Branch and Bound Algorithm Selection by Performance Prediction

Lionel Lobjois and **Michel Lemaître**

ONERA-CERT/DCSD - ENSAE
2, avenue Édouard Belin – BP 4025 – 31055 Toulouse cedex 4 – France
{Lionel.Lobjois,Michel.Lemaitre}@cert.fr

Abstract

We propose a method called *Selection by Performance Prediction* (SPP) which allows one, when faced with a particular problem instance, to select a Branch and Bound algorithm from among several promising ones. This method is based on Knuth's sampling method which estimates the efficiency of a backtrack program on a particular instance by iteratively generating random paths in the search tree. We present a simple adaptation of this estimator in the field of combinatorial optimization problems, more precisely for an extension of the *maximal constraint satisfaction* framework. Experiments both on random and strongly structured instances show that, in most cases, the proposed method is able to select, from a candidate list, the best algorithm for solving a given instance.

Introduction

The Branch and Bound search is a well-known algorithmic schema, widely used for solving combinatorial optimization problems. A lot of specific algorithms can be derived from this general schema. These can differ in many ways. For example, one can use different static or dynamic orderings for variables and values. Likewise, the computation of a lower bound (in the case of minimization) at each branch node is often a compromise between speed and efficiency of the induced cut, and several variants are potentially appropriate. Thus, each algorithm is a combination of several particular features. It is generally difficult to predict the precise behavior of a combinatorial algorithm on a particular instance. In actual practice, one can observe that the range of computation times used by the candidate algorithms to solve a particular instance is often very wide. Faced with a particular instance to be solved, often in a limited time, one must choose an algorithm without being sure of making the most appropriate choice. Bad decisions may lead to unacceptable running times.

 This work was partially supported by the French Délégation Générale à l'Armement, under contract DRET 94/002 BC 47.
The authors thank Gérard Verfaillie and Thomas Schiex for helpful discussions during this work.

In this paper, we propose a method called *Selection by Performance Prediction* (SPP) to select, for each particular problem instance, the most appropriate Branch and Bound algorithm from among several promising ones. We restrict ourselves to the case of constraint optimization problems expressed in the Valued CSP framework (Schiex, Fargier, & Verfaillie 1995), which is an extension of the *maximal constraint satisfaction* framework, as explained in the next section. The proposed SPP method is based on an old and very simple idea (Knuth 1975) allowing one to statistically estimate the size of a search tree by iterative sampling. It gives surprisingly good results on both strongly structured and random problem instances. Estimating each candidate algorithm on the very instance to be solved is the key to a successful choice.

This paper is organized as follows. We first introduce the VCSP framework and describe Knuth's method of estimation. We show how this estimation can be used for Branch and Bound algorithms. Then we introduce the SPP method, and show some experimental results on both strongly structured and random problem instances. Lastly, after the review of some related works, we state our conclusions and discuss future directions.

Valued CSPs

A Constraint Satisfaction Problem (CSP) instance is defined by a triple (X, D, C), where X is a set of *variables*, D is a set of finite *domains* for the variables, and C is a set of *constraints*. A constraint is defined by a subset of variables on which it holds and by a subset of allowed tuples of values. A *solution* of an instance is a complete assignment — an assignment of values to all of the variables — which satisfies all of the constraints. Many CSP instances are so constrained that no solution exists. In this case, one can search for a solution maximizing the number of satisfied constraints. This is the maximal constraint satisfaction framework introduced by (Freuder & Wallace 1992). This framework can be further generalized by giving a weight or a *valuation* to each constraint, mirroring the importance one gives to its satisfaction. The cost of a complete assignment is the *aggregation* of the valuations of the unsatisfied constraints. We then search for

a solution minimizing this cost. This extension of the CSP model is called the Valued CSP (VCSP) framework (Schiex, Fargier, & Verfaillie 1995). In this paper, we only consider Σ-VCSPs, for which the aggregation operator is the ordinary sum. Algorithms for maximal constraint satisfaction (Freuder & Wallace 1992; Larrosa & Meseguer 1996) are easily extended to VCSPs.

Knuth's method of estimation

Knuth's method (Knuth 1975) is based on a statistical estimation of the quantity $\varphi \stackrel{\text{def}}{=} \sum_{x \in \text{nodes}(T)} f(x)$, where T is any tree. Among other quantities this method can estimate the number of nodes in a search tree ($f(x) = 1$), or the total running time ($f(x)$ being the time spent on the node x).

