Building Structure into Local Search for SAT

Duc Nghia Pham[1,2] and **John Thornton**[1,2] and **Abdul Sattar**[1,2]
[1] Safeguarding Australia Program, National ICT Australia Ltd.
[2] Institute for Integrated and Intelligent Systems, Griffith University, Australia
{duc-nghia.pham, john.thornton, abdul.sattar}@nicta.com.au

Abstract

Local search procedures for solving satisfiability problems have attracted considerable attention since the development of GSAT in 1992. However, recent work indicates that for many real-world problems, complete search methods have the advantage, because modern heuristics are able to effectively exploit problem structure. Indeed, to develop a local search technique that can effectively deal with variable dependencies has been an open challenge since 1997.

In this paper we show that local search techniques can effectively exploit information about problem structure producing significant improvements in performance on structured problem instances. Building on the earlier work of Ostrowski *et al.* we describe how information about variable dependencies can be built into a local search, so that only independent variables are considered for flipping. The cost effect of a flip is then dynamically calculated using a dependency lattice that models dependent variables using *gates* (specifically *and, or* and *equivalence* gates). The experimental study on hard structured benchmark problems demonstrates that our new approach significantly outperforms the previously reported best local search techniques.

1 Introduction

A fundamental challenge facing local search researchers in the satisfiability (SAT) domain is the increasing scope and performance of complete search methods. While for most of the 1990s it was taken for granted that local search was the most practical method for solving large and complex real world satisfiability problems [Selman *et al.*, 1992; Kautz and Selman, 1996; Béjar and Manyà, 2000], the latest generation of complete SAT solvers have turned the tables, solving many structured problems that are beyond the reach of local search (e.g. [Zhang *et al.*, 2001; Eén and Biere, 2005]).[1] So it is of basic importance to the area that local search techniques learn from the success of the newer complete methods.

The current research goes to the core of this problem by modelling problem structure *within* a local search procedure. This involves a two-part process: firstly problem structure must be recognized within the original problem representation (here we are considering satisfiability problems expressed in conjunctive normal form (CNF)). And secondly this structure must be represented within a local search procedure in such a way that the local neighbourhood of possible moves will only contain structure respecting flips.

The ideas behind this approach come from two sources. Firstly, there is the recent work on modelling constraint satisfaction problems (CSPs) as SAT problems [Pham *et al.*, 2005]. Here the presence of CSP multivalued variables is automatically detected in the clausal structure of a CNF problem. This information is then embedded within a local search in such a way that for each group of binary valued SAT variables corresponding to a single multivalued CSP variable, only one SAT variable will be true at any one time. This enforces that the underlying CSP variable is always instantiated with a single domain value. The SAT-based local search achieves this by doing a two-flip look-ahead whenever it encounters a literal associated with a CSP domain value. In effect this look-ahead turns off the current CSP domain value with one flip and turns on a new value with a second flip. A significant advantage of this approach (when encoding binary CSPs) is that the cost of such a double-flip equals the sum of the costs of the individual flips, because these flip pairs will never appear in the same conflict clause. For this reason, CSP structure exploiting algorithms can be easily embedded within an existing SAT local search architecture and add negligible processing overhead to an individual flip. As we shall see, more general structure exploiting approaches do cause interactions between literals within the same clauses, and so require more sophisticated flip cost calculation procedures.

The second source for the current research comes from Ostrowski *et al.*'s [2002] work on the extraction of *gates* from CNF encoded SAT problems. These gates represent relationships between SAT variables of the form $y = f(x_1, \ldots, x_n)$ where $f \in \{\Leftrightarrow, \wedge, \vee\}$ and y and x_i are Boolean variables from the original problem. If such a gate is recognized, the value of y is determined by $x_1, \ldots, x_n$, and can be removed. Ostrowski *et al.* used this method to simplify a range of struc-

[1] More information on the performance of complete search versus local search is available from the SAT competitions (http://www.satcompetition.org)

tured SAT benchmark problems, producing significant performance gains for their systematic DPLL solver. However, to the best of our knowledge, such structure exploiting approaches have not been applied in the local search domain.

