On Hybrid SAT Solving Using Tree Decompositions and BDDs

Sathiamoorthy Subbarayan Lucas Bordeaux
Youssef Hamadi

March 2006

Technical Report
MSR-TR-2006-28

The goal of this paper is to study the benefits of hybridizing the CNF SAT Solvers with BDDs. Towards this we define a metric for the level of hybridization based on a tree decomposition of an input CNF. We also present a new linear time algorithm on BDDs, which is useful for efficient conflict analysis in any BDD-based hybrid SAT solver. Experiments on an implementation of our hybrid solver shows when to use such a hybridization.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

1 Introduction

Our work is motivated by the fact that BDDs [1] and resolution-based CNF solvers are different reasoning methods [2]. Some constraints are easy for BDD representations while they are hard for CNF solvers and vice versa. The complexity of solving a SAT instance can be expressed as exponential in the *tree width* [3] of the instance. Hence, when the tree width of a SAT instance is quite low, we can use this fact to guide a SAT solver. In this paper, we present a hybrid SAT solver combining BDDs and CNF reasoning methods with tree decomposition techniques. Although attempts have been made in the past to combine some of the these techniques [4, 5], we are not aware of any previous work mixing all the three techniques. Our work takes care of the problems in the previous approaches, such as ad hoc methods to build BDDs [4], suffering from high cost for obtaining tree decompositions [5].

The goal of this paper is to study the benefits of our hybridization. We first create a tree decomposition of an input CNF instance of n variables. Then, based on a parameter $h \in [0,..,2^n]$, we make a *heuristic cut* in the tree decomposition. The cut will be such that for each subtree below the cut the clauses K *covered* by the subtree can be represented by a BDD of size at most *h*nLits*, where *nLits* is the number of literal occurrences in the clauses K. For each subtree below the cut its BDD will be created and the corresponding clauses in the input CNF will be replaced by the BDD. This results in a hybrid representation. We then have a hybrid SAT solver of the form used in [4], which has to do propagation and conflict analysis in this hybrid representation to determine satisfiability. The contributions of this paper include a new linear-time algorithm for finding minimal necessary assignments for implications by BDDs, which will be used by the hybrid conflict learning process.

At h=0 (h=2^n) the hybrid solver will be a pure CNF (BDD) solver. The intermediate values of h result in a really hybrid solver.

An implementation of the above mentioned method is used in an experimental study of the tradeoffs between BDD and CNF representations. In some instances, we observe very low tree width which should naturally help in hybrid SAT solving.

The rest of the paper is organized as follows. The necessary background on tree-decomposition methods and BDDs are given in Section 2. The process of creating the hybrid representation, the hybrid SAT solver and the new BDD algorithm useful for conflict analysis are presented in Section 3. The subsequent section presents the experimental results, which is followed by related work and some concluding remarks.

2 Background

2.1 Tree Decomposition

Let (V,C) be a CNF SAT problem, where V is the set of propositional variables and C is the set of clauses in the CNF. Let the notation *vars*(c) denote the set of variables occurring in the literals of a clause $c \in C$. Then, the *tree decomposition* [3] of the SAT problem is a tree T=(N,E), where N is the set of nodes in the tree and E is the set of edges in the tree. Each node $n \in N$ will be *associated* to a set *vars*(n), where *vars*(n) $\subseteq V$. A node n is said to *contain* the variables *vars*(n). Additionally, a tree decomposition has to satisfy the following two conditions:

1. for any clause $c \in C$, there exists a node $n \in N$ such that *vars*(c) $\subseteq$ *vars*(n). i.e., each clause needs to be *covered* by at least one node in the tree.

2. for any variable $v \in V$, the nodes in N containing the variable v form a *connected subtree* in the tree T.

The *size* of a node in the tree decomposition is the number of variables in it. The *tree width* of a tree decomposition ω' is the size of the largest sized node of the tree decomposition minus one. The *tree width* of a SAT instance ω is then the smallest number among the tree widths of all possible tree decompositions of the instance. It is well known that finding the tree width of a SAT instance is NP-hard [3]. Hence, heuristics are employed to approximate the tree decomposition.

Given a tree decomposition of a SAT problem, the SAT instance can be solved by a dynamic programming method [6] with complexity exponential in the tree width of the given tree decomposition. This is the fact we can exploit to improve the current SAT solvers. Since finding the best tree decomposition is NP-hard, we can use one among the several available heuristics to obtain a good tree decomposition and build a SAT solver to use the decomposition. For a set of problem instances of same tree width and increasing instance sizes, the performance of the tree width based SAT solvers depend only upon the ability of the heuristic to find good decompositions. Exploiting tree width could be useful in cases like bounded model checking [7], where with the increase in the size of the bound, the tree width of the corresponding instances should not differ a lot, as a similar circuit structure is repeated for each increase in bound.

2.1.1 The min-degree heuristic

We use minimum-degree (min-degree) [8] heuristic for obtaining tree decompositions. The min-degree heuristic uses a graph in which all the variables will have a node. There will be an edge between two nodes if the corresponding two variables occur together in an input CNF clause. Given such a graph, the following steps will be repeated until all nodes are eliminated from the graph:

1. select a node n with minimum degree.

2. form a clique between the node n and all of its neighbors, by adding an edge between two neighbors of n, if required.

3. remove (eliminate) the node n and all the edges connected to it from the graph.

