YIELDS: A Yet Improved Limited Discrepancy Search for CSPs *

Wafa Karoui[1,2], Marie-José Huguet[1], Pierre Lopez[1], and Wady Naanaa[3]

[1]Univ. de Toulouse, LAAS-CNRS, 7 avenue du colonel Roche, 31077 Toulouse, France
[2]Unité ROI, Ecole Polytechnique de Tunisie, La Marsa, Tunisie
[3]Faculté des Sciences de Monastir, Boulevard de l'environnement, Tunisie
{wakaroui,huguet,lopez}@laas.fr, naanaa.wady@planet.tn

Abstract. In this paper, we introduce a Yet ImprovEd Limited Discrepancy Search (YIELDS), a complete algorithm for solving Constraint Satisfaction Problems. As indicated in its name, YIELDS is an improved version of Limited Discrepancy Search (LDS). It integrates constraint propagation and variable order learning. The learning scheme, which is the main contribution of this paper, takes benefit from failures encountered during search in order to enhance the efficiency of variable ordering heuristic. As a result, we obtain a search which needs less discrepancies than LDS to find a solution or to state a problem is intractable. This method is then less redundant than LDS.
The efficiency of YIELDS is experimentally validated, comparing it with several solving algorithms: Depth-bounded Discrepancy Search, Forward Checking, and Maintaining Arc-Consistency. Experiments carried out on randomly generated binary CSPs and real problems clearly indicate that YIELDS often outperforms the algorithms with which it is compared, especially for tractable problems.

1 Introduction and motivations

Constraint Satisfaction Problems (CSPs) provide a general framework for modeling and solving numerous combinatorial problems. Basically, a CSP consists of a set of variables, each of which can take a value chosen among a set of potential values called its domain. The constraints express restrictions on which combinations of values are allowed. The problem is to find an assignment of values to variables, from their respective domains, such that all the constraints are satisfied [4][19].

CSPs are known to be NP-complete problems. Nevertheless, since CSPs crop up in various domains, many search algorithms for solving them have been developed. In this paper, we are interested in complete methods which have the advantage of finding at least a solution to a problem if such a solution exists. A widely studied class of complete algorithms relies to depth first search (DFS).

* To be published in the proceedings of The Fourth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'07), Brussels, Belgium.

Forward Checking (FC) [9] and *Maintaining Arc-Consistency* (MAC) [17] are two sophisticated algorithms belonging to the DFS class. Each of them enforces during search a kind of local consistency to prune the search tree and therefore to fasten problem solving. Another algorithm belonging to the DFS class is *Limited Discrepancy Search* (LDS) [10]. Since value ordering heuristics cannot avoid bad instantiations (*i.e.,* choosing, for a given variable, a value that does not participate in any solution), LDS tackles this problem by gradually increasing the number of less-preferred options from the heuristic view-point (*discrepancies*) [13][15][20].

Lots of recent research works try to improve this type of methods. One important area is concerned with learning from failures [2][8]. In this context and in order to design a more efficient search method we proposed a subtantial improvement of LDS. We have then defined *YIELDS* (Yet Improved Limited Discrepancy Search) [12]. YIELDS integrates constraint propagation as well as a variable order learning scheme in order to reduce the size of the search tree. The goal is to minimize the number of discrepancies needed to obtain a solution or to state a problem is intractable. More precisely, this paper is dedicated to the refinement of the learning mechanism of YIELDS and its evaluation on various types of instances including randomly generated data and real problems.

The paper is organized as follows. The next section specifies the main concepts and reviews the existing methods to solve constraint satisfaction problems, especially those following Depth-First Search. Section 3 is the core of the paper: it describes how to better explore the search space using Yet Improved Discrepancy Search. Experimental experience is reported in Section 4. Section 5 describes related works and Section 6 concludes the paper.

2 Background

Constraint Satisfaction Problem. A CSP is defined by a tuple (X, D, C) where: $X = \{X_1, \ldots, X_n\}$ is a finite set of variables; $D = \{D_1, \ldots, D_n\}$ is the set of domains for each variable, each D_i being the set of discrete values for variable X_i; $C = \{C_1, \ldots, C_m\}$ is a set of constraints.

