Exploiting Weak Dependencies in Tree-based Search

Alejandro Arbelaez, Youssef Hamadi

Exploiting Weak Dependencies in Tree-based Search

Alejandro Arbelaez
Microsoft-INRIA, joint-lab, Parc Orsay Université
28, rue Jean Rostand 91893 Orsay Cedex, France
alejandro.arbelaez@inria.fr

Youssef Hamadi
Microsoft Research, UK
7 JJ Thomson avenue, Cambridge CB3 0FB, United Kingdom
youssefh@microsoft.com

ABSTRACT

In this work, our objective is to heuristically discover a simplified form of functional dependencies between variables called *weak dependencies*. Once discovered, these relations are used to rank the variables. Our method shows that these relations can be detected with some acceptable overhead during constraint propagation. More precisely, each time a variable y gets instantiated as a result of the instantiation of x, a weak dependency (x, y) is recorded. As a consequence, the weight of x is raised, and the variable becomes more likely to be selected by the variable ordering heuristic. Experiments on a large set of problems show that on the average, the search trees are reduced by a factor 3 while runtime is decreased by 31% when compared against dom-wdeg, one of the best dynamic variable ordering heuristic.

Keywords

Constraint Programming, Constraint Satisfaction Problems, Tree-Search , Functional Dependencies

1. INTRODUCTION

The relationships between the variables of a combinatorial problem are key to its resolution. Among all the possible relations, explicit constraints are the most straightforward and were widely used. For instance, they are used to support classical look-ahead and look-back schemes. During look-ahead, they can limit the maintenance of some level of consistency to some locality. During look-back, they can improve the backtracking by jumping to related and/or guilty decisions. These relationships are also used in dynamic variable ordering (dvo) to relate the current variable selection to past decisions (e.g., [3]), or to give preference to the most constrained parts of the problem, etc.

Recently, backdoors have been illustrated. A backdoor can be informally defined as a subset of the variables such that, once assigned values, the remaining instance simplifies to a computationally tractable class. Backdoors can be explained by the presence of a particular relation between variables, e.g., functional dependencies. Unfortunately, detecting backdoors can be computationally expensive [4], and their exploitation is often restricted to restart-based strategies like in modern SAT solvers [11].

In this work, our objective is to heuristically discover a simplified form of functional dependencies between variables called *weak dependencies*. Once discovered, these relations are used to rank the importance of each variable. Our method assumes that these relations can be detected with low overhead during constraint propagation. More precisely, each time a variable y gets instantiated as a result of the instantiation of x, a weak dependency (x, y) is recorded. As a consequence, the weight of x is raised, and the variable becomes more likely to be selected by the variable ordering heuristic.

In the following section, we start with some background definitions. Section three describes our new heuristic. Section four presents experimental results. Finally, before the general conclusion, section five presents related work.

2. BACKGROUND

In this section, we briefly introduce definitions and notation used hereafter.

DEFINITION 2.1. *A Constraint Satisfaction Problem (CSP) is a triple* (X, D, C) *where,*

- $X = \{X_1, X_2, \ldots, X_n\}$ *represents a set of* n *variables.*
- $D = \{D_1, D_2, \ldots, D_n\}$ *represents the set of associated domains, i.e., possible values for the variables.*
- $C = \{C_1, C_2, \ldots, C_m\}$ *represents a finite set of constraints.*

Each constraint C_i is associated to a set of variables $vars(C_i)$, and is used to restrict the combinations of values between these variables. Similarly, each variable X_i is related to a set of constraints $prop(X_i)$. The arity of a constraint C_i corresponds to $|vars(C_i)|$, and the degree of a variable X_i corresponds to $|prop(X_i)|$.

Solving a CSP involves the finding of a solution, i.e., an assignment of values to variables such that all the constraints are satisfied. If a solution exists, the problem is stated as satisfiable, and unsatisfiable otherwise.

A depth-first search backtracking algorithm can be used to tackle CSPs. At each step a value is assigned to some variable. Each assignment is combined with a look-ahead process called constraint propagation which can reduce the

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

```
1: Q = {p1, p2, ...}
2: while Q ≠ {} do
3:    p = choose(Q);
4:    run(p);
5:    for all X_i ∈ vars(p) s.t. D_i was narrowed do
6:       schedule(Q, p, X_i);
7:    end for
8: end while
```

Figure 1: Classic propagation engine

domains of the remaining variables. Constraint propagation is usually based on some constraint network property which determines its locality and therefore its computational cost. Arc-consistency is widely used, and the result of its combination with backtrack search is called MAC for maintaining arc-consistency [9].