Let $S = \langle x_1, x_2, \ldots \rangle$ be a random path from the root x_1 to a terminal node, in which the successor of each internal node is randomly selected according to a uniform distribution. Let $\hat{\varphi}(S) \stackrel{\text{def}}{=} \sum_{x_i \in S} w(x_i) f(x_i)$, where $w(x_i) \stackrel{\text{def}}{=} \prod_{k=1}^{i-1} d(x_k)$, and $d(x_k)$ is the number of successors of x_k. $\hat{\varphi}$ is an unbiased estimate of φ. This is formally expressed as $\mathbf{E}\,\hat{\varphi} = \varphi$ (the expected value of the random variable $\hat{\varphi}$ is φ). The variance of $\hat{\varphi}$ is

$$\mathbf{V}\hat{\varphi} = \sum_{x \in \text{nodes}(T)} w(x) \sum_{1 \le i \le j \le d(x)} \left(\varphi(x^{(i)}) - \varphi(x^{(j)}) \right)^2 \quad (1)$$

where $x^{(i)}$ is the i^{th} successor of x, $\varphi(x) = \sum_{y \in \text{nodes}(T_x)} f(y)$, and T_x is the subtree rooted in x. The expression for the variance shows that it can be quite large, all the larger as the tree is unbalanced. Of course, one can get a better estimate of φ by repeatedly sampling the tree. Let $\hat{\varphi_n}$ be the mean of $\hat{\varphi}(S_i)$ over n successive random paths S_i. We still have $\mathbf{E}\,\hat{\varphi_n} = \varphi$, but the variance is now reduced to $\mathbf{V}\hat{\varphi_n} = \mathbf{V}\hat{\varphi}/n$. When sampling search trees, experiments show that the distribution of $\hat{\varphi}$ cannot be considered as a common one, hence it is difficult to provide a good confidence interval for $\hat{\varphi}$. However, Chebyshev's inequality[1] gives a confidence interval for φ with a probability of error less than $1/c^2$: $\Pr(|\hat{\varphi_n} - \varphi| \ge c\sqrt{\mathbf{V}\hat{\varphi}/n}) < 1/c^2$. In practice $\mathbf{V}\hat{\varphi}$ is unknown and must be estimated from the n random paths S_i using the well-known formula $\hat{\mathbf{V}}\hat{\varphi} = \frac{1}{n-1}\sum_{i=1}^{n}(\varphi(S_i) - \hat{\varphi_n})^2$.

In his paper, Knuth suggests a refinement called *importance sampling*, in which the successor of a node is selected according to a weighted distribution (instead of a uniform one), the weight of each successor being an estimate of the corresponding $\varphi(x^{(i)})$. Knuth's method has been improved in different ways by (Purdom 1978) and (Chen 1992). These improvements are based on a deep knowledge of the structure of problem instances.

[1] It can be used because it does not make any assumption on the actual distribution of the random variable $\hat{\varphi}$.

```
DFBB(ub_0)
    c* ← ub_0
    success ← false
    SEARCH(1)
SEARCH(i)
    if i ≤ nb-variables
    then v_i ← VARIABLE-CHOICE(i)
         for each value k in Current-Domain[v_i]
              A[v_i] ← k
              PROPAGATE(i)
              b ← BOUND(i)
              if b < c* then SEARCH(i + 1)
              UNPROPAGATE(i)
    else success ← true
         A* ← A
         c* ← COST(A)
```

Figure 1: Depth First Branch and Bound search.

In this paper, we choose to keep close to the original and simplest prediction method.

Estimating the Performance of a Branch and Bound Algorithm

In this section, we will show how Knuth's estimation method can be used to predict the running time of a Depth First Branch and Bound algorithm for solving a particular VCSP instance. This prediction is based on the estimation of the number of nodes in the tree developed during the search.

Figure 1 shows the pseudo-code of a Depth First Branch and Bound algorithm. It looks for a complete assignment $A^\star$ of minimal cost $c^\star$ less than an initial upper bound ub_0. If such an assignment does not exist (because ub_0 is less than or equal to the optimal cost) then the algorithm ends with *success* equal to **false**. The current partial assignment, involving variables $v_1, v_2, \ldots v_i$, is stored in $A[1..i]$. PROPAGATE(i) is a procedure which, like forward-checking, propagates the choices already made for the assigned variables onto the domains of the unassigned ones. This propagation may result in value deletions in the domains of future variables, and thus may improve the subsequent lower bound computation. BOUND(i) returns a lower bound of the cost of any complete extension of the current partial assignment. UNPROPAGATE(i) simply restores the domains.

