HAL
open science

Continuous Search in Constraint Programming

Alejandro Arbelaez, Youssef Hamadi, Michèle Sebag

To cite this version:

Alejandro Arbelaez, Youssef Hamadi, Michèle Sebag. Continuous Search in Constraint Programming. 22th International Conference on Tools with Artificial Intelligence, Oct 2010, Arras, France. inria-00515137

HAL Id: inria-00515137

https://inria.hal.science/inria-00515137v1

Submitted on 17 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Continuous Search in Constraint Programming

Alejandro Arbelaez
Microsoft-INRIA joint-lab,
Orsay, France
alejandro.arbelaez@inria.fr

Youssef Hamadi
Microsoft Research,
Cambridge, United Kingdom,
LIX École Polytechnique,
F-91128 Palaiseau, France
youssefh@microsoft.com

Michele Sebag
Project-team TAO,
INRIA Saclay Île-de-France,
LRI (UMR CNRS 8623),
Orsay, France
michele.sebag@inria.fr

***Abstract*—This work presents the concept of Continuous Search (CS), which objective is to allow any user to eventually get their constraint solver achieving a top performance on their problems. Continuous Search comes in two modes: the functioning mode solves the user's problem instances using the current heuristics model; the exploration mode reuses these instances to train and improve the heuristics model through Machine Learning during the computer idle time. Contrasting with previous approaches, Continuous Search thus does not require that the representative instances needed to train a good heuristics model be available beforehand. It achieves lifelong learning, gradually becoming an expert on the user's problem instance distribution. Experimental validation suggests that Continuous Search can design efficient mixed strategies after considering a moderate number of problem instances.**

I. 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. Additional tricks, e.g. adding redundant and channeling constraints, or using some global constraint (depending on your constraint solver) which can efficiently do part of the job, are required to achieve efficiency. Such tricks are far from being obvious, unfortunately; they do not change the solution space, and users with a classical mathematical background might find it hard to see why adding redundancy helps.

For this reason, 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. Parameter tuning indeed appears to be conceptually simple (i/ try different parameter settings on representative problem instances, ii/ pick up the setting yielding best average performance). Still, most users would easily consider instances which are not representative of their problem, and get misled.

The goal of the presented work is to allow any user to eventually get their constraint solver achieving a top performance on their problems. The proposed approach is based on the original concept of Continuous Search (CS), gradually building a heuristics model tailored to the user's problems, and mapping a problem instance onto some appropriate parameter setting. A main contribution compared to the state of the art (see [23] for a recent survey; more in section III) is to relax the requirement of a large set of representative problem instances to be available beforehand to support offline training. The heuristics model is initially empty (set to the initial default parameter setting of the constraint solver) and it is enriched along a lifelong learning approach, exploiting the problem instances submitted by the user to the constraint solver.

Formally, CS interleaves two functioning modes. In production or exploitation mode, the instance submitted by the user is processed by the constraint solver; the current heuristics model is used to parameterize the constraint solver depending on the instance at hand. In learning or exploration mode, CS reuses the last submitted instance, running other heuristics than the one used in production mode in order to find which heuristics would have been most efficient for this instance. CS thus gains some expertise relative to this particular instance, which is used to refine the general heuristics model through Machine Learning (section II-B). During the exploration mode, new information is thus generated and exploited in order to refine the heuristics model, in a transparent manner: without requiring the user's input and by only using the idle computer's CPU cycles.

The paper claim is that the CS methodology is realistic (most computational systems are always on, especially production ones) and compliant with real-world settings, where the solver is critically embedded within large and complex applications. The CS computational cost must be balanced against the huge computational cost of offline training [28]. Finally, lifelong learning appears a good way to construct an efficient and agnostic heuristics model, and able to adapt to new modelling styles or new classes of problem.

The paper is organized as follows. Background material is presented in Section II. Section III introduces the Continuous Search paradigm. Section IV details the proposed algorithm. Section V reports on its experimental validation. Section VI discusses related work and the paper concludes with some perspectives for further studies.

II. Background and Notations

This section briefly introduces definitions used in the rest of the paper.

A. Constraint Satisfaction Problems

A constraint Satisfaction Problem (*CSP*) is a triple (X, D, C) where, X represents a set of variables, D a set

of associated domains (i.e., possible values for the variables) and C a finite set of constraints.

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. A depth-first search backtracking algorithm can be used to tackle CSPs. At each step of the search, an unassigned variable X and a valid value v for X are selected, the exploration of variables/values is combined with a look-ahead strategy able to narrow the domains of the variables and reduce the remaining search space through constraint propagation. Restarting the search engine [12], [17] helps to reduce the effects of early mistakes in the search process. A restart is done when some cutoff limit in the number of failures (backtracks) is met (i.e., at some point in the search tree), before restarting the search each heuristic stores its ranking metrics in order to start the next tree-based search.

