

# Extending Unit Propagation Look-Ahead of DPLL Procedure

Anbulagan

Logic and Computation Program, Canberra Research Laboratory  
National ICT Australia Limited  
Locked Bag 8001, Canberra, ACT 2601, Australia  
anbulagan@nicta.com.au

**Abstract.** The DPLL (Davis-Putnam-Logemann-Loveland) procedure is one of the most effective methods for solving SAT problems. It is well known that its efficiency depends on the choice of the branching rule. Different branching rules are proposed in the literature. Unit propagation look-ahead (UPLA) branching rule was one of the main improvements in the DPLL procedure (e.g., [10]). The UPLA branching rule integrated in *satz* SAT solver [10] performs a series of variable filtering process at each node as a static variable filtering agency. In this paper we introduce and experiment with dynamic variable filtering (DVF) based branching rule which extends the UPLA heuristic process for doing more filtering and choosing a best branching variable from an irreducible sub-formula. To enhance the performance of DVF branching rule, we integrate neighborhood variable ordering heuristic (NVO) for exploring only the neighborhood variables of the current assigned variable. Experimental results of DVF+NVO branching rule on a number of real-world benchmark instances and quasigroup problems prove our approaches to be useful in many circumstances.

## 1 Introduction

The satisfiability (SAT) problem is central in mathematical logic, artificial intelligence and other fields of computer science and engineering. In conjunctive normal form (CNF), a SAT problem can be represented as a propositional formula  $\mathcal{F}$  on a set of Boolean variables  $\{x_1, x_2, \dots, x_n\}$ . A literal  $l$  is then a variable  $x_i$  or its negated form  $\bar{x}_i$ , and a clause  $c_i$  is a logical *or* of some literals such as  $x_1 \vee x_2 \vee \bar{x}_3$ . A propositional formula  $\mathcal{F}$  consists of a logical *and* of several clauses, such as  $c_1 \wedge c_2 \wedge \dots \wedge c_m$ , and is often simply written as a set  $\{c_1, c_2, \dots, c_m\}$  of clauses.

Given  $\mathcal{F}$ , the SAT problem involves testing whether all the clauses in  $\mathcal{F}$  can be satisfied by some consistent assignment of truth values  $\{true, false\}$  to the variables. If this is the case,  $\mathcal{F}$  is satisfiable; otherwise it is unsatisfiable.

One of the best known and most widely used algorithms to solve SAT problems is the DPLL (Davis-Putnam-Logemann-Loveland) procedure [3]. Many SAT solvers such as Posit [5], Tableau [2], *satz* [10], and *cnfs* [4] are based

on this procedure. DPLL essentially enumerates all possible solutions to a given SAT problem by setting up a binary search tree and proceeding until it either finds a satisfying truth assignment or concludes that no such assignment exists. It is well known that the search tree size of a SAT problem is generally an exponential function of the problem size, and that the branching variable selected by a branching rule at a node is crucial for determining the size of the sub-tree rooted at that node. A wrong choice may cause an exponential increase of the sub-tree size. Hence, the actual performance of a DPLL procedure depends significantly on the effectiveness of the branching rule used.

In general, the branching rules compute  $w(x_i)$  and  $w(\bar{x}_i)$ , where the function  $w$  measures the quality of branching to literals  $(x_i)$  or  $(\bar{x}_i)$ . The DPLL procedure should select the branching variable  $(x_i)$  such that  $w(x_i)$  and  $w(\bar{x}_i)$  are the highest. Whether to branch on  $(x_i)$  or  $(\bar{x}_i)$  is only important for the satisfiable problems because the literals chain created, when the truth value is appropriately assigned to the branching variables, will reduce the search tree size. We can reduce the search tree size that a DPLL procedure explores if we extend the branching rule with an appropriate heuristic.

Much of the research on DPLL has focussed on finding clever branching rules to select the branching variable that most effectively reduces the search space. Among them, Li and Anbulagan have performed a systematic empirical study of unit propagation look-ahead (UPLA) heuristics in [10] and integrated the optimal UPLA in *satz* SAT solver. The effectiveness of UPLA in *satz* has made this solver one of the best solvers for solving hard random and a number of real-world SAT problems.

