HAL
archives-ouvertes.fr

Online Heuristic Selection in Constraint Programming

Alejandro Arbelaez, Youssef Hamadi, Michèle Sebag

▶ **To cite this version:**

Alejandro Arbelaez, Youssef Hamadi, Michèle Sebag. Online Heuristic Selection in Constraint Programming. International Symposium on Combinatorial Search - 2009. 2009. <inria-00392752>

HAL Id: inria-00392752
https://hal.inria.fr/inria-00392752

Submitted on 8 Jun 2009

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.

Online Heuristic Selection in Constraint Programming

Alejandro Arbelaez
MSR-INRIA Joint Centre,
Orsay, France
alejandro.arbelaez@inria.fr

Youssef Hamadi
Microsoft Research,
7 JJ Thomson Avenue,
CB3 0FB, Cambridge,
United Kingdom
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 paper presents our first attempt to apply Support Vector Machines to the problem of automatically tuning CP search algorithms. More precisely, we exploit instances features to dynamically adapt the search strategy of a CP solver in order to more efficiently solve a given instance. In these preliminary results, adaptation is restricted to restart points, and the number of times the strategy changes is also restricted. We report very encouraging results where our adaptation outperforms what is currently considered as one of the state of the art dynamic variable selection strategy.

1 Introduction

Constraint Programming (CP) is a powerful paradigm which allows the resolution of many complex problems, such as scheduling, planning, and configuration. One main feature of this formalism is its use of a glass-box approach to problem solving. Constraint solvers since the beginning are *opens*, and expose their parameters to a properly trained 'Constraint Programmer'. What seemed a correct standpoint in 1990, at a time where the number of applications was pretty small, is seen today as a major weakness (Puget 2004).

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

Because of that, most users are left with the tedious task of tuning the search parameters of their constraint solver, which both is time-consuming and requires some expertise.

This paper is interested in automatically tuning CP search algorithms using Machine Learning algorithms, more specifically Support Vector Machines (SVM) (Vapnik 1998). The problem instance and the current state of the search is described through a set of features inspired from (Hutter et al. 2009); the solution provided by the ML algorithm is used to select the CP strategy at each restart point. We report very encouraging results, showing that this dynamic adaptation of the search strategy can outperform what is currently considered as one of the state of the art dynamic variable selection strategy.

The paper is organized as follows. Background material related to CSP and Search are presented in Section 2. Section 3 introduces Machine Learning and Support Vector Machine algorithms. Section 4 presents the proposed ML-based approach for adaptive CP solving, and Section 5 reports on the experimental results. Section 6 discusses the approach w.r.t. related work, and Section 7 concludes with some research perspectives.

2 Background

This section introduces the CSP notations and the standard backtrack search strategy.

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.*

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.

2.2 The Search Strategy

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 (Schulte 2002). At each step of the search process, an unassigned variable x and a valid value v for x are selected and constraint $x = v$ is added to the search process. In case of unfeasibility, the search backtracks and can

undo previous decisions with new constraints, e.g., $x \neq v$. The search thus explores a so-called search tree, where each leaf-node corresponds to a solution. Clearly, in the worst-case scenario the search process requires to explore an exponential space. Therefore, it is necessary to combine the exploration of variables/values with a *look-ahead* strategy to narrow the domains of the variables and reduce the remaining search space through constraint propagation.

Restarting the search engine (Gomes, Selman, and Kautz 1998) is very efficient because it 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 is met (i.e., at some point in the search tree).

2.3 Search Heuristics

This section briefly reviews the basic ideas and principles behind the last generation of CSP heuristics. As we pointed out above, a CSP heuristic includes a variable/value selection procedure. Classical value ordering strategies can be summarized as follows: *min-value* selects the minimum value, *max-value* selects the maximum value and *mid-value* selects the median value in the remaining domain. Usually variable selection heuristics are more important and comprehend more sophisticated algorithms.