Figure 1 describes a classic constraint propagation engine [10]. In this algorithm, constraints are managed as propagators[1] in a propagation queue, Q. This structure represents the set of propagators that need to be revised. Revising a propagator corresponds to the enforcement of some consistency level on the domains of the associated variables.

Initially, Q is set to the entire set of constraints. This is used to enforce the arc-consistency property before the search process. During depth-first exploration, each decision is added to an empty queue, and propagated through this algorithm.

The function *choose* removes a propagator $p \in Q$, *run* applies the filtering algorithm associated to p, and *schedule* adds $prop(X_i)$ to Q. The algorithm terminates when the queue is empty. A fix-point is reached and more propagations can only appear as the result of a tree-based decision.

DEFINITION 2.2. *$f(X, y)$ is a functional dependency between the variables in the set X and the variable y if and only if for each combination of values in X there is precisely one value for y satisfying f.*

Many constraints of arity k can be seen as functional dependencies between a set of $k-1$ variables and some remaining variable y. For instance, the arithmetic constraint $X + Y = Z$, gives the dependencies $f(\{X, Y\}, Z)$, $f(\{X, Z\}, Y)$, and $f(\{Y, Z\}, X)$. There are also many exceptions like the constraint $X \neq Y$, where in the general case, one variable is not functionally dependent of the other one.

3. EXPLOITING WEAK DEPENDENCIES IN TREE-BASED SEARCH

3.1 Weak dependencies

Our objective is to take advantage of functional dependencies during search. We propose to heuristically discover a weaker form of relation called *weak dependency* between pairs of variables. A weak dependency is observed when a variable gets instantiated as the result of another instantiation. Our new dvo heuristic records these weak dependencies and exploit them to prioritize the variables during the search process.

[1]In the following, we will use this as a synonym for constraint.

DEFINITION 3.2. *During constraint propagation with the Algorithm presented in Figure 1, we call (X, Y) a weak dependency if the two following conditions hold:*

1. *Y is instantiated as the result of the execution of a propagator p.*
2. *p was inserted in Q as the result of the instantiation of X.*

PROPERTY 3.1. *Weak dependency relations (X, Y) can be recorded as the result of the execution of a propagator p* iff *$X \in vars(p)$ and $Y \in vars(p)$.*

The proof is straightforward if we consider the Algorithm presented in Figure 1.

3.3 Example

To illustrate our definition, we consider the following set of constraints:

- $p_1 \equiv X_1 + X_2 < X_3$
- $p_2 \equiv X_1 \neq X_4$
- $p_3 \equiv X_4 \neq X_5$

With the domains, $D_1 = D_2 = D_4 = D_5 = \{0, 1\}$ and $D_3 = \{1, 2\}$.

The initial filtering does not remove any value and the search process has to be started. Assuming that the search is started on X_1 with value 1, the propagator $X_1 = 1$ is added to Q, and after its execution the domain D_1 has been narrowed, so that it is necessary to schedule p_1 and p_2.

Running p_1 sets X_2 to 0, and X_3 to 2, and gives the weak dependencies (X_1, X_2) and (X_1, X_3). Afterwards, p_2 sets X_4 to 0 which corresponds to (X_1, X_4). Finally, the narrowing of D_4 schedules p_3 which sets X_5 to 1, and gives the weak dependency (X_4, X_5).

Weak dependencies are binary, therefore they only roughly approximate functional dependencies. For example, with the constraint $X+Y = Z$ they will never record $(\{X, Y\}, Z)$. On the other hand, weak dependencies exploit the current domains of the variables and can record relations which are not true in general but hold in particular cases. For instance, the propagator p_3 above creates (X_4, X_5). This represents a real functional dependency since the domains of the variables are binary and equal.