The problem facing a local search approach to implementing gates is one of efficiency. Ostrowski *et al.*'s approach can detect independent variables whose values determine a set of dependent variables via clausal gate connections. Using this information, we can implement a local search that only flips the values of independent variables and then dynamically calculates the effects of these flips on the overall solution cost. However, a local search needs to know the flip cost of all candidate flips in advance of making a move. Generally this is achieved by maintaining a make cost and a break cost for each literal in the problem (i.e. the number of clauses that will become true if a literal is flipped and the number of clauses that will become false). These costs are then updated after each flip. A major advantage of a SAT local search is the speed with which these cost effects can be calculated (this is achieved using clever data structures that exploit the SAT CNF problem structure) [Tompkins and Hoos, 2004]. However, taking an approach that only flips independent variables renders the standard SAT local search architecture redundant. In this case, finding the potential cost of an independent variable flip requires us to solve a mini-SAT problem involving all the affected dependent variables and their associated clauses.

The rest of the paper is organized as follows: we first explain how the cost of flipping an independent variable in a local search can be efficiently calculated using a dependency lattice data structure. This structure models the various dependencies in the original problem and dynamically calculates which independent variables, when flipped, will cause a clause to change its truth value. Details of the construction and operation of this lattice are given in the next two sections. To evaluate the usefulness of this new approach, we conduct an empirical study that examines several of the structured SAT benchmarks that have proved to be the most difficult for local search in the past. Finally, we discuss the significance of our results and indicate some future directions of research.

2 Gates and Dependencies

In the following discussion we shall broadly follow the terminology used in [Ostrowski *et al.*, 2002]. Firstly, consider the following CNF formula:

$$(\neg a \vee b \vee c \vee d) \wedge (a \vee \neg b) \wedge (a \vee \neg c) \wedge (a \vee \neg d)$$

Here, for the clauses to be satisfied, if b and c and d are *all* false then a must necessarily be false, otherwise a must necessarily be true. This is an example of an "or" gate, because the value of a is determined by truth value of $(b \vee c \vee d)$ and can be represented as

$$a = \vee(b, c, d)$$

Similarly, if we reverse the signs of the literals we get:

$$(a \vee \neg b \vee \neg c \vee \neg d) \wedge (\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg a \vee d)$$

Now for the formula to be satisfied, if b and c and d are *all* true then a must necessarily be true, otherwise a must necessarily be false. This example of an "and" gate can be represented as

$$a = \wedge(b, c, d)$$

A third commonly occurring clausal structure is the "equivalence" gate (or "xnor" gate), illustrated as follows:

$$(a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c)$$

Here, in order to satisfy the formula, a will be true iff b and c are both true or both false, i.e. if they are equivalent, otherwise if b and c differ then a will be false. This can be represented as

$$a = \Leftrightarrow (b, c)$$

Finally, an "xor" gate is the negation of an "equivalence" gate, illustrated as follows:

$$(\neg a \vee \neg b \vee \neg c) \wedge (a \vee b \vee \neg c) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee b \vee c)$$

where a will only be false if b and c are equivalent. This can be represented as

$$a = \oplus(b, c)$$

If an "equivalence" or an "xor" gate depends on more than two variables, then equivalence or difference is calculated in a pairwise fashion, i.e. if $a = \Leftrightarrow (b, c, d)$ and b and c are false, then $b \Leftrightarrow c$ is true, and if d is true, then $d \Leftrightarrow (b \Leftrightarrow c)$ is true and therefore the gate is true. In general, we represent an "xor" gate as $y = \oplus(x_1, \ldots, x_n)$, an "equivalence" gate as $y = \Leftrightarrow (x_1, \ldots, x_n)$, an "or" gate as $y = \vee(x_1, \ldots, x_n)$ and an "and" gate as $y = \wedge(x_1, \ldots, x_n)$. Here y is the *dependent* variable because its value is determined by the *independent* variables $x_1, \ldots, x_n$. For the sake of simplicity, we treat an "xor" gate (e.g. $a = \oplus(b, c)$) as a special case of an "equivalence" gate (e.g. $\neg a = \Leftrightarrow (b, c)$) in the rest of the paper.

3 The Dependency Lattice

As Ostrowski *et al.* [2002] have already described, the process of recognizing gates in a CNF problem can be reduced to searching for the appropriate clausal structures during a preprocessing phase. This information can then be used to classify the dependent and independent variables. For a complete search method, this means the search space can immediately be reduced to only consider independent variable instantiations, as all dependent variable values can be automatically fixed by propagation.