The UPLA branching rule of *satz* performs a series of variable filtering process at each node as a *static* variable filtering agency. The UPLA heuristic itself carries out one of the following actions during two propagations of a free variable at each search tree node: detecting a contradiction earlier, simplifying the formula, or weighing the branching variable candidates. Intuitively, at a node, the formula simplification process by UPLA can cause the previously selected branching variable candidates become ineffective. To handle the problem, in this paper we introduce and experiment with dynamic variable filtering (DVF) based branching rule.

The key idea underlying this new branching rule is to further detect failed literals that would remain undiscovered using a UPLA branching rule, before choosing a branching variable. In other words, we perform more reasoning in the open space between the UPLA heuristic and the MOMS (Maximum Occurrences in clause of Minimum Size) heuristic in the actual DPLL branching rule. To test this idea, we use *satz215* (the best version of the *satz* DPLL procedure) where we simply replace its branching rule by a new branching rule. The new rule allows filtering of free variables, and at the same time reduces the sub-formula size at each node until the filtering process is saturated. Then, it chooses a best branching variable from an irreducible sub-formula.

Since the DVF based branching rule examines all free variables many times at each node, we attempt to limit the number of free variables examined by only

exploring the neighborhood variables of the current assigned variable. For this purpose, we additionally integrate the neighborhood variable ordering (NVO) heuristic for enhancing the performance of DVF.

The experimental results of DVF+NVO branching rule on a number of real-world benchmark instances and quasigroup problems prove our approaches to be useful in many circumstances. This study also raises a number of other possibilities for enhancing the performance of DVF+NVO branching rule to solve more SAT problems, e.g., by avoiding redundant unit propagation searches for variables remain unchanged between iteration of UPLA heuristic.

In the next section we describe the background of this work in more detail. In section 3 we present the DVF based branching rule and its extension, which integrates the NVO heuristic. In section 4, we present some experimental results to give a picture of the performance of our new branching rules on a number of structured SAT problems. Finally, we conclude the paper with some remarks on current and future research.

## 2 Unit Propagation Look-Ahead Based Branching Rule

The UPLA heuristic plays a crucial role in a DPLL procedure and is used to reach dead-ends earlier with the aim of minimising the length of the current path in the search tree. The earlier SAT solvers which used the power of UPLA partially were *POSIT* [5] and *Tableau* [2]. Then Li and Anbulagan conducted a systematic empirical study to explore the real power of the UPLA heuristic and integrated the optimal UPLA heuristic in a SAT solver called *satz* [10]. The success of *POSIT*, *Tableau*, and *satz* in solving hard random 3-SAT and a number of real-world problems shows the effectiveness of this heuristic.

We distinguish the UPLA heuristic from the conventional unit propagation procedure (UP) that is usually used in DPLL as follows: UP is executed to reduce the size of a sub-formula possessing unit clauses *after* a branching variable is selected, while UPLA is integrated in the branching rule itself and is executed at each search tree node. In figure 1, we present a branching rule which integrates the UPLA heuristic on top of the MOMS heuristic.