An instantiation of a subset of variables corresponds to an assignment of these variables by a value from their domain. An instantiation is said to be complete when it concerns all the variables from X. Otherwise, it is called a partial instantiation. A solution is a complete instantiation satisfying the constraints. An inconsistency in the problem is raised as soon as a partial instantiation cannot be extended to a complete one.

Tree Search Methods. Methods based on Depth-First Search are generally used to quickly obtain a solution of a CSP. One has to successively instantiate variables which leads to the development of a search tree in which the root corresponds to uninstantiated variables and leaves to solutions. DFS methods are based on several key-concepts:

- At each node of the tree, one has to determine how to choose a variable to be instantiated and how to choose its value; this can be processed by variable and value ordering heuristics, respectively.
- After each instantiation, one has to define what level of constraint propagation can be used to prune the tree.
- Moreover, when a dead-end occurs (*i.e.,* an inconsistency) one has to specify how to backtrack in the tree to develop another branch or to restart in order to continue the search.

This type of methods is usually stopped either as soon as a solution is obtained or when the complete tree has been explored without finding any solution. In the worst case, it needs an exponential time in the number of variables.

Chronological Backtracking (CB) is a well-known method based on DFS for solving CSPs. CB extends a partial instantiation by assigning to a new variable a value which is consistent with the previous instantiated variables (look-back scheme). When an inconsistency appears it goes back to the latest instantiated variable trying another value.

Limited Discrepancy Search (LDS) is based on DFS for variable instantiation and on the concept of discrepancy to expand the search tree. A discrepancy is realized when the search assigns to a variable a value which is not ranked as the best by the value ordering heuristic. The LDS method is based on the idea of gradually increasing the number of allowed discrepancies while restarting:

- It begins by exploring the path obtained by following the values advocated by the value ordering heuristic: this path corresponds to zero discrepancy.
- If this path does not lead to a solution, the search explores the paths that involve a single discrepancy.
- The method iterates increasing the number of allowed discrepancies.

For binary trees, counting discrepancies is a quite simple task: exploring the branch associated with the best boolean value, according to a value ordering heuristic, involves no discrepancy, while exploring the remaining branch implies a single discrepancy. For non binary trees, the values are ranked according to a value ordering heuristic such that the best value has rank 1; exploring the branch associated to value of rank $k \geq 1$ leads to make $k - 1$ discrepancies.

LDS can be stopped either as soon as a first solution is found or when the complete tree is expanded using the maximum number of allowed discrepancies. Note that an improvement of LDS is Depth-bounded Discrepancy Search (DDS) which first favours discrepancies at the top of the search tree (*i.e.,* on the most important variables).

Ordering heuristics. They aim to reduce the search space to find a solution. Depending on their type they provide an order for the selection of the next variable to consider or the next value for a variable instantiation. These heuristics can be static (*i.e.,* the orders are defined at the beginning of the search) or dynamic (*i.e.,* the orders may change during search). The efficiency of DFS methods such as CB or LDS clearly depends on the chosen ordering heuristic.

For CSPs, several common variable and value ordering heuristics are defined such as dom (*i.e., min-domain*) or dom/deg for variable ordering, and min-conflict for value ordering (see [2] and [8]).

Propagations. To limit the search space and then to speed up the search process, constraint propagation mechanisms can be joined to each variable instantiation. The goal of these propagations is to filter the domain of some not yet instantiated variables. Various levels of constraint propagation can be considered in a tree search method. The most common are:

- *Forward Checking* (FC) which suppresses inconsistent values in the domain of not yet instantiated variables linked to the instantiated one by a constraint.
- *Arc-Consistency* (AC) which corresponds to suppress inconsistent values in the domain of all uninstantiated variables.

These constraint propagation mechanisms can be added both in Chronological Backtracking and in Discrepancy Search. In the rest of the paper, CB-FC refers to the CB method including FC propagation while CB-AC includes AC propagations (CB-AC corresponds to MAC algorithm [17]).