However, for a local search, there is no built-in propagation mechanism. In fact, a local search strategy precludes propagation because it deliberately allows inconsistent assignments to persist. To exploit the information inherent in dependent variable relationships requires us to remove them from the domain of "free" variables while still including their effect in the overall cost of a particular flip. To achieve this we have developed a *dependency lattice* data structure. This lattice is formed as a result of analyzing the original CNF problem into *independent variables*, and relationships between *internal* and *external* gates. Firstly, an independent variable is not simply a variable that determines the value of a dependent variable in a gate relationship, because such determining variables can in turn be determined in another gate. An independent variable is a variable that is *never* determined in any gate relationship. Secondly, an internal gate is any gate that can be recognized within the structure of the original CNF formula, and thirdly, an external gate is a gate where the dependent variable represents a clause from the original CNF formula that is not part of any internal gate. To clarify these concepts, consider the following CNF formula example:

$$(\neg g_1 \vee v_2 \vee v_3) \wedge (g_1 \vee \neg v_2) \wedge (g_1 \vee \neg v_3) \wedge$$
$$(g_2 \vee \neg v_3 \vee \neg v_4) \wedge (\neg g_2 \vee v_3) \wedge (\neg g_2 \vee v_4) \wedge$$
$$(g_3 \vee g_1 \vee g_2) \wedge (\neg g_3 \vee \neg g_1 \vee g_2) \wedge$$
$$(\neg g_3 \vee g_1 \vee \neg g_2) \wedge (g_3 \vee \neg g_1 \vee \neg g_2) \wedge$$
$$(v_1 \vee g_1)$$

which is equivalent to:

$$(g_1 = \wedge(v_2, v_3)) \wedge (g_2 = \vee(v_3, v_4)) \wedge$$
$$(g_3 = \Leftrightarrow (v_1, v_2)) \wedge (c_1 = \vee(v_1, g_1))$$

where c_1 is an additional variable that depends on the clause $(v_1 \vee g_1)$ (i.e. if $(v_1 \vee g_1)$ is true, c_1 is true, otherwise c_1 is false). In general, each original CNF clause that is not subsumed within a gate dependency is represented by an additional variable, that then subsumes the clause in an external "or" gate dependency.

Having translated our original problem into a set of gates, we can now represent it as the dependency lattice in Figure 1:

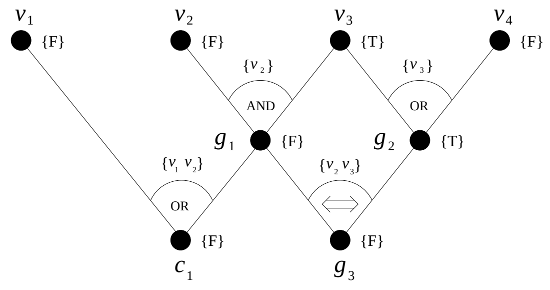

Figure 1: An example dependency lattice.

Here the original variables have become nodes in the lattice, such that nodes $v_1 \ldots v_4$ correspond to the independent variables, nodes $g_1 \ldots g_3$ to the internal gates and node c_1 to an external gate. In this form, a satisfying assignment to the original CNF formula is equivalent to an assignment of the independent variables such that all the c_i external nodes evaluate to *true*. This result follows trivially from the structure of the lattice, which implements the structure of the internal gates and therefore ensures that all internal gate relationships are necessarily satisfied. When all the external nodes evaluate to true, this means all the remaining CNF clauses that were not subsumed as internal gates are true, and hence that a satisfying assignment has been found.

The purpose of the lattice is to embody the structure of gate dependencies in such a way that the cost of flipping an independent variable can be efficiently calculated. This is analogous to existing local search SAT solvers, except that existing solvers are only equipped to handle "or" gate dependencies.

3.1 Calculating Flip Costs

To illustrate the process of cost calculation, we have instantiated the independent variables in Figure 1 as follows: $v_1 \leftarrow$ *false*, $v_2 \leftarrow$ *false*, $v_3 \leftarrow$ *true* and $v_4 \leftarrow$ *false*. Moving down the lattice from v_2 and v_3 to the "and" gate at node g_1 it follows that the values of v_2 and v_3 make this gate variable *false*. Similarly, moving down from v_3 and v_4 to g_2 we can see that the corresponding "or" gate variable is *true*. Then following down from the gates at g_1 and g_2 to the "equivalence" gate at g_3 we can see that this gate variable is *false*, and so on. In this way, the lattice reflects the necessary consequences of the independent variable instantiations on the rest of the problem.