Given a variable  $x_i$ , the UPLA heuristic examines  $x_i$  by adding the two unit clauses possessing  $x_i$  and  $\bar{x}_i$  to  $\mathcal{F}$  and independently making two unit propagations. These propagations result in a number of newly produced binary clauses, which are then used to weigh the variable  $x_i$ . This is calculated in figure 1, using the function  $diff(\mathcal{F}_1, \mathcal{F}_2)$  which returns the number of new binary clauses in  $\mathcal{F}_1$  that were not in  $\mathcal{F}_2$ . Let  $w(x_i)$  be the number of new binary clauses produced by setting the variable to *true*, and  $w(\bar{x}_i)$  be the number of new binary clauses produced by setting the variable to *false*. When there is no contradiction found during the two unit propagations, then variable  $x_i$  will be piled up to the branching variable candidates stack  $\mathcal{B}$ . The DPLL procedure then uses a MOMS heuristic to branch on the variable  $x_i$  such that  $w(\bar{x}_i)*w(x_i)*1024+w(\bar{x}_i)+w(x_i)$  is the highest. The branching variable selected follows the two-sided Jeroslow-Wang (J-W) rule [7] designed to balance the search tree.

```

 $\mathcal{B} := \emptyset;$ 
For each free variable  $x_i$ , do
Begin
  let  $\mathcal{F}'_i$  and  $\mathcal{F}''_i$  be two copies of  $\mathcal{F}$ 
   $\mathcal{F}'_i := \text{UP}(\mathcal{F}'_i \cup \{x_i\}); \mathcal{F}''_i := \text{UP}(\mathcal{F}''_i \cup \{\bar{x}_i\});$ 
  If both  $\mathcal{F}'_i$  and  $\mathcal{F}''_i$  contain an empty clause then backtrack();
  else if  $\mathcal{F}'_i$  contains an empty clause then  $x_i := \text{false}; \mathcal{F} := \mathcal{F}'_i;$ 
  else if  $\mathcal{F}''_i$  contains an empty clause then  $x_i := \text{true}; \mathcal{F} := \mathcal{F}''_i;$ 
  else
     $\mathcal{B} := \mathcal{B} \cup \{x_i\}; w(x_i) := \text{diff}(\mathcal{F}'_i, \mathcal{F})$  and  $w(\bar{x}_i) := \text{diff}(\mathcal{F}''_i, \mathcal{F});$ 
End;

For each variable  $x_i \in \mathcal{B}$ , do  $\mathcal{M}(x_i) := w(\bar{x}_i) * w(x_i) * 1024 + w(\bar{x}_i) + w(x_i);$ 
Branch on the free variable  $x_i$  such that  $\mathcal{M}(x_i)$  is the highest.

```

**Fig. 1.** The UPLA based branching rule

The UPLA heuristic also allows the earlier detection of the so-called failed literals in  $\mathcal{F}$ . These are literals  $l$  where  $w(l)$  counts an empty clause. For such variables, DPLL immediately tries to satisfy  $\bar{l}$ . If there is a contradiction during the second unit propagation, DPLL will directly perform backtracking, else the size of the sub-formula is reduced which allows the selection of a set of best branching variable candidates at each node in search tree.

So, during two propagations of a free variable through the UPLA heuristic, the following three circumstances can occur:

- The free variable selected becomes a candidate for the branching variable.
- Only one contradiction is found during two unit propagations, meaning the size of formula  $\mathcal{F}$  will be reduced during the other successful unit propagation process.
- Two contradictions are found during two unit propagations causing the search to backtrack to an earlier instantiation.

Daniel Le Berre suggested the further detection of implied literals within UPLA heuristic [9], resulting in the latest and best version of *satz*, *satz215*. The *satz215* DPLL procedure generally uses a reasoning based on unit propagation to deduce implied literals in order to simplify  $\mathcal{F}$  before choosing a best branching variable. For example, if  $\mathcal{F}$  contains no unit clause but two binary clauses  $(x \vee y)$  and  $(x \vee \bar{y})$ , unit propagation in  $\mathcal{F} \cup \{\bar{x}\}$  leads to a contradiction. Therefore,  $x$  is an implied literal and could be used to simplify the formula  $\mathcal{F}$  directly.

Intuitively, at a node, the formula simplification process of UPLA in *satz215* can cause the previously selected branching variable candidates become ineffective. To handle the problem, in the next section we propose a new branching rule which does more reasoning to choose a best branching variable from an irreducible sub-formula. We term this reasoning technique the *dynamic variable filtering* (DVF) heuristic.