In this paper, we consider five well known variable selection heuristics. *min-dom* [14] selects the variable with the smallest domain, *wdeg* [4] selects the variable which is involved in the highest number of failed constraints, *dom-deg* selects the variable which minimizes the ratio $\frac{dom}{deg}$, *dom-wdeg* [4] selects the variable which minimizes the ratio $\frac{dom}{wdeg}$ and *impacts* [20] selects the (variable, value) pair which maximizes the reduction of the remaining search space. While only deterministic heuristics will be considered in this paper, the proposed approach can be extended to randomized algorithms by following the approach proposed in [15].

B. Supervised Machine Learning

Supervised Machine Learning exploits data labelled by the expert to automatically build hypotheses emulating the expert's decisions [25]. Only the binary classification case will be considered in the following. Formally, a learning algorithm processes a training set $\mathcal{E} = \{(x_i, y_i), x_i \in \Omega, y_i \in \{1, -1\}, i = 1 \ldots n\}$ made of n examples (x_i, y_i), where x_i is the example description (e.g. a vector of values, $\Omega = \mathbb{R}^d$) and y_i is the associated label; example (x, y) is referred to as positive (respectively, negative) iff y is 1 (resp., -1). The learning algorithm outputs a hypothesis $f : \Omega \mapsto Y$ associating to each example description x a label $y = f(x)$ in $\{1, -1\}$. Among ML applications are pattern recognition, ranging from computer vision to fraud detection [18], game playing [11], or autonomic computing [21].

Among the prominent ML algorithms are *Support Vector Machines* (SVM) [7]. Linear SVM considers real-valued positive and negative instances ($\Omega = \mathbb{R}^d$) and constructs the separating hyperplane which maximizes the margin, i.e. the minimal distance between the examples and the separating hyperplane. The margin maximization principle provides good guarantees about the stability of the solution and its convergence towards the optimal solution when the number of examples increases.

The linear SVM hypothesis $f(x)$ can be described from the sum of the scalar products between the current instance x and some of the training instances x_i, called support vectors:

$$f(x) = < w, x > +b = \sum \alpha_i < x_i, x > +b$$

The SVM approach can be extended to non-linear spaces, by mapping the instance space Ω into a more expressive feature space $\Phi(\Omega)$. This mapping is made implicit through the so-called *kernel trick*, by defining $K(x, x') = < \Phi(x), \Phi(x') >$; it preserves all good SVM properties provided the kernel be positive definite. Among the most widely used kernels are the Gaussian kernel ($K(x, x') = exp\{-\frac{||x-x'||^2}{\sigma^2}\}$) and the polynomial kernel ($K(x, x') = (< x, x' > +c)^d$). More complex separating hypotheses can be built on such kernels,

$$f(x) = \sum \alpha_i K(x_i, x) + b$$

using the same learning algorithm core as in the linear case. In all cases, a new instance x is classified as positive (respectively negative) if $f(x)$ is positive (resp. negative).

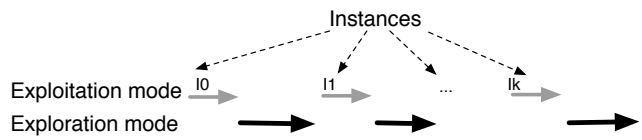


Figure 1. Continuous Search scenario

III. Continuous Search in Constraint Programming

The Continuous Search paradigm, illustrated on Figure 1, considers a functioning system governed from a heuristics model (which could be expressed as e.g., a set of rules, a knowledge base, a neural net). The core of continuous search is to exploit the problem instances submitted to the system along a 3-step process:

1) unseen problem instances are solved using the current heuristics model;
2) these instances are solved with other heuristics, yielding new information. This information associates to the description x of the example (accounting for the problem instance and the heuristics), a boolean label y (the heuristics improves/does not improve on the current heuristics model);
3) the training set $\mathcal{E}$, augmented with these new examples (x, y), is used to revise or relearn the heuristics model.

The Exploitation or production mode (step 1) aims at solving new problem instances as quickly as possible. The Exploration or learning mode (steps 2 and 3) aims at learning a more accurate heuristics model.

Definition 1: A continuous search system is endowed with a heuristics model, which is used as is to solve the current problem instance in production mode, and which is improved using the previously seen instances in learning mode.

Initially, the heuristics model of a continuous search system is empty, that is, it is set to the default settings of the search

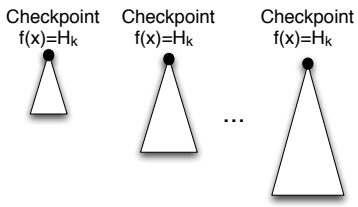


Figure 2. *dyn-CS*: selecting the best heuristic at each restart point

system. In the proposed CS-based constraint programming, the default setting is a given heuristics noted *DEF* in the following (section IV). Assumedly, *DEF* is a reasonably good strategy on average; the challenge is to improve on *DEF* for the particular types of instances which have been encountered in production mode.

IV. Dynamic Continuous Search

The Continuous Search paradigm is applied to a restart-based constraint solver, defining the *dyn-CS* algorithm. After a general overview of *dyn-CS*, this section details the different modules thereof.