To calculate the cost of flipping a particular independent variable v_i we need to know how many external gate variables will become false and how many will become true as a result of the flip. This is achieved by storing at each gate node the set of independent variables that, if flipped, would cause the gate variable to change its truth value. For example, node g_1 stores the independent variable v_2, signifying that if v_2 was flipped then g_1 would change from $false$ to $true$. Similarly, g_2 can only become $false$ if v_3 is flipped. Moving down the lattice, we can see that g_3 would become $true$ if either g_1 or g_2 were to change values. As flipping v_2 will change the truth value of g_1 and flipping v_3 will change the truth value of g_2, either of these flips will also change the truth value of g_3, so g_3 inherits v_2 and v_3 into its variable set. Node c_1 similarly inherits v_1 and v_2, as a change in either variable will make c_1 $true$.

Once we have the correct variable sets for each of the external gates we can read off the make cost and break cost for each independent variable simply by counting the number of times a variable appears in a false external gate (= make cost) and the number of times it appears in a true external gate (= break cost). So, in our example, both v_1 and v_2 have a make cost of one, with all other make and break costs equal to zero.

3.2 The General Case

In realistic problems it can easily happen that the same independent variable appears in multiple branches leading to the same gate. To handle such cases we require more general definitions of how the variable sets for each gate type are composed.

Firstly, if an "and" gate is $true$ then all its parent nodes must be $true$, therefore any change in a parent node's truth value will also change the truth value of the gate. This means the gate's variable set (V) should inherit the union of all the parent variable sets (P_{sets}), as follows:

$$\textbf{if } \textit{true}(\text{AND}) \textbf{ then } V \leftarrow \cup(\textit{true}(P_{sets}))$$

Alternatively, if an "and" gate is $false$ then only if all its parent nodes become $true$ will the gate become $true$. This requires that all false parent nodes become $true$ and no true parent node becomes $false$, as follows:

$$\textbf{if } \textit{false}(\text{AND}) \textbf{ then } V \leftarrow \cap(\textit{false}(P_{sets})) \setminus \cup(\textit{true}(P_{sets}))$$

The rules for an "or" gate can be similarly defined in reverse:

$$\textbf{if } \textit{false}(\text{OR}) \textbf{ then } V \leftarrow \cup(\textit{false}(P_{sets}))$$
$$\textbf{if } \textit{true}(\text{OR}) \textbf{ then } V \leftarrow \cap(\textit{true}(P_{sets})) \setminus \cup(\textit{false}(P_{sets}))$$

For all "equivalence" gates with no more than two parents, only if the truth value of a single parent changes will the value of the gate change, as follows:

$$V \leftarrow \cup(P_{sets}) \setminus \cap(P_{sets})$$

In general, an "equivalence" gate is $true$ iff the count of its $true$ parents has the same parity as the count of all its parents. As we did not discover any "equivalence" gates with more than two parents in our problem set, we did not implement the more-than-two-parent case.

In addition, we did find rare problem instances where a single variable was dependent on more than one gate. In these circumstances, we added an additional dependent variable for each extra gate and connected these variables to the first variable via additional "equivalence" gates.

3.3 Implementation

The motivation behind the dependency lattice is to develop an efficient method to update the make and break costs for the independent variables. Clearly there is a potential for the additional work of calculating flip costs using the lattice to outweigh the benefits of reducing the size of the search space (i.e. by eliminating dependent variables and internal gate clauses from the problem). Therefore it is of significance exactly how the lattice is updated. In our current implementation we do this by representing the total set of independent variables at each node using an n_{ind} bit pattern, such that if the i^{th} independent variable is in the variable set of a particular node, then the i^{th} position of the bit pattern for that node will be set. Using this representation we can efficiently implement the set operations necessary to propagate the effects of flipping an independent variable through the lattice. This propagation starts when an independent variable has been selected for flipping. Beginning at the lattice node corresponding to this variable, the update works down the lattice to all the connected internal nodes. For example, if v_3 is flipped in Figure 1, then nodes g_1 and g_2 will be selected for update. The update process then performs the appropriate set operations on the parent variable sets to produce an updated variable set and truth value for the selected gate. If the truth value of a node and the contents of its variable set remain unchanged after an update then no further propagation is required from that node. Otherwise the process continues until it reaches the external nodes. Then, if an external node's variable set is changed, this updates the make and break costs of any affected independent variables and the process terminated.