Figure 2 shows the pseudo-code of the procedure ESTIMATE-NB-NODES(ub_0,n) which estimates the number of nodes that will be developed by the call DFBB(ub_0). The procedure VARIABLE-CHOICE used in both SEARCH (figure 1) and SAMPLE (figure 2) chooses the next variable, using an appropriate heuristic. It should be stressed that the structure of the SAMPLE procedure is simply obtained from the structure of the SEARCH procedure by changing the **for** loop into a single random value choice.

```
ESTIMATE-NB-NODES(ub_0,n)
    c* ← ub_0
    φ̂_n ← 0
    for j = 1 to n
        w ← 1
        φ̂ ← 0
        SAMPLE(1)
        φ̂_n ← φ̂_n + φ̂
    φ̂_n ← φ̂_n/n
    return φ̂_n
SAMPLE(i)
    if i ≤ nb-variables
    then v_i ← VARIABLE-CHOICE(i)
        k ← randomly select a value
              in Current-Domain[v_i]
        A[v_i] ← k
        w ← w · |Current-Domain[v_i]|
        φ̂ ← φ̂ + w
        PROPAGATE(i)
        b ← BOUND(i)
        if b < c* then SAMPLE(i + 1)
        UNPROPAGATE(i)
```

Figure 2: Estimating the size of a DFBB search tree through iterative sampling.

As mentioned by Knuth in his paper, "the estimation procedure does not apply directly to branch-and-bound algorithms". To better understand this, one should note that SEARCH updates the current upper bound $c^\star$ each time it finds a better complete assignment, thus allowing for a better subsequent pruning of the search tree. On the contrary, SAMPLE never updates its initial upper bound: it estimates the size of a tree which would be generated by a search process in which the upper bound remained constant. Hence, the sampled search tree does not correspond exactly to the actual search tree.

Search efforts between the regular version and the constant upper bound version of a Branch and Bound algorithm can differ tremendously. One extreme case occurs when the initial upper bound is set to infinity: whereas the regular version finds good solutions and consequently prunes its search tree, the constant upper bound version has to explore all complete assignments. On the other hand, if the initial upper bound is less than or equal to the optimal cost, both versions develop exactly the same tree. Eventually, good estimates need low upper bounds. In practice, one can execute an incomplete method like a local search first in order to get a low upper bound of the optimal cost. This upper bound will help the estimation process as well as the resolution itself.

The SAMPLE procedure of figure 2 makes it possible to estimate the size (number of nodes) of the search tree. However, we are in fact more interested in an estimate of the running time. A simple way of estimating this running time is to estimate first the average time

```
SPP(I,L,t_μ,t_s,ub_0)
  for each BB_i in L
      μ̂ ← ESTIMATE-TIME-PER-NODE(I,BB_i,t_μ,ub_0)
      φ̂_n ← ESTIMATE-NB-NODES(I,BB_i,t_s,ub_0)
      time_i ← μ̂ · φ̂_n
  return BB_i such as time_i is minimal
```

Figure 3: The Selection by Performance Prediction (SPP) method.

μ spent on a node. To do this, we run the target algorithm during a brief interval of time and then deduce an estimate of the average time per node. According to our experiments, such a simple procedure is sufficient to produce reasonable estimates.

Experiments with several instances and algorithms show that the variance of $\hat{\varphi}$ is generally very large. Hence, it seems difficult to produce useful confidence intervals for the number of developed nodes (and hence for the running time). The main contribution of this paper is to show empirically that the SPP method works well in practice despite this huge variance and the difference between sampled and actual search trees.

Selecting the Best Algorithm

In this section, we give a detailed description of the "Selection by Performance Prediction" method (SPP). Given an instance to be solved and a list of promising candidate Branch and Bound algorithms, we would like to select the best possible algorithm from among the candidates, that is, the algorithm which will solve the instance within the shortest time.