Figure 2 depicts the general scheme of *dyn-CS*. The constraint-based solver involves several restarts of the search. A restart is launched after the number of backtracks in the search tree reaches a user-specified threshold. The search stops after a given time limit. Before starting the tree-based search and after each subsequent restarts, the description x of the problem instance is computed (section IV-A). We will call checkpoints the calculation of these descriptions.

In production mode, the heuristics model f is used to compute the heuristic $f(x)$ to be applied for the entire checkpoint window, i.e., until the next restart. Not to be confused with the *choice point* which selects a variable/value pair at each node in the search tree, *dyn-CS* selects the most promising heuristic at a given checkpoint and uses it for the whole checkpoint window. In learning mode, other combination of heuristics are applied (section IV-D) and the eventual result (depending on whether the other heuristics improved on heuristics $f(x)$) leads to build training examples (section IV-C). The augmented training set is used to relearn the heuristics model $f(x)$.

A. Representing instances: feature definition

At each checkpoint (or restart), the description of the problem instance is computed including static and dynamic features.

While a few of these descriptors had already been used in SAT portfolio solvers [15], [28], many descriptors had to be added as CSPs are more diverse than SAT instances: SAT instances only involve boolean variables and clauses, contrasting with CSPs using variables with large domains, and a variety of constraints and pruning rules.

1) Static Features: Encode the general description of a given problem instance; they are computed once for each instance as they are not modified along the resolution process. The static features also allow one to discriminate between types of problems, and different instances.

- **Problem definition** (4 features): Number of variables, constraints, variables assigned/not assigned at the beginning of the search.
- **Variables size information** (6 features): `Size prod`, `sum`, `min`, `max`, `mean` and `variance` of all variables domain size.
- **Variables degree information** (8 features): `min`, `max`, `mean` and `variance` of all variables degree (resp. variables' domain/degree)
- **Constraints Information** (6 features): The degree (or arity) of a given constraint c is represented by the total number of variables involved in c. Likewise the size of c is represented by the product of its corresponding variables domain sizes. Taking into account this information, the following features are computed `min`, `max`, `mean` of constraints size and degree.
- **Filtering cost category** (8 features): Each constraint c is associated a category[1]. In this way, we compute the number of constraints for each category. Intuitively each category represents the implementation cost of the filtering algorithm. *Cat* = {*Exponential*, *Cubic*, *Quadratic*, *Linear expensive*, *Linear cheap*, *Ternary*, *Binary*, *Unary*}. Where *Linear expensive* (resp. *cheap*) indicates the complexity of a linear equation constrain and the last three categories indicate the number of variables involved in the constraint. More information about the filtering cost category can be found in [10].

2) Dynamic features: Two kinds of dynamic features are used to monitor the performance of the search effort at a given checkpoint: global statistics describe the progress of the overall search process; local statistics check the evolution of a given strategy.

- **Heuristic criteria** (15 features): each heuristic criteria (e.g., *wdeg*, *dom-wdeg*, *impacts*) is computed for each variable; their `prod`, `min`, `max`, `mean` and `variance` over all variables are used as features.
- **Constraints weight** (12 features): likewise report the `min`, `max`, `mean` and `variance` of all constraints weight (i.e., constraints *wdeg*). Additionally the `mean` for each filtering cost category is used as feature.
- **Constraints information** (3 features): `min`, `max` and `mean` of constraint's *run-prop*, where *run-prop* indicates the number of times the propagation engine has called the filtering algorithm of a given constraint.
- **Checkpoint information** (33 features): for every checkpoint$_i$ relevant information from the previous checkpoint$_{i-1}$ (when available) is included into the feature vector. From checkpoint$_{i-1}$ we include the total number of nodes and maximum search depth. From the latest non-failed node, we consider the total number of assigned variables, satisfied constraints, sum of variables

[1]Out of 8 categories, detailed in http://www.gecode.org/doc-latest/reference/classGecode_1_1PropCost.html

wdeg (resp. size and degree) and product of variables degree (resp. *domain*, *wdeg* and *impacts*) of non assigned variables. Finally using the previous 11 features the `mean` and `variance` is computed taking into account all visited checkpoints.

The attributes listed above include a collection of 95 features.

B. Feature pre-processing

Feature pre-processing is a most important step in Machine Learning [26], which can significantly improve the predictive accuracy of the learned hypothesis. Typically, the descriptive features detailed above are on different scales; the number of variables and/or constraints can be high while the Impact of (variable, value) is between 0 and 1. A data normalization step, scaling down feature values in $[-1, 1]$ (*minmax-normalization*) is used.

Although selecting the most informative features might improve the performance, in this paper we do not consider any feature selection algorithm, and only features that are constant over all examples are removed as they offer no discriminant information.

C. Learning and using the heuristics model

The selection of the best heuristic for a given problem instance is formulated as a binary classification problem, as follows. Let $\mathcal{H}$ denote the set of k candidate heuristics, two particular elements in $\mathcal{H}$ being *DEF* (the default heuristics yielding reasonably good results on average) and *dyn-CS*, the (dynamic) ML-based heuristics model initially set to *DEF*.