### 3 Dynamic Variable Filtering Based Branching Rule

The main objective of using UPLA in *satz215* DPLL procedure is to detect contradictions earlier or to find a set of best branching variable candidates. In reality, UPLA heuristic in *satz215* performs a series of variable filtering processes at each node as a static variable filtering agency, because it will only perform between one to three filtering processes at each node (depending on the search tree height). During the filtering process, some variables are assigned the value *true* or *false* through a forced unit propagation when a contradiction occurs during another unit propagation. Note that the UPLA examines a free variable by performing two unit propagations. This process will automatically reduce the size of sub-formula and collect the (almost) best branching variable candidates at each node of the search tree.

```

Do
   $\mathcal{F}_{init} := \mathcal{F}; \mathcal{B} := \emptyset;$ 
  For each free variable  $x_i$ , do
    Begin
      let  $\mathcal{F}'_i$  and  $\mathcal{F}''_i$  be two copies of  $\mathcal{F}$ 
       $\mathcal{F}'_i := \text{UP}(\mathcal{F}'_i \cup \{x_i\}); \mathcal{F}''_i := \text{UP}(\mathcal{F}''_i \cup \{\bar{x}_i\});$ 
      If both  $\mathcal{F}'_i$  and  $\mathcal{F}''_i$  contain an empty clause then backtrack();
      else if  $\mathcal{F}'_i$  contains an empty clause then  $x_i := false; \mathcal{F} := \mathcal{F}'_i;$ 
      else if  $\mathcal{F}''_i$  contains an empty clause then  $x_i := true; \mathcal{F} := \mathcal{F}''_i;$ 
      else
         $\mathcal{B} := \mathcal{B} \cup \{x_i\}; w(x_i) := \text{diff}(\mathcal{F}'_i, \mathcal{F})$  and  $w(\bar{x}_i) := \text{diff}(\mathcal{F}''_i, \mathcal{F});$ 
      End;
    Until ( $\mathcal{F} = \mathcal{F}_{init}$ );

  For each variable  $x_i \in \mathcal{B}$ , do  $\mathcal{M}(x_i) := w(\bar{x}_i) * w(x_i) * 1024 + w(\bar{x}_i) + w(x_i);$ 
  Branch on the free variable  $x_i$  such that  $\mathcal{M}(x_i)$  is the highest.

```

**Fig. 2.** The DVF based branching rule

Our work is based on the insight that the size of a sub-formula during the variable filtering process can be further reduced in the UPLA based DPLL procedures. Here, we propose a new heuristic called the *dynamic variables filtering* (DVF) heuristic that further filters the free variables and at the same time reduces the sub-formula size at each node until the filtering process is saturated. We illustrate the new branching rule powered by DVF heuristic in figure 2.

We expect this new heuristic to perform better than the UPLA heuristic in terms of reducing the search tree size. To verify this, we carried out an empirical study and modified the branching rule of the DPLL procedure *satz215*<sup>1</sup> for our purpose. The *satz215* DPLL procedure is the best version of *satz* in our

<sup>1</sup> available from [www.laria.u-picardie.fr/~cli/EnglishPage.html](http://www.laria.u-picardie.fr/~cli/EnglishPage.html)

experiments. A new DPLL procedure based on the DVF heuristic, *ssc34*, are proposed.

The *ssc34* solver is the same as the *satz215* solver, except we replace the branching rule used in *satz215* with the DVF heuristic based branching rule. It performs the variable filtering process until the sub-formula cannot be further reduced at each node before a branching variable selected. In fact, *ssc34* examines the free variables many times using the UPLA heuristic at each search tree node. One might think that this saturation process is very costly, but it is not the case.

### 3.1 Neighborhood Variable Ordering Heuristic

Since DVF based branching rule of *ssc34* examines all free variables many times using the UPLA heuristic at each node, we attempt to limit the number of free variables examined by only exploring the neighborhood variables of the current assigned variable. For this purpose, we create the *ssc355* DPLL procedure by integrating a simple *neighborhood variable ordering* (NVO) heuristic in *ssc34*. Bessi ere et. al. [1] proposed a formulation of the dynamic variable ordering heuristic in the CSP domain that takes into account the properties of the neighborhood of the variable. The main objective of our simple NVO heuristic in *ssc355* is to restrict the number of variables examined by UPLA in the DVF heuristic.

