Progressive Focusing Search

Nicolas Prcovic [1] and **Bertrand Neveu** [2]

Abstract. This paper deals with value ordering heuristics used in a complete tree search algorithm for solving binary constraint satisfaction problems. Their aim is to guide the search towards a solution. First, we show the limits of the traditional prospective approach, which uses the size of the domains of the still unassigned variables. In an advantageous context, where arc consistency is maintained and allows the time spent by the dynamic value ordering to be negligible, the speedup is poor when the problems are hard. Then, we present a new value ordering heuristic based on a learning-from-failure scheme. Instead of making a choice a priori, an interleaving search follows every sub-tree to gather information. After this learning phase, the algorithm focuses on the most promising one. This new algorithm, named Progressive Focusing Search, is compared to Interleaved Depth First Search and appears to be efficient for problems on the phase transition complexity peak.

1 Introduction

For solving a Constraint Satisfaction Problem (CSP) on finite domains by a complete method, tree search algorithms based on depth first search (DFS) are commonly used. In order to limit the combinatorial explosion, different improvements of the standard backtracking algorithm have been proposed. The main improvements are domains filtering, backjumping and the use of variable ordering heuristics.

For finding only one solution, value ordering heuristics can also be used, but they are generally considered to be less important for reducing the search tree. We will first present in section 2 some existing value ordering heuristics. Until now, the main idea for choosing a value is to study the effects of a choice in a prospective approach, as does the Promises solutions heuristic (MP)[6]. The effects of instantiating the current variable with every value are partially propagated, before choosing the most promising one. We propose another approach based on a learning from failure scheme, using some knowledge gathered during the search. In section 3, we show the limits of the MP value ordering heuristic. The gains due to this heuristic are impressive when the problems have many solutions, but become negligible at the peak of complexity. The reason is the following : the heuristic only has a partial information at the beginning of the search and can make disastrous mistakes.

We propose in section 4 a new kind of heuristics for the first choices. We generate a subpart of each subtree where the first variables are instantiated by each value in their domain. The information gathered during such beginning of the search in each subtree is then used to choose the most promising subtree to be first entirely explored. We define a new algorithm named Progressive Focusing Search (PFS), that implements this idea. In section 5, we compare it with Interleaved Depth First Search (IDFS) [10] on random CSPs.

[1] LSIS, Université Aix-Marseille III, France
[2] INRIA-CERMICS, Route des lucioles, BP93, 06902 Sophia-Antipolis, France

2 Previous work on value ordering heuristics

In a standard tree search algorithm, two choices are made at each step. First, the next variable to instantiate is chosen. The problem is then decomposed into k subproblems, one for each value in the domain of that variable. The value choice determines thus which subproblem will be explored first by DFS.

The effects of these two choices are complementary. The variable choice has an effect on the search tree itself and the value choice on the branches that will be explored. The main goal in using variable ordering heuristics is to build a tree as small as possible, i.e. limiting the branching factors and in case of a choice leading to a failure, finding this failure as soon as possible [15]. The hybrid heuristics as *min-domain+degree* and *min-domain/degree*[1] based on the domain size and the degree of the variables mix these two effects.

The goal of using value ordering heuristics is to avoid making a bad choice, i.e. selecting a subtree without solution. Therefore, value ordering heuristics use partial information to select the value to assign to the current variable. This information can be computed once at the beginning of the solving process or dynamically during the search. The main information used by the dynamic heuristics is the size of the domains of the future variables (i.e. the yet unassigned variables). Domain sizes vary indeed when the search algorithm, such as MAC[3][13], performs a filtering in order to maintain arc consistency.

This information is used in diverse ways in the following heuristics. *LVO-MC* [4] selects the value with the greatest sum of numbers of compatible values in the domains of future variables. *Highest support* [9] selects the value with the greatest sum of normalized sizes of future variable domains by taking into account the inconsistencies. With *Max-domain-size* [5], the values are ordered in the decreasing order of the minimum domain size of the future variables. The idea is to avoid fragile subproblems with small domains. *ES Estimate solutions* [3] estimates the number of potential solutions by solving a tree relaxation of the remaining problem. *Promises solutions* (MP) [6] is a simple case of the preceding heuristic : the relaxation of the remaining problem is the problem made of only the constraints connecting the current variable to future variables. The value with the greatest product of domain sizes of future variables is chosen.