A tree decomposition can be obtained when all the nodes are being eliminated by the above method. Each clique formed during the elimination process will have a corresponding node in the tree decomposition. A tree node will contain all the variables in its corresponding clique. The edges between the nodes in the tree decomposition are obtained as follows. In the step-3 of the above elimination process, when the node n with k neighbors is removed, the k neighbors will still form a clique of size k in the remaining graph. All such k-cliques can be *marked* and each one of them can be associated with the tree node formed during the immediate previous step. When a new clique is formed in step-2, if it contains any marked k-clique, then the tree node corresponding to the new clique will have an edge with each one of the tree nodes associated with the contained marked cliques.

Note that, in a tree decomposition, when the variables contained in a tree node n_1 form a subset of the variables contained in its adjacent node n_2, then the node n_1 can be removed after adding an edge between n_2 and each one of the neighbors of n_1.

Using the largest sized node in a tree decomposition obtained by min-degree as the root node, we can arrange the nodes in the tree and mark the leaf nodes accordingly. During a traversal along any path from a leaf node to the root node, the structure of the min-degree heuristic decompositions are such that the number of variables contained in a node in the path will keep increasing. Due to elimination of the minimum degree nodes, the size of the cliques created during the elimination process tend to increase with each node elimination. The leaf nodes would have been created earlier in the elimination process and the root node at the end.

2.2 Binary Decision Diagrams

A binary decision diagram (BDD) [1] is a rooted directed acyclic graph with two terminal nodes marked with 0 and 1. The non-terminal nodes are associated with a propositional variable and each one of them will have two outgoing edges, one solid and one dashed. The solid (dashed) edges are used when the corresponding variables take true (false) value. Figure 1 shows a sample BDD. The solid (dashed) edge is also mentioned as high (low) edge.

Given a BDD representing a propositional function, each path from the root node to the 1-terminal node represents one or more solutions to the function. A non-occurrence of a variable's node in such a path would mean that the variable is a don't care variable and hence it can take both true and false values. Similarly, paths from the root to the 0-terminal node represent the non-solutions of the function. We are only interested in a special type of BDD called reduced ordered binary decision diagram(ROBDD), in which the nodes in the BDD

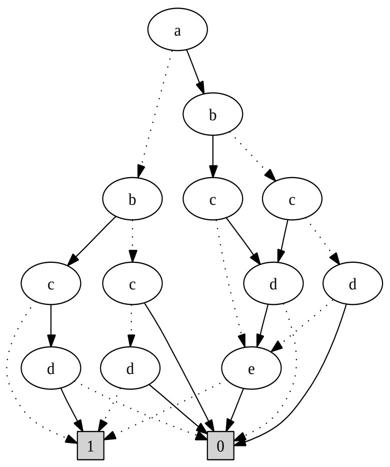

Figure 1: A Binary Decision Diagram

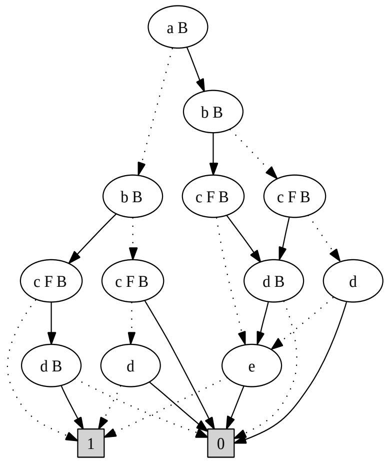

Figure 2: An example for the Minimal Reason-Vars algorithm

obey a selected variable order in all paths in the BDD, and isomorphic nodes are merged and represented by a single node. Hereafter, we simply mention ROBDD as BDD.

The size of a BDD b, denoted by $|b|$, is the number of nodes in the BDD. Given two BDDs, b_1 and b_2, representing two Boolean functions, the time and space complexity of the *conjoin* (*disjoin*) operation between them is O($|b_1|*|b_2|$) [9]. The time and space complexity of existential quantification of a variable from a BDD of size $|b|$ is O($|b|*|b|$) [9]. Given a CNF clause, its equivalent BDD can be obtained by disjoin operations between the BDDs representing the literals in the clause.

Let the assignment of a propositional value *val* to a propositional variable *var* be denoted by the pair (*var*,*val*). A path from a BDD node to one of the terminal nodes is said to *contain* an assignment pair (*var*,*val*), if the path contains an edge corresponding to the assignment pair or there is no *var* node in the path. A path is said to *violate* an assignment pair (*var*,*val*) if the path does not contain (*var*,*val*).

The nice property of BDDs is their ability to succinctly represent several practical propositional functions, due to which they have been widely used in the verification area. Hence, we use BDDs to represent the partial solution spaces of tree nodes in our decomposition based SAT solver.

The size of the BDD representing a function can vary exponentially based on the variable order used [9]. Since, a typical SAT instance will have a lot of variables, in our work, we do not spend time on finding good variable order. We just use the order in which the variables are numbered in a DIMACS-format CNF instance.

3 The Hybrid SAT Solver

3.1 The Hybrid Representation

Given a tree decomposition of a SAT problem, the satisfiability of the problem can be determined as follows:

1. For each leaf node in the tree, create a BDD of the node by conjoining the BDDs representing the clauses covered by the leaf node.
2. A tree node is *qualified* for this step if the BDDs for each one of its child nodes have been created. If there is any qualifying node n, create a BDD by conjoining all the BDDs of the child nodes of n along with the BDDs representing each one of the clauses covered by n. Then, existentially quantify out any variable v from the so obtained BDD, if the variable v only occurs in the subtree rooted at n. The resulting BDD is associated with n.
3. Repeat the previous step if there is any qualifying tree node for the step.

At the end of the above procedure, if the BDD for the root node represents *false* then the input problem is unsatisfiable, otherwise satisfiable.

