A heuristic based on domain-splitting nogoods from restarts

Luís Baptista[1,2] and Francisco Azevedo[1]

[1]CENTRIA, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal
fmaa@fct.unl.pt
[2]C3i, Instituto Politécnico de Portalegre, Portugal
lmtbaptista@gmail.com

Abstract. Inspired by Boolean Satisfiability Problems (SAT), Constraint Satisfaction Problems (CSP) are starting to use restart techniques associated with learning nogoods widely. Recent developments show how to learn nogoods from restarts and that these nogoods are of major importance when solving a CSP. Using a backtracking search algorithm, with domain-splitting branching, nogoods are learned from the last branch of the search tree, immediately before the restart occurs. This type of nogoods, named ds-nogoods, is still not proven to be effective in solving CSP. Nevertheless, information retained within ds-nogoods can be used in heuristic decisions. Inspired by activity-based heuristics of SAT solvers, we propose and evaluate a variable selection heuristic based on ds-nogoods. Experimental results show that this allows some problems to be solved more efficiently.

Keywords: constraint, restarts, heuristic, nogoods, domain-splitting

1 Introduction

The use of restart techniques associated with learning nogoods in solving hard CSPs is not widely used. On the other hand, the impressive progress in propositional satisfiability problems (SAT) has been achieved using restarts and nogoods recording. SAT and CSP share many solving techniques [1]. But, as noted in [2] the interest of the CSP community in restarts and nogood recording is growing.

The main contribution of this paper shows that nogoods recorded from restarts can improve backtrack search algorithms with domain-splitting branching. Nogoods recorded from restarts in backtrack search algorithms with 2-way branching scheme were shown to be of great importance when solving CSPs [3]. These nogoods were generalized to backtrack search algorithms that use other branching schemes [4], e.g., domain-splitting branching – domain-splitting (ds) nogoods. But, the importance of these ds-nogoods recorded from restarts, are still not proven to be effective.

In this paper we present a more formal description of ds-nogoods [4], analyze the space complexity of ds-nogoods, and show that ds-nogoods can be useful to improve search. Inspired by activity-based heuristics of SAT solvers [5], we use activity of variables involved in ds-nogoods to improve the variable selection heuristic. We

show, empirically, that this new heuristic can effectively improve the search algorithm. We also present results showing the interplay of this heuristic and different restart strategies.

The rest of the paper is organized as follows. Section 2 gives background about constraint satisfaction problems and backtrack search algorithms for solving them. In section 3 we explain ds-nogoods and in section 4 we present our new heuristic. Finally in section 5 we present and discuss our results and in section 6 we present conclusions and future work.

2 Background

A Constraint Satisfaction Problem is a triple, (*X*,*D*,*C*), consisting of a set of variables *X*, each with a domain of values, the set of domain *D*, and a set of constraints *C* on a subset of these variables. In this paper we will consider that the CSP has finite domains (CP(FD)). An assignment to all the variables that satisfies every constraint is a solution to the CSP. A problem is unsatisfiable if it does not have a solution. A propositional satisfiability problem (SAT) is a particular case of a CSP with Boolean variables and constraints defined by propositional logic expressed in conjunctive normal form.

Complete backtrack search algorithms are widely used for solving CSPs. At each node, a variable with no value assigned is selected, based on a variable selection heuristic, and a value is selected, from the actual domain of the variable, based on a value selection heuristic. Different branching schemes could be used. The d-way branching scheme creates one branch for each value of the variable. The 2-way branching scheme creates two branches, the left one for value assignment and the right one for value refutation. The domain splitting branching scheme [6] splits the domain in two sets, typically based on the lexicographic order of the values, and branches on those sets. Set branching refers to any branching scheme that splits the domain values in different sets, based on some similarity criterion [7], and branches on those sets.

The use of restart techniques and learning nogoods is still considered of small importance. A backtrack search algorithm is randomized by introducing a fixed amount of randomness in the branching heuristic [8]. The algorithm is repeatedly run (restart), each time limiting the maximum number of backtracks to a cutoff value. A good cutoff value eliminates the heavy-tail phenomena, but unfortunately such a value has to be found empirically [8]. A simple restart strategy can increment the cutoff value, by a constant, after each restart, thus guarantying completeness. Another restart policy, which is widely considered to work well in practice, was proposed in [9], where the cutoff values are geometrically increased by a geometric factor.