For computing for each value of the current variable the domain sizes of future variables, one has to propagate the effect of the variable instantiation with this value. In other words, one has to perform the propagation made by the Forward-checking algorithm in

[3] In this paper, when we will refer to the MAC algorithm, it will be only the degree of filtering performed (arc-consistency) when a value is assigned to a variable and not the binary tree feature of the algorithm.

branches that maybe would not be explored, which can be costly.

Therefore, several static value ordering heuristics have been defined, where the values are ordered before the search. The static least-conflicts (SLC) heuristics [5] orders the values, by the number of conflicts, in which they take part, the static max-promises (SMP), orders them by the product of number of compatible values, and the mean field heuristic performs some preprocessing based on Mean Field Theory [2].

The main part of reported results have been obtained with the Forward-checking [7] algorithm. It was observed that a dynamic value ordering heuristic has an important overhead and becomes useful only when the problem is difficult enough (i.e. needs more than 1000000 constraint checks in [4]). But, now, the problems are mostly solved by a MAC-like algorithm maintaining his strong level of consistency, and we will see that the overhead to compute a dynamic heuristic and reorder the domains becomes negligible.

No experimental study has definitively showed that one value ordering heuristic is better than the others. We decided to study the behavior of the promises solutions (MP) heuristic, which gives the number of solutions in a subtree, if there were no constraints between the future variables. We also decided to use MAC : the information about the domain sizes of future variables is then more accurate than with FC and the quality of the MP heuristic should increase.

3 Effects of the MP value ordering heuristic

We performed some experiments on binary random CSPs. We restrained the tests in zones where CSPs have solutions and where the problems are not too easy. When MAC alone solves a problem without any backtracking, there is no need of value ordering heuristics. These heuristics have also no reason to be used for problems without solutions.

3.1 Experimental results on random CSPs

We performed some tests using the C++ CSP Library by Hulubei [8], a library for solving binary CSPs: we implemented the static max-promises heuristic (SMP) and the dynamic max-promises heuristic (MP). In order to study the importance of the value choice for the first variable, we also performed tests where the max-promises heuristic is used only for the first choice, all the others choices being done in the natural ordering of the domains : we named this first choice heuristic H1 and H0 the algorithm run without any value ordering heuristic.

The tests were performed on random problems, using the generator of the library. A problem class is defined by 4 parameters : the number of variables N, the size of the domains S, the ratio of existing constraints, determined by the density parameter D, and the ratio of forbidden value pairs (tightness parameter T).

First, we ran tests with problems made of 50 variables and a domain size equal to 15. We made the density and the tightness parameters vary, in order to obtain difficult problems, while being solvable in a reasonable amount of time (a few minutes per problem instance).

For each parameter set, we performed tests with the H0, H1, SMP, and MP heuristics. The variable ordering heuristic used was the dynamic *min-domain/degree* heuristic, which selects the variable having the minimum of the domain size divided by the degree, the degree being the number of constraints between the variable to be chosen and the future variables. The filtering algorithm was the built-in arc consistency filtering *AC3*.

We report in the left part of table 1 the depth of the highest backtrack that occurred for a set of 100 problems with the different heuristics. A result $i(k)$ means that the $i-1$ first choices were good choices for all the 100 problem instances and it was not needed to backtrack on them to find a solution and that k problems among 100 needed to backtrack at this level i. This can be seen as a measure of the distance from the peak of complexity, corresponding to the phase transition zone, between solvable and unsolvable problems [14]. When the i reported for the H0 algorithm is deep enough, it means that for all problem instances, we did not meet any problem where the $i-1$ first choices caused a failure. We are then in a zone with many solutions. Conversely, when backtracks occur at the first level, i.e. for some problem instances the first choice led to a subtree without solution, the problems are close to the peak of complexity.

The result presented in the right part of table 1 is the total cpu time in minutes spent for finding one solution in all the 100 problem instances for each parameter set (we used a Pentium III 500).

Table 1. Depth of highest backtrack (left), with the number of problems with a backtrack at this depth, and cpu time in mn (right) spent for solving 100 instances with N = 50, S= 15, density D and tightness T.