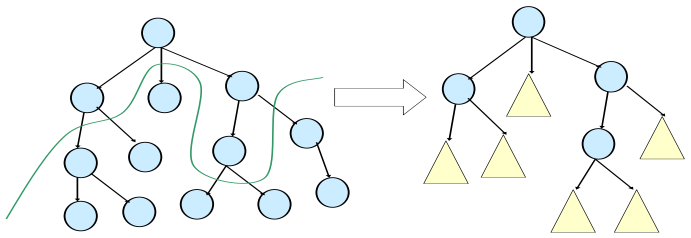
Figure 3: The Tree Decomposition to the Hybrid Representation conversion

Although the BDDs can succinctly represent the solution spaces of small SAT problems, for large problems the above procedure will typically fail due to large memory requirements to store BDDs. Hence, we modify the above procedure such that BDDs of restricted size only are created. Towards this we parameterize the BDD creation process by h, where h is an integer in the range $[0,..,2^n]$, n is the number of variables in the input SAT problem.

For a given h, the BDD for a tree node is created only if the size of the BDD is at most h*$nLits$, where $nLits$ is the number of literals in the clauses covered by the subtree rooted at the tree node. Whenever a BDD is created in step-1 or step-2 of the above procedure, the size restriction is checked. If it is not violated, then the BDD will be accepted and the clauses covered by the tree node will be removed from the CNF. Otherwise, the BDD will be discarded and the corresponding node will be marked as unqualified for step-2.

After such parametrization, the above procedure results in a hybrid representation of the SAT problem using some BDDs and some CNF clauses. Note that when we accept a BDD created for a tree node, then the BDDs for all of its child nodes can be discarded. Due to the properties of tree decomposition, this is enough to determine satisfiability. When a satisfying solution is required in case of a satisfiable input, we cannot discard the BDDs of the child nodes. In this work we just focus on determining satisfiability, since extending our work to find a satisfying solution is trivial.

When $h=0$, only BDDs of size zero will be accepted, i.e. either true or false. This would be equivalent to a pure CNF representation. When $h=2^n$, BDDs of all sizes will be accepted, hence, given enough time and memory, the BDD of the root node will be created. This is equivalent to a pure BDD SAT solver with interleaved conjoins and quantifications. The intermediate values of h would result in a real hybrid representation.

The choice for h makes a cut in the tree decomposition as shown in the Figure 3. A triangle in the figure represents a BDD that replaces a subtree below the cut. All the clauses covered by each subtree below the cut will be replaced by a BDD in the hybrid representation.

3.2 The Solver

To build a zChaff [10] style SAT solver to work on this hybrid representation we need to port the techniques like watching-based lazy propagation [11], conflict-directed backtracking [12] and conflict clause learning [12, 10] from the pure CNF representation to the hybrid representation.

We use the VSIDS [11] heuristic for making branching decisions in the hybrid SAT solver. The two-literal watching based lazy propagation [12] is used in the CNF part. The propagation in a BDD is done by a very naive procedure which traverses the nodes in the BDD. During the traversal the procedure visits only those nodes that can be reached by the assignments that have been made and marks the possible assignments for any unassigned variable in the tree node corresponding to the BDD. In case an unassigned variable has only one of its possible Boolean choices marked during the process, it results in an implication by the BDD. In case both the choices of an unassigned variable are unmarked during the process then it results in a conflict. The complexity of this trivial function is linear in the number of nodes in the BDD.

When conflicts occur in the search process, we use the 1-UIP [10] type conflict clause learning. Since the 1-UIP conflict learning procedure results in an *asserting clause* [12, 10], the SAT solver backtracks to the asserting decision level after learning a conflict clause.

In the 1-UIP learning scheme, when a variable implication at the latest decision level is involved in a conflict, all the assignments prior to the implication which are *enough* to make the implication need to be found. We call the set of variables in the assignments which are enough for an implication *reason-vars* set for the implication. Finding reason-vars is trivial for an implication by a CNF clause, as all the variables in the clause, except the implied one, are reason-vars for the implication.

In case of an implication made by a BDD, this process is non-trivial. All the variables in the BDD that were assigned a value before the implication would themselves constitute a reason-vars set. But such a set of reason-vars is not useful as only a small subset among them might still be a reason-vars set and using a smaller sized reason-vars set would result in smaller conflict learnt clauses and hence better propagation. Hence, we would like to have a reason-vars set whose size is *minimum*.

3.3 A Linear Time Algorithm for Minimal Reason-Vars

We are not aware of any BDD algorithm that can find a minimum or even minimal reason-vars set in linear-time. We give an algorithm to this problem, that can obtain a *minimal* reason-vars set. The complexity of the algorithm is linear in the number of nodes in the BDD.

Given a set of assignments $A=\{(var_{a_1},val_{a_1}),..,(var_{a_k},val_{a_k})\}$, a BDD b and an implication (var_i,val_i), where the assignments A result in the BDD to imply (var_i,val_i). Our goal is to find a minimal subset of variables in A, whose assignments are enough for the BDD to imply (var_i,val_i).

Let the m variables contained in the tree node corresponding to the BDD b be $\{var_{c_1},..,var_{c_m}\}$. Let the variable order used in the BDD be $var_{c_l} < var_{c_{l+1}}$.

In a BDD each path from the root node to the 1-terminal represents one or more solutions. That the pair (var_i,val_i) was implied by the BDD b means that there is no path in the BDD that contains $(var_i,\neg val_i)$, such that the path does not violate any of the assignments of A.