In (Boussemart et al. 2004), Boussemart et al. propose *wdeg* and *dom-wdeg* heuristics. The former one selects the variable that is most involved in failed constraints. A weight is associated to each constraint and incremented each time the constraint fails. Using this information *wdeg* selects the variable whose weight is maximal. The latter one, *dom-wdeg*, is a mixture of the current domain and the weight degree of the variable, choosing the variable that minimize the ratio $\frac{dom}{wdeg}$, where dom denotes the current size of the domain of the variable.

In (Refalo 2004), Refalo proposes the *impact* dynamic variable-value selection heuristic. The rationale of impact is to measure the size of the search space given by the Cartesian product of the variables (i.e., $|v_1| \times ... \times |v_n|$). Using this information the impact of a variable is averaged over all previous decisions in the search tree and the variable with the highest impact is selected.

It is also worth mentioning another category of dynamic variable heuristics that correspond to *min-dom* and *dom-deg*. The former so-called "First-Fail Principle: try first where you are more likely to fail" chooses the variable with minimum size domain, while the latter selects the variable with the smallest domain that is involved in most of the constraints (i.e., minimizing $\frac{dom}{deg}$).

3 Supervised Machine Learning

Supervised Machine Learning exploits data labelled by the expert to automatically build hypotheses emulating the expert's decisions (Vapnik 1995). Formally, given a training set $\mathcal{E} = \{(x_i, y_i), x_i \in X, y_i \in Y, i = 1 \ldots n\}$ made of n examples (x_i, y_i), where x_i is an instance (e.g. a vector of values) and y_i is the associated label, a learning algorithm builds a hypothesis $f : X \mapsto Y$ associating to each instance x a label $y = f(x)$ in Y. In the binary classification case, which will be considered in the following, the label space Y is binary; instance x is referred to as positive (respectively, negative) iff the associated label y is 1 (resp. 0). Among ML applications are pattern recognition, ranging from computer vision to fraud detection (Larochelle and Bengio 2008). game playing (Gelly and Silver 2007), or autonomic computing (Rish, Brodie, and et al 2005).

Among the prominent ML algorithms are *Support Vector Machines* (SVM) (Cristianini and Shawe-Taylor 2000). Linear SVM considers real-valued positive and negative instances ($X = \mathbb{R}^D$) and constructs the separating hyperplane which maximizes the margin (Fig. 3), 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.

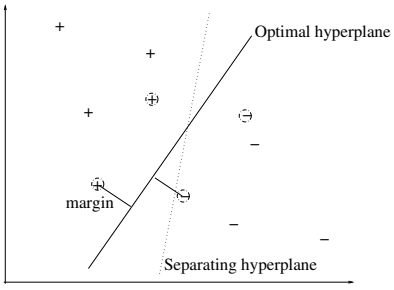


Figure 1: Linear Support Vector Machine. The optimal hyperplane is the one maximizing the minimal distance to the examples; the examples achieving this minimal distance are called support vectors.

The linear SVM hypothesis $h(x)$ 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 so-called *kernel trick* enables to extend the good SVMs properties beyond linear spaces, mapping the instance space X into a more expressive feature space $\Phi(X)$, provided that the scalar product in this feature space can be described in terms of a kernel on X. 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).

The quality of a hypothesis f is most often assessed from its accuracy on a test set (disjoint from the training set), i.e. the fraction of instances that are correctly classified by h ($f(x_i) = y_i$)). Learning a high quality hypothesis mostly 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 positive and the negative classes.

4 Online heuristic selection in CP

The goal of the paper is to tackle CSP heuristic selection as a supervised Machine Learning problem. The rationale for this approach is that the search for a universal (or killer) strategy is an idealistic fantasy: as stated in (Correira and Barahona 2008) one algorithm might stand out as the most promising one for a given family of problem-instances; however another algorithm might become more effective when considering another family of problems. For this reason, we thus aim at determining (learning) online the heuristics most appropriate to the current problem instance.