D,T	H0	H1	SMP	MP	H0	H1	SMP	MP
.89,.10	4(3)	4(1)	7(7)	7(3)	1143	996	147	149
.75,.12	4(9)	4(4)	6(9)	6(1)	1140	980	196	156
.52,.16	4(2)	4(3)	6(7)	7(9)	133	130	39	21
.41,.20	2(2)	3(5)	4(1)	5(11)	323	206	61	57
.40,.20	4(16)	4(3)	5(2)	5(1)	72	39	13	11
.32,.25	2(20)	2(2)	2(1)	3(6)	337	187	98	83
.31,.25	2(2)	3(5)	4(4)	4(3)	66	42	17	10
.30,.25	4(5)	4(3)	5(1)	6(2)	10	6	2	2

We can see that the MP heuristic is very efficient for all these problems, that are far enough from the peak of complexity. The gain obtained is a factor between 4 and 8. The overhead due to the computation of the heuristic is in fact negligible : the dynamic version of the heuristic is almost always better than the static one. We can also remark in analyzing the H1 results that a significant part of the gain is often due to the first choice.

3.2 Inefficiency at the peak of complexity

A second set of tests has been performed closer to the peak of complexity with domain sizes of 15 and 8. Then, the gain factor due to the use of the MP heuristic goes down to 1.4 or 1.2 (see table 2).

As noted before, the subtree with the highest number of promises has the greater probability to contain solutions. But, this subtree has also the probability to have the greatest size and when the heuristic fails, the damage is greater.

Still in the zone of problems with solutions, but closer to the peak of complexity, such a failure occurs higher in the tree and can have disastrous effects. This can explain why the heuristic is inefficient near the peak, although the cases where it makes a mistake are much fewer than those obtained by H0. Let us analyze in detail the results near the peak, for example for $S = 15$, $D = 0.20$, and $T = 0.36$, the heuristic makes 18 erroneous choices for selecting the value for the first variable. Without heuristic, an error occurs 50 times at the first level in the set of 100 problem instances. But the longest time needed for solving any instance is 14 minutes with the heuristic and 2 minutes without the heuristic. We can also remark that these erroneous first choices are critical for the global time for solving the 100 instances. Indeed, the heuristic has no effect in subtrees with no solutions : these subtrees, when they are chosen, are to be entirely explored.

In the problems far enough from the peak in the feasible zone there always appeared a difference in the minimum depth levels where a mistake is done by the algorithm with and without the heuristic. The dynamic heuristic, informed by the former choices in the current branch makes fewer mistakes and is more efficient.

Table 2. Results at the complexity peak for problems with 50 variables, a domain size S, a density D and a tightness T.

S,D,T	H0	H1,SMP,MP	H0	H1	SMP	MP
15,.25,.31	1(38)	1(16)	592	512	407	502
15,.20,.36	1(50)	1(18)	140	124	126	121
15,.14,.44	1(44)	1(12)	18	12	10	8
8,.50,.13	1(51)	1(26)	149	109	107	108
8,.44,.15	1(49)	1(27)	97	88	82	87
8,.435,.15	1(30)	1(9)	46	35	31	31
8,.315,.20	1(39)	1(18)	16	15	14	14

We see conversely that the MP dynamic heuristic is not able to improve the results at the peak of complexity : it makes too often a disastrous bad choice at the first level. The first choices are crucial, and the heuristic cannot be improved by a top-down information in the current branch. Since this kind of heuristic becomes inefficient, instead of making a definitive choice for the first subtree at the beginning of the search, we propose to gather some information by beginning to explore each subtree. Then, with this bottom-up information, we will choose the subtree to explore first completely. In the next section, we will present an algorithm that implements this idea.

4 Ordering values by learning from failure

We want to design a specific value ordering for the variables at the top of the search tree, where the MP heuristic is not effective. For these variables the prospective approach will be substituted by an empirical learning from failure approach. We will therefore define a statistic, computed on a sample of each subtree at level d, and based on the average branch length (ABL) i.e. the length between the top of the subtree and a failure. We will first present a method where this statistic is precomputed and further the Progressive Focusing Search (PFS) algorithm which updates this statistic during the search and allows itself to change the subtree to explore if ABL of current subtree becomes bad.

4.1 The ABL statistic

We started with a basic idea : (1) run DFS on a subtree at level d and stop it when a fixed number b of leaves has been generated, then run DFS on another subtree for b other leaves and so on until every subtree at level d has been visited, (2) extract an empirical information from the sample of b leaves generated in each subtree and rank the subtrees (i.e. order the values of the first variable), (3) finish running DFS on the remaining branches of the best ranked subtree, then on the second best ranked subtree and so on, until a solution is found or the tree is exhausted.