3 The proposed approach

3.1 Overcoming the limits of LDS

The objective of our approach is to minimize the number of discrepancies needed to reach a solution or to declare that the problem is intractable. To do that, we propose to use the dead ends encountered during a step of the LDS method to order the problem variables for the following steps. In fact in LDS, only the heuristic on the order of variables selects the path in the search space (see Algorithms 1 and 2). This means that when we increment the number of discrepancies and reiterate LDS, we have frequently the same initial variable to instantiate. If we assume that this variable is the failure reason and that it eliminates values required for the solution, it is useless to develop again its branch.

Algorithm 1 LDS(X, D, C, k-max, Sol)

```
1: k ← 0
2: Sol ← NIL
3: while (Sol = NIL) and (k ≤ k_max) do
4:    Sol ← LDS_iteration(X, D, C, k, Sol)
5:    k ← k+1
6: end while
7: return Sol
```

To avoid this kind of situations, we associate a weight, initially equal to zero, to each variable. This weight is incremented every time this variable fails because

Algorithm 2 LDS_iteration(X, D, C, k, Sol)

```
1: if X = ∅ then
2:    return Sol
3: else
4:    x_i ← BestVariable(X) // variable instantiation heuristic
5:    v_i ← BestValue(D_i, k) // value instantiation heuristic
6:    if v_i ≠ NIL then
7:       D' ← Update(X\{x_i}, D, C, (x_i, v_i)) // constraint propagation
8:       I ← LDS_iteration(X\{x_i}, D', C, k, Sol ⋃ {(x_i, v_i)})
9:       if I ≠ NIL then
10:         return I
11:      else
12:         if k > 0 then
13:            D_i ← D_i\{v_i}
14:            return LDS_iteration(X, D, C, k-1, Sol) // can diverge
15:         end if
16:      end if
17:   end if
18:   return NIL
19: end if
```

of the limit on the number of allowed discrepancies: we cannot diverge on this variable despite its domain of values is not empty. In the following iterations, this variable will be privileged and will be placed higher in the branch developed by the LDS method. Like this, we will avoid the situation of inconsistency caused by this variable. Therefore, the choice of variable to be instantiated is based, first, on the usual heuristic (dom for example), second, on the variable weight, finally, if tied variables still remain, the order of indexation can be considered.

Thus, by introducing the notion of weight, our purpose is to correct the heuristic for variable instantiation guiding it to variables concretely constrained. These variables greatly influence the solution search. Therefore, we correct mistakes of the heuristic by adding the weight notion which can be considered as a type of dynamic learning. Like this, we can exploit previous failures and take useful information for the following steps. To speed up the process, difficult and intractable subproblems are pushed up at the top of the search tree.

This improvement of LDS method besides its effects on the variable ordering, stops the LDS iterations when an inconsistency is found. In fact, if an inconsistency arises with k allowed discrepancies, other iterations, from k+1 to the maximum number of discrepancies, are unnecessary since they will discover again the same inconsistency.

For LDS, when we authorize a fixed number of discrepancies we can consume, completely or not, these authorized discrepancies. If the totality of discrepancies is consumed and no solution is found, a new iteration of LDS is launched incrementing the number of allowed discrepancies. In contrast, if the allowed discrepancies are not consumed, it is not necessary to continue to reiterate LDS with a greater number of discrepancies even if no solution has been found. In such situation, one can be sure that the problem is intractable (all the feasible

values of each variable have been tried without using the number of allowed discrepancies).

3.2 The YIELDS algorithm

The YIELDS method is based on a learning from failures technique. This learning produces a new way to go all over the search space contributing to speed up the resolution. Moreover, the propagation mechanisms lead us to stop the search before we reach the maximum number of discrepancies in the case of intractable problems, without missing solution if it does exist.

The completeness of YIELDS can be proved: if the problem has a solution, YIELDS does find it. In fact, when the problem is tractable, the learning technique has produced a permutation on the order of variables. The iteration of YIELDS which has discovered the solution, called YIELDS(k), is based on a variable ordering O, learnt during the previous iterations. We can say that YIELDS(k) is equivalent to CB-FC directly associated with the variable ordering O.