Given any path from the root node of BDD to the 1-terminal that contains $(var_i,\neg val_i)$, there will be at least one edge in the path that violates an assignment in A. Among them we can pick the edge that is nearest to the 1-terminal and label it as a *frontier edge*. If a node has one of its outgoing edges labeled as a frontier edge, then the node could be labeled as a *frontier node*. Considering all the paths in the BDD which contain $(var_i,\neg val_i)$, we can label all the frontier nodes in the BDD. This can be done in linear time, as a BDD is a rooted directed acyclic graph.

Given a node n from the BDD, the propositional term *frontier(n)* evaluates to *true* if and only if n is labeled as a frontier node. In our algorithm for finding minimal reason-vars, we will mark some nodes with the *blue* color. The term *blue(n)* evaluates to true if and only if the node is marked with blue color. Let *var(n)* denote the variable associated with the BDD node n.

The pseudo code of our algorithm for finding minimal reason-vars is shown in Algorithm 1. Additional low-level details of the algorithm and arguments on the correctness and complexity of the algorithm are in the appendix of this paper.

An example for the minimal reason-vars algorithm is shown in Figure 2. In the example, the set of variables in the tree node of the BDD is $\{a,b,c,d,e\}$. The BDD in the example is the same as the one in Figure 1, but with frontier labelings and blue marks. An entry "F" ("B") in a node of the BDD refers to a frontier label (blue mark). In the example, the set $A=\{(a,1),(c,1),(e,0)\}$ and

Algorithm 1 An algorithm to find Minimal Reason-Vars

```
REASONVARS($A$, $b$, ($var_i$,$val_i$))
Label all frontier nodes
Mark the root node as blue
for $j = 1$ to $m$ do
    BMN:=$\{n|\exists n \in b$.blue($n$)$\wedge$(var($n$)=$var_{c_j}$)$\}$
    if $\nexists n\in$BMN.frontier($n$) then
        for all $n \in$ BMN do
            if $i = c_j$ then
                Mark the $\neg val_i$ child node of $n$ with blue
            else
                Mark both the child nodes of $n$ with blue
            end if
        end for
    else
        for all $n \in$ BMN do
            Mark the $val_j$ child node of $n$ with blue
        end for
    end if
end for
M:=$\{$var($n$)$|\exists n \in b$.blue($n$)$\wedge$frontier($n$)$\}$
return M
```

(var_i,val_i)=(d,*1*). The set M returned by the algorithm is $\{c\}$. At the end of the reason-vars algorithm, a node in the BDD will be marked with blue, if the node can be reached from the root node without using an edge that violates an assignment of the variables in M and also without using a (var_i,val_i) edge. Note that, during a run of the reason-vars algorithm, a frontier node n will not be marked blue, if there is an edge (var_x,val_x) in all the paths between n and the root node, such that $\exists k \in b$.blue(k)$\wedge$frontier(k)$\wedge$(var(k)=var_x) and (var_x,$\neg val_x$)$\in A$.

4 Experiments

A prototype of the hybrid SAT solver was implemented in C++. The BDD part of the solver was implemented using the BuDDy [13] package. Experiments were done on six sets of benchmark instances. The *barrel* [14], *longmult* [14], *cnt* [15] and *w08* [15] instances are bounded model checking instances, while the *x1* [15] instances are xor-chain verification instances and the *pipe_k* [16] instances are microprocessor verification instances. All the longmult, barrel, x1 and pipe_k instances are unsatisfiable instances, while all the cnt instances are satisfiable. In case of the w08 instances, the w08_10 instance is unsatisfiable, while the rest are satisfiable.

The experiments were performed on an Intel Xeon 3.2GHz dual processor machine, running linux and with 4GB RAM. For each experiment at most 3600 seconds was given for solving an instance. A "TO" entry in the table mentions a timeout.

The benchmark statistics are shown in Table 1. In the table, the first column lists the instance name. The following three columns list the number of variables (#vars), clauses (#cls) and literals (#lits) in the corresponding instance. The next two columns list the number of decisions (#des) and time taken (in CPU seconds) by the *Minisat Solver* (version 1.14) [17, 18] to solve the instances. The next two columns list the time taken for obtaining a tree decomposition using the min-degree heuristic and the width (ω') of the decomposition. The last column list the percentage of (ω'/#vars). From the table, we can infer that in many cases the cost of finding the tree decompositions is quite affordable. In some of the cases the tree decompositions have very small tree-width, especially in case of the cnt instances. The rate at which ω' increases within each family of instances is quite small when compared with the rate of #vars increase. In some of the families, with the increase in the instance size, there is a considerable decrease in the (ω'/#vars) value. This is especially the case in the bounded model checking instances.

The implementation of our hybrid solver is mainly intended for studying the benefits due to different levels of hybridization. Since our implementation is not optimized for speed, it cannot be compared with well-optimized SAT solvers like Minisat. Also, our implementation does not have some of the techniques like clause deletion [11], clause minimization [19] and random restarts [11].

We did the experiments on our six set of instances with six different values for h (the hybridization parameter): $\{0,1,2,4,8,2^n\}$. The results of the experiments are listed in the Table 2. All the CPU time listings in the table include the time taken for obtaining a min-degree decomposition for the corresponding instances.

When h=0, the hybrid solver behaves as a pure CNF solver, as BDDs of size zero are alone allowed. At level h=2^n, since the hybrid representation creation process alone will determine satisfiability, there is no requirement for the #des column. The "MO" (memory out) entries in this column mean aborts of the corresponding experiments due to the memory limitations. Except the case of cnt and x1 instances, the h=2^n experiments have to be aborted.

Since BDDs can provide higher level of consistency than the pure CNF representations, we would expect fewer decisions when we allow larger BDDs. But as we can infer from the tables, in most of the cases, with the increase in the level of hybridization there is no effect on the number of decisions. The exceptions are cnt and x1 instances. Hence, even if we can find a better way of doing unit propagation in BDDs, it might not be useful, due to almost no reduction in the number of decisions required in the hybrid solver.