The principle of the method is very simple: we estimate the running time of each candidate algorithm on the instance; then we select the algorithm which gives the smallest expected running time. Figure 3 shows a pseudo-code of the proposed SPP method. $\mathcal{I}$ is the instance to be solved. $\mathcal{L}$ is the list of candidate algorithms. t_μ is the time allocated for estimating μ and t_s the time for estimating the size of each search tree. ub_0 is the initial upper bound used both for the actual search and for the estimation process.

ESTIMATE-TIME-PER-NODE($\mathcal{I}$,BB$_i$,t_μ,ub_0) runs the algorithm BB$_i$ for a time t_μ on the instance $\mathcal{I}$ using ub_0 as initial upper bound. It returns an estimate $\hat{\mu}$ of the average running time per node for BB$_i$. ESTIMATE-NB-NODES($\mathcal{I}$,BB$_i$,t_s,ub_0) samples during a time t_s the tree developed by the algorithm BB$_i$ using ub_0 as constant upper bound. It returns the mean value $\hat{\varphi_n}$ of the n random paths that have been generated. ESTIMATE-NB-NODES is similar to ESTIMATE-NB-NODES of figure 1 (however, while SAMPLE in figure 2 is given a fixed n, ESTIMATE-NB-NODES is given instead a time limit).

Although the SPP method is very simple, it works surprisingly well in practice. Two reasons may explain this success. First, the estimator is unbiased: it produces on average good estimates despite a huge vari-

ance. Second, the method does not depend on absolute performance predictions: since it compares the constant upper bound version of each candidate algorithm, pessimistic estimates due to a poor upper bound are pessimistic for all algorithms.

Experiments

In this section, we describe some experiments of the SPP method both on random and strongly structured instances[2]. We selected four well known algorithms as candidates. Good descriptions of these algorithms can be found in (Freuder & Wallace 1992), (Wallace 1994) and (Larrosa & Meseguer 1996). BB_1 is a *forward-checking* (P-EFC3) with the widely used dynamic variable ordering: *minimum current domain* as first heuristic and *decreasing degree* to break ties. Its value ordering is *increasing IC* (Freuder & Wallace 1992). BB_2, BB_3 and BB_4 are *forward-checking* with *Directed Arc Consistency Counts* for lower bound computation (P-EFC3+DAC2 described in (Larrosa & Meseguer 1996)). They all use *increasing IC + DAC* as value ordering but they differ on their static variable ordering: BB_2 uses *decreasing forward degree* (FD in (Larrosa & Meseguer 1996)) with *max-cardinality* (Tsang 1993, p 179) as second criteria, BB_3 uses *minimum width* (Tsang 1993, p 164) and BB_4 uses *decreasing degree*. Our experience on several problems shows that these algorithms appear to be among the best Branch and Bound algorithms available today for strongly structured instances.

Since we are mainly interested in solving realistic problems, we chose for these experiments the field of Radio Link Frequency Assignment Problems (RLFAP) (Cabon *et al.* 1998). We used sub-instances of CELAR instances 6 and 7 which are probably the two most difficult instances of the set[3]. The next table summarizes some properties of these sub-instances:

name	# of variables	# of values	# of constraints	optimal cost
$\mathcal{I}_1$	16	44	207	159
$\mathcal{I}_2$	14	44	300	2669
$\mathcal{I}_3$	16	44	188	10310

In the following experiments, we address three different cases depending on the initial upper bound. The first case corresponds to the common situation in which ub_0 is an upper bound of the optimal cost provided by a simple local search. In the second case, ub_0 is the optimal cost itself. Such a situation may occur when the optimal cost is easily found by a local search but there is no proof of its optimality. In the last case, we try to prove that ub_0 is a lower bound of the optimal cost. This may be helpful to bound the optimal cost when it is impracticable (de Givry & Verfaillie 1997).

[2]All experiments have been done on a SUN Sparc5 with 64 Mo of RAM using CMU Common Lisp.

[3]The original instances and the sub-instances are available at `ftp://ftp.cs.unh.edu/pub/csp/archive/code/benchmarks/FullRLFAP.tgz`.