## 4 Experimental Evaluation

To evaluate the effectiveness of our proposed filtering techniques, we compare *ssc34* and *ssc355* DPLL procedures with *satz215* on a number of structured SAT benchmark instances. These instances are well known in the literature and taken from different domain problems, such as bounded model checking, circuit verification, planning, scheduling, security and quasigroup problems. All instances have been downloaded from SATLIB ([www.satlib.org](http://www.satlib.org)) except the *lg\** problems which have been downloaded from SIMLIB ([www.mrg.dist.unige.it/star/sim/](http://www.mrg.dist.unige.it/star/sim/)). The test consists of 125 instances where 57 of them are satisfiable and the other 68 are unsatisfiable. The number of variables of those instances varies from 317 to 939,040 and the number of clauses varies from 27 to 228,329. This experiment was conducted on a Intel Pentium 4 PC with a 3 GHz CPU under Linux. The run time is expressed in seconds. The time limit, to solve a problem, is set to 3600 seconds.

### 4.1 Performance on Real-World Instances

In table 1, we present the comparative results of *satz215* (uses UPLA heuristic), *ssc34* (uses DVF heuristic) and *ssc355* (uses DVF+NVO heuristics) on the well known real-world instances. The table shows the search tree size (number of branching nodes) and the run time (in seconds) required to solve a given problem. The bracketed numbers in the problem column indicate the number of instances