In case of the cnt instances, the reason for the success of h=2^n level should be mainly due to the very low tree width. For example, the cnt10 instance has 20470 variables, but its tree width is just 38 and such a tree decomposition can be obtained in just 3 seconds using min-degree. Interestingly, the rate at which the tree width of the cnt instances increase is very small when compared with the corresponding increase in the number of variables. For example cnt10 has

Instance	Original			Minisat		MinDeg		
	#vars	#cls	#lits	#des	Time	Time	ω'	(ω'/#vars)%
barrel5	1407	5383	14814	25464	1	0.78	171	12.15
barrel6	2306	8931	24664	98669	4	3.41	289	12.53
barrel7	3523	13765	38112	214373	22	7.24	407	11.55
barrel8	5106	20083	55716	164743	78	21.96	585	11.46
barrel9	8903	36606	102370	3671194	164	72.42	874	9.82
longmult7	3319	10335	24957	15656	3	0.65	86	2.59
longmult8	3810	11877	28667	55962	17	0.71	102	2.68
longmult9	4321	13479	32517	93112	37	0.84	99	2.29
longmult10	4852	15141	36507	123972	55	0.93	104	2.14
longmult11	5403	16863	40637	144381	71	1.06	115	2.13
longmult12	5974	18645	44907	150683	79	1.30	107	1.79
longmult13	6565	20487	49317	161321	84	1.54	120	1.83
longmult14	7176	22389	53867	128639	82	1.55	120	1.67
longmult15	7807	24351	58557	95544	50	1.82	126	1.61
cnt05	316	1002	2738	673	0.01	0.02	14	4.43
cnt06	762	2469	6753	452	0.04	0.04	21	2.76
cnt07	1786	5856	16016	20840	0.3	0.10	23	1.29
cnt08	4089	13531	36991	79958	2	0.33	24	0.59
cnt09	9207	30678	83822	363761	24	1.14	32	0.35
cnt10	20470	68561	187229	1620603	228	3.31	38	0.19
x1_16	46	122	364	6068	0.3	0.00	10	21.74
xl_24	70	186	556	174806	1	0.00	17	24.29
xl_32	94	250	748	557869	3.3	0.01	22	23.40
xl_36	106	282	844	1934814	13	0.01	25	23.58
xl_40	118	314	940	11777704	106	0.00	25	21.19
xl_44	130	346	1036	67154414	719	0.01	28	21.54
xl_48	142	378	1132	12738588	112	0.00	26	18.31
xl_56	166	442	1324		TO	0.01	38	22.89
xl_64	190	506	1516		TO	0.02	34	17.89
xl_72	214	570	1708		TO	0.01	42	19.63
xl_80	238	634	1900		TO	0.01	47	19.75
x1_96	286	762	2284		TO	0.01	65	22.73
xl_128	382	1018	3052		TO	0.03	66	17.28
2pipe_k	860	6693	18633	6566	0.1	2	114	13.26
3pipe_k	2391	27405	78127	66110	3	30	273	11.42
4pipe_k	5095	79489	229675	1363155	1160	208	483	9.48
5pipe_k	9330	189109	551125		TO	1070	950	10.18
6pipe_k	15346	408792	1199026	4858021	350	TO	1383	9.01
7pipe_k	23909	751116	2211464		TO	TO	1939	8.11
w08_10	71615	248504	604106	34694	18	177	1731	2.42
w08_14	120367	425316	1038230	245482	398	482	2476	2.06
w08_15	132555	469519	1146761	304040	569	535	2153	1.62