When the problem is intractable, YIELDS stops the search with anticipation. The last iteration does not consume all allowed discrepancies: it can be compared to a call to CB-FC because the bound on discrepancies was not at the origin of the break. Like this, the method is complete (see Algorithms 3 and 4).

Algorithm 3 YIELDS(X, D, C, k_max, Sol)

```
1: k ← 0
2: Sol ← NIL
3: Exceed ← False
4: while (Sol = NIL) and (k ≤ k_max) do
5:    Sol ← YIELDS_iteration(X, D, C, k, Sol)
6:    k ← k+1
7:    if !Exceed then
8:       exit
9:    end if
10: end while
11: return Sol
```

The principle of YIELDS is exactly the same as LDS: it considers, initially, branches of the tree which cumulate the smallest number of discrepancies. The first difference is that a weight (initially the same for all variables) is associated to each variable and every time a variable fails because of the limit on discrepancies, its weight is incremented to guide next choices of the heuristic. The second difference is that the number of discrepancies is not blindly incremented until the maximum of discrepancies allowed by the search tree is reached (as it is done in LDS). Thus, the new method consumes less discrepancies than LDS or even DDS.

Algorithm 4 YIELDS_iteration(X, D, C, k, Sol)

```
1: if X = ∅ then
2:    return Sol
3: else
4:    x_i ← First_VariableOrdering(X,Weight)
5:    v_i ← First_ValueOrdering(D_i, k)
6:    if v_i ≠ NIL then
7:       D' ← Update(X\{x_i}, D, C, (x_i, v_i))
8:       I ← YIELDS_iteration(X\{x_i}, D', C, k, Sol ⋃ {(x_i, v_i)})
9:       if I ≠ NIL then
10:         return I
11:      else
12:         if k > 0 then
13:            D_i ← D_i \{v_i}
14:            return YIELDS_iteration(X, D, C, k-1, Sol)
15:         else
16:            Weight[x_i] ← Weight[x_i]+1
17:            Exceed ← True // impossible to diverge
18:         end if
19:      end if
20:   end if
21:   return NIL
22: end if
```

Definition 1. *Let $P = (X, D, C)$ be a binary CSP of n variables and w_i a weight associated to each variable x_i. The weight vector W of P is the vector composed of weights of all variables of the problem:*

$$W(P) = [w_0, w_1, ..., w_{n-1}]$$

Definition 2. *Let $W1$ and $W2$ be two weight vectors of P, a binary CSP. The variation of weights of P is given by ΔW the vector difference between W1 and W2:*

$$\Delta W(P) = W2(P) - W1(P)$$

Proposition 1. *Let assume that variable weights are initially equal and that they are incremented every time that we do not found a variable value which respects the limit on the number of authorized discrepancies. Let consider two successive iterations of YIELDS for the resolution of P a binary CSP. If the variation of weights $\Delta W(P)$ between these iterations is equal to the null vector, then we can be sure that:*

1. *The process of learning comes to end.*
2. *P is an intractable problem.*

Proof: Since $\Delta W(P) = 0$ the last iteration was not interrupted because of the limit on the number of authorized discrepancies (see Algorithm 2). In addition, it is obvious that an iteration of YIELDS without the limit on the number of discrepancies corresponds to CB-FC which is a complete method. Therefore, if the last iteration corresponds to a complete method and that no solution has been found yet, the problem is intractable. □

3.3 Illustrative examples

As an example for an intractable problem, let consider a CSP composed of three variables x_0, x_1, x_2 and four values $0, 1, 2, 3$ presented by its incompatibility diagram (see Figure 1).