4.1 Characterizing CSP instances

In order to do so, a set of descriptive features are proposed to describe every CSP instance. In total, 57 features are considered, divided into *static* and *dynamic* features. *Static features* are computed once for each CSP instance. *Dynamic features* are dynamically updated while executing the backtracking algorithm, reflecting the search state and the performance of each particular heuristic at each node in the search tree. – It is important to note that a modification in the checkpoint policy might change the category of some features. –

Some of the presented features are those defined in SATzilla (Xu et al. 2007). However, since CSP is more general than SAT solving, defining suitable features to describe instances is a bit harder. For example, we have to face large domains, variety of constraints with varying computational cost and pruning capacity.

Static Features The main goal of this set of features is to distinguish one instance from another one, so that using these features we try to get an overall description of each problem-instance.

- # of variables, # of constraints, # assigned variables
- average, standard deviation min and max values of *dom/deg*, *dom*, *deg* grouping by variables.
- avarage, standard deviation of *dom*, *dom/deg* grouping by constraints.
- $log(deg(x_1), \times \ldots \times deg(x_n))$
- $log(dom(x_1) \times \ldots \times dom(x_n))$
- average, standard deviation min and max of constraint's arity

Dynamic Features Since we dynamically compute the best heuristic and we are using algorithms as a *glass-box*, we actually use heuristics information as features values. In this way, we gather the following statistic information as dynamic features:

- average, standard deviation, min (and max) values of *wdeg*, *dom/wdeg*, *impacts*, grouping by variables
- $log(wdeg(x_1) \times \ldots \times wdeg(x_n))$
- average, standard deviation, min (and max) values of *run-prop*, grouping by constraints, where *run-prop(c_i)* represents the number of times the propagation engine has called the filtering algorithm of c_i.

The computation of all these features can be obtained almost for free when maintaining each heuristic.

Features Pre-Processing One of the most important steps in Machine Learning systems is *data pre-processing* (Witten and Frank 2005), governing the accuracy of the learned hypothesis (especially when the number of examples is limited). The features that are constant over all examples are removed as they offer no discriminant information; other features are normalized using *minmax-normalization*, scaling down their value range to $[-1, 1]$. At this point it is important to mention that one of the key success factors in SATzilla system is its strong feature engineering phase, involving (i) the creation of compound features involving all pairwise combinations of features; (ii) the forward selection of the most informative subset of features.

4.2 The new search process

A modified backtrack search procedure is shown in Algorithm 1. The algorithm starts with the problem definition s and systematically adds (and removes) constraints in order to find a solution or to prove that the problem is unsatisfiable. *select-variable-h* and *select-value-h* functions (lines 5 and 6) are used to suggest a variable and a value according to a given heuristic h.

This algorithm also includes a procedure *predict-the-best-heuristic* (line 4): from the description of the current node in the search tree, the learned hypothesis is used to predict the heuristic most appropriate to that node. In practice it can be too time-consuming to determine the best heuristic at each node in the search tree; therefore a checkpoint policy is defined (parameterized by the user; e.g., set a checkpoint after every 30 failed nodes) and the best heuristic to be executed is computed after the learned hypothesis at each checkpoint.

Algorithm 1 backtracking-with-heuristic-selection(s)

```
1: if s = SOLUTION then
2:     return s
3: end if
4: predict-the-best-heuristic h ∈ {h_1, ..., h_k}
5: x ← select-variable-h(s)
6: v ← select-value-h(x)
7: add-and-propagate(x = v) to s
8: result ←backtracking-with-heuristic-selection(s)
9: if result = FAILURE then
10:    remove-and-backtrack(x = v) from s
11:    add-and-propagate(x ≠ v) to s
12:    return backtracking-with-heuristic-selection(s)
13: end if
14: return SOLUTION
```

Static Feat.	*Dynamic Feat.*	*checkpoint-id*	*Heu. Info*

Figure 2: Representation of a learning example

4.3 Predicting the best heuristic

As mentioned earlier on, the problem of selecting the best heuristic is formulated as a binary classification problem. Let $\mathcal{H}$ denote the set of k candidate heuristics, including the (supposedly best) default heuristics h_{def}.