Definition 2: Each training example $p_i = (x_i, y_i)$ is generated by applying some heuristics h ($h \in \mathcal{H}, h \neq$ *dyn-CS*) at some checkpoint in the search tree of a given problem instance. Description x_i ($\in \mathbb{R}^{97}$) is made of the static feature values describing the problem instance, the dynamic feature values computed at this check point and describing the current search state, and two additional features: *checkpoint-id* gives the number of checkpoints up to now and *cutoff-information* gives the cutoff limit of the next restart. The associated label y_i is positive iff the associated runtime (using heuristic h instead of *dyn-CS* at the current checkpoint) improves on the heuristics model-based runtime (using *dyn-CS* at every checkpoint); otherwise, label y_i is negative.
If the problem instance cannot be solved (whatever the heuristics used, i.e., time out during the exploration and exploitation modes), it is discarded (since the associate training examples do not provide any relevant information).

In production mode, the hypothesis f learned from the above training examples (their generation is detailed in next subsection) is used as follows:

Definition 3: At each checkpoint, for each $h \in \mathcal{H}$, the description x_h and the associated value $f(x_h)$ are computed. If there exists a single h such that $f(x_h)$ is positive, it is selected and used in the subsequent search effort. If there exists several heuristics with positive $f(x_h)$, the one with maximal value is selected[2]. If $f(x_h)$ is negative for all h, the default heuristic *DEF* is selected.

D. Generating examples in Exploration mode

The Continuous Search paradigm uses the idle computer's CPU cycles to explore different heuristic combinations on the last seen problem instance, and see whether one could have done better than the current heuristics model on this instance. The rationale for this exploration is that improving on the last seen instance (albeit meaningless from a production viewpoint since the user already got a solution) will deliver useful indications as to how to best deal with further similar instances. In this way, the heuristics model will expectedly be tailored to the distribution of problem instances actually dealt with by the user.

The CS exploration proceeds by slightly perturbing the heuristics model. Let *dyn-CS* $^{-i,h}$ denote the policy defined as: use heuristics model *dyn-CS* at all checkpoints except the i-th one, and use heuristic h at the i-checkpoint.

Algorithm 1 Exploration-time(instance: $\mathcal{I}$)

1: $\mathcal{E} = \{\}$ //initialize the training set
2: **for all** i in checkpoints($\mathcal{I}$) // loop over checkpoints ($\mathcal{I}$) **do**
3: **for all** h in $\mathcal{H}$ // loop over all heuristics **do**
4: Compute x describing the current checkpoint and h
5: **if** $h \neq$ *dyn-CS* **then**
6: Launch *dyn-CS* $^{-i,h}$
7: Define $y = 1$ iff *dyn-CS* $^{-i,h}$ improves on *dyn-CS* and -1 otherwise
8: $\mathcal{E} \leftarrow \mathcal{E} \cup \{x, y\}$
9: **end if**
10: **end for**
11: **end for**
12: return $\mathcal{E}$

Algorithm 1 describes the proposed Exploration mode for Continuous Search. A limited number (10 in this paper) of checkpoints in the *dyn-CS* based resolution of instance $\mathcal{I}$ are considered (line 2); for each checkpoint and each heuristic h (distinct from the *dyn-CS*), a lesion study is conducted, applying h instead of *dyn-CS* at the i-th checkpoint (heuristics model *dyn-CS* $^{-i,h}$); the example (described from the i-th checkpoint and h) is labelled positive iff *dyn-CS* $^{-i,h}$ improves on *dyn-CS*, and added to the training set $\mathcal{E}$, once the exploration mode for a given instance is finished the hypothesis model is updated by retraining the SVM including the feature pre-processing as stated in section IV-B.

E. Imbalanced examples

It is well known that one of the heuristics often performs much better than the others for a particular distribution of

[2]The rationale for this decision is that the margin, i.e. the distance of the example w.r.t the separating hyperplane, is interpreted as the confidence of the prediction [25].

problems [6]. Accordingly, negative training examples considerably outnumber the positive ones (it is difficult to improve on the winning heuristics). This phenomenon, known as *Imbalanced distribution*, might severely hinder the SVM algorithm [1]. Two simple ways of enforcing a balanced distribution in such cases, intensively examined in the literature and considered in earlier work [3], are to over-sample examples in the minority class (generating additional positive examples by Gaussianly perturbing the available ones) and/or undersample examples in the majority class.

Another options is to use prior knowledge to rebalance the training distribution. Formally, instead of labeling an example positive (resp, negative) iff the associated runtime is strictly less (resp. greater) than that of the heuristic model, we consider the difference between the runtimes. If the difference is less than some tolerance value dt, then the example is relabeled as positive.

The number of positive examples and hence the coverage of the learned heuristics model increase with dt; in the experiments (Section V), dt is set to 1 minute iff *time-exploitation* (time required to solve a given instance in production mode) is greater than 1 minute, otherwise dt is set to *time-exploitation*.

V. Experimental validation

This section reports on the experimental validation of the proposed Continuous Search approach. All tests were conducted on Linux Mandriva-2009 boxes with 8 GB of RAM and 2.33 Ghz Intel processors.

A. Experimental setting

The presented experiments consider 496 CSP instances taken from different repositories.