If we follow the process of flipping v_2 in Figure 1, this will alter the variable set at g_1 from $\{v_2\}$ to $\{v_2, v_3\}$ and change g_1 to *true.* As the variable set at g_1 has changed these effects are now propagated to the internal node g_3 which becomes *true*. Now we have a situation where both parents of g_3 share the same variable, i.e. g_1 now has $\{v_2, v_3\}$ and g_2 has $\{v_3\}$. In this case if v_3 were flipped then both parents of g_3 would change their truth value and still remain equivalent, leaving g_3 unchanged. Hence g_3 only inherits v_2 into its variable set. Finally, the external node c_1 changes its truth value to *true* and changes its variable set from $\{v_1, v_2\}$ to $\{v_2, v_3\}$. This change then causes the make costs of v_1 and v_2 to be reduced by one and the break costs of v_2 and v_3 to be increased by one. The process terminates with all external nodes set to *true,* meaning that a satisfying solution has been found. This situation is illustrated in Figure 2:

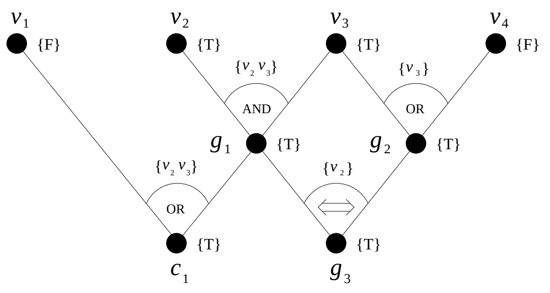

Figure 2: A dependency lattice solution.

4 Experimental Validation

In order to validate our approach to handling gate dependencies, we implemented a *non-CNF* version of AdaptNovelty$^+$ [Hoos, 2002] that operates on the new dependency lattice platform. We then evaluated the performance of this algorithm on a selection of SATLIB ssa7552, parity 16- and 32-bit problems.[2] Table 1 firstly shows the effect of gate detection on the number of variables and clauses in the problem set, detailing the number of independent and dependent gates in the corresponding non-CNF dependency lattices and the total time taken for each conversion.

We then compared the performance of our new non-CNF AdaptNovelty$^+$ algorithm against the original CNF based variant by running the two algorithms 100 times each on the ssa7552 and par16 instances and 10 times each on the par32 instances. Table 2 shows the success rate, the average number of flips and time in seconds taken to solve these instances. Each run was timed out after 1 hour for the ssa7552 and par16 instances and after 24 hours for the par32 instances.

As shown in Table 2, the non-CNF version of AdaptNovelty$^+$ significantly outperforms its original CNF counterpart both in terms of flips and time. Indeed, the new non-CNF approach is at least 100 times better than the original CNF approach on these instances. In addition, this is the first time that a local search solver has managed to find solutions for all the parity 32-bit problems within 24 hours. The only other SLS method that can solve these problems is DLM2005 [Wah and Wu, 2005]. However, DLM could only solve the compacted par32-*-c instances, producing only 1 successful run out of 20 attempts and taking nearly 33 hours to find a solution.

As all else has been left unchanged between the two versions of AdaptNovelty$^+$, we must conclude that the extraordinary performance gains are due to the successful recognition and exploitation of variable dependencies in the new non-CNF approach. As shown in Figure 3, the dependency lattice

[2]Available from http://www.satlib.org

Problem	CNF		Extracted			non-CNF		
	#vars	#clauses	#fixed	#eq	#and/or	#input	#output	#seconds
ssa-038	1,501	3,575	40	1,031	23	407	1,137	0.016
ssa-158	1,363	3,034	186	894	7	276	642	0.011
ssa-159	1,363	3,032	132	932	11	288	683	0.015
ssa-160	1,391	3,126	25	1,016	19	331	855	0.015
par16-1	1,015	3,310	408	560	31	16	91	0.011
par16-2	1,015	3,374	383	585	31	16	91	0.013
par16-3	1,015	3,344	395	573	31	16	91	0.011
par16-4	1,015	3,324	396	572	31	16	91	0.012
par16-5	1,015	3,358	388	580	31	16	91	0.016
par32-1	3,176	10,277	758	2,261	125	32	247	0.031
par32-2	3,176	10,253	784	2,235	125	32	247	0.034
par32-3	3,176	10,297	781	2,238	125	32	247	0.032
par32-4	3,176	10,313	791	2,228	125	32	247	0.032
par32-5	3,176	10,325	791	2,228	125	32	247	0.034

Table 1: The effects of the gate extracting algorithm on the ssa7552-$*$ and parity problems. This table shows the number of "fixed", "equivalence" ($\#eq$) and "and/or" gates extracted from each instance. The number of "independent" and "external dependent" gates of each processed non-CNF instance are described in the table as $\#input$ and $\#output$, respectively.