Domain	Problem	#Vars	#Cls	Search Tree Size			Run Time		
				satz215	ssc34	ssc355	satz215	ssc34	ssc355
BMC	barrel5	1407	5383	1072	1200	<b>472</b>	33.28	<b>20.13</b>	38.41
	barrel6	2306	8931	4304	2600	<b>2560</b>	270.53	<b>102.70</b>	407.41
	barrel7	3523	13765	12704	<b>2643</b>	8656	1895.57	<b>594.00</b>	3344.53
	longmult8	3810	11877	11931	10881	<b>7449</b>	<b>234.91</b>	1012.22	485.56
	longmult9	4321	13479	18453	14447	<b>10917</b>	<b>459.39</b>	1825.39	1131.52
	longmult10	4852	15141	23854	n/a	<b>13207</b>	<b>735.78</b>	> 3600	1311.43
	longmult11	5403	16863	28951	n/a	<b>14558</b>	<b>997.61</b>	> 3600	1617.30
	longmult12	5974	18645	29574	n/a	<b>15268</b>	<b>1098.05</b>	> 3600	1819.64
	longmult13	6565	20487	28686	n/a	<b>15278</b>	<b>1246.26</b>	> 3600	2126.64
	longmult14	7176	22389	29721	n/a	<b>15598</b>	<b>1419.23</b>	> 3600	2419.84
	longmult15	7807	24351	32719	n/a	<b>17375</b>	<b>1651.43</b>	> 3600	3002.06
	queueinvar12	1112	7335	276	195	<b>94</b>	<b>0.81</b>	4.05	3.24
	queueinvar14	1370	9313	1019	399	<b>169</b>	<b>1.96</b>	12.68	6.45
	queueinvar16	1168	6496	293	287	<b>110</b>	<b>1.05</b>	6.23	4.06
	queueinvar18	2081	17368	5695	<b>1566</b>	1797	<b>11.51</b>	72.85	35.61
queueinvar20	2435	20671	8865	2607	<b>2238</b>	<b>18.85</b>	144.60	70.35	
CIRCU	eq_checking (34)	18055	31162	11677	2961	<b>2834</b>	5.68	5.90	<b>5.41</b>
	par16 (10)	6740	23350	5813	5894	<b>4717</b>	<b>11.75</b>	23.85	24.77
PLAN	bw_large.c	3016	50457	4	4	15	<b>1.33</b>	20.75	21.99
	bw_large.d	6325	131973	705	n/a	<b>466</b>	<b>220.26</b>	> 3600	1081.17
	hanoi4	718	4934	8055	8197	<b>4462</b>	<b>4.91</b>	18.00	13.77
	lg28	7022	212453	n/a	n/a	<b>37</b>	> 3600	> 3600	<b>14.60</b>
	lg283	7268	227148	n/a	n/a	<b>93</b>	> 3600	> 3600	<b>30.29</b>
	lg284	7268	227293	n/a	n/a	<b>33</b>	> 3600	> 3600	<b>15.15</b>
	lg285	7295	228325	n/a	n/a	<b>37</b>	> 3600	> 3600	<b>15.16</b>
	lg286	7295	228329	n/a	n/a	<b>39</b>	> 3600	> 3600	<b>17.12</b>
	lg291	6668	166247	n/a	n/a	<b>3072</b>	> 3600	> 3600	<b>885.44</b>
	log.a	828	6718	12640	0	0	2.64	0.13	<b>0.12</b>
	log.b	843	7301	6	293	0	<b>0.08</b>	0.54	0.10
	log.c	1141	10719	507	1632	1	0.35	3.47	<b>0.23</b>
	log.d	4713	21991	0	520	1	<b>0.67</b>	51.31	5.53
SCHED	e0ddr2-10-by-5-1	19500	103887	1	1	29	<b>14.97</b>	40.75	153.03
	e0ddr2-10-by-5-4	19500	104527	n/a	1	n/a	> 3600	<b>57.34</b>	> 3600
	enddr2-10-by-5-1	20700	111567	0	0	1	<b>35.66</b>	74.11	127.55
	enddr2-10-by-5-8	21000	113729	0	2	222	<b>48.10</b>	58.02	108.64
	ewddr2-10-by-5-1	21800	118607	0	0	2	<b>24.58</b>	57.04	125.81
ewddr2-10-by-5-8	22500	123329	0	3	0	<b>23.25</b>	46.43	60.68	
SECUR	cnf-r1 (8)	2920867	35391	225	0	0	1.40	2.64	<b>1.05</b>
	cnf-r2 (8)	2986215	63698	17	0	0	2.42	4.05	<b>2.20</b>
	cnf-r3-b1-k1.1	21536	8966	2008485	<b>1265</b>	3551	2965.68	<b>70.65</b>	124.32
	cnf-r3-b1-k1.2	152608	8891	n/a	3002	<b>1500</b>	> 3600	174.00	<b>52.62</b>
	cnf-r3-b2-k1.1	152608	17857	128061	0	0	792.08	1.05	<b>0.88</b>
	cnf-r3-b2-k1.2	414752	17960	181576	0	0	1253.54	1.19	<b>1.09</b>
	cnf-r3-b3-k1.1	283680	26778	31647	0	0	447.66	1.89	<b>1.51</b>
	cnf-r3-b3-k1.2	676896	27503	38279	0	0	600.35	2.25	<b>1.64</b>
	cnf-r3-b4-k1.1	414752	35817	11790	0	0	347.51	3.00	<b>2.41</b>
cnf-r3-b4-k1.2	939040	35963	20954	0	0	623.98	3.37	<b>2.71</b>	

Table 1. Run time (in seconds) and search tree size of real-world SAT problems. The best performances are in bold.

solved for that class of problems. For those problems, the *#Vars*, *#Cls*, *Search Tree Size*, and *Time* indicate the sum from all instances solved.

Bounded model checking (BMC) is the problem of checking if a model satisfies a temporal property in paths with bounded length  $k$ . We experiment with SAT-encoded BMC domain problems. We select the most representative *barrel\**, *longmult\** and *queueinvar\** instances from this domain. All instances are unsatisfiable. The results on BMC problems indicate that *ssc355* has its best performance, in term of search tree size, even though it still suffers from run time point of view. This means that on BMC domain problems, the DVF+NVO branching rule performs well to choose a best branching variable from an irreducible sub-formula.

We solve the equivalence verification and parity instances from circuit domain (CIRCU) problems. All equivalence verification instances are unsatisfiable and all parity instances are satisfiable. The *ssc355* has its best performance from search tree size point of view.

We solve also the blocks world, hanoi and logistics instances from planning domain (PLAN) problems. The *lg\** problems are unsatisfiable, while the other problems of this domain are satisfiable. The results on those problems indicate that *ssc355* has its best performance in general. The DVF+NVO based DPLL procedure can solve the *lg\** problems, while the *satz215* and *ssc34* unable to solve those problems in the given time limit.

