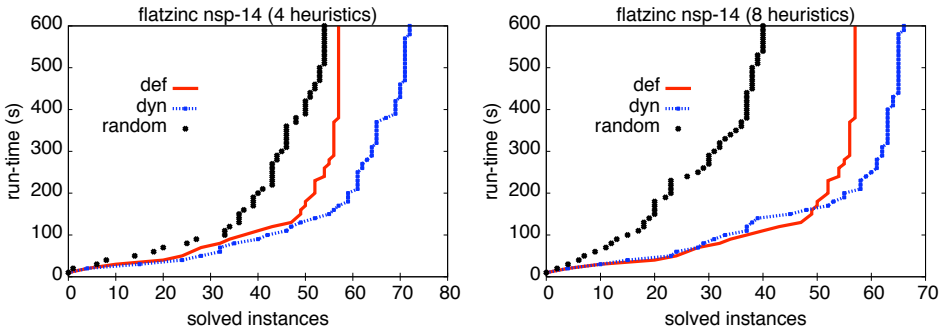

Fig. 2. Nurse Scheduling-14 (nsp-14), Note that this data shows the performance of the continuous search approach with a particular ordering of the problem instances

5 Related work

In this section, we describe some related work that has been proposed to integrate Machine Learning Algorithms into CSP and related areas such as: SAT and *Quantified Boolean Formulas* (QBF).

SATzilla [10] is a well known SAT portfolio solver which is built upon a set of features, in general words SATzilla includes two kinds of features: basic features such as number of variables, number of propagators, etc. and local search features which actually probe the search space in order to estimate the difficulty of each problem-instance. The goal of SATzilla is to learn a runtime predictor using a simple linear regression model.

CPHydra [6], one of the best constraint solvers in the lastest CSP competition[4] is a portfolio approach based on case-based reasoning. Broadly speaking CPhydra maintains a database with all solved instances (so-called *cases*). Later on, once a new instance arrives a set of similar cases C is computed and the heuristic that is able to solve the majority of instances in C is selected. The main drawback of this portfolio approach is that due to its high complexity to

[4] http://www.cril.univ-artois.fr/CPAI08/

select the best solver, it is limited to a small number of solvers (in competition settings less than 6 solvers were used).

Our work is related to [8] in a way that they also apply machine learning techniques to perform on-line combination of heuristics into search tree procedures. Their paper proposes to use a multinomial logistic regression method in order to maximize the probability of predicting the right heuristic at different states of the search procedure. Unfortunately, this work requires an important number of training instances to get enough generalization of the target distribution of problems and does not fulfill all the requirements of the Continuous Search settings.

6 Conclusion

This paper has presented an introduction to Continuous Search, this new concept includes an online algorithm to adaptively tune a constraint solver. At different states of the search, the instance feature is provided with dynamic information collected by the search engine to dynamically adapt the search strategy of a well known CP solver in order to more efficiently solve the current instance. The results in this paper show that the online approximation to deal with continuous search outperforms a very good default heuristic for solving CSPs.

7 Acknowledgements

We would like to thank the anonymous reviews and Michele Sebag for helpful discussions of the integration of Machine Learning and Constraint Programming.

References

1. A. Arbelaez, Y. Hamadi, and M. Sebag. Online heuristic selection in constraint programming. In *Symposiumon Combinatorial Search (SoCS)*, 2009.
2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *ECAI'04*, 2004.
3. N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, 2000.
4. R. M. Haralick and G. L. Elliott. Increasing tree search efficientcy for constraint satisfaction problems. In *Artificial Intelligence*, 1980.
5. F. Hutter and Y. Hamadi. Parameter adjustment based on performance prediction: Towards an instance-aware problem solver. Number MSR-TR-2005-125, Cambridge, UK, Jan 2005. Microsoft-Research.
6. E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *AICS'08*, 2008.
7. P. Refalo. Impact-based search strategies for constraint programming. In *CP'04*, 2004.
8. H. Samulowitz and R. Memisevic. Learning to solve qbf. In *AAAI'07*, 2007.
9. K. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 41(1), 2008.
10. L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla-07: The design and analysis of an algorithm portfolio for sat. In *CP'07*, 2007.