Definition 2 *To each checkpoint in the search tree and each heuristic $h \neq h_{def}$ is associated a training example $p_i = (x_i, y_i)$. Description x_i ($x_i \in \mathbb{R}^d$) involves the static features describing the CSP instance, the dynamic features associated to the current state of the search, and the heuristic information made of k boolean features, all set to 0 except the one corresponding to the considered heuristic h. Label y_i is positive iff the runtime corresponding to using h at the current checkpoint is less than using the default heuristic h_{def}, otherwise y_i is negative.*

Fig. 2 shows the representation of the learning examples, including the heuristic information (*Heu*) and the checkpoint counter (*checkpoint-id*) giving the number of checkpoints up to now, thus indicating the age of the search and the difficulty of problem instance. Note that all CSP instances that cannot be solved by any of the strategies are removed from the training set.

4.4 Imbalanced examples

It is well known that one of the strategies often performs much better than the others for a particular distribution of problems (Correira and Barahona 2008). For this reason, negative examples considerably outnumber the positive ones. This phenomenon, known as *Imbalanced distribution* (Akbani, Kwek, and Japkowicz 2004), might severely hinder the learning algorithm: if the training set includes 99% examples of one class (say positive), then a hypothesis classifying all examples as positive can be considered as very accurate since its accuracy is 99%. To alleviate this problem, many approaches have been proposed in the Machine Learning litterature; the most widely used are based on modifying the distribution of the training examples using over or under-sampling:

- *over-sampling* proceeds by increasing the number of examples in the minory class; new examples generated by perturbing the examples in the minority class are generated.
- *under-sampling* proceeds by decreasing the number of examples in the majority class, to enforce an equal representation of positive and negative examples.

While classical approaches based on under- and over-sampling proceed by removing/perturbing examples uniformly selected in the training set, our approach was guided by prior knowledge about the learning goal. Actually, since the point is to discriminate between more and less effective heuristics, it was considered that the most informative examples are those for which the run-time of the challenging heuristics is very different from that of the defaut heuristics. The over-sampling approach thus selected with higher priority the most informative positive examples, to be perturbed and added to the training set; symmetrically, the under-sampling approach selected with higher priority the non-informative negative examples, to be removed. Experimentally (section 5), it will be seen that the *under-sampling* approach outperforms the *over-sampling* one.

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

Figure 3: Candidate heuristics; the default heuristics is the first one.

5 Experiments

This section describes the experimental validation of the proposed approach. The experimental setting is described before listing the considered CSP problem families, and reporting the empirical results.

5.1 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 (Chang and Lin 2001). All CSP heuristics are home-made implementations of the Gecode-2.1.1, presented in Section 2.

Three CSP adaptive strategies have been experimented, respectively considering the first 2, 4 and 8 strategies in Table 3. In all cases, the default heuristics is the first one, using *dom/wdeg* for variable selection and *min value* for value selection.

The training examples are generated by replacing the default heuristics by another candidate heuristics (Table 3) in exactly one checkpoint for each problem instance. It will be seen that replacing the default heuristics even a single time might greatly improve the performance of the search.

The learned hypothesis f was used at runtime as follows. At each checkpoint (restart), the instance(s) corresponding to the other candidate heuristics are considered; if a heuristics is deemed to be more efficient than the default one (instance classified as positive by f), then this heuristics is used instead of the default one. In case several heuristics are deemed to be more efficient, one of them is selected at random.

All experiments were repeated independently 10 times (same parameters and different random seeds, using 4-fold cross validation), averaging the results over the 10 runs. All the results were performed on a 8 machines cluster running

Linux Mandriva 2008, 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.

5.2 CSP problem families

We used three different sets of problems for our experiments. *quasi-group with holes* is a set of 100 instances, randomly generated using *lsencode* (Achlioptas et al. 2000) of order 28 with 321 holes each one. The second and third families of instances are a collection of 200 *nurse scheduling problems* from the MiniZinc-0.7.1 repository[1] named *nsp-14* and *nps-28*.