It seems reasonable to assume that the subtree that has the longest branch on average will contain the largest number of solutions. Let us call $D_{i,j}$, the depth of the j^{th} branch of the i^{th} subtree. We can measure the average length $D_i(b) = \frac{\sum_{j=1}^{j=b} D_{i,j}}{b}$ of the sample of b branches generated for the i^{th} subtree. The subtrees can be ordered by that $D_i(b)$ criterion. So, the Average Branch Length (ABL) criterion is the empirical way to order the values we decided to use.

An important problem is : how to fix the value b ? The ABL criterion can allow us to reach a better value ordering than MP if the sample is large enough. If the sample size is too small, the resulting value ordering will be not as good as the MP value ordering and the search will be less efficient. If it is too large, the time spent for finding the value ordering could be longer than the time saved on the tree search. Increasing b makes the choices more accurate but the potential gain smaller. If b is set to infinity, we fall back into a standard DFS. So, we will enhance the current algorithm by unifying the sampling phase and the search phase.

4.2 The PFS algorithm

The idea is to update each average branch length during the search phase. The new criterion is $D_i(b_i)$, the average length of the b_i generated branches in the subtree i during the sampling and the search phases. The subtree i with $D_i(b_i) = max_j(\{D_j(b_j)\})$ will be searched until $D_i(b_i)$ becomes lower than another $D_j(b_j)$ or the subtree i is exhausted. Then the search will be continued in the subtree which has the greatest $D_i(b_i)$ value. The advantage of continuing to update $D_i(b_i)$ during the search phase is to allow the algorithm to "change its decision" if it finds that the subtree it chose is not as promising as it first appeared to be. As each b_i will grow, each $D_i(b_i)$ value will become more stable and the search will stay stuck in a subtree until its exhaustion. This mechanism allows us to eliminate the sampling phase, which no longer needs to achieve a completely satisfying value ordering before starting the search. During the search phase, each time a backtrack occurs in subtree i, the current $D_i(b_i)$ value is updated and if the subtree i does not have the greatest value anymore then the algorithm switches to the subtree with the new greatest value.

Since the sampling is now integrated into the search phase, we can skip the sampling phase. One branch is generated into each subtree and the search starts in the subtree whose first branch is the longest. Then, the subtree may be switched if its average branch length becomes lower than another. However, we have to consider the possibility that the first branch of a subtree is much shorter than the following branches. In this case, the subtree could be never visited anymore. In order to force the algorithm to visit several branches in each subtree, we will measure a biased average.

The final version of the algorithm is the same as the one we have presented before except that, instead of $D_i(b_i)$, the new statistic for switching between the subtrees is $E_i(b_i) = \frac{K.n+\sum_{j=1}^{j=b_i} D_{i,j}}{K+b_i}$, where K is a constant parameter and n is the number of variables of the CSP. $E_i(b_i)$ is a biased average as it computes the average depth of the branches as if a preprocessing phase had generated K branches of length n. This algorithm, called Progressive Focusing Search (PFS), is given in figure. 1. PFS starts the search by interleaving it on all the subtrees beginning at depth d The subtrees at depth d are ordered in the priority queue Q according to their E_i value. During the search, as the average length of the branches of these subtrees becomes more and more significant, it will progressively alter the interleaving by focusing on the most promising subtree: the subtrees with the best average branch length will be visited more often until it gradually falls into the previous mechanism, where the search will be maintained in a subtree as long as it has the greatest $D_i(b_i)$ value.

Here are the properties of the $E_i(b_i)$ statistic and their consequences on the tree search:

- As $\forall D_{i,b_i} : D_{i,b_i} \leq n$, we have $\forall b_i : E_i(b_i) > D_i(b_i)$. $E_i(0) = n$ and $E_i(b_i)$ converges to $D_i(b_i)$ as b_i increases.
  This means that searching into subtree i makes b_i increase and $E_i(b_i) - D_i(b_i)$ decrease. So, $E_i(b_i)$ tends to decrease even if

```
PFS(d):
1   Insert the k nodes at depth d into the stack array Stack.
2   Set Q to [1, 2, ..., k]
3   s ← remove_max(Q)
4   WHILE top(Stack[s]) is not a solution
5       node ← pop(Stack[s])
6       IF node = Failure
7           update E_s
8           insert(s, Q)
9           s ← remove_max(Q)
10      ELSE
11          push(children(node), Stack[s])
12      IF Stack[s] = ∅
13          IF Q = ∅
14              return Failure
15          ELSE
16              s ← remove_max(Q)
17  return top(Stack[s])
```

Figure 1. Progressive Focusing Search

$D_i(b_i)$ remains constant. This makes the search switch artificially between all the subtrees.