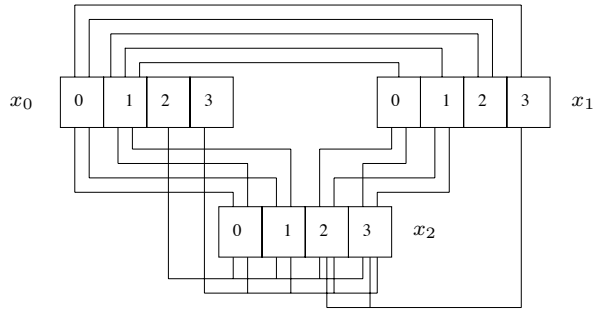

Fig. 1. Incompatibility diagram

The variable ordering initially follows the ascending order, then it is based first on min-domain order (dom), then on min-domain plus weights order (see Table 1), while min-conflict heuristic is applied for value ordering. In this example, the weights do not influence the variables order which is always the same (ascending order). Reminding the way retained for counting discrepancies (see Section 2), the maximum number of discrepancies is here equal to 9.

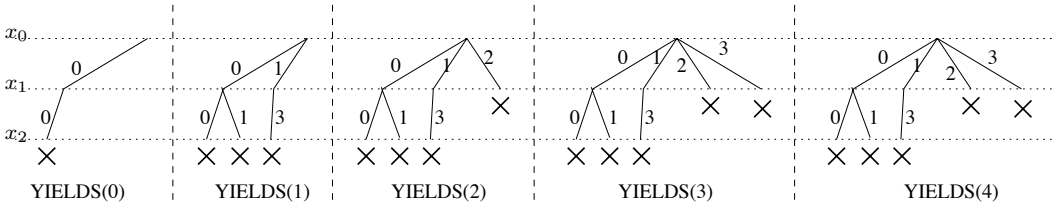

Fig. 2. Illustration of YIELDS on an intractable problem

Weight	Initial	YIELDS(0)	YIELDS(1)	YIELDS(2)	YIELDS(3)	YIELDS(4)
$W[x_0]$	0	1	2	3	4	4
$W[x_1]$	0	1	3	3	3	3
$W[x_2]$	0	0	0	0	0	0

Table 1. Variable weights for the intractable problem

From iterations YIELDS(0) till YIELDS(4), YIELDS develops the same search trees as LDS (see Figure 2). In YIELDS(4), we can see that even if we authorize 4 discrepancies, only 3 are used. The iterations of the YIELDS method are interrupted but not because of the limit on discrepancies. In such a context, YIELDS stops the search. LDS would continue iterations until LDS(9) and would repeat exactly the same search tree.

As an example for a tractable problem, let consider the CSP (X, D, C) defined by $X = \{x_0, x_1, x_2\}$, $D = \{D_0, D_1, D_2\}$ where $D_0 = D_1 = D_2 = \{0, 1, 2, 3, 4\}$. The set of contraints C is represented by the following set of incompatible tuples: $\{(x_0, 0), (x_1, 4)\} \cup \{(x_0, 0), (x_2, 4)\} \cup \{(x_0, 1), (x_1, 4)\} \cup \{(x_0, 1), (x_2, 4)\}$ $\cup \{(x_0, 2), (x_1, 4)\} \cup \{(x_0, 2), (x_2, 4)\} \cup \{(x_0, 3), (x_1, 4)\} \cup \{(x_0, 3), (x_2, 4)\} \cup$ $\{(x_0, 4), (x_2, 2)\} \cup \{(x_0, 4), (x_2, 3)\} \cup \{(x_1, 0), (x_2, 0)\} \cup \{(x_1, 0), (x_2, 1)\} \cup \{(x_1, 0), (x_2, 2)\}$ $\cup \{(x_1, 0), (x_2, 3)\} \cup \{(x_1, 1), (x_2, 0)\} \cup \{(x_1, 1), (x_2, 1)\} \cup \{(x_1, 1), (x_2, 2)\} \cup$ $\{(x_1, 1), (x_2, 3)\} \cup \{(x_1, 2), (x_2, 0)\} \cup \{(x_1, 2), (x_2, 1)\} \cup \{(x_1, 2), (x_2, 2)\} \cup \{(x_1, 2), (x_2, 3)\}$ $\cup \{(x_1, 3), (x_2, 0)\} \cup \{(x_1, 3), (x_2, 1)\} \cup \{(x_1, 3), (x_2, 2)\} \cup \{(x_1, 3), (x_2, 3)\}$.