Nurse Scheduling Results related to the nsp-14 problem are presented in Figure 4(a). It shows the performance of the dynamic combination of heuristics against the default one. Here the dynamic approach is able to solve 3.2, 1.8 and 1.1 more instances when considering one, three and seven heuristics (from left to right).

Results related to nsp-28 problem (more complex than nsp-14) are presented in Figure 4(b). It shows that the dynamic strategy outperforms the default one solving on average 1.5 more instances for one heuristic, 0.2 more instances for three heuristics and 0.3 more instances for seven heuristics.

5.3 Quasigroups

Results related to the qwh problem are presented in Figure 4(c). It shows that the performance of the dynamic approach is very competitive when compared against the default heuristic. Here, on average the dynamic approach is able to solve 0.275 more instances when considering one heuristic, and the default strategy can solve 0.45 and 0.7 more instances taking into account three and seven heuristics respectively. We think that the homogeneous nature of these problems make them more challenging for our classifier.

It can be observed that in general the dynamic approach is able to solve more instances that the default one. However the performance goes down as the number of strategies increases. The main explanation of this phenomenon is that we are not using any sophisticated strategy for breaking ties. In case of ties (i.e., if several heuristics are predicted to outperform the default one) we pick one strategy at random. We are currently studying different approaches to break ties, selecting the algorithm with largest decision value as in One-Against-All SVM classifiers.

Our results in the integration of SVM into a constraint solver are promising, especially because up to now we are only considering replacing the default heuristic in a single restart. We are currently investigating to replace the default heuristic in more than a single restart and preliminary results are promising.

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

6 Related work

In this section, we describe previously proposed work that has been used in CSP and related areas such as: SAT and *Quantied Boolean Formulas* QBF.

SATzilla (Xu et al. 2007) 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 (O'Mahony et al. 2008), one of the best Constraint Solvers in the lastest CSP competition[2] is a portfolio approach based on case-based reasoning (Aamodt and Plaza 1994). 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 which 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 select the best solver, it is limited to a small number of solvers (in competition settings less than 6 solvers were used).

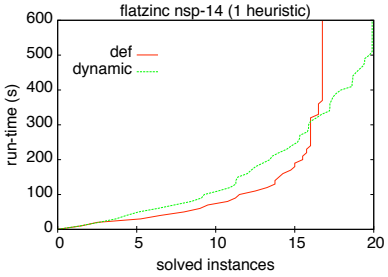
Our work is related to (Samulowitz and Memisevic 2007) in a way that they also apply machine learning techniques to perform on-line combination of heuristics into search tree procedures. This paper proposes to use a multinomial logistic regression method in order to maximize the probability of predicting 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.

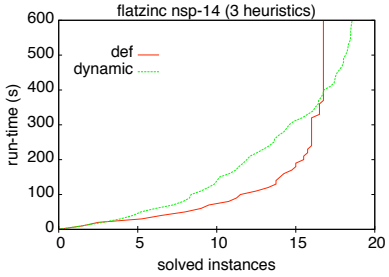
The Quickest First Principle (QFP) (James E. Borrett and Walsh 1995) is also a methodology for combining CSP heuristics. QFP relies on the fact that easy instances can frequently be solved by simple algorithms, while exceptionally difficult instances will require more complex heuristics. In this context, it is necessary to pre-define an execution order of heuristics and the switching mechanics is set according to the thrashing indicator, once the thrashing value of the current strategy reach some cutoff value, it becomes necessary to continue the search procedure with the following heuristic in the sequence. Despite QFP is a very elegant approach, this static methodology does not actually learn any information about solved instances.