Table 1: Benchmark statistics

Instance	**h=0**		**h=1**		**h=2**		**h=4**		**h=8**		**h=2^n**
	#des	**Time**	**#des**	**Time**	**#des**	**Time**	**#des**	**Time**	**#des**	**Time**	**Time**
barrel5	10181	2	6248	4	5971	5	5759	216	6493	32	MO
barrel6	29120	13	22241	18	26566	28	24018	258	25563	285	MO
barrel7	75471	84	54947	87	43701	45	46940	316	72380	609	MO
barrel8	86185	172	74935	115	66350	102	74517	961	107803	1701	MO
barrel9	345055	493	231851	530	426801	1186	478656	1718		TO	MO
longmult7	23153	14	15456	27	14289	44	15688	870	17520	197	MO
longmult8	52817	59	42156	95	35198	124	40944	234	35307	511	MO
longmult9	84480	137	66481	181	74399	339	73674	461	81309	1195	MO
longmult10	113642	201	91055	282	118978	560	123713	787	113580	1369	MO
longmult11	127923	224	112036	422	127331	728	133042	1026	136856	2060	MO
longmult12	121660	197	159430	702	129012	785	131310	1193	132840	1625	MO
longmult13	114271	176	150467	706	182749	1248	133606	1277	126893	1993	MO
longmult14	145838	308	132870	636	153947	1239	142029	1602	131025	2632	MO
longmult15	117011	162	113217	519	113872	719	113408	1144	95167	1663	MO
cnt05	627	0.2	5	0.2	0	0.2	0	0.16	0	0.19	0.2
cnt06	3218	0.3	1465	0.4	0	0.3	0	0.22	0	0.22	0.3
cnt07	18055	0.8	11837	3	3224	2	68	0.39	0	0.36	0.5
cnt08	90636	11	44991	17	35507	22	7333	630	0	0.69	1
cnt09	354998	96	246166	184	143212	150	137478	269	113592	300	3
cnt10	1590226	2085	1076313	1859	726391	1410	634009	2291		TO	12
x1_16	1422	0.2	0	0.5	0	0.5	0	0.5	0	0.5	0.5
xl_24	29074	2	0	0.5	0	0.5	0	0.5	0	0.5	0.5
xl_32	471350	796	0	0.5	0	0.5	0	0.5	0	0.5	0.5
xl_36		TO	0	0.5	0	0.5	0	0.5	0	0.5	0.5
xl_40		TO	0	0.5	0	0.5	0	0.5	0	0.5	0.5
xl_44		TO	0	0.5	0	0.5	0	0.5	0	0.5	0.5
xl_48		TO	0	0.5	0	0.5	0	0.5	0	0.5	0.5
xl_56		TO	0	0.5	0	0.5	0	0.5	0	0.5	0.5
xl_64		TO	0	0.5	0	0.5	0	0.5	0	0.5	0.5
xl_72		TO	0	0.5	0	0.5	0	0.5	0	0.5	0.5
xl_80		TO	0	0.5	0	0.5	0	0.5	0	0.5	0.5
x1_96		TO	0	0.5	0	0.5	0	0.5	0	0.5	0.5
xl_128		TO	0	0.5	0	0.5	0	0.5	0	0.5	0.5
2pipe_k	12788	1	6856	4	7329	9	8203	18	7240	15	MO
3pipe_k	148601	38	85927	153	55548	193	173175	1063	109413	1238	MO
4pipe_k	788461	667	678733	2787	438606	3560		TO		TO	MO
5pipe_k	1856324	2943		TO		TO		TO		TO	MO
6pipe_k		TO		TO		TO		TO		TO	MO
7pipe_k		TO		TO		TO		TO		TO	MO
w08_10	34536	58	24570	187	17925	262	17116	394	15528	625	MO
w08_14	300409	1148	203378	2456		TO		TO		TO	MO
w08_15	294824	817	181341	1683	106069	1860	92761	2537		TO	MO

Table 2: Experiments with different values of h

more than double the number of variables in cnt09, but the tree width of cnt10 is just 20% more than the tree width of cnt09.

In case of the x1 instances, which are known to be very hard for DPLL solvers, pure CNF is not able to scale as high as the BDD hybrids. By observing the (ω'/#vars) column, the tree width for these instances does not seem to be very low. Hence, the BDD representation should have been the main reason for the scalability of the hybrid solver in the x1 instances.

5 Related Work

In [4], the authors converted each one of the clauses in an input CNF SAT problem into a BDD. Then a heuristic was used to conjoin the obtained BDDs. During the conjoining process, a variable will be existentially quantified out if it occurs in only one BDD. The conjoining process was restricted such that BDDs of size at most 100 nodes were only created. Then a hybrid SAT solver was implemented to decide satisfiability, in which the conflict learnt clauses will be in CNF form. In [4], we can observe that the restriction of creating BDDs of size at most 100 is ad hoc. We have defined an improved measure *h*nLits*, which is willing to create a BDD of size larger than 100 if the BDD replaces enough literals. This is more reasonable than a constant size limit. Also, most of the interesting instances used in the experiments of [4] are non-publicly available industrial instances. This makes it hard to infer conclusions from their experiments. All the instances we use and the descriptions of their origin are publicly available. Unlike [4], we use a tree decomposition to guide the BDDs creation process. This results in conversion of some parts of an input CNF into BDDs, while the rest remains in CNF form.

In [5], the authors used tree decomposition techniques to guide a CNF SAT solver, such that conflict learning creates clauses of size bounded by the width of a tree decomposition. The authors of [5] evaluated two heuristics to obtain tree decompositions: MINCE [20] and min-degree. Upon a preliminary evaluation the authors found out that the tree width of the tree decomposition by MINCE is generally smaller than that by the min-degree heuristic. Although the authors had also observed that min-degree heuristic is very fast than the MINCE, they dumped min-degree in favor of MINCE, mentioning the importance of smaller tree width. But the problem with MINCE, as observed in [5], is that the cost of obtaining MINCE tree decompositions of a SAT instance very often exceeds the actual cost of solving the instance using the state-of-the-art SAT solving techniques. Hence, we use min-degree heuristic in our work. Although min-degree might give poorer decompositions compared to the MINCE heuristic, when compared to the number of variables in an instance, the width of a decomposition obtained by min-degree is still very small (see Table 1).

In [4], the authors did not give the details of their method for generating minimal reason-vars set, and the complexity of their method was also not mentioned. Recently in [21], the authors have given a method for finding minimal reason-vars set, but the complexity of their method is quite high, as their method uses several existential quantification operations. Also, their method creates new BDD nodes, which should preferably be avoided. In our case, we use existential quantifications and new BDD node creations only while creating the hybrid representation. After that, during hybrid SAT solving, the BDDs remain static.

6 Conclusion

We have presented an evaluation of a hybrid SAT solver. From the experiments on the different levels of hybridization, we can fairly conclude that there are two cases where the hybridization certainly helps: the first case when the instances have very low tree width; the second case when the properties of instances make them hard for DPLL and easy for BDD based solvers. In all other cases, even allowing very small BDDs does not help. This is mainly due to the high cost of unit propagation in BDDs and no significant reduction in the number of required decisions. Therefore, even if one can find a better way of doing unit propagation in BDDs, as it will not reduce the decisions required, it might not improve the overall efficiency.

A nice future work topic is to study the effect of our hybrid solver using tree decompositions and BDDs on finding *all* solutions. Indeed, by storing the BDDs created for each node in the tree decompositions, we can have an implicit representation of all solutions. In ideal cases like the satisfiable cnt instances, we are able to build the BDDs up to the root node.