Nogood recording was introduced in [10], where a nogood is recorded when a conflict occurs during a backtrack search algorithm. Those recorded nogoods were used to avoid exploration of useless parts of the search tree. Contrary to CSP, learning is an important feature of SAT solver algorithms. Important progress in SAT solvers was due to the use of restarts, conflict clause recording [5, 11] and the use of very efficient data structures [5].

Standard nogoods correspond to variable assignments, but recently, a generalization of standard nogoods, that also uses value refutations, has been proposed by [12, 13]. They show that these generalized nogoods allow learning more useful nogoods from global constraints. This is an important point since state of the art CSP solvers rely on heavy propagators for global constraints.

Recently the use of standard nogoods and restarts in the context of CSP algorithms was applied with great success [3, 14]. They record a set of nogoods after each restart (at the end of each run). Those nogoods are computed from the last branch of the search tree before the restart. So, the already visited tree is guaranteed not to be visited again. This approach is similar to the one used for SAT, where clauses are recorded, from the last branch of the search tree before the restart (search signature) [15].

3 Domain-Splitting Nogoods

In this section we present Domain-Splitting (ds) nogoods as described in [4], but now using a more formal description, and extend the work with a space complexity analysis.

A ds-nogood is a generalization of the work presented in [3] about nogood recording from restarts, in the context of backtrack search algorithms with 2-way branching. In [4], using a bactrack search algorithm with domain-splitting branching, and adapting the concepts described in [3], a nogood recorded from a restart uses domain splitting decisions instead of assignment decisions.

Consider a search tree built by a backtracking search algorithm with a domain splitting branching scheme. As for the 2-way branching scheme this is also a binary tree. But now the domain is split lexicographically in one of the values.

Definition 1. Let $P=(X,D,C)$ be a CSP, $x_i \in X$ be a variable and $v_i \in d_i$ ($d_i \in D$) be a value from the domain of the variable. The constraint $x_i \leq v_i$ is called a positive decision and corresponds to constraining the variable to the left part of the domain. The negation of the positive decision, $\neg(x_i \leq v_i)$, is called a negative decision, $x_i > v_i$, and corresponds to constraining the variable to the right part of the domain. Also, the negation of a negative decision is a positive decision.

In a search tree, positive decisions are taken in the left branch and negative decisions in the right branch.

Definition 2. Let $\Sigma = \langle \delta_1, \ldots, \delta_m \rangle$ be a sequence of decisions. The sequence of positive and negative decisions of a variable $x \in X$ are denoted by $pos_x(\Sigma)$ and $neg_x(\Sigma)$, respectively. The set with the last decision of a sequence is denoted by $last(\Sigma)$ ($last(\Sigma) = \{\delta_m\}$ and $last(\langle\rangle) = \varnothing$).

Definition 3. Let $\Sigma = \langle \delta_1, \ldots, \delta_i, \ldots, \delta_m \rangle$ be a sequence of decisions and δ_i is a negative decision. The sequence $\langle \delta_1, \ldots, \delta_i \rangle$ is a negative last decision (nld) subsequence.

Proposition 1. Let P be a CSP, $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ be an nld-subsequence of decisions taken along a branch of the search tree, and $\Sigma' = \langle \delta_1, \ldots, \neg\delta_i \rangle$ be a sequence derived

from Σ where the last decision is negated and converted to a positive decision. The set created with all decisions of Σ', $\Delta = \{\delta_1, \ldots, \neg\delta_i\}$, is a ds-nogood.

Proof. In the search tree, positive decisions are taken first, so, if a negative decision δ_i appears then the subtree corresponding to the positive decision $\neg\delta_i$ was refuted.