3.4 Computing weak dependencies

We can represent weak dependencies as a weighted digraph relation among the variables of the problem, where the nodes of the graph are the variables and the edges indicate weak dependencies relations between two variables, i.e, when there is an edge between two variables X and Y, the direction of the edge shows the relation and its weight indicates the number of observed occurrences of that relation.

In a propagation centered approach [6] each variable has a list of dependent propagators and each propagator knows its variables (see Figure 2).

In this way, once the domain of a variable is narrowed it is necessary to schedule its associated propagators into the propagator pool. Since we are interested in capturing weak dependencies, we have to track the reasons for constraint propagation. More specifically, when a propagator

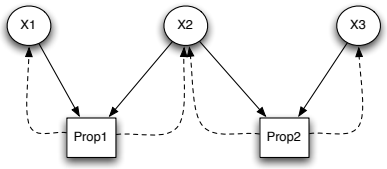

Figure 2: Variables and propagators

gets activated as the result of the direct assignment of some variable, we need to keep a reference to that variable. Since the assignment of several variables can activate a propagator, we might have to keep several references.

```
1: enqueue(Q, prop(X_i));
2: if |D_i| = 1 then
3:    dependencies(p.assigned, X_i);
4:    for all p' in prop(X_i) do
5:       p'.assigned.add(X_i);
6:    end for
7: end if
```

Figure 3: Schedule(Queue Q, Propagator p, Variable Xi)

A modified *schedule* procedure is shown in Figure 3. The algorithm starts by enqueueing all the propagators associated to a given variable X_i into the propagators pool. If the propagator p was called as the result of the assignment of X_i ($|D_i| = 1$), a weak dependency is created between each variable of the set $p.assigned$ and X_i. Variables from this set are the ones whose assignment was the reason for propagating p. After that, a reference to X_i is added to its propagators $prop(X_i)$. This is done to ensure that if these propagators assign other variables, a subsequent call to the schedule procedure will be able to create dependencies between X_i and these variables.

3.5 The domFD dynamic variable ordering

In the previous section, we have seen that a generic constraint propagation algorithm can be modified to compute weak dependencies. As we pointed out above, weak dependencies can be seen as a weighted digraph relation among the variables. Using this graph, we propose to define a function $FD(X_i)$ which computes the out-degree weight of a variable X_i taking into account only uninstantiated variables.

$$FD(X_i) = \sum_{X_j \in \Gamma^+(X_i)} weight(X_i, X_j) \quad (1)$$

Where $\Gamma^+(x)$ (resp. $\Gamma^-(x)$) represents the set of outgoing (resp. ingoing) edges from (resp. to) x in the graph of dependencies. It is also important to note that when there is no outgoing edge associated to X_i we assume $FD(X_i) = 1$.

Given the definition of FD, we propose to define $domFD$, a new dvo heuristic based on both: the observed weak dependencies of the problem and the well-known fail-first *mindom* heuristic:

$$domFD(X_i) = \frac{|X_i|}{FD(X_i)} \quad (2)$$

Then, the heuristic selects the variable whose $domFD$ value is minimal.

3.6 Complexities of domFD

3.6.1 Space

We know from Property 3.1 that dependencies are created between variables which share a constraint. Therefore, computing the weak dependency graph requires in the worst case a space proportional to the space used for the representation of the problem. Assuming n variables and m constraints, the space is proportional to $n + m$.

3.6.2 Time

The computation of weak dependencies is tightly linked to constraint propagation. The original schedule procedure only enqueues the propagators related to X_i in Q, therefore its original cost is $O(m)$. Our new procedure creates dependencies each time a variable gets instantiated. Dependencies between variables can be recorded as the result of the instantiation of one or several variables. In the latter case, up to $n-1$ dependencies can be created since the instantiation of up to $n-1$ variables can be responsible for the scheduling of the current propagator (line 3 in Algorithm of Figure 3). Once dependencies are created, the propagators associated to X_i need to reference it. Here the cost is bounded by m. Overall, the time complexity of the new schedule procedure is $O(n + m)$.

We now have to consider the cost of maintaining the weak dependency graph. Since our heuristic only considers the weights related to the variables which are not instantiated we have to disconnect variables from the graph when they get a value, and we have to reconnect them when the search backtracks. This can be done incrementally.