Overall, we hope that our investigations would have clarified the relative interest of CNF and BDD representations.

References

[1] Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **8** (1986) 677–691

[2] Groote, J.F., Zantema, H.: Resolution and binary decision diagrams cannot simulate each other polynomially. Discrete Applied Mathematics **130** (2003) 157–171

[3] Bodlaender, H.L.: Discovering treewidth. In: Proceedings of SOFSEM. (2005) 1–16

[4] Damiano, R.F., Kukula, J.H.: Checking satisfiability of a conjunction of BDDs. In: Proceedings of DAC. (2003) 818–823

[5] Bjesse, P., Kukula, J., Damiano, R., Stanion, T., Zhu, Y.: Guiding SAT diagnosis with tree decompositions. In: Proceedings of Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Selected Revised Papers, Springer LNCS (2004) 315–329

[6] Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)

[7] Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proceedings of TACAS. (1999) 193–207

[8] Heggernes, P., Eisenstat, S., Kumfert, G., Pothen, A.: The computational complexity of the minimum degree algorithm. In: Proceedings of NIK 2001 - 14th Norwegian Computer Science Conference. (2001) 98 – 109

[9] Bryant, R.E.: Symbolic boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys **24** (1992) 293–318

[10] Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: Proceedings of ICCAD. (2001) 279–285

[11] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC'01). (2001)

[12] Silva, J.P.M., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48** (1999) 506–521

[13] Lind-Nielsen, J.: BuDDy - A Binary Decision Diagram Package. `http://sourceforge.net/projects/buddy` (online)

[14] CMU-BMC instances. `http://www.cs.cmu.edu/~modelcheck/bmc.html` (online)

[15] SAT 2002 competition instances. `http://www.satlib.org` (online)

[16] Microprocessor Benchmarks. `www.ece.cmu.edu/~mvelev/sat_benchmarks.html` (online)

[17] Een, N., Sorensson, N.: An extensible SAT-solver. In: Proceedings of SAT. (2003) 502–518

[18] Minisat SAT Solver. `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/Main.html` (online)

[19] Een, N., Sorensson, N.: Minisat a SAT solver with conflict-clause minimization. In: Posters of SAT. (2005)

[20] Aloul, F., Markov, I., Sakallah, K.A.: MINCE: A static global variable-ordering heuristic for sat search and bdd manipulation. Journal of Universal Computer Science **10** (2004) 1562–1596

[21] Hawkins, P., Stuckey, P.J.: A hybrid BDD and SAT finite domain constraint solver. In: Proceedings of PADL. (2006) 103–117

APPENDIX

A The Linear Time Algorithm for Minimal Reason-Vars

Given a set of assignments A=$\{(var_{a_1},val_{a_1}),..,(var_{a_k},val_{a_k})\}$, a BDD and an implication (var_i,val_i), where the assignments A results in the BDD to imply (var_i,val_i). Our goal is to find a minimal subset of variables in A, whose assignments are enough for the BDD to imply (var_i,val_i).

In a BDD each path from the root node to the 1-terminal represents one or more solutions. The pair (var_i,val_i) was implied by the BDD means that there is no path in the BDD that contains $(var_i,\neg val_i)$, such that the path does not violate any of the assignments of A.

Given any path from the root node of BDD to the 1-terminal that contains $(var_i,\neg val_i)$, there will be at least one edge in the path, that violates an assignment in A. Among them we can pick the edge that is nearest to the 1-terminal and label it as a *frontier edge*. If a node has one of its outgoing edges labeled as a frontier edge, then the node could be labeled as a *frontier node*.

Considering all the paths in the BDD which contain $(var_i,\neg val_i)$, we can label all the frontier nodes in the BDD. This can be done in linear time, as a BDD is a rooted directed acyclic graph. Now *mark* the root node of the BDD. Then, in the sequence of the variable order used in the BDD, look at the marked nodes of each variable v in the BDD. If v=var_i, then mark the child nodes of the marked nodes of v, which can only be reached using the $\neg val_i$ edge. This is because, when we find the minimal reason-vars set we can ignore the solutions paths not containing $(var_i,\neg val_i)$. If the variable v is not var_i, and *none* of the marked nodes of v is a frontier node, then mark both the child nodes of each one of the marked nodes of v. If *any* of the marked nodes of the variable v is a frontier node, then for each node n of those marked nodes, mark the child node of n, which can only be reached using the outgoing edge of n that contains the assignment for v. Note that the 1-terminal cannot be marked, as we ignore all the solution paths not containing $(var_i,\neg val_i)$, and in the rest of the paths there will be at least one *marked frontier node*. Now, let M be the set of variables, such that for each $v \in$ M, there is at least one node of v in the BDD, which is a *marked frontier node*. The set M is a minimal reason-vars set. The following low-level descriptions can be used in checking the correctness and the linear-time complexity of the whole algorithm.

The algorithm we call *GetReasonVars* is shown in Figure 4. In the figure (var_i,val_i) is denoted by (impliedVar,impliedValue).

The GetReasonVars procedure requires five variables to be associated with each node in a BDD. The five variables are: *highcolor*, *lowcolor*, *color*, *colorMarked* and *isFrontier*. The first three variables can be assigned a value from the set {GREEN,RED,BLUE}, while the last two variables are Boolean variables. Apart from these variables, each node will also have a variable *varId* denoting the variable of the node. The notation ”b.highNode” refers to the BDD node at the end of the high edge of the BDD node b.