As an example, consider the sequence of decisions before the restart (the last branch of the search tree), $\langle v \leq a, w > b, y > b, x \leq c, w > a, z > b \rangle$. We can extract the nld-subsequences, $\langle v \leq a, w > b \rangle$, $\langle v \leq a, w > b, y > b \rangle$, $\langle v \leq a, w > b, y > b, x \leq c, w > a \rangle$ and $\langle v \leq a, w > b, y > b, x \leq c, w > a, z > b \rangle$. The corresponding ds-nogoods are then, $\{v \leq a, w \leq b\}$, $\{v \leq a, w > b, y \leq b\}$, $\{v \leq a, w > b, y > b, x \leq c, w \leq a\}$ and $\{v \leq a, w > b, y > b, x \leq c, w > a, z \leq b\}$, respectively.

Similarly to reduced nld-nogoods [3], we can also have reduced ds-nogoods, considering only positive decisions. As an example, consider again the ds-nogoods of the last paragraph, then the reduced ds-nogoods are $\{v \leq a, w \leq b\}$, $\{v \leq a, y \leq b\}$, $\{v \leq a, x \leq c, w \leq a\}$, $\{v \leq a, x \leq c, z \leq b\}$, respectively.

As explained in [4] (reduced) ds-nogoods have potentially more pruning power than (reduced) nld-nogoods, because they use a more compact representation, since one decision can represent more than one decision of the nld-nogoods.

3.1 Simplifying ds-nogoods

By construction, a ds-nogood does not contain two opposite decisions, e.g., $x \leq a$ and $x > a$. But a ds-nogood can have more than one decision on the same variable.

Proposition 2. Let P be a CSP, $\Delta = \{\delta_1, \ldots, \delta_i\}$ be a ds-nogood and $\Sigma' = \langle \delta_1, \ldots, \delta_i \rangle$ be the sequence of decisions that create Δ. The simplified version of Δ is the ds-nogood $\Delta' = \cup_{x \in X} (last(pos_x(\Sigma')) \cup last(neg_x(\Sigma'))$. And the simplified version of the reduced version of Δ is the reduced ds-nogood $\Delta'' = \cup_{x \in X} last(pos_x(\Sigma'))$.

Proof. As already noted, by construction, nogoods do not contain two opposite decisions. A decision in the search tree will narrow the domain of a variable, decreasing the upper bound, a positive decision, or increasing the lower bound, a negative decision. Subsequent decisions on the same variable will further narrow the domain, hence, it suffices to maintain, for each variable, only the last positive and last negative decision, and safely remove the other decision.

As an example, consider the decisions over variable w in the ds-nogood $\{v \leq a, w > b, y > b, x \leq c, w > a, z \leq b\}$, $w > b$ and $w > a$. It is easy to see, that the decision $w > a$ subsumes $w > b$; because decision $w > a$ is made after $w > b$ we know that $a > b$; and we can safely remove decision $w > b$ Thus, a great compaction can be obtained with simplified ds-nogoods.

Proposition 3. Let P be a CSP, n the number of variables, d the size of the domains, and Σ be the sequence of decisions taken along a branch of the search tree. The space complexity to record all the ds-nogoods of Σ is $O(n^2d)$. The space complexity to record all the reduced ds-nogoods of Σ is also $O(n^2d)$.

Proof. The number of negative decisions in any branch is $O(nd)$. For each negative decision a ds-nogood (or a reduced version) is extracted. Because of Propositions 1 and 2 the size of any ds-nogood or any reduced ds-nogood is $O(n)$. So, the resulting space complexity is $O(n^2d)$.

4 Using ds-nogoods

When ds-nogoods are extracted from the last branch of the search tree, before the restart, they could be posted as constraints in the solver, avoiding exploration of the already searched tree. Trying to do this we were unable to show the usefulness of using ds-nogoods. Indeed, in all the empirically evaluation conducted we do not observe improvements in the search algorithm with ds-nogoods. Nevertheless, ds-nogoods retain information that could be used to help backtrack search algorithms.

Inspired by activity-based heuristic of SAT solvers [5], that use information of the activity of literals in conflict clauses (nogoods), we propose using ds-nogoods in the variable selection heuristic.

During search we maintain a counter for each variable. These counters maintain the number of times a variable appears in ds-nogoods, as the search evolves. Variables with higher activity are preferred. As in SAT solvers we divide the countings by two, from time to time (we use an interval of four restarts). In this way, the search concentrates efforts on the more recent variables appearing in more ds-nogoods.