- **nsp**: 100 *nurse-scheduling* instances from the MiniZinc[3] repository.
- **bibd**: 83 *Balance Incomplete Block Design* instances from the XCSP[4] repository, translated into Gecode using Tailor[5]
- **js**: 130 *Job Shop* instances from the XCSP repository.
- **geom**: 100 *Geometric* instances from the XCSP repository.
- **lfn**: 83 *Langford-number* instances, translated into Gecode using global and channelling constraints.

The learning algorithm used in the experimental validation of the proposed approach is a Support Vector Machine with Gaussian kernel, using the libSVM implementation with default parameters[6]. All considered CSP heuristics (Section II) are home-made implementations integrated in the Gecode 2.1.1 [10] constraint solver. *dyn-CS* was used as a heuristics model on the top of the heuristics set $\mathcal{H} = \{$*dom-wdeg*, *wdeg*, *dom-deg*, *min-dom*, *impacts* $\}$, taking *min-value* as value selection heuristic. The cutoff value used to restart the search was initially set to 1000 and the cutoff increase policy to $\times 1.5$, the same cutoff policy is used in all the experimental scenarios.

Continuous Search was assessed comparatively to the best two dynamic variable ordering heuristics on the considered problems, namely *dom-wdeg* and *wdeg*. It must be noted that Continuous Search, being a lifelong learning system, will depend on the curriculum, that is the order of the submitted instances. If the user "pedagogically" starts by submitting informative instances first, the performance in the first stages will be better than if untypical and awkward instances are considered first. For the sake of fairness, the performance reported for Continuous Search on each problem instance is the median performance over 10 random orderings of the CSP instances.

B. Practical performances

Figure 3 highlights the Continuous Search results on Langford-number problems, comparatively to *dom-wdeg* and *wdeg*. The x-axis gives the number of problems solved and the y-axis presents the cumulated runtime. The (median) *dyn-CS* performance (grey line) is satisfactory as it solves 12 more instances than *dom-wdeg* (black line) and *wdeg* (light gray line). The dispersion of the *dyn-CS* results depending on the instance ordering is depicted from the set of dashed lines. Indeed traditional portfolio approaches such as [15], [22], [28] do not present such performance variations as they assume a complete set of training examples to be available beforehand.

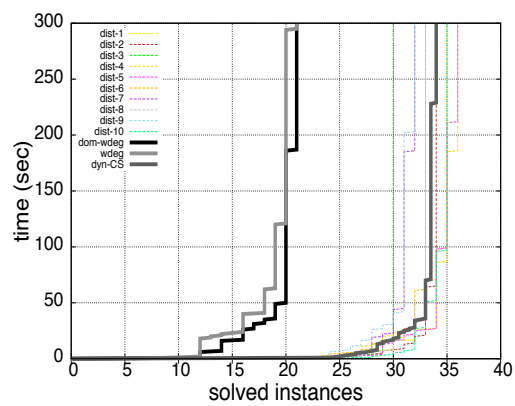
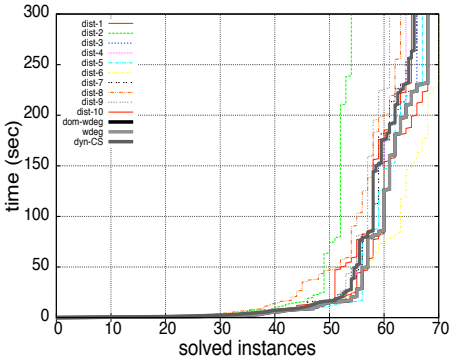


Figure 3. Langford-number (lfn): Number of instances solved in less than 5 min with *dyn-CS*, *wdeg*, and *dom-wdeg*. Dashed lines illustrate the performance of *dyn-CS* for a particular instance ordering.

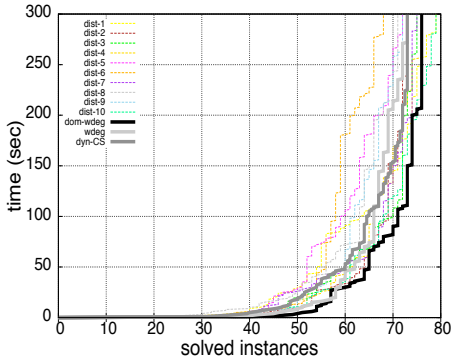
Figure 4 depicts the performance of *dyn-CS*, *dom-wdeg* and *wdeg* on all other problem families, respectively (bibd, js, nsp, and geom). On the bibd (Figure 4(a)) and js (Figure 4(b)) problems, the best heuristics is *dom-wdeg*, solving 3 more instances than *dyn-CS*. Note that *dom-wdeg* and *wdeg* coincide on bibd since all decision variables are boolean.