The *ssc34* DPLL procedure can solve all job shop scheduling instances from scheduling domain (SCHED) problems. All instances are satisfiable. The *satz215* and *ssc355* fail to solve the problem *e0ddr2-10-by-5-4* in one hour.

We solve the data encryption standard (DES) instances of security domain (SECUR) problems. These are SAT-encoding of cryptographic key search problem. All instances are satisfiable. The *ssc355* DPLL procedure has its best performance on those problems. While the UPLA branching rule has the difficulty to solve those problems.

The simplistic version of NVO heuristic performed well on *longmult\** instances of BMC domain and the instances of planning domain. These results encourage us to explore further the power of NVO heuristic. Moreover, the DVF+NVO branching rule can solve all the problems in our experiment, except the *e0ddr2-10-by-5-4* job shop scheduling problem. While UPLA branching rule fails to solve 8 problems and the DVF one fails to solve 13 problems in given time limit.

## 4.2 Performance on Quasigroup Problems

The quasigroup problems were given by Fujita, Slaney, and Bennett in their award-winning IJCAI paper [6]. The best way to view a quasigroup problem is in terms of the completion of a Latin square. Given  $N$  colors, a Latin square is defined by an  $N$  by  $N$  table, where each entry has a color and where there are no repeated colors in any row or column.  $N$  is called the *order* of the quasigroup.

In table 2, we present the comparative results of *satz215*, *ssc34* and *ssc355* on the well known quasigroup problems. The column SAT, in the table, denotes

Problem	SAT	#Vars	#Cls	Search Tree Size			Run Time		
				satz215	ssc34	ssc355	satz215	ssc34	ssc355
qg1-07	Y	343	68083	<b>2</b>	4	<b>2</b>	5.31	5.25	<b>4.61</b>
qg1-08	Y	512	148957	2644	8	<b>1</b>	36.48	23.17	<b>21.71</b>
qg2-07	Y	343	68083	1	1	<b>2</b>	6.26	6.11	<b>5.41</b>
qg2-08	Y	512	148957	2788	6067	<b>2380</b>	<b>33.17</b>	65.34	53.27
qg3-08	Y	512	10469	18	<b>0</b>	<b>0</b>	<b>0.10</b>	0.21	0.18
qg3-09	N	729	16732	1034	736	<b>695</b>	<b>2.51</b>	4.24	6.64
qg4-08	N	512	9685	<b>30</b>	<b>30</b>	<b>30</b>	<b>0.09</b>	0.21	0.17
qg4-09	Y	729	15580	82	<b>0</b>	9	<b>0.25</b>	0.40	0.34
qg5-09	N	729	28540	<b>1</b>	<b>1</b>	<b>1</b>	0.40	0.58	<b>0.36</b>
qg5-10	N	1000	43636	2	<b>1</b>	2	<b>0.71</b>	0.89	0.81
qg5-11	Y	1331	64054	3	<b>0</b>	1	<b>1.39</b>	3.60	7.90
qg5-12	N	1728	90919	3	<b>1</b>	2	<b>2.55</b>	4.45	4.23
qg5-13	N	2197	125464	669	3150	<b>245</b>	<b>75.55</b>	2192.63	154.75
qg6-09	Y	729	21844	1	<b>0</b>	<b>0</b>	<b>0.25</b>	0.51	0.87
qg6-10	N	1000	33466	3	<b>1</b>	<b>1</b>	0.45	0.57	<b>0.44</b>
qg6-11	N	1331	49204	63	<b>19</b>	41	<b>1.86</b>	4.67	4.14
qg6-12	N	1728	69931	1024	925	<b>720</b>	<b>29.10</b>	276.70	70.41
qg7-09	Y	729	22060	<b>0</b>	<b>0</b>	<b>0</b>	<b>0.35</b>	0.69	0.87
qg7-10	N	1000	33736	<b>1</b>	<b>1</b>	2	<b>0.50</b>	0.66	0.57
qg7-11	N	1331	49534	7	<b>1</b>	4	<b>1.12</b>	1.41	1.51
qg7-12	N	1728	70327	154	<b>42</b>	88	<b>5.89</b>	23.42	10.26
qg7-13	Y	2197	97072	<b>24</b>	1405	492	<b>3.33</b>	1316.80	98.37