- $E_i(b_i+1) = \frac{K.n+\sum_{j=1}^{j=b_i+1} D_{i,j}}{K+b_i+1} = \frac{K+b_i}{K+b_i+1} E_i(b_i) + \frac{D_{i,b_i+1}}{K+b_i+1}$.

  Thus, $E_i(b_i+1) < E_i(b_i)$ iff $E_i(b_i) > D_{i,b_i+1}$.
  At the beginning of the search, if $K \gg b_i$ then $K.n \gg \sum_{j=1}^{j=b_i} D_{i,j}$ and $E_i(b_i) \simeq n > D_{i,b_i+1}$. So, $E_i(b_i)$ will decrease and the subtrees will be often switched. After a while, when $K \ll b_i$, we will have $K.n \ll \sum_{j=1}^{j=b_i} D_{i,j}$ and $E_i(b_i) \simeq D_i(b_i)$. So, $E_i(b_i)$ will decrease iff $D_{i,b_i+1} < D_i(b_i)$. This means that $E_i(b_i)$ is more likely to decrease at the beginning of the search than later. So, the interleaving will be progressively reduced all along the search.
- If $K = 0$ then $E_i(b_i) = D_i(b_i)$. If $K \gg 1$ then $E_i(b_i) \simeq n$. So, tuning K changes the speedup of convergence of $E_i(b_i)$ to $D_i(b_i)$, that is, the speedup of focusing on the subtree with the greatest $D_i(b_i)$ value.

4.3 Comparison with IDFS

PFS can be compared to Interleaved Depth First Search (IDFS) [10]. IDFS has been designed to take into account the increasing quality of the value ordering heuristic as the depth increases. Its behavior can be opposed to the one of DFS. When meeting a dead end, DFS undoes the *last* choices first whereas IDFS undoes the *first* choices first. The justification is simple : the shallowest choices are the least trustworthy so the deepest choices are more to be preserved. There are two versions of IDFS : Pure IDFS and Limited IDFS.

Pure IDFS exactly behaves as presented. Limited IDFS restrains this interleaving to a fixed depth d for a limited number of subtrees. It exactly simulates the order of examination of the nodes by a parallel tree search, with the interleaving depth and the number of active subtrees corresponding to the distribution depth and the number of processes.

In the rest of the paper, when we will refer to IDFS, it will implicitly be the limited version. IDFS does not make any choice until the interleaving maximum depth d is reached. Every value for the d first variables is indeed tried in an interleaved way.

Notice that IDFS is more efficient than DFS in the context we are studying. It has be shown theoretically and practically [10, 11, 12] that IDFS can outperform DFS when the quality of the value ordering heuristic increases with depth, which is the case of the dynamic promises solutions heuristic (MP). Experimental results in the next section will confirm that again.

At the beginning of the search, PFS behaves somewhat like Limited IDFS, in the sense that it generates one branch in each subtree and will not generate another branch into that subtree before it has visited all the other subtrees. When the tree search starts, every $E_i(0) = n$. So, the first subtree is selected, one branch is generated until depth $D_{1,1}$(where a backtrack occurs) and $E_1(1)$ is set to a value lower than n. So, the search has to be continued into the second subtree, and the same process is repeated until one branch has been generated into each subtree. Then, the subtree j with the greatest $E_i(1)$ value is selected and $E_j(2)$ is all the more likely to become the lowest value that K is high. When $K \gg b_i$, if $b_i > b_j$, $E_i(b_i)$ is often lower than $E_j(b_j)$. If $b_i = b_j$, $(E_i(b_i) > E_j(b_j)) \equiv (D_i(b_i) > D_j(b_j))$. So, PFS with a high value for K will interleave the search as much as IDFS but not exactly in the same order: a subtree which has generated less branches than others will tend to be chosen, and if all the subtrees have generated the same number of branches, then the one having the greatest $D_i(b_i)$ will be selected first. A PFS with a high value for K can be seen as an IDFS which always re-orders the subtrees thanks to their $D_i(b_i)$ values after having generated one branch in all of them. This means that PFS with a high value for K should generate a little less branches and nodes than IDFS. Now, we will see why a lower value for K and a progressive focusing of the tree search may lead to a higher performance.