In this example, we use the same ordering heuristics as previously. Applying CB-FC to solve this CSP, the resulting search tree consists of 24 expanded nodes (EN) (see Figure 3). Applying LDS, we obtain a bigger search tree of 95 EN. If we apply YIELDS, we obtain a search tree of only 13 EN (see Figure 4) due to the increasing of x_1 priority which contributes to speed up the search.

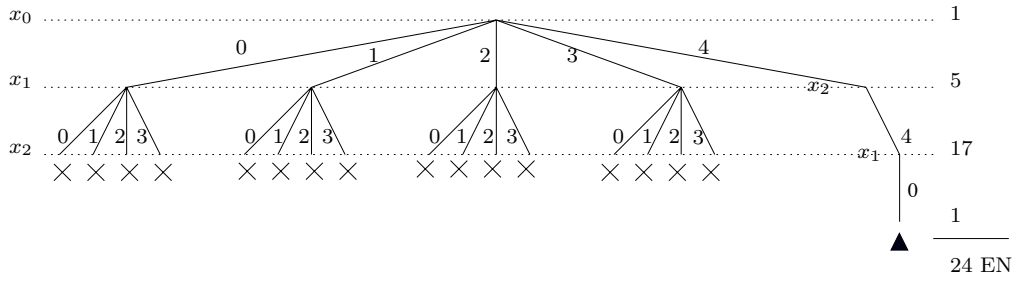

Fig. 3. CB-FC search tree

Weight	Initial	YIELDS(0)	YIELDS(1)	YIELDS(2)
$W[x_0]$	0	1	2	2
$W[x_1]$	0	1	3	3
$W[x_2]$	0	0	0	0

Table 2. Variable weights for the tractable problem

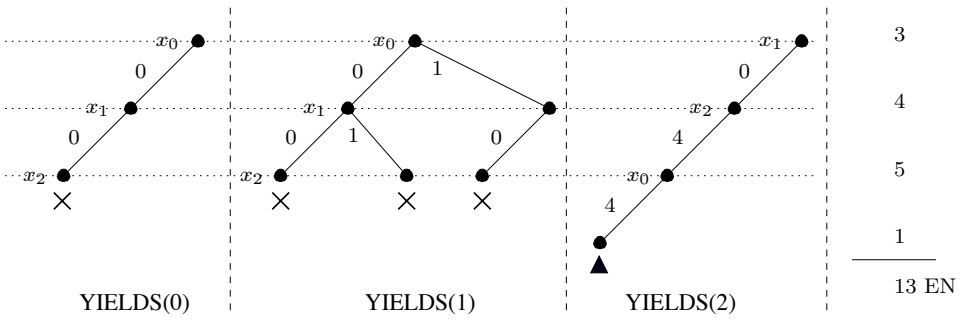

Fig. 4. YIELDS search tree

4 Experimental results

The problems investigated in our experiments are random binary CSPs, latin squares, and job-shop problems. We compared YIELDS with standard versions of DDS, CB-FC, and CB-AC. The arc-consistency algorithm underlying CB-AC is AC-3.1 [1][21]. The variable ordering heuristic used by all algorithms is dom. For value ordering, we used the min-conflict heuristic. The evaluation criteria are the number of expanded nodes (NED) and CPU time in seconds. Reported results are expressed as average values. All algorithms were implemented in C++. They were run under Windows XP Professional on a 1.6 GHz PC having 1 Go of RAM.

Random Binary CSPs:

For random binary CSPs, we used the generator developed by Frost *et al.* which is available in [3]. Problems are generated according to model B. We experimented on problems involving $n = 30$ variables, a uniform domain size of $d = 25$. The problem density p_1 (*i.e.,* the ratio of the number of constraints in the constraint graph over that of all possible constraints) varied from 0.1 (sparse problems: from line 1 to line 3 in Table 3) to 0.3 (dense problems: from line 4 to the end in Table 3). The constraint tightness p_2 (*i.e.,* the ratio of the number of disallowed tuples over that of all possible tuples) varied so that we obtain instances around the peak of complexity. The size of samples is 100 problem instances for each data point.