Practically, we do not have to physically remove a variable from the dependency graph, we can just offset the weight of the recorded dependencies between other variables and that variable. For instance, when X_i gets instantiated as the result of a tree decision or as the result of constraint propagation, we only need to update the out degrees of variables $X_j \in \Gamma^-(X_i)$. The update is done by decreasing their associated counter $X_j.FD$ by $weight(X_j, X_i)$. These counters represent the number of times the weak dependency (X_j, X_i) was observed during the search process. During backtracking, X_i gets back its domain, and we just have to "reconnect" the associated $X_j \in \Gamma^-(X_i)$ by adding $weight(X_j, X_i)$ to $X_j.FD$. Since a variable can be linked to m propagators, an update of the dependency graph cost $O(m)$. In the worst case, each branching holds no propagation and therefore at each node, the cost of updating the dependency graph is $O(m)$.

Finally, selecting the variable which minimizes $domFD$ can cost an iteration over n variables if no special data structure is used.

Now if we consider all the operations, constraint propagation with the new schedule procedure, disconnecting a single variable, and selection of the variable which minimizes $domFD$, we have $O(n + m)$ - as opposed to $O(m)$ initially.

4. EXPERIMENTS

In this section, we propose to study the performance of $domFD$ when compared to *dom-wdeg*, a recently introduced heuristic able to focus on the difficult parts of a problem [2].

In dom-wdeg, the priority is given to variables which are frequently involved in failed constraints. A weight is added to each constraint and updated (i.e, incremented by one) each time a constraint fails. Using this value variables are selected based on their domain size and their total associated weight. X_i, the selected variable minimizes dom-wdeg(X_i)= $|X_i|/\sum_{c \in prop(X_i)} weight(c)$.

This heuristic is used in the Abscon solver which appeared to be the most robust in the last CSP-competition[2] where it finished 1 time first, 3 times second, 3 times third, and 2 times fourth, when compared against 15 other solvers.

To compare domFD against the powerful dom-wdeg, we implemented them in gecode-2.0.1 [5] and used them to tackle several problems. Since gecode is now widely used, we decided to take from the Internet problems already encoded for the gecode library. We paid attention to the fact that overall our problems cover a large set of gecode's constraints.

We used 35 instances coming from 9 different benchmark families. They involve satisfaction, counting, and optimization problems. They were solved using the default gecode's branch-and-prune strategy, and a modified restart technique based on the default strategy. In the tests, the value selection ordering was gecode's `INT_VAL_MIN`, which returns the minimal value of a domain. All the experiments were performed on a MacBook-Pro 2.4GHz Intel Core 2 Duo, under Ubuntu linux 7.10 and gcc version 4.0.1. A time-out (TO) of 10 minutes was used for each experiment.

4.1 The problems

In the following, we list the different benchmark families. When they are described on `www.csplib.org`, we only present the number in the library. Note that for all problems (except Quasigroup) the model and its implementation is the one proposed in the gecode examples[3].

- Quasigroup, *qwh*, problem 3 of CSPLib.
- Golomb-ruler, *gol-rul*, problem 6 of CSPLib.
- All-interval, *all-int*, problem 7 of CSPLib.
- Nonogram, *nono*, problem 12 of CSPLib.
- Magic-square, *magic-squ*, problem 19 of CSPLib.
- Langford-number, *lfn*, problem 24 of CSPLib.
- Sport league tournament, *sport-lea*, problem 26 of CSPLib.
- Balanced Incomplete Block Design, *bibd*, problem 28 of CSPLib.
- Crowded-chess, *crow-ch*, this problem consists in arranging n queens, n rooks, $2n-1$ bishops and k knights on a $n \times n$ chessboard, so that queens cannot attack each others, no rook can attack another rook and no bishop can attack another bishop. Note that two queens (in general two pieces of the same type) are attacking each other even if there is a bishop (in general another piece of different type) between them.

[2]http://www.cril.univ-artois.fr/CPAI06/round2/results/ranking.php?idev=6

[3]Available from http://www.gecode.org/gecode-doc -latest/group__ExProblem.html.

When an instance is solved, the number of nodes in the tree(s), the number of fails and the time in seconds are reported. If the 10 minutes time-out is reached, TO is reported.