We do not use only the activity of the variable in the variable selection heuristic. Instead, we use *dom*, a variable selection heuristic based on the fail-first principle, that chooses the variable with the smallest domain, and break ties with activity of the variables (variables with more activity are preferred). Actually, we compute a heuristic value for each variable i, based on the domain of i, dom_i, and the activity of i, act_i,

$$dom_i + \frac{1}{act_i + 1} \tag{1}$$

Finally it is important to say that our ds-nogoods are simplified.

5 Results

In our empirical study we use the Comet System , using the constraint programming solver over finite domains, in a dual core Pentium (E5200) at 2.5 GHz with 2GB of memory, running a 64 bits linux system.

We use instances of talisman squares. A talisman square is a magic square of size n but with constraints stating that the difference between two adjacent cells must be greater or equal than some constant, k. For each of the instances we run three algorithms: without restarts; with restarts; with restarts and heuristic based on ds-nogoods. Each algorithm is run 100 times for each of the instances.

All the algorithms use domain-splitting search, and a value heuristic that chooses the splitting value randomly, from the actual domain of the variable. To evaluate the

execution of the algorithms we use the number of fails needed to find a solution. If the algorithms could not find a solution within 100000 fails, the execution is aborted.

The first algorithm, without restarts, uses *dom*, a variable selection heuristic based on the fail-first principle, that chooses the variable with the smallest domain, breaking ties randomly. The second algorithm is equal to the first one plus restarts, but chooses randomly between the variables with the two best heuristic values. We use a restart strategy with initial cutoff of 1000 fails and after each restart we increment the cutoff by 5 fails. The third algorithm is equal to the second one, but the variable selection heuristic uses information from nogoods, as described in the previous section, and we choose randomly between the variables with the three best heuristic values. We use nogoods only to compute the weight of variables. We do not post nogoods as constraints in the solver.

Table 1 below summarizes the results of running the three algorithms. The first column indicates the instances used, where the first number is the size *n* of the talisman and the second number is the minimum difference *k* between adjacent cells. The next columns define, respectively, for each algorithm, the average number of fails, the average runtime, in milliseconds, and the number of runs aborted. When computing the average of the runtime the aborted runs are included. This runtime depends on the algorithm used, and, even for the same algorithm, it is not the same for all runs with 100000 fails.

Table 1. Average number of fails, runtime and aborts, for 100 runs

Talis-man	without restarts			restarts			restarts + ds-nogoods		
(n;k)	#fails	time	Aborts	#fails	time	aborts	#fails	time	aborts
(4;1)	333	24	0	586	41	0	796	55	0
(5;1)	35076	2580	8	32624	2608	7	33025	2731	8
(6;1)	82851	6890	76	38395	3518	5	33311	3064	7
(7;1)	77451	7355	71	29432	3160	3	25246	2622	3
(8;1)	99008	11028	99	63089	7885	35	39481	4518	9
(9;1)	98112	11871	98	79017	10623	55	44614	5752	14
(10;1)	100000	13440	100	88829	13577	81	80283	11412	60

As we can see, the first two instances are easier, and all the algorithms have equivalent performances. This means that, for instances that are easy, the use of restarts and ds-nogoods are not useful. But, when instances are starting to be harder, as in talisman (6;1) and (7;1), the use of restarts is essential. In these cases the number of aborts decreases by one order of magnitude when using restarts. Still, for these instances adding nogoods information is not useful.

However, if instances are hard, as the last three talismans, the use of restarts and nogoods are crucial for the success of the third algorithm. In talisman (8;1) we have again a reduction of one order of magnitude of the number of aborts, when comparing with the first algorithm.

Results in table 1 could indicate an easy-hard phase transition phenomena. If the instances are easy then our proposed techniques are not useful, but when the instances

are in the hard region then our proposed techniques are crucial for solving the instances.