We ran SPP 5000 times on each instance using $\mathcal{L}$=$\{BB_1,BB_2,BB_3,BB_4\}$, $t_\mu = 1$ second and $t_s = 3$ seconds. In order to check predictions we then ran the complete algorithms using upper bounds given to SPP. The next table shows, for each instance and each algorithm, the running time in seconds of the complete search using the given upper bound and *nbc*, the number of times the algorithm was selected. For instance, algorithm BB_3 solved instance $\mathcal{I}_1$ to optimality in 377 seconds and was selected 4660 times among the 5000 runs of SPP:

	$\mathcal{I}_1$		$\mathcal{I}_2$		$\mathcal{I}_3$	
ub_0	159		2000		10413	
opt	159		2669		10310	
	time	*nbc*	time	*nbc*	time	*nbc*
BB_1	580000[6]	0	1200	4939	2116	441
BB_2	1961	170	18149	31	3200	229
BB_3	377	4660	40299	0	2162	881
BB_4	1015	170	18271	30	1010	3449

These experimental results are very encouraging: for all instances, SPP is able to find the best algorithm in the majority of runs. When it does not select the best algorithm, it generally selects a good one and rarely the worst. As could be guessed, SPP is unable to distinguish two algorithms which have close running times. More generally, the more the actual running times differ, the easier it is for SPP to select the best algorithm.

To give a more precise idea of the quality of the SPP method, we propose to compare it with two alternative approaches in terms of expected running time. The first approach, RANDOM, makes a random choice in the list of the candidate algorithms. When several algorithms seem to be suitable for the instance, one can pick one of them at random. The expected running time one obtains using this approach is simply the mean of the four running times. The second approach, INTERLEAVED, runs all the candidate algorithms in an interleaved way on the same processor as proposed in (Huberman, Lukose, & Hogg 1997; Gomes & Selman 1997). The first algorithm which finishes the search stops the others. For this approach, the expected running time is four times the running time of the best algorithm. We approximated the expected running time one obtains using SPP with the simple formula $\sum nbc_i \cdot t_i / \sum nbc_i$, where nbc_i is the number of times SPP selects algorithm BB_i and t_i is the actual running time of the complete solving using BB_i.

Figure 4 compares the running times of each algorithm and the expected running time using RANDOM, INTERLEAVED and SPP on our three instances. For SPP, we added the cost of the estimation process which is $4 \cdot (1+3) = 16$ seconds to the expected running time (this appears in a darker grey).

According to these experiments, SPP definitely outperforms RANDOM and INTERLEAVED approaches on these instances. In each case, the expected running

[6]This time is not the actual running time of BB_1 on $\mathcal{I}_1$, but an estimation using $t_\mu = 3$ minutes and $t_s = 1$ hour.

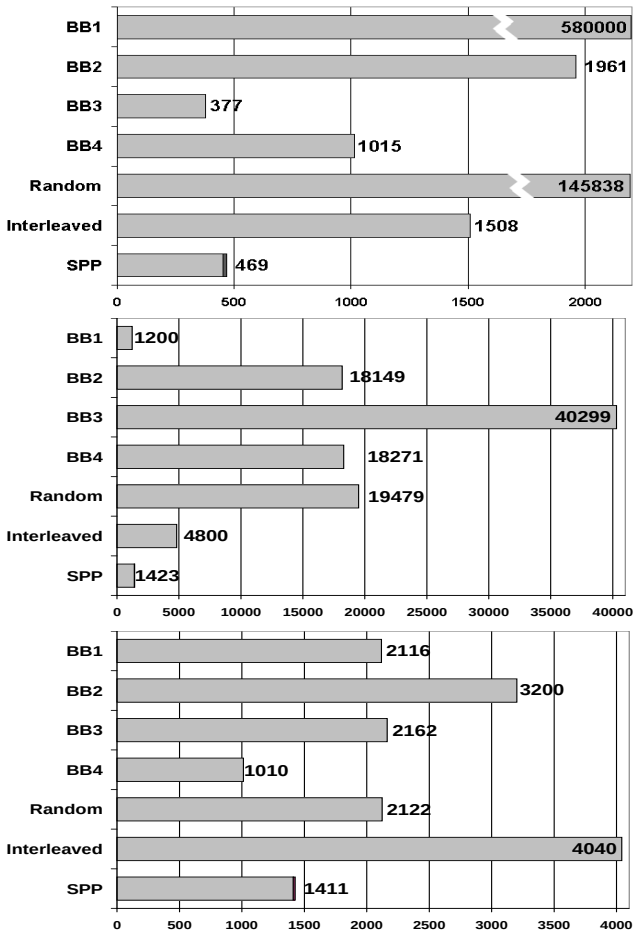

Figure 4: Expected running times using different approaches when solving instances $\mathcal{I}_1$, $\mathcal{I}_2$ and $\mathcal{I}_3$.

time using SPP is very close to the running time of the best algorithm: wrong selections have a small influence on the expected running time since they occur rarely.