4.2 Searching for all solutions or for an optimal solution

The first part of Table 1, presents results related to the finding of all the solutions of all-interval problems of order 11 to 14. We can observe that the trees generated with domFD are usually far smaller than the ones generated by dom-wdeg. Most of the time, domFD runtime is also better. However, the time per nodes is more important for our heuristic. For instance, on all-int-14, dom-wdeg does 89973 nodes/s while domFD runs at 54122 nodes/s.

The second part of the table presents results for the optimal Golomb-rulers of orders 10 to 12. Here, we can observe that order 10 is easier for dom-wdeg, but sizes trees are comparable. Order 11, and 12 give the advantages to domFD, with far smaller search trees and better runtimes. As before, the time per node is more important for our heuristic (31771 vs 35852 on gol-rul-11).

4.3 Searching for a solution with a classical branch-and-prune strategy

Experiments related to the finding of a first solution are presented in Table 2. They show results for respectively, quasi-groups, balance incomplete block design, Langford numbers, and nonograms.

4.3.1 Quasi-groups

Three instances of order 30 with 316 unassigned positions were produced with the generator presented in [1]. On these instances, domFD always generates smaller search trees. When this difference is large enough e.g., second instance, the runtime is also better.

4.3.2 Balance incomplete block design

Our heuristic always explores smaller trees which allows better runtimes. Interestingly the third instance is solved in 0.03 seconds by domFD while dom-wdeg cannot solve it in 10 minutes.

4.3.3 Langford numbers

On these problems, domFD is always superior to dom-wdeg. For instance, lfn-3-10 can be solved by both heuristics but the performance of domFD is far better: 190 times faster.

4.3.4 Nonograms

Table 2 shows results for the nonogram problem. Three instances of orders 5, 8, and 9 were generated. Here again, the trees are systematically smaller with domFD and when the difference is large enough runtimes are always better.

4.4 Searching for a solution with a restart-based branch-and-prune strategy

Restart-based searches are very efficient since they can alleviate the effects of early bad decisions. Therefore, it is important to test our new heuristic with a restart strategy.

A restart is done when some cutoff limit in the number of fails is met, i.e., at some node in a tree. There, the actual domFD-graph is stored and used to start the next

Table 1: All solutions and optimal solution

Instance	*dom-wdeg*			*domFD*		
	#nodes	#failures	time (s)	#nodes	#failures	time (s)
all-int-11	100844	50261	0.93	52846	26262	**0.81**
all-int-12	552668	276003	6.92	211958	105648	**3.45**
all-int-13	2.34M	1.17M	**26.13**	1.64M	821419	29.74
all-int-14	15.73M	7.86M	**174.83**	11.27M	5.63M	208.23
gol-rul-10	93732	46866	**1.97**	102910	51449	2.70
gol-rul-11	2.77M	1.38M	77.26	1.77M	889633	**55.71**
gol-rul-12	12.45M	6.22M	404.92	6.97M	3.48M	**266.28**

Table 2: First solution, branch-and-prune strategy

Instance	*dom-wdeg*			*domFD*		
	#nodes	#failures	time (s)	#nodes	#failures	time (s)
qwh-30-316-1	1215	603	**0.22**	234	115	0.32
qwh-30-316-2	48141	24063	8.09	10454	5220	**3.62**
qwh-30-316-3	6704	3347	**1.11**	2880	1437	1.15
bibd-7-3-2	100	39	**0.01**	65	28	**0.01**
bibd-7-3-3	383	180	0.03	96	42	**0.01**
bibd-7-3-4	—	—	TO	132	56	**0.03**
lfn-3-9	168638	84316	6.16	7527	3760	**0.26**
lfn-2-19	—	—	TO	1.64M	822500	**43.05**
lfn-3-10	2.21M	1.10M	87.15	12440	6218	**0.46**
nono-5	1785	879	0.12	491	239	**0.11**
nono-8	17979	8983	3.54	1084	537	**0.54**
nono-9	248	115	**0.04**	120	58	0.12

tree-based search. This allows the early selection of well ranked variables. The same technique is used with dom-wdeg, and the next search tree can branch early on well ranked variables.

This part presents results with a restart-based branch-and-prune where the cutoff value used to restart the search was initially set to 1000, and the cutoff increase policy to $\times 1.2$. The same 10 minutes timeout was used.