Problem	CNF based			non-CNF based		
	% solved	#flips	#seconds	% solved	#flips	#seconds
ssa-038	86%	180,937,822	838.790	100%	2,169	15.131
ssa-158	100%	303,377,289	419.054	100%	439	1.203
ssa-159	95%	118,865,143	499.300	100%	460	1.396
ssa-160	100%	154,646,089	377.383	100%	1,284	5.562
par16-1	100%	148,475,195	684.972	100%	2,455	0.489
par16-2	98%	331,148,102	788.312	100%	2,724	0.570
par16-3	100%	381,887,299	801.501	100%	1,640	0.320
par16-4	100%	79,196,974	779.877	100%	3,217	0.626
par16-5	100%	390,162,552	604.379	100%	7,938	1.630
par32-1	0%	n/a	$> 24h$	80%	n/a	48,194.033
par32-2	0%	n/a	$> 24h$	100%	n/a	13,204.536
par32-3	0%	n/a	$> 24h$	100%	n/a	17,766.822
par32-4	0%	n/a	$> 24h$	100%	n/a	9,487.728
par32-5	0%	n/a	$> 24h$	100%	n/a	23,212.755

Table 2: The results of solving the ssa7552-* and parity problems using the CNF and non-CNF based versions of AdaptNovelty$^+$. The $\#flips$ and $\#seconds$ are the average number of flips and seconds taken to solve each instance. For the original AdaptNovelty$^+$ the $\#flips$ values have been approximated as the flip counter maximum was exceeded.

of the par8-1 instance is highly connected. Access to this extra knowledge enables the new solver to maintain the consistency of the dependent variables and hence to efficiently navigate the search space and find a solution. The structure of the variable dependencies is otherwise flattened out and hidden in the original CNF representation. This means that CNF based SLS solvers must expend considerable extra effort to discover solutions that respect these dependencies.

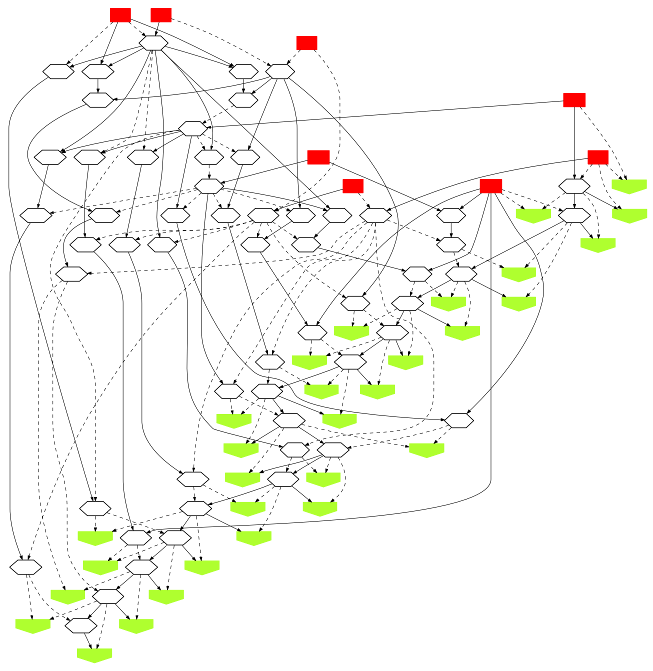
Figure 3: The dependency lattice of the par8-1 instance. In this graph, the *independent* gates are depicted as shaded *rectangular* boxes and the *dependent* "equivalence", "and" and "or" gates are represented as *hexagon*, *house* and *inverse house* shaped boxes, respectively. External dependent gates are also lightly shaded. A solid arrow outputs the gate value, while a dashed arrow outputs the negation of the gate value.