To validate our approach on more instances, we experimented with the SPP method on a set of random instances. The goal of the experiment was to solve sequentially all instances of the set as quickly as possible. For this experiment, we chose to tackle the case where there are only two promising candidates for solving the whole set, neither algorithm clearly dominating the other. We restricted $\mathcal{L}$ to $\{BB_1, BB_2\}$ and generated 238 instances according to the model described in (Smith 1994) modified to allow valued constraints. These instances contain 20 variables and 10 values per variable; the graph connectivity is 50% and the tightness of each constraint is 90%. Constraints are uniformly valued in the set $\{1, 10, 100, 1000, 100000\}$. With such parameters, both algorithms have nearly equal chances to be the best.

For each instance, we first ran a simple Hill-Climbing to find an upper bound. Then we ran BB_1 and BB_2 on each instance with the given upper bound and recorded

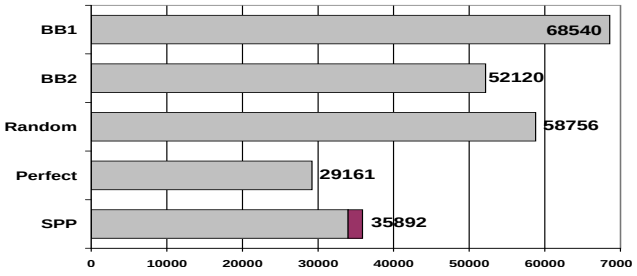

Figure 5: Cumulated running times on 238 random instances ($t_\mu = 1$, $t_s = 3$)

their running times: 103 instances were best solved by BB_1 as opposed to 135 for BB_2. Finally, we used approaches RANDOM and SPP to select an algorithm for each instance. Figure 5 shows the cumulated running times using, for each instance, algorithm BB_1, BB_2, the one selected by RANDOM, and the one selected by SPP[4]. To emphasize the performance of the SPP method, we also show the cumulated running times one would obtain using the hypothetical perfect selection method (a method which could choose the best algorithm for each instance).

One important point stressed by this last experiment is that it is better to use the algorithm selected by the SPP method for each instance, than to use the best algorithm on average for all instances. As a matter of fact, large experiments on a class of instances may indicate which algorithm is the best on average for this class. Nevertheless, our experimental results clearly show that each instance has to be solved with an appropriate algorithm. Moreover, SPP seems to be a very accurate selecting method since it is close to a perfect one, at least on this set of instances.

To summarize, experimental results confirm the interest of the SPP method. It allows one to use an appropriate algorithm, avoiding exceptional behaviors which can lead to unacceptable running times. Hence, it may save great amounts of time even when the best algorithm for the class of the instance is known.

Related work

(Bailleux & Chabrier 1996) estimate the number of solutions of constraint satisfaction problem instances by iterative sampling of the search tree.

In the context of a telescope scheduling application, (Bresina, Drummond, & Swanson 1995) use Knuth's sampling method to estimate the number of solutions which satisfy all hard constraints. But the main use of the sampling method is for statistically characterizing scheduling problems and the performance of schedulers. A "quality density function" provides a background against which schedulers can be evaluated.

[4]For SPP we added its running time which is $2 \cdot (1+3) = 8$ seconds per instance.

Works mentioned below do not make use of Knuth's estimator, but are related to this work in some way.

An adaptive method, aiming at automatically switching to good algorithms, has been proposed by (Borret, Tsang, & Walsh 1996). Despite similar goals, the adaptive method and the sampling method are different. The former one is based on a thrashing prediction computed during the regular execution of an algorithm. When such a thrashing is likely to occur, another algorithm is tried sequentially. Conversely, the SPP method, once its choice made, runs only one algorithm: the one which has the best chance of success.

Heading in yet a different direction, both (Gomes, Selman, & Crato 1997) and (Rish & Frost 1997) show that, in the case of random unsatisfiable CSPs, the *log-normal* distribution is a good approximation of the distribution of computational effort required by backtracking algorithms. We, too, observed a "heavy-tailed" distribution for the random variable $\hat{\varphi}$, but were unable to identify it. Note that the distribution of $\hat{\varphi}$ on a particular instance and the distribution of φ on a *class* of instances are two different distributions.