On nsp (Figure 4(c)), *dyn-CS* solves 9 more problems than *dom-wdeg*, but is outperformed by *wdeg* by 11 problems. On geom (Figure 4(d)), *dyn-CS* improves on the other heuristics,

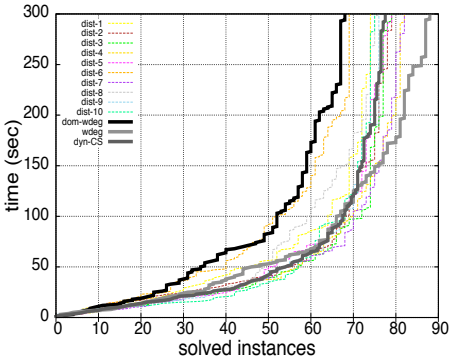
[3] http://www.g12.cs.mu.oz.au/minizinc/download.html
[4] http://www.cril.univ-artois.fr/∼lecoutre/benchmarks.html
[5] http://www.cs.st-andrews.ac.uk/∼andrea/tailor/
[6] http://www.csie.ntu.edu.tw/∼cjlin/libsvm/



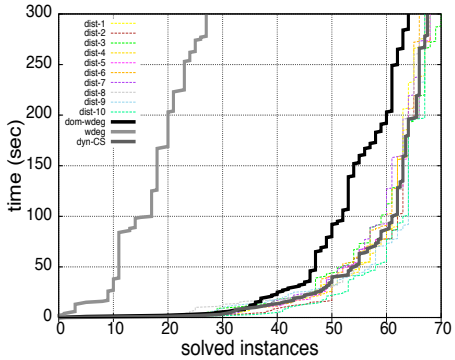
(a) Balance incomplete block designs (bibd)



(b) Job-shop (js)



(c) Nurse scheduling problem (nsp)



(d) Geometric (geom)

Figure 4. Number of instances solved in less than 5 minutes, with same legends as in Fig. 3.

solving respectively 3 more instances and 40 more instances than *dom-wdeg* and *wdeg*.

These results suggest that *dyn-CS* is most often able to pick up the best heuristics on a given problem family, and sometimes able to significantly improve on the best of the available heuristics.

All experimental results are summarized in Table I, reporting for each considered heuristics the number of instances solved (#sol), the total computational cost for all instances (time, in hour), and the average time (avg-time, in minutes) per instance, over all problem families. These results confirm that *dyn-CS* outperforms *dom-wdeg* and *wdeg*, solving respectively 18 and 41 instances more out of 315. Furthermore, it shows that *dyn-CS* is slightly faster than the other heuristics, with an average time of 2.11 minutes, against respectively 2.39 for *dom-wdeg* and 2.61 for *wdeg*. It is also worth mentioning that the total CPU time required to complete the exploration (or learning) mode after solving a given instance was on average no longer than 2 hours.

Additionally, a random heuristic selection scenario was also experimented (i.e., executing 10 times each instance with a uniform heuristic selection and reporting the median value over the 10 runs). The random selection strategy was able to solve 278 out of 496 instances, 19 instances less than *dom-wdeg* and 37 instances less than *dyn-CS*.

Another interesting lesson learned from the experiments concerns the difficulty of the underlying learning problem, and the generalization error of the learned hypothesis. The generalization error in the Continuous Search framework is estimated by 10-fold Cross Validation on the whole training set (including all training examples generated in exploration mode). Table II reports on the predictive accuracy of the SVM algorithm (with same default setting) on all problem families, with an average accuracy of 67%. As could have been expected, the predictive accuracy is correlated to the performance of Continuous Search: the problems with best accuracy and best performance improvement are geom and lfn.

To give an order of idea, 62% predictive accuracy was reported in the context of SATzilla [28], aimed at selecting of the best heuristic in a portfolio.

A direct comparison of the predictive accuracy might however be biased. On the one hand SATzilla errors are attributed to the selection of some near-optimal heuristics, after the

Table I
TOTAL SOLVED INSTANCES

Problem	*dom-wdeg*			*wdeg*			*dyn-CS*		
	#sol	time(h)	avg-time(m)	#sol	time(h)	avg-time(m)	#sol	time(h)	avg-time(m)
nsp	68	3.9	2.34	**88**	**2.6**	**1.56**	77	2.9	1.74
bibd	**68**	**1.8**	**1.37**	68	1.8	1.37	65	2.0	1.44
js	**76**	**4.9**	**2.26**	73	5.1	2.35	73	5.2	2.4
lfn	21	5.2	3.75	21	5.3	3.83	**33**	**4.1**	**2.96**
geom	64	3.9	2.34	27	6.8	4.08	**67**	**3.3**	**1.98**
Total	297	19.7	2.39	274	21.6	2.61	**315**	**17.5**	**2.11**

authors; on the other hand, Continuous Search would involve several selection steps (in each checkpoint) and could thus compensate from earlier errors.

Table II
PREDICTIVE ACCURACY OF THE HEURISTICS MODEL (10-FOLD CROSS VALIDATION)

bibd	nsp	geom	js	lfn
63.2%	58.8%	76.9%	63.6%	73.8%

C. The power of adaptation

Our second experimental test combines instances from different domains in order to show how CS is able to adapt to changing problems distribution. Indeed, unlike classical portfolio-based approaches which can only be applied if the training and exploitation sets come from the same domain, CS can adapt to changes and provide top performances even if the problems change.