PFS can be seen as a tree search which behaves like IDFS at the beginning and progressively focuses on the most promising subtree, behaving after a while like DFS. If we decide to use the average branch length heuristic (ABL) for the d first variables and MP for the $n-d$ remaining variables, then :

- At the beginning of the search, ABL is poorly informed and uncertain. The quality of MP is better. Then, IDFS is more to be used than DFS. So, PFS is right in starting by imitating IDFS.
- As the ABL estimates converge to the right values, there will be a moment during the search where the quality of ABL will be better than the quality of MP. Since the quality of the heuristic will not decrease between depth 1 and depth $d+1$, continuing to interleave the search in the subtrees at depth d will become a waste. Then, DFS will become a better algorithm to apply after that moment. So, PFS is right in imitating DFS at this moment.

A progressive change makes the algorithm stable. The efficiency of PFS depends on the speed of the focusing. Thus, attention has to be paid to the tuning of K.

5 Experimental results

Experiments were made using the same random CSP generator as before. The problems were chosen so as to remain at the left of the complexity peak, where most problems have solutions. For each set of CSP parameters, we ran several algorithms on the same 100 instances of problems all having at least one solution. All algorithms maintained arc-consistency, used the dom/degree dynamic variable ordering heuristic and the dynamic max-promises solutions heuristic. The algorithms were : DFS, IDFS$_d$ and PFS$_d(K)$, where d is the

depth of interleaving. Our goal was not to find the optimal parameters to maximize the gains of PFS over DFS and IDFS but to begin to study how this gain varies when approaching the complexity peak or when d and K vary.

We began by checking if PFS could be more efficient on problems of same size (50 variables, domain size 8) as in section 3 at the left of the complexity peak (see table 3).

Table 3. Experiments with $N = 50$, $S = 8$, $D = 5/64$ and T varying. We always reported the *total* constraint checks (in millions) over the 100 random problem instances.

T	1000/1275	1020/1275	1040/1275	1060/1275
DFS	2060	2290	7872	21070
$IDFS_1$	**541**	894	3093	8720
$IDFS_2$	1233	1555	3808	8176
$PFS_1(5)$	707	**680**	**2246**	8721
$PFS_2(5)$	1216	1452	3487	**8029**

We can notice that when problems became harder, PFS was more efficient and interleaving at depth 2 became better than at depth 1. In order to see what could happen if the interleaving depth increased again, we continued the tests on problems with $S = 4$ in order to limit the number of stacks (see table 4).

Table 4. Variation of problem hardness and interleaving depth. Experiments with $N = 100$, $S = 4$ and various (D, T) settings. For the parameter set (0.166, 0.125), two problems had no solution.

D,T	.158,.125	.162,.125	.166,.125	.327,.0625	.336,.0625
DFS	109	598	1306	384	2005
$IDFS_1$	102	509	1302	198	1230
$IDFS_2$	60.1	298	979	142	605
$IDFS_3$	73.9	308	718	197	597
$PFS_1(5)$	91.9	520	1483	**121**	1223
$PFS_2(5)$	**49.9**	**195**	973	137	505
$PFS_3(5)$	69.7	204	**597**	175	**444**

We also made K vary. When K was high ($K = 1000$), PFS_d was always slightly better (saving up less than 1%) than $IDFS_d$. When K had small values, we could obtain even better results than with $K = 5$ but the gain was not high. For example, for the problems in the third column of table 4, the results varied between 563 and 659 when $1 \leq K \leq 10$.

We can notice that :

- IDFS outperformed DFS on hard problems.
- A deeper interleaving was better when the problems became harder.
- PFS best outperformed DFS (savings of up to 78%) and IDFS (up to 35%) when problems were hard.
- The efficiency of PFS over DFS seems to increase as the problems become harder.

We noticed that the PFS results had a lower standard deviation than DFS and IDFS. When the problem was easily solved by DFS, IDFS and PFS produced more nodes. When the problem took a long time to be solved by DFS (because large subtrees without solution were explored), PFS usually produced much less nodes.