But, even for instances in the easy region, we can use the third algorithm, because this algorithm is equivalent, or better than the other two, concerning the runtime and aborts. Thus, this algorithm is suitable for practical application, since the computational overhead of the use of restarts and ds-nogoods is compensated by performance improvements. As can be observed in table 1 the third algorithm allows solving the instances faster, sometimes two times faster, or in the easy instances in equivalent time.

Table 2. Minimum number of fails, runtime and aborts, for 100 runs

Talis-man	without restarts			restarts			restarts + ds-nogoods		
(n;k)	#fails	time	aborts	#fails	time	aborts	#fails	time	aborts
(4;1)	1	0	0	3	0	0	2	0	0
(5;1)	45	10	8	35	10	7	279	30	8
(6;1)	177	20	76	43	10	5	612	50	7
(7;1)	1462	100	71	344	40	3	1804	190	3
(8;1)	864	100	99	1830	220	35	2754	300	9
(9;1)	1606	160	98	1156	230	55	1869	290	14
(10;1)	100000	7910	100	6333	1160	81	4596	670	60

In table 2 we can see the minimum number of fails and the minimum runtime that the algorithms used to solve the instances, in one of the 100 runs. We present again the number of aborts for reference. This table allows us to see that even for hard instances, e.g., (8;1) and (9;1), there was at least one run where the algorithm needed only a small amount of fails (milliseconds) to solve the instances, when comparing with the average case.

Table 3. Number of fails in the last restarts of each run

Talisman	restarts #fails in last restart		
(n;k)	average	min	max
(4;1)	306,2	0	989
(5;1)	518,3	8	1302
(6;1)	526,9	14	1251
(7;1)	558,5	27	1333
(8;1)	649	62	1176
(9;1)	660,1	155	1289
(10;1)	682,8	218	1117

In table 3 we summarize results concerning the number of fails the algorithm with restarts used in the last restart, just before the solution has been found. Aborts of the algorithm were not considered. We show the average number of fails and the minimum and maximum number of fails that occurs in the last restart of each run. We only consider this algorithm because the restarts are completely independent. The same

could not be stated for the algorithm that uses restarts and ds-nogoods, because we use information from the past restarts in the variable selection heuristic, which will make the future decision dependent from the past ones. The results in this table allow us to strengthen what we stated for table 2. Actually, even the hard instance, i.e., (8;1), (9;1) and (10;1), had been solved, at least in one case, with a (really) small number of fails. But if we look at the maximum fails we can still conclude the same, because, in the worst case, all the instances solved used only a number of fails between 1000 and 1400. This means that restarts are important in this problem, but also, that a restart strategy should not increment the cutoff value too much.

Table 4. Comparing different restart strategies

strategy	(8;1)			(9;1)			(10;1)		
	#fails	time	aborts	#fails	time	aborts	#fails	time	aborts
+1 (95)	35144	4059	7	53941	7066	26	80319	14218	63
+5 (82)	39482	4518	9	44614	5752	14	80283	11412	60
+10 (73)	40192	5485	15	44104	7008	15	82386	15029	66
+20 (62)	33995	4698	6	51050	8040	24	76950	14065	56
+50 (46)	37236	5117	11	51569	8012	25	80040	14337	63
+100 (36)	50209	6974	19	48784	7418	26	79455	13797	64
*2 (6)	75173	9611	62	81465	11512	69	94013	15659	90
*1.5 (9)	57805	7477	41	70436	10103	52	93896	15302	87
*1.1 (25)	47151	6018	19	62378	9466	39	88565	14848	72
*1.05 (36)	41443	5559	13	51927	7912	26	79059	12657	63
*1.01 (69)	39349	4606	9	50275	6406	22	80625	11392	59

Table 4 compares different restart strategies for the harder instances. Restart strategies are identified by a plus signal (+) and a value, representing a linear incremental strategy on the cutoff by the value specified, or by a star signal (*) and a value, representing a geometrical incremental strategy on the cutoff by the value specified. The number in parenthesis indicates the number of restarts needed until the predefined limit of 100000 fails was reached. All the strategies start with a cutoff of 1000. We use the third algorithm (restarts + ds-nogoods). For each instance the average number of fails, the average runtime in milliseconds, and the number of aborts are presented.