Algorithm portfolio design (Huberman, Lukose, & Hogg 1997; Gomes & Selman 1997) aims at combining several algorithms by running them in parallel or by interleaving them on a single processor.

(Minton 1996) addresses the problem of specializing general constraint satisfaction algorithms and heuristics for a particular application.

Conclusion and future work

We have proposed a simple method for selecting a Branch and Bound algorithm from among a set of promising ones. It is based on the estimation of the running times of those algorithms on the particular instance to be solved. We provided experimental results showing that the SPP method is a cheap and effective selection method. This efficient performance has been empirically demonstrated, in the field of constraint optimization problems, both on random and strongly structured problem instances.

Clearly, improvements on the proposed method must be sought in the estimation process itself. A better knowledge of the structure of problems to be solved would probably make it possible to better estimate running times. Improvements like *importance sampling* (Knuth 1975), *partial backtracking* (Purdom 1978) or *heuristic sampling* (Chen 1992) merit further investigations into the field of constraint optimization problems.

There is no doubt that this method can also be applied to inconsistent CSP instances, because a proof of inconsistency implies a complete search as well. Besides, it would be interesting to investigate the application of the proposed method to consistent CSP instances.

Experimental results clearly show that each instance is best solved with a particular algorithm. This confirms the interest of adapting general algorithms to suit each instance.

References

Bailleux, O., and Chabrier, J. 1996. Approximate Resolution of Hard Numbering Problems. In *Proc. AAAI-96*.

Borret, J. E.; Tsang, E. P. K.; and Walsh, N. R. 1996. Adaptive Constraint Satisfaction : The Quickest First Principle. In *Proc. ECAI-96*, 160–164.

Bresina, J.; Drummond, M.; and Swanson, K. 1995. Expected Solution Quality. In *Proc. IJCAI-95*, 1583–1590.

Cabon, B.; de Givry, S.; Lobjois, L.; Schiex, T.; and Warners, J. 1998. Benchmark Problems: Radio Link Frequency Assignment. *To appear in Constraints*.

Chen, P. 1992. Heuristic Sampling: a Method for Predicting the Performance of Tree Searching Programs. *SIAM Journal on Computing* 21(2):295–315.

de Givry, S., and Verfaillie, G. 1997. Optimum Anytime Bounding for Constraint Optimization Problems. In *Proc. AAAI-97*.

Freuder, E., and Wallace, R. 1992. Partial Constraint Satisfaction. *Artificial Intelligence* 58:21–70.

Gomes, C. P., and Selman, B. 1997. Practical aspects of algorithm portfolio design. In *Proc. of Third ILOG International Users Meeting*.

Gomes, C. P.; Selman, B.; and Crato, N. 1997. Heavy-Tailed Distributions in Combinatorial Search. In *Proc. CP-97*, 121–135.

Huberman, B. A.; Lukose, R. M.; and Hogg, T. 1997. An economics approach to hard computational problems. *Science* 275:51–54.

Knuth, D. 1975. Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation* 29(129):121–136.

Larrosa, J., and Meseguer, P. 1996. Expoiting the Use of DAC in MAX-CSP. In *Proc. CP-96*, 308–322.

Minton, S. 1996. Automatically Configuring Constraint Satisfaction Programs : A Case Study. *Constraints* 1:7–43.

Purdom, P. 1978. Tree Size by Partial Backtracking. *SIAM Journal on Computing* 7(4):481–491.

Rish, I., and Frost, D. 1997. Statistical Analysis of Backtracking on Inconsistent CSPs. In *Proc. CP-97*, 150–162.

Schiex, T.; Fargier, H.; and Verfaillie, G. 1995. Valued Constraint Satisfaction Problems : Hard and Easy Problems. In *Proc. IJCAI-95*, 631–637.

Smith, B. 1994. Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In *Proc. ECAI-94*, 100–104.

Tsang, E. 1993. *Foundations of Constraint Satisfaction*. London: Academic Press Ltd.

Wallace, R. 1994. Directed Arc Consistency Preprocessing. In *Proc. of the ECAI-94* Constraint Processing *workshop (LNCS 923)*. Springer. 121–137.