That IDFS outperforms DFS on hard problems confirms the interest of avoiding to choose between the possible values for the first variables. Interleaving is a good option when the quality of the value ordering heuristic is increasing with depth. The performances of PFS over IDFS show that there is a time when focusing on a single subtree becomes better than keeping on interleaving.

The gains of PFS over IDFS were not very high. One reason can be that the random CSPs we tested were regular. So, there was not always a really better subtree to focus on in order to accelerate the search. We expect PFS to exhibit better performances on less balanced CSPs. As the average branch length criterion is very simple, a more accurate criterion should also produce better results.

6 Conclusion and perspectives

We have shown the inefficiency of the prospective approach of value ordering for hard problems with solutions. We have begun to explore the possibilities of a learning from failure scheme. Simply measuring the average branch length of subtrees for the value ordering made PFS obtaining better results than DFS or IDFS. Yet, the ABL criterion is very simple. To enhance the results of PFS over IDFS, there must be some more accurate criteria that still have to be found. It would be also useful to find an automatic way to fix K to its best value. We intend to study this point by trying two approaches: (1) precomputing K thanks to the CSP characteristics or (2) adjusting K during the search on criteria that still have to be determined.

We think the empirical approach for (re)ordering values is promising for accelerating the search in hard problems. Where the traditional prospective approach becomes inefficient, an empirical approach can replace it advantageously.

REFERENCES

[1] C. Bessière and J.C. Régin, 'MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems', in *Proc of CP'96*, volume 1113 of *LNCS*, pp. 61–75, (1996).

[2] B. Cabon, G. Verfaillie, D. Martinez, and P. Bourret, 'Using Mean Field Methods for Boosting Backtrack Search in Constraint Satisfaction Problems', in *Proceedings of the* 12^{th} *European Conference on Artificial Intelligence*, pp. 165–169, (1996).

[3] R. Dechter and J. Pearl, 'Network–based heuristics for constraint–satisfaction problems', *Artificial Intelligence*, **34**, 1–38, (1988).

[4] D. Frost and R. Dechter, 'Look-ahead Value Ordering for Constraint Satisfaction Problems', in *Proceedings of the* 14^{th} *International Joint Conference on Artificial Intelligence*, (1995).

[5] Daniel H. Frost, *Algorithms and Heuristics for Constraint Satisfaction Problems*, Ph.D. dissertation, University of California - Irvine, 1997.

[6] P. A. Geelen, 'Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems', in *Proceedings of the* 10^{th} *European Conference on Artificial Intelligence*, (1992).

[7] R. M. Haralick and G. L. Elliott, 'Increasing Tree Search Effi ciency for Constraint Satisfaction Problems', *Artificial Intelligence*, **14**, 263–313, (1980).

[8] T. Hulubei, 'The CSP Library', http://www.cs.unh.edu/~tudor/csp/, University of New Hampshire, (1999).

[9] J. Larrosa and P. Meseguer, 'Optimization-based Heuristics for Maximal Constraint Satisfaction', in *Constraint Programming CP'95*, volume LNCS 976, pp. 103–120, (1995).

[10] P. Meseguer, 'Interleaved Depth-First Search', in *Proceedings of the* 15^{th} *International Joint Conference on Artificial Intelligence*, pp. 1382–1387, (1997).

[11] P. Meseguer and T. Walsh, 'Interleaved and Discrepancy Based Search', in *Proc. of the* 13^{th} *European Conf. on Artificial Intelligence*, (1998).

[12] N. Prcovic and B. Neveu, 'Ensuring a Relevant Visiting Order of the Leaf Nodes during a Tree Search', in *Constraint Programming, CP'99*, volume LNCS 1713, pp. 361–374, (1999).

[13] D. Sabin and E. C. Freuder, 'Contradicting Conventional Wisdom in Constraint Satisfaction', in *Proceedings of the* 10^{th} *European Conference on Artificial Intelligence*, pp. 125–129, (1994).

[14] B. M. Smith and M. E. Dyer, 'Locating the Phase Transition in Binary Constraint Satisfaction Problem', *Artificial Intelligence*, **81**, 155–181, (1996).

[15] B. M. Smith and S. A. Grant, 'Trying Harder to Fail First', in *Proceeedings of the* 13^{th} *European Conference on Artificial Intelligence*, (1998).