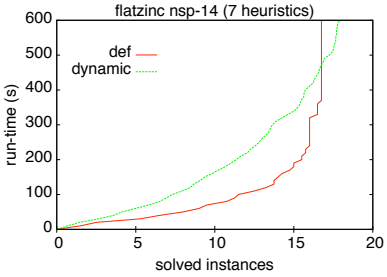
The purpose in *The Adaptive Constraint Engine* (ACE) (Epstein et al. 2002) is to unify the decision of several heuristics in order to guide the search process. In this way, each heuristic votes for a possible variable/value decision to solve a CSP, therefore, a global controller is going to select the most appropriate pair variable/value according to previously (off-line) learnt weights associated to each heuristic. Unfortunately, Epstein et al. did not present any experimental scenario taking into account any restart strategy which nowadays is vital in a constraint solver.

In (Carchrae and Beck 2005) authors propose to use low-knowledge information (general information common to all

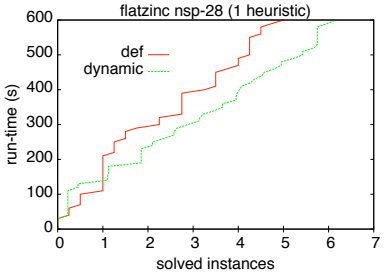
[2] http://www.cril.univ-artois.fr/CPAI08/

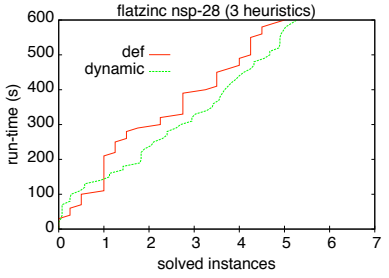


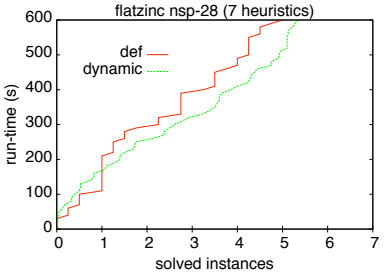




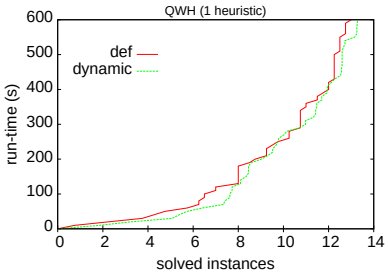
(a) Nurse Scheduling-14 (nsp-14)

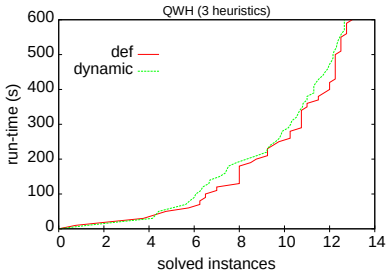


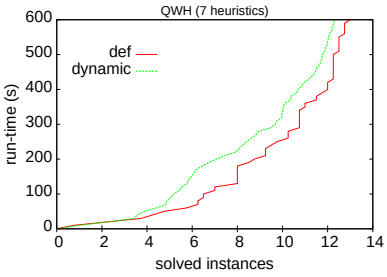




(b) Nurse Scheduling-28 (nsp-28)







(c) Quasi-groups (qwh)

optimization problems) in order to speed up the resolution process of optimization algorithms. Here the authors suggest to combine heuristics based on the quality of their solutions on time. In this way, heuristics that quickly produce better solutions are going to have more runnable time. It is important to note that the main different with ours, is that we are working with high-knowledge features to build the combination of heuristics, and this methodology is only applicability to optimization problems while ours is applicable to satisfiability and optimization problems.

Finally, Combining Multiple Heuristics Online (Streeter, Golovin, and Smith 2007) and Portfolio with deadlines for backtracking search (Wu and Beek 2008) are designed to build a scheduler policy in order to switch the execution of *black-box* solvers during the resolution process. However, in these related papers the switching mechanics is learnt/defined beforehand, while our approach relies on the use of machine learning algorithms to on-the-fly switch the execution of heuristics.

7 Conclusions and Future Work

This paper has presented a first attempt to use Machine Learning algorithms, specifically Support Vector Machines, to adaptively tune a CP search algorithm. At each restart, the instance features provide the dynamic information collected by the heuristics, e.g., weights for $dom - wdeg$ or impact. These features are used to dynamically adapt the search strategy of a well known CP solver in order to more efficiently solve the current instance.