Table III
TOTAL SOLVED INSTANCES

Problem	#Sol	time (h)	Problem	#Sol	time (h)
nsp-geom‡	55	4.1	lfn-bibd‡	23	5.3
nsp-geom†	67	3.4	lfn-bibd†	63	2.3

In this context, Table III reports the results on the geom (left) and bibd (right) problems by considering the following two scenarios. In the first scenario, we are going to emulate a portfolio-based search which would use the wrong domain to train. In *nsp-geom*‡, CS incrementally learns while solving the 100 nsp instances, and then solves one by one the 100 geom instances. However, when switching to this second domain, incremental learning is switched off, and checkpoints adaptation uses the model learnt on nsp. In the second scenario, *nsp-geom*† we solve nsp, then geom instances one by one, but this time, we keep the incremental learning on when switching from the first domain to the second one - as if CS was not aware of the transition.

As we can see in the first line of the Table, training on the wrong domain gives poor performance (55 instances solved in 4.1 hours). At contrary, the second line shows that CS can recover from training on the wrong domain thanks to its incremental adaptation (solving 67 instances in 3.4 hours). The right part of the Table reports similar results for the bibd problem.

As can be observed in nsp-geom† and lfn-bibd†, CS successfully identifies the new distribution of problems solving respectively the same number and 2 less instances than geom and bibd when CS is only applied to this domain starting from scratch. However the detection of the new distribution introduces an overhead in the solving time (see results for single domain in Table I).

VI. RELATED WORKS

This section briefly reviews and discusses some related works, devoted to heuristic selection within CP and SAT solvers.

SATzilla [28] is a well known SAT portfolio solver which is built upon a set of features. Roughly speaking SATzilla includes two kinds of basic features: general 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 for a given algorithm. The goal of SATzilla is to learn a run-time prediction function by using a linear regression model. In the same direction of SATzilla in [13] Haim et al., build the portfolio taking into account several restarts policies for a set of well known SAT solvers.

CPHydra [19] is a portfolio approach based on case-based reasoning; it maintains a database with all solved instances (so-called *cases*). Later on, once a new instance I arrives a set of similar cases C is computed and based on C it builds a switching policy selecting a set of CSP solvers that maximizes the possibilities of solving I within a given amount of time.

The approach most similar to the presented one is that of [22], who likewise apply Machine Learning techniques to perform on-line combination of heuristics into search tree procedures. Unfortunately, this work requires an important number of training instances to build a model with good generalization property.

In [5] low-knowledge is used to select the best algorithm in the context of optimization problems, this work assumes a black-box optimization scenario where the user has no information about the problem or even about the domain of the problem, and the only known information is the output (i.e., solution cost for each algorithm in the portfolio). Unfortunately this mechanism is only applicable to optimization problems and cannot be used to solve CSPs.

The purpose in *The Adaptive Constraint Engine* (ACE) [9] is to unify the decision of several heuristics in order to guide the search process. In this way, each heuristic votes for a possible

variable/value decision to solve a CSP. Afterwards, a global controller selects the most appropriate pair variable/value according to previously (offline) learnt weights associated to each heuristic. The authors however did not present any experimental scenario taking into account any restart strategy, although these nowadays are an essential part of constraint solvers

Combining Multiple Heuristics Online [24] and Portfolios with deadlines [27] are designed to build a scheduler policy in order to switch the execution of *black-box* solvers during the resolution process. However, in these papers the switching mechanics is learnt/defined beforehand, while our approach relies on the use of machine learning to on-the-fly switch the execution of heuristics.

Finally, in [2] and [16] the authors studied the automatic configuration problem which objective is to find the best parameters of a given algorithm in order to efficiently solve a class of problems.

VII. Discussion and Perspectives

The main contribution of the presented approach, the Continuous Search framework aims at designing a heuristics model tailored to the user problem distribution, allowing her to get top performance from the constraint solver. The representative instances needed to train a good heuristics model are not assumed to be available beforehand; they are gradually built and exploited to improve the current heuristics model, by stealing the idle CPU cycles of the computing system. Metaphorically speaking, the constraint solver uses its spare time to play against itself and gradually improve its strategy along time; further, this expertise is relevant to the real-world problems considered by the user, all the more so as it directly relates to the problem instances submitted to the system.

The experimental results suggest that Continuous Search is able to pick up the best of a set of heuristics on a diverse set of problems, by exploiting the incoming instances; in 2 out of 5 problems, Continuous Search swiftly builds up a mixed strategy, significantly overcoming all baseline heuristics. With the other classes of problems, its performance is comparable to the best two single heuristics. Our experiments also showed the capacity of adaptation of CS. Moving from one problem domain to another one is possible thanks to its incremental learning capacity. This capacity is a major improvement against classical portfolio-based approaches which only work when offline training and exploitation use instances from the same domain.

Further work will investigate the use of Active Learning [8] in order to select the most informative training examples and focus the exploration mode on the most promising heuristics. Another point regards the feature design; better features would be needed to get a higher predictive accuracy, governing the efficiency of the approach.