Table 3 presents the results for magic square, crowded chess, sport league tournament, quasi-groups, and bibd problems.

4.4.1 Magic square

Instances of orders 5 to 11 were solved. Clearly, domFD is the only heuristic able to solve large orders within the time limit. For example, dom-wdeg cannot deal orders greater than 8, while our technique can. The reduction in the search tree sizes is very significant, e.g., on mag-squ-8, dom-wdeg develops 35.18M nodes and domFD 152466, which allows it to be more than 100 times faster.

4.4.2 Crowded chess

As before, domFD can tackle large problems while dom-wdeg cannot.

4.4.3 Sport league tournament

If we exclude the last instance, domFD is always better than dom-wdeg.

4.4.4 Quasi-groups

Here, on most problems, domFD generates smaller search trees, and can return a solution more quickly. On the hardest problem, (order 35), domFD is nearly two time faster.

4.4.5 Balanced incomplete block design

Here domFD performs very well, with both smaller search trees and small runtime.

4.5 Synthesis

Table 4 summarizes the performance of the heuristics. These results were generated by only taking into account the problems which can be solved by both domFD and dom-wdeg i.e., we removed 6 instances which cannot be solved by dom-wdeg.

Table 4: Synthesis of the experiments

heuristic	*average*			
	#nodes	#failures	time (s)	nodes/s
dom-wdeg	2.14M	1.07M	56.99	37664
domFD	717202	358419	39.53	18139

We can observe that the search trees generated by domFD are on the average three times smaller. The difference in the number of fails is similar. Finally, even if domFD is 2 times slower on the time per node, it is 31% faster overall.

Technically, our integration into gecode is quite straightforward and not particularly optimized. For instance we use *Leda*[4], an external library to maintain the graph, while a bespoke light class with the right set of features should be used. The way we record weak dependencies is also not optimized and requires extra data structures whose accesses could be easily improved, e.g., the *assigned* list of variables shown in the Algorithm of Figure 3. For all these reasons, we think that it must be possible to increase the speed of our heuristic by some factor.

[4]www.algorithmic-solutions.com.

Table 3: First solution, restart-based strategy

Instance	*dom-wdeg*			*domFD*		
	#nodes	#failures	time (s)	#nodes	#failures	time (s)
mag-squ-5	2239	1113	**0.02**	3025	1505	0.06
mag-squ-6	33238	16564	0.32	4924	2440	**0.08**
mag-squ-7	9963	4868	**0.20**	33422	16614	0.86
mag-squ-8	35.18M	17.59M	460.40	152446	75987	**4.51**
mag-squ-9	—	—	TO	66387	32951	**1.64**
mag-squ-10	—	—	TO	83737	41607	**2.17**
mag-squ-11	—	—	TO	8.52M	4.26M	**374.62**
crow-ch-7	2029	1002	**0.04**	3340	1656	0.22
crow-ch-8	16147	8036	0.67	2041	1002	**0.14**
crow-ch-9	129827	64788	**6.15**	228480	114089	37.97
crow-ch-10	—	—	TO	1134052	566761	**263.01**
sport-lea-14	4746	2327	0.68	4814	2359	**0.65**
sport-lea-16	28508	14073	4.05	3913	1912	**0.61**
sport-lea-18	546475	272510	101.70	51680	25549	**10.72**
sport-lea-20	182074	90355	**36.69**	2.07M	1.03M	514.18
qwh-30-316-1	1215	603	**0.22**	234	115	0.32
qwh-30-316-2	118348	59104	20.06	8828	4397	**2.7**
qwh-30-316-3	8944	4451	1.68	3114	1552	**1.01**
qwh-35-405	2.38M	1.19M	562.62	475053	237369	**236.05**
bibd-7-3-2	100	39	**0.01**	65	28	**0.01**
bibd-7-3-3	383	180	0.03	96	42	**0.01**
bibd-7-3-4	6486	3210	0.79	132	56	**0.03**

We also did some experiments to see if the computation of domFD could be cheaply approximated. We used a counter with each variable to record the number of times that variable was at the origin of a weak dependency. This represents an approximation of domFD since the counter considers dependencies on instantiated variables. Unfortunately, this - fast - approximation is always beaten by domFD on large instances.