In these preliminary results, adaptation is restricted to restart points, and the number of times the strategy changes is also restricted. First experimental results are very encouraging since our dynamic adaptation mechanism outperforms what is currently considered as one of the state of the art dynamic variable selection strategy.

Our choice of a Machine Learning approach is motivated by our long term goal of defining and implementing Continuous Search. In this paradigm, the solver uses all its idle time to generate new examples (e.g. by trying a candidate heuristics at some restart point) and improve its current hypothesis after the additional information gathered from these new examples. Accordingly, the current approach will be extended to incorporate online learning algorithms, incrementally refining and adapting the current hypothesis on the basis of the additional examples available.

References

Aamodt, A., and Plaza, E. 1994. Case-based reasoning: Foundational issues, methodological variations, and systems approaches. AI Communications.

Achlioptas, D.; Gomes, C. P.; Kautz, H. A.; and Selman, B. 2000. Generating satisfiable problem instances. In *AAAI'00*.

Akbani, R.; Kwek, S.; and Japkowicz, N. 2004. Applying support vector machines to imbalanced datasets. In *ECML'04*.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais., L. 2004. Boosting systematic search by weighting constraints. In *ECAI'04*.

Carchrae, T., and Beck, J. C. 2005. Applying machine learning to low-knowledge control of optimization algorithms.

Chang, C.-C., and Lin, C.-J. 2001. *LIBSVM: a library for support vector machines*. Software available at `http://www.csie.ntu.edu.tw/ cjlin/libsvm`.

Correira, M., and Barahona, P. 2008. On the efficiency of impact based heuristics. In *CP'08*.

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

Epstein, S. L.; Freuder, E. C.; Wallace, R.; Morozov, A.; and Samuels, B. 2002. The adaptive constraint engine. In *CP'02*.

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in UCT. In *Proc. International Conference on Machine Learning, ICML07*, 273–280. ACM Press.

Gomes, C.; Selman, B.; and Kautz, H. 1998. Boosting combinatorial search through randomization. In *AAAI'98*.

Hutter, F.; Hamadi, Y.; Hoos, H. H.; and Leyton-Brown, K. 2009. Performance prediction and automated tuning of randomized and parametric algorithms. *CP'09*.

James E. Borrett, E. P. K. T., and Walsh, N. R. 1995. Adaptive constraint satisfaction: The quickest first principle. Technical report.

Larochelle, H., and Bengio, Y. 2008. Classification using discriminative restricted boltzmann machines. In *Proc. Int. Conf. on Machine Learning, ICML08*, 536–543.

O'Mahony, E.; Hebrard, E.; Holland, A.; Nugent, C.; and O'Sullivan, B. 2008. Using case-based reasoning in an algorithm portfolio for constraint solving. 19th Conference on Artificial Intelligence and Cognitive Science.

Puget, J.-F. 2004. Constraint programming next challenge: Simplicity of use. In *CP'04*.

Refalo, P. 2004. Impact-based search strategies for constraint programming. In *CP'04*.

Rish, I.; Brodie, M.; and et al, S. M. 2005. Adaptive diagnosis in distributed dystems. *IEEE Trans. on Neural Networks* 16:1088–1109.

Samulowitz, H., and Memisevic, R. 2007. Learning to solve qbf. In *AAAI'07*.

Schulte, C. 2002. *Programming Constraint Services*. Springer-Verlag.

Streeter, M.; Golovin, D.; and Smith, S. F. 2007. Combining multiple heuristcs online. In *AAAI'07*.

Vapnik, V. 1995. *The nature of statistical learning theory*.

Vapnik, V. 1998. The statistical learning theory.

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

Wu, H., and Beek, P. V. 2008. Portfolios with deadlines for backtracking search.

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2007. Satzilla-07: The design and analysis of an algorithm portfolio for sat. In *CP'07*.