References

[1] R. Akbani, S. Kwek, and N. Japkowicz, "Applying support vector machines to imbalanced datasets," in *ECML*, ser. LNCS, vol. 3201. Pisa, Italy: Springer, Sept 2004, pp. 39–50.

[2] C. Ansótegui, M. Sellmann, and K. Tierney, "A gender-based genetic algorithm for the automatic configuration of algorithms," in *CP*, ser. LNCS, vol. 5732. Lisbon, Portugal: Springer, Sept 2009, pp. 142–157.

[3] A. Arbelaez, Y. Hamadi, and M. Sebag, "Online heuristic selection in constraint programming," in *International Symposium on Combinatorial Search (SoCS)*, Lake Arrowhead, USA, July 2009.

[4] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais., "Boosting systematic search by weighting constraints," in *ECAI*. Valencia, Spain: IOS Press, Aug 2004, pp. 146–150.

[5] T. Carchrae and J. C. Beck, "Applying machine learning to low-knowledge control of optimization algorithms," *Computational Intelligence*, vol. 21, no. 4, pp. 372–387, 2005.

[6] M. Correira and P. Barahona, "On the efficiency of impact based heuristics," in *CP*, ser. LNCS, vol. 5202. Sydney, Australia: Springer, Sept 2008, pp. 608–612.

[7] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, 2000.

[8] S. Dasgupta, D. Hsu, and C. Monteleoni, "A general agnostic active learning algorithm," in *NIPS*. Vancouver, British Columbia, Canada: MIT Press, Dec 2007.

[9] S. L. Epstein, E. C. Freuder, R. Wallace, A. Morozov, and B. Samuels, "The adaptive constraint engine," in *CP*, ser. LNCS, vol. 2470. NY, USA: Springer, Sept 2002, pp. 525–542.

[10] Gecode Team, "Gecode: Generic constraint development environment," 2006, available from `http://www.gecode.org`.

[11] S. Gelly and D. Silver, "Combining online and offline knowledge in UCT," in *ICML*, ser. ACM International Conference Proceeding Series, vol. 227. Corvalis, Oregon, USA: ACM, June 2007, pp. 273–280.

[12] C. Gomes, B. Selman, and H. Kautz, "Boosting combinatorial search through randomization," in *AAAI/IAAI*, 1998, pp. 431–437.

[13] S. Haim and T. Walsh, "Restart strategy selection using machine learning techniques," in *SAT*, ser. LNCS, vol. 5584. Swansea, UK: Springer, June 2009, pp. 312–325.

[14] R. M. Haralick and G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," in *IJCAI*, San Francisco, CA, USA, 1979, pp. 356–364.

[15] F. Hutter, Y. Hamadi, H. H. Hoos, and K. Leyton-Brown, "Performance prediction and automated tuning of randomized and parametric algorithms," in *CP*, ser. LNCS, vol. 4204. Nantes, France: Springer, Sept 2006, pp. 213–228.

[16] F. Hutter, H. H. Hoos, and T. Stützle, "Automatic algorithm configuration based on local search," in *AAAI*. Vancouver, British Columbia, Canada: AAAI Press, July 2007, pp. 1152–1157.

[17] H. A. Kautz, E. Horvitz, Y. Ruan, C. P. Gomes, and B. Selman, "Dynamic restart policies," in *AAAI/IAAI*, 2002, pp. 674–681.

[18] H. Larochelle and Y. Bengio, "Classification using discriminative restricted boltzmann machines," in *ICML*, ser. ACM International Conference Proceeding Series, vol. 307. Helsinki, Finland: ACM, June 2008, pp. 536–543.

[19] 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*, Aug 2008.

[20] P. Refalo, "Impact-based search strategies for constraint programming," in *CP*, ser. LNCS, vol. 3258. Toronto, Canada: Springer, Sept 2004, pp. 557–571.

[21] I. Rish, M. Brodie, and S. M. et al, "Adaptive diagnosis in distributed dystems," *IEEE Trans. on Neural Networks*, vol. 16, pp. 1088–1109, 2005.

[22] H. Samulowitz and R. Memisevic, "Learning to solve qbf," in *AAAI*. Vancouver, British Columbia: AAAI Press, July 2007, pp. 255–260.

[23] K. Smith-Miles, "Cross-disciplinary perspectives on meta-learning for algorithm selection," *ACM Comput. Surv.*, vol. 41, no. 1, 2008.

[24] M. Streeter, D. Golovin, and S. F. Smith, "Combining multiple heuristics online," in *AAAI*. Vancouver, British Columbia, Canada: AAAI Press, July 2007, pp. 1197–1203.

[25] V. Vapnik, *The Nature of Statistical Learning*. Springer Verlag, 1995.

[26] I. H. Witten and E. Frank, *Data Mining - Practical Machine Learning Tools and Techniques*. Elsevier, 2005.

[27] H. Wu and P. V. Beek, "Portfolios with deadlines for backtracking search," in *IJAIT*, vol. 17, no. 5, 2008, pp. 835–856.

[28] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "The design and analysis of an algorithm portfolio for sat," in *CP*, ser. LNCS, vol. 4741. Providence, RI, USA: Springer, Sept 2007, pp. 712–727.