5. PREVIOUS WORK

In [2], the authors have proposed *dom-wdeg*, an heuristic which gives priority to variables frequently involved in failed constraints. It adds a weight to each constraint which is updated (i.e, incremented by one) each time the constraint fails. Using this value variables are ranked according to domain size and associated weight. X_i, the selected variable minimizes dom-wdeg(X_i)= $|X_i|/\sum_{c \in prop(X_i)} weight(c)$. As shown in the previous section, domFD is superior to dom-wdeg on many problems. Interestingly, while dom-wdeg can only learn information from conflicts, domFD can also learn from successful branchings. This is an important difference between these two techniques.

In [8], Refalo proposes the *impact* dynamic variable-value selection heuristic. The rational here is to maximize the reduction of the remaining search space. In this context an *impact* is computed taking into account the reduction of the search space due to an instantiated variable. Impact also considers values, and can therefore select the best instantiation instead of the best variable. With domFD, a variable is well ranked if its instantiation has generated several others instantiation. This is equivalent to an important pruning of the search space. In that respect domFD is close to impact. However, its principle is the dynamic exploitation of functional dependencies, not the explicit quantification of search space reductions. More generally, since dvo heuristics are all based on some understanding of the *fail-first* principle they are all aiming at an important reduction of the search space.

To improve SAT solving, [7] proposes a new pre-processing step that exploits the structural knowledge that is hidden in a CNF formula. It delivers an hybrid formula made of clauses together with a set of equations of the form $y = f(x_1, ..., x_n)$. This set of functional dependencies is then exploited to eliminate clauses and variables, while preserving satisfiability. This work detects real functions while our heuristic observes weak dependencies. Moreover, it uses a pre-processing step while we perform our learning during constraint propagation.

6. CONCLUSION

In this work, our goal was to heuristically discover a simplified form of functional dependencies between variables called *weak dependencies*. Once discovered, these relations are used to rank the branching variables. More precisely, each time a variable y gets instantiated as a result of the instantiation of x, a weak dependency (x, y) is recorded. As a consequence, the weight of x is raised, and the variable becomes more likely to be selected by the variable ordering heuristic.

Experiments done on 35 problems coming from 9 benchmarks families showed that on the average domFD reduces search trees by a factor 3 and runtime by 31% when compared against dom-wdeg, one of the best dynamic variable ordering heuristic. domFD is also more expensive to compute since it puts some overhead on the propagation engine. However, it seems that our implementation can be improved, for example by using incremental data structures to record potential dependencies in the propagation engine.

Our heuristic learns from successes, allowing a quick exploitation of the solver's work. In a way, this is complementary to dom-wdeg which learns from failures. Moreover, both techniques rely on the computation of *mindom*. Combining their respective strengths seems obvious. We did ex-

tensive experiments around a new mixture, $dom(x)/(wdeg(x)+FD(x))$ but found out that domFD was better than this straightforward combination.

7. REFERENCES

[1] D. Achlioptas, C. P. Gomes, H. A. Kautz, and B. Selman. Generating satisfiable problem instances. In *AAAI/IAAI*, pages 256–261. AAAI Press / The MIT Press, 2000.

[2] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *ICAI*, pages 146–150, 2004.

[3] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[4] B. N. Dilkina, C. P. Gomes, and A. Sabharwal. Tradeoffs in the complexity of backdoor detection. In C. Bessiere, editor, *CP'07*, volume 4741 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2007.

[5] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[6] M. Z. Lagerkvist and C. Schulte. Advisors for incremental propagation. In C. Bessiere, editor, *CP'07*, volume 4741 of *Lecture Notes in Computer Science*, pages 409–422. Springer, 2007.

[7] R. Ostrowski, E. Gregoire, B. Mazure, and L. Sais. Recovering and exploiting structural knowledge from CNF formulas. In *CP'02*, volume 2470 of *Lecture Notes in Computer Science*, pages 185–199. Springer, 2002.

[8] P. Refalo. Impact-based search strategies for constraint programming. In *CP'04*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004.

[9] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI*, pages 125–129, 1994.

[10] C. Schulte and M. Carlsson. Finite domain constraint programming systems. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 14, pages 495–526. Elsevier Science Publishers, 2006.

[11] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *IJCAI*, pages 1173–1178, 2003.