For all considered problems, the results clearly indicated that YIELDS outperforms DDS on sparse and dense problems (in Table 3, "$\gg$" means that execution times are of several hours).

For sparse problems, YIELDS is faster than CB-AC and CB-FC, albeit CB-AC develops less nodes.

For dense problems, YIELDS is also faster than CB-AC and CB-FC. However the advantage is less significant as we move toward dense problems. If we isolate tractable problems (last three lines in Table 3), results become particularly interesting and YIELDS clearly outperforms other considered methods.

instances	DDS		YIELDS		CB-FC		CB-AC	
$<n,d,C,T>$	NED	CPU	NED	CPU	NED	CPU	NED	CPU
<30,25,44,531>(35% sat)	10210510	90.61	6273	0.04	1427970	10.2	71	0.08
<30,25,44,526>(48% sat)	21876033	207.8	8526	0.06	1732513	11.92	250	0.19
<30,25,44,518>(73% sat)	1447242	11.72	3543	0.02	178168	1.26	270	0.21
<30,25,131,322>(39% sat)	$\gg$	$\gg$	1342742	12	1898943	16.45	203862	152.66
<30,25,131,320>(56% sat)	$\gg$	$\gg$	1360525	11.76	1374413	11.92	94277	79.92
<30,25,131,318>(74% sat)	$\gg$	$\gg$	1503581	12.39	1577180	13.24	54870	39.9
<30,25,131,322>(sat)	$\gg$	$\gg$	326739	3.07	1101429	9.37	46583	35
<30,25,131,320>(sat)	$\gg$	$\gg$	337996	3.05	827566	6.98	55165	58.46
<30,25,131,318>(sat)	$\gg$	$\gg$	341994	3.12	843548	7.06	16876	11.87

Table 3. Random binary CSPs instances

For intractable problems, CB-FC remains the better method.

Latin Squares and Job-Shop Scheduling problems:
For the job-shop problems, we investigated the Sadeh instances [18]. For tested instances, YIELDS is clearly better than CB-FC and CB-AC (see Table 4).

instances	CB-FC		YIELDS		CB-AC	
	NND	CPU	NND	CPU	NND	CPU
enddr1-10-5-1	802233	362	68997	<1	53186	897
enddr1-10-5-10	176015	94	57	<1	113486	457
ewddr2-10-5-1	156388	58	910	<1	92729	480
ewddr2-10-5-10	104032	41	55	<1	64535	372
e0ddr1-10-5-1	1262247624	13030	17133261	113	6262916	1752

Table 4. Job-Shop instances

We also studied experiments on *Latin Squares* obtained by the generator of [7]. Selected problems have an order of 10. Results showed that YIELDS is always faster than all considered methods (see Table 5).

5 Related works

Many research works try to improve known methods integrating learning from failures. In this context, the following methods were proposed:

1. Squeaky Wheel Optimization (SWO) [11] which is a general optimization approach for local search. In SWO, a greedy constructor produces an initial solution in which difficult elements are identified and guides the construction

instances	CB-FC		YIELDS		CB-AC	
	NND	CPU	NND	CPU	NND	CPU
qg.1030	7940160	74	158808	16	68276	72.5
qg.1032	26070985	239	128424	71	80215	105
qg.1034	400490	3.91	1775	0.02	181934	212
qg.1036	18976	0.19	364	0.01	13609	17.6
qg.1038	22795	0.21	114	0.01	14393	18

Table 5. Latin Squares instances

of a new solution (the process is iterated until some stopping criterion is met). This strategy has not completeness guarantee.

2. Impact-Based Search (IBS) [16] which is also a general search method based on a probing-like integer programming technique. In IBS, the reduction of the search space following a variable instantiation is used to prioritize the variables to consider. This method differs from ours by the used information for learning and by the nature of restarts.
3. F-O-Opt (failure-driven algorithm for Open Hidden-variable Weighted Constraint Optimization Problems) which is one of the algorithm proposed in [5] for open constraint optimization. The context for this search method is dynamic and constraints are updated while searching so used learning technics are local and different.
4. Last Conflict reasoning (LC) [2] which is a learning search method. This method was improved by Grimes and Wallace in [8] including restarts to the original method. Unlike our method, this method learns from constraints and, in Grimes improvement, added restarts are not relied to problem properties. In our method, learning is based on variables and restarts are based on dicrepancies. Gathered information on discrepancies variation may represent an additional information on the considered problem and contribute to accelerate the search.