**Table 2.** Run time (in seconds) and search tree size of quasigroup problems. The best performances are in bold.

the status of solution which indicates satisfiable ("Y") or unsatisfiable ("N"). The *ssc355* DPLL procedure has a best performance for most of the problems, in terms of search tree size, while the *satz215* has a best performance from the run time point of view. These results explain that the NVO heuristic of *ssc355* performs well in solving the quasigroup problems with larger neighborhoods, while its inefficiency comes from the redundant unit propagation searches.

## 5 Conclusion

UPLA branching rule fails to choose the best branching variable because it limits the variable filtering process. Its ineffectiveness makes many real-world problems difficult for the DPLL procedure. In order to improve the power of UPLA, we have proposed the DVF and DVF+NVO branching rules which perform more variable filtering at each node.

The experimental results of DVF+NVO branching rules on a number of real-world benchmark instances and quasigroup problems prove our approaches to be useful in many circumstances. The DPLL procedure based on DVF+NVO branching rule performed well particularly on planning and security problems.

The work presented in this paper is a first attempt at building an efficient SAT solver. In our future work, we envisage at least three further improvements of our current approach. Firstly, it is clear that savings can be made by avoiding redundant unit propagation searches for variables that remain unchanged between iterations of UPLA. Secondly, further improvements of the NVO heuristic appear promising, as our first implementation is fairly simplistic. Finally, we are also looking at integrating a backjumping technique into DVF.

## Acknowledgments

We would like to thank Andrew Slater for helping us to run the experiments. We would also like to thank the anonymous reviewers for their valuable comments and suggestions on a previous version of this paper. National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

## References

1. Bessière, C., Chmeiss, A., and Sais, L. *Neighborhood-based Variable Ordering Heuristics for the Constraint Satisfaction Problem*. In Proceedings of Seventh International Conference on Principles and Practice of Constraint Programming, 2001, Paphos, Cyprus, pp. 565-569.
2. Crawford, J. M., and Auton, L. D. Experimental Results on the Crossover Point in Random 3SAT. *Artificial Intelligence Journal*, 1996, Vol. 81, no. 1-2.
3. Davis, M., Logemann, G. and Loveland, D. *A Machine Program for Theorem Proving*. Communication of ACM 5 (1962), pp. 394-397.
4. Dubois, O., and Dequen, G. *A Backbone-search Heuristic for Efficient Solving of Hard 3-SAT Formulae*. In Proceedings of 17th International Joint Conference on Artificial Intelligence, 2001, Seattle, Washington, USA.
5. Freeman, J. W. *Improvements to Propositional Satisfiability Search Algorithms*. Ph.D. Dissertation, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, (1995).
6. Fujita, M., Slaney, J., and Bennett, F. *Automatic Generation of Some Results in Finite Algebra*. In Proceedings of 13th International Joint Conference on Artificial Intelligence, 1993, Chambéry, France, pp.
7. Hooker, J. N., Vinay, V. *Branching Rules for Satisfiability*. Journal of Automated Reasoning, 15:359-383, 1995.
8. Jeroslow, R., Wang, J. *Solving Propositional Satisfiability Problems*. Annals of Mathematics and AI, 1, 1990, pp. 167-187.
9. Le Berre, D. *Exploiting the Real Power of Unit Propagation Lookahead*. In Proceedings of Workshop on the Theory and Applications of Satisfiability Testing, 2001, Boston University, MA, USA.
10. Li, C. M., and Anbulagan. *Heuristics Based on Unit Propagation for Satisfiability Problems*. In Proceedings of 15th International Joint Conference on Artificial Intelligence, 1997, Nagoya, Aichi, Japan, pp. 366-371.