The results in table 4 are not conclusive about the best linear strategy. But in the geometrical strategies we can conclude that a high geometrical factor (fast growing of the cutoff value) has a poor behavior when compared to lower geometrical factors. And, we can also observe that the geometrical strategy only has a performance compared with the linear strategy when the geometrical factor is very low and the number of restarts starts to be near the ones of the incremental strategy. Restarting means learning information from ds-nogoods, and so, the algorithm needs to restart, at least, a minimum number of restarts. Hence, we can conclude that, for these instances, frequent restarts are better, and a linear incremental restart strategy is better than a geometrical one.

6 Conclusions and Future Work

The use of restarts with nogoods recording in backtrack search algorithms for solving CSPs is starting to be considered of great importance. But the use of ds-nogoods in a domain splitting backtrack search algorithm with restarts is still not proven to be effective in solving CSPs. In this paper we present a formal description of ds-nogoods, propose a variable selection heuristic based on activity of variables in ds-nogoods, and analyze restart strategies.

From the empirical evaluation of the proposed heuristic, we can conclude that the use of restarts can improve the performance of the search algorithm. But, for harder instances, restarts are not enough, and the use of our proposed heuristic is crucial for solving those hard instances. We also show that frequent restarts are better than slow restarts.

We know that a domain splitting branching is useful for optimization problems and when the domains sizes of the variables are very large. So, in the near future we expect to empirically evaluate other problems where ds-nogoods could be useful. But this work is included in a wider research project, whose aim is to study the use of restarts in constraint programming with finite domains. We will evaluate the interplay of different techniques associated with restarts, namely, nogoods, search restart strategies, randomization and heuristics.

Acknowledgments. This work is supported by the Portuguese "Fundação para a Ciência e a Tecnologia" and "Instituto Politécnico de Portalegre" (SFRH/PROTEC/49859/2009).

References

1. Bordeaux, L., Hamadi, Y., Zhang, L.: Propositional Satisfiability and Constraint Programming: A comparative survey. ACM Comput. Surv. 38, 12 (2006).
2. Lecoutre, C.: Constraint Networks: Techniques and Algorithms. Wiley-ISTE (2009).
3. Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Nogood recording from restarts. IJCAI. pp. 131–136. Morgan Kaufmann Publishers Inc., Hyderabad, India (2007).
4. Baptista, L., Azevedo, F.: Domain-Splitting Generalized Nogoods from Restarts. In: Antunes, L. e Pinto, H. (eds.) Progress in Artificial Intelligence. pp. 679–689. Springer Berlin / Heidelberg (2011).
5. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. DAC. pp. 530–535. ACM (2001).
6. Dincbas, M., Hentenryck, P.V., Simonis, H., Aggoun, A., Graf, T., Berthier, F.: The Constraint Logic Programming Language CHIP. FGCS'88. pp. 693–702 (1988).
7. Balafoutis, T., Paparrizou, A., Stergiou, K.: Experimental Evaluation of Branching Schemes for the CSP. CP - TRICS Workshop. pp. 1–12 (2010).
8. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. Proceedings of the fifteenth national conference on Artificial intelligence. pp. 431–437. American Association for Artificial Intelligence, Madison, Wisconsin, United States (1998).

9. Walsh, T.: Search in a Small World. Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence. pp. 1172–1177. Morgan Kaufmann Publishers Inc. (1999).
10. Dechter, R.: Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. Artif. Intell. 41, 273–312 (1990).
11. Baptista, L., Silva, J.P.M.: Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability. CP. pp. 489–494. Springer-Verlag (2000).
12. Katsirelos, G., Bacchus, F.: Unrestricted Nogood Recording in CSP Search. CP. pp. 873–877 (2003).
13. Katsirelos, G., Bacchus, F.: Generalized NoGoods in CSPs. AAAI. pp. 390–396 (2005).
14. Lecoutre, C., Saïs, L., Tabary, S., Vidal, V.: Recording and Minimizing Nogoods from Restarts. JSAT. 1, 147–167 (2007).
15. Baptista, L., Lynce, I., Marques-Silva, J.: Complete Search Restart Strategies for Satisfiability. IJCAI-SSA. (2001).