6 Conclusion and further work

In this paper we present a novel method, Yet Improved Discrepancy Search (YIELDS), which takes advantages from failures to guide the search. The goal of this method is to correct the variable ordering heuristic exploiting some fails and detects whether a problem is intractable without doing all the iterations of LDS. We propose an effective YIELDS algorithm and describe how to integrate it into a classical LDS algorithm.

An experimental study carried out on numerous random and real CSPs have shown how it is possible to obtain good results.

In the near future, we plan to set up an association of two learning ways, weights and no-goods which, in our opinion, will constitute a helpful tool for the proposed method. In addition, we think that a careful computational study on

other known benchmarks will present an interesting issue to better illustrate the usefulness of YIELDS. Comparisons with some related works are also planned.

References

1. C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings IJCAI-01*, pages 309–315, Seattle, USA, 2001
2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings ECAI'04*, pages 146–150, Valencia, Spain, 2004
3. D. Frost, C. Bessière, R. Dechter, and J.-C. Régin, Random uniform CSP generators, `http://www.lirmm.fr/~bessiere/generator.html`
4. R. Dechter. *Constraint processing*, Morgan Kaufmann, San Francisco, 2003
5. B. Faltings and S. Macho-Gonzalez, Open constraint satisfaction. In Van Hentenryck, P., ed., *Proceedings of CP'2002, LNCS No. 2470*, Springer, pages 356–370, 2002
6. I.P. Gent and P. Prosser. Inside MAC and FC. APES Research Group Report APES-20-2000, 2000
7. C.P. Gomes. Generator of Quasigroup Completion Problem and related problems, `http://www.cs.cornell.edu/gomes/new-demos.htm`
8. D. Grimes and R. J. Wallace. Learning from failures in constraint satisfaction search. *AAAI Workshop on Learning for Search*, Boston, Massachusetts, USA, 2006
9. R. Haralick and G. Elliot, Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence*, 14:263–313, 1980
10. W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *Proceedings IJCAI-95*, pages 607–613, Montréal, Canada, 1995
11. D. Joslin and D. Clements. Squeaky Wheel Optimisation. In *Proceedings Sixteenth National Conference on Artificial Intelligence-AAAI'98*, pages 340–346, 1998.
12. W. Karoui, M.J. Huguet, P. Lopez et W. Naanaa. Amélioration par apprentissage de la recherche à divergences limitées. In *Proceedings JFPC'05*, pages 109–118, Lens, France, 2005
13. R.E. Korf. Improved limited discrepancy search. In *Proceedings AAAI-96/IAAI-96, Vol. 1*, pages 286–291, Portland, Oregon, USA, 1996
14. C. Likitvivatanavong, Y. Zhang, J. Bowen, and E.C. Freuder. Arc-consistency in MAC: A new perspective. In *Proceedings First International Workshop on Constraint Propagation and Implementation*, Toronto, Canada, 2004
15. N. Prcovic. Quelques variantes de LDS. In *Proceedings JNPC'02*, pages 195–208, Nice, France, 2002
16. P. Refalo. Impact-based search strategies for constraint programming. In Wallace, M., ed., *Principles and Practice of Constraints Programming-CP'04, LNCS No. 3258*, Springer, pages 557–571, 2004
17. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings PPCP-94*, Seattle, USA, 1994
18. N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86:1–41, 1996
19. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press Ltd, London, 1993

20. T. Walsh. Depth-bounded discrepancy search. In *Proceedings IJCAI-97*, pages 1388–1395, Nagoya, Japan, 1997
21. Y. Zhang and R.H.C. Yap. Making AC-3 an optimal algorithm. In *Proceedings IJCAI-01*, pages 316–321, Seattle, USA, 2001