The procedure takes three arguments. The first one ”*tn*” denotes the tree

```
GetReasonVars(TreeNode tn, BddNode b, int[] isRequired)
1   MarkReasonColor(b)
2   b.color=BLUE
3   forall varId contained in tn /* loop in the BDD variable order */
4       blueMarkedNodes=GetBlueMarkedNodes(varId,b)
5       forall node in blueMarkedNodes
6           if(node.isFrontier)
7               isRequired[varId]=TRUE
8       if(isRequired[varId]==FALSE)
9           forall node in blueMarkedNodes
10              if(varId!=impliedVar || impliedValue!=TRUE)
11                  node.highNode.color=BLUE
12              if(varId!=impliedVar || impliedValue!=FALSE)
13                  node.lowNode.color=BLUE
14      else
15          forall node in blueMarkedNodes
16              if(value[varId]==TRUE)
17                  node.highNode.color=BLUE
18              else
19                  node.lowNode.color=BLUE
```

Figure 4: The GetReasonVars algorithm.

node whose BDD made the implication. The second one "b" denotes the root node of the BDD. The last one "*isRequired*" is an array of Boolean values indexed by variables. Before calling the procedure, the isFrontier and colorMarked variables in all the nodes of the BDD needs to be initialized to FALSE. All the entries in the isRequired array needs to be FALSE as well. When the procedure returns, the variables contained in the tree node "tn", whose isRequired value is set to TRUE will be a minimal reason-vars set.

The procedure calls another procedure called *MarkReasonColor* shown in Figure 5. The MarkReasonColor algorithm is a recursive procedure which takes a BDD Node "b" as input. When the procedure returns, the "b.color" will be set to GREEN if there is a path from b to the 1-terminal not containing (var_i,val_i), and the path does not violate any of the assignments in A. The MarkReasonColor procedure assigns TRUE value to the isFrontier variable of a node, only if there exists a path from the node to the 1-terminal node by violating the assignment of the varId of the node, without violating any other assignment in A and the path does not contain (var_i,val_i). We label a node whose isFrontier is set to TRUE as a *frontier node*. When the call made to the MarkReasonColor from the GetReasonVars procedure returns, the color of the root node of the BDD will have been set a RED value, otherwise there would not have been an implication of (var_i,val_i).

At the line 2 of the GetReasonVars procedure, the color of the root node of the input BDD is set to BLUE. At the end of the procedure, the BLUE marked nodes are those that can be reached from the root node without using any frontier edge. The line 3 starts a loop on the list of variables contained in the tree node "tn", where the variables in the list are ordered based on the variable ordering used in the BDD. The procedure *GetBlueMarkedNode(varId,b)* would return a list of all the varId variable nodes in "b", which are marked with BLUE color. Note that, in a loop for a variable varId, isRequired[varId] is set to TRUE, only if there exists at least one frontier node of varId, which node can be reached by a path from the root node of the input BDD, with the path not containing an another frontier node with its isRequired value set to FALSE.

When the GetReasonVars procedure returns, all the variable contained in the "tn" tree node, whose isRequired value is set to TRUE will form a minimal reason-vars set.

```
MarkReasonColor(BddNode b)
if(b==1-terminal)  b.color=GREEN; b.colorMarked=TRUE; return
if(b==0-terminal)  b.color=RED; b.colorMarked=TRUE; return
if(b.highNode.colorMarked==FALSE) MarkReasonColor(b.highNode)
if(b.lowNode.colorMarked==FALSE)  MarkReasonColor(b.lowNode)
b.highColor=b.highNode.color; b.lowColor=b.lowNode.color
if(b.varId==impliedVar)
    if(impliedValue==TRUE)
        b.highColor=RED
    else
        b.lowColor=RED
else
    if(value[b.varId]==FALSE)
        if(b.highColor==GREEN)
            b.highColor=RED; b.isFrontier=TRUE
    else if(value[b.varId]==TRUE)
        if(b.lowColor==GREEN)
            b.lowColor=RED; b.isFrontier=TRUE
    if(b.highColor==GREEN || b.lowColor==GREEN)
        b.color=GREEN
    else
        b.color=RED
b.colorMarked=TRUE; return
```

Figure 5: The MarkReasonColor algorithm.

A.0.1 The correctness of GetReasonVars

The correctness of GetReasonVars procedure can be observed from the facts:

1. any of the paths from the root node of the BDD to the 1-terminal, that

contains the assignment (var_i,$\neg val_i$) violate at least one among the assignments of the variables in the reason-vars set obtained by using GetReasonVars procedure. This shows that the reason-vars set is *enough* for the implication.

2. removing any variable from the obtained reason-vars set will result in a path from a BLUE marked frontier node to the 1-terminal node without violating the assignment to the other variables in the reason-vars set. Since, this implies that the assignment pair (var_i,val_i) is not an implication, the reason-vars set obtained by GetReasonVars is *minimal*.

A.0.2 The complexity of GetReasonVars

The initialization of isRequired to FALSE values needs to be done only once, when the hybrid SAT solver is started. After each call to GetReasonVars, only the isRequired entries of the variables in the corresponding tree node needs to be reset to FALSE. The initialization of isFrontier and colorMarked values of the BDD nodes could be done in a traversal of the BDD nodes in linear-time. The total number of calls to MarkReasonVars is linear as the function is invoked at most once for a node. An implementation of the GetBlueMarkedNodes will also need only linear time; we can maintain a list of BLUE marked nodes for each varId, a node is entered into the list at most once and the list of BLUE marked nodes for a variable is read at most once. Since, each line inside the loop at line 3 of GetReasonVars is used at most once for each node in the BDD, the complexity of the GetReasonVars procedure is linear in the number of nodes in the BDD.