Problem	CNF based			non-CNF based		
	% solved	#flips	#seconds	% solved	#flips	#seconds
bart26	100%	216	0.001	100%	215	3.585
bart27	100%	233	0.001	100%	228	4.537
bart28	100%	222	0.001	100%	222	4.191
bart29	100%	244	0.001	100%	249	5.602
bart30	100%	266	0.001	100%	251	6.677

Table 3: The results of solving the Aloul's bart FPGA problems using the CNF and non-CNF based versions of AdaptNovelty$^+$. The $\#flips$ and $\#seconds$ are the average number of flips and seconds taken to solve each instance.

However, the cost of maintaining consistency between the newly discovered gates in our non-CNF approach is also significant. To measure this, we conducted an additional experiment using a set of bart FPGA problems that exhibit no dependency structure [Aloul *et al.*, 2002]. Table 3 shows our non-CNF AdaptNovelty$^+$ solver to be more than $1,000$ times slower on these problems. However, this performance deficit can be partly explained by the initial cost of searching for gate dependencies in the original CNF representation, and hence will become less significant for problems where the solution time significantly exceeds the preprocessing time. We also used the built-in C++ set class to update the lattice which could be replaced by more efficient, special purpose data structures and operators. Finally, it would be trivial to implement a switch that automatically reverts to using a CNF based solver when the proportion of gate dependencies falls below a given threshold.

5 Conclusion

In conclusion, we have introduced a new dependency lattice platform that effectively maintains the consistency between independent and dependent variables (or gates) during the execution of a local search. Based on this platform, our new non-CNF version of AdaptNovelty$^+$ can solve many hard structured benchmark problems significantly faster than its original CNF based counterpart. In addition, this non-CNF AdaptNovelty$^+$ variant is the first local search solver able to reliably solve all five par32 instances within 24 hours. By exploiting variable dependencies within a local search and by solving the par32 problems we have also successfully addressed two of the ten challenges in propositional reasoning and search (#2 and #6) presented in [Selman *et al.*, 1997].

In future work, we expect that non-CNF implementations of the latest clause weighting local search solvers (such as PAWS [Thornton *et al.*, 2004] and SAPS [Hutter *et al.*, 2002]) will further extend the state-of-the-art in local search techniques. In fact, the extension of these solvers using our dependency lattice is very straightforward. Instead of counting the number of external dependent gates that will be made or broken if an independent gate is flipped, we simply sum the corresponding weights of the dependent gates.

Another future research direction is to develop new heuristics that further exploit the gate dependencies when selecting the next variable to flip. With these improvements, we expect that local search techniques will be able to match the performance of the state-of-the-art DPLL solvers on the more structured industrial benchmark problems.

Acknowledgments The authors would like to acknowledge the financial support of National ICT Australia (NICTA) and the Queensland government. NICTA is funded through the Australian Government's *Backing Australia's Ability* initiative and also through the Australian Research Council.

References

[Aloul *et al.*, 2002] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *Proceedings of the 39th Design Automation Conference (DAC-02)*, pages 731–736, 2002.

[Béjar and Manyà, 2000] Ramoón Béjar and Felip Manyà. Solving the round robin problem using propositional logic. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pages 262–266, 2000.

[Eén and Biere, 2005] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, pages 61–75, 2005.

[Hoos, 2002] Holger H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02)*, pages 635–660, 2002.

[Hutter *et al.*, 2002] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP-02)*, pages 233–248, 2002.

[Kautz and Selman, 1996] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1194–1201, 1996.

[Ostrowski *et al.*, 2002] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Saïs. Recovering and exploiting strutural knowledge from CNF formulas. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP-02)*, pages 185–199, 2002.

[Pham *et al.*, 2005] Duc Nghia Pham, John Thornton, Abdul Sattar, and Adelraouf Ishtaiwi. SAT-based versus CSP-based constraint weighting for satisfiability. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 455–460, 2005.

[Selman *et al.*, 1992] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, 1992.

[Selman *et al.*, 1997] Bart Selman, Henry Kautz, and David McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 50–54, 1997.

[Thornton *et al.*, 2004] John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for SAT. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-04)*, pages 191–196, 2004.

[Tompkins and Hoos, 2004] Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *SAT (Selected Papers)*, pages 306–320, 2004.

[Wah and Wu, 2005] Benjamin W. Wah and Zhe Wu. Penalty formulations and trap-avoidance strategies for solving hard satisfiability problems. *J. Comput. Sci. Technol.*, 20(1):3–17, 2005.

[Zhang *et al.*, 2001] Lintao Zhang, Conor Madigan, Matthew Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design*, pages 279–285, 2001.