Cost-Effective Hyper-Resolution for Preprocessing CNF Formulas

Roman Gershman and Ofer Strichman

Technion, Haifa, Israel
gershman@cs.technion.ac.il
ofers@ie.technion.ac.il

Abstract. We present an improvement to the HYPRE preprocessing algorithm that was suggested by Bacchus and Winter in SAT 2003 [1]. Given the power of modern SAT solvers, HYPRE is currently one of the only cost-effective preprocessors, at least when combined with some modern SAT solvers and on certain classes of problems. Our algorithm, although it produces less information than HYPRE, is much more efficient. Experiments on a large set of industrial Benchmark sets from previous SAT competitions show that HYPERBINFAST is always faster than HYPRE (sometimes an order of magnitude faster on some of the bigger CNF formulas), and achieves faster total run times, including the SAT solver's time. The experiments also show that HYPERBINFAST is cost-effective when combined with three state-of-the-art SAT solvers.

1 Introduction

Given the power of modern SAT solvers, most CNF preprocessing algorithms are mostly not cost-effective time-wise. Since these solvers are so effective in focusing on the important information in a given CNF, it is particularly challenging to find the right balance between the amount of effort invested in preprocessing and the quality of information gained, in order to positively impact the overall solving time.

The best currently available preprocessor that we are aware of is HYPRE in [1], which is cost-effective when combined with some of the modern SAT solvers, although it fails to be so with very large CNF files. We present a new preprocessing algorithm HYPERBINFAST for CNF formulas which is an improvement of the HYPRE algorithm. Our algorithm makes the preprocessing of CNF formulas cost-effective time-wise by relaxing the optimality constraints presented in the original algorithm. Experiments show that giving up optimality generally improves the overall solving time (preprocessing + SAT). Another advantage of HYPERBINFAST is that it is implemented as an anytime algorithm that can be stopped either according to a predetermined timeout or according to a heuristic function that decides when to stop it by measuring its progress. Due to limit of space we refer the interested reader to the full version of this article [2] for a more detailed description of this heuristic, as well as more experimental results, comparison to previous work and a detailed comparison to HYPRE.

2 Definitions and Motivation

Definition 1 (Binary Implications graph). *Given a CNF formula φ with a set of binary clauses B, a* Binary Implications Graph *is a directed graph $G(V, E)$ such that $v \in V$ if and only if v is a literal in φ, and $e = (u, v)$ is an edge if and only if B contains a clause $(\overline{u}, v)$.*

A Binary Implications Graph allows us to follow implications through binary clauses. Note that for each binary clause (u, v), both $(\overline{u}, v)$ and $(\overline{v}, u)$ are edges in this graph. Thus, the total number of edges in the initial (before further processing) graph is twice the size of B. This is what we refer to as the *symmetry* of Binary Implications Graphs. Binary Implication Graphs are *Static* and are not related to the standard implication graphs that describe the progress of Unit Propagation (UP - also known as BCP).

Definition 2 (Binary Transitive Closure of a literal). *Given a literal v, a set of literals denoted by $BTC(v)$ is the* Binary Transitive Closure of v *if it contains exactly those literals that are implied by v through the Binary Implications Graph.*

Definition 3 (Failed literal). *A literal v is called a* Failed Literal *if setting its value to $TRUE$ and applying UP causes a conflict.*

Definition 4 (Propagation closure of a literal). *Given a non-failed literal u, a set of literals denoted by $UP(u)$ is the* Propagation closure of u *if it contains exactly those literals that are implied through UP by u in the given CNF (not only the binary clauses).*

It is easy to see that $BTC(v) \subseteq UP(v)$ for every literal v, because $UP(v)$ is not restricted to what can be inferred from binary clauses. Note that $v \in UP(u)$ implies that $u \rightarrow v$ and hence $\overline{v} \rightarrow \overline{u}$, but it is not necessarily the case that $\overline{u} \in UP(\overline{v})$, due to the limitations of UP. For example, in the set of clauses $(\overline{x} \vee y), (\overline{x} \vee z), (\overline{y} \vee \overline{z} \vee w)$, it holds that $x \rightarrow w$ and hence $\overline{w} \rightarrow \overline{x}$, but UP detects only the first direction. It disregards $\overline{w} \rightarrow \overline{x}$ because $\overline{w}$ does not invoke any unit clause. Hence, UP lacks the symmetry of Binary Implications Graphs.

3 The HyperBinFast Algorithm

Algorithm HyperBinFast iterates over all root nodes in the Binary Implications Graph ($roots(v)$ denotes the set of all ancestor roots of v in such a graph). It has two main stages. In the first stage (lines 4 - 5) it iteratively finds equal literals (by detecting SCCs and unifying their vertices to a single 'representing literal'), propagates unit clauses, and simplifies the clauses in the formula. Simplification in this context corresponds to substituting literals by their representative literal in all clauses (not only binary), removing literals that are

```
HyperBinFast
 1: Mark all root vertices as weak;
 2: while there are weak roots, unit clauses, or binary cycles do
 3:     while there are unit clauses or binary cycles do
 4:         Detect all SCCs and collapse each one of them to a single node;
 5:         Propagate all unit clauses and simplify all clauses accordingly;
 6:         For each new binary clause $(u, v)$ mark as weak $roots(\overline{u})$ and $roots(\overline{v})$;
 7:     Choose a weak root node $v$;
 8:     $FailedLiteral$ =FastVisit ($v$);
 9:     Undo assignments caused by BinaryWalk and clear nQueue;
10:     if $FailedLiteral \neq$ undefined then
11:         Add unit clause $(\overline{FailedLiteral})$;
12:     Mark $v$ as strong;
```

```
FastVisit (Literal $v$)
 1: $res \leftarrow$ BinaryWalk ($v$, NULL);
 2: if $res \neq undefined$ then
 3:     return $res$
 4: while !nQueue.empty() do
 5:     Literal $p \leftarrow$ nQueue.pop_front();
 6:     for each n-ary clause $\in$ watched(p) do                    $\triangleright$ $n > 2$
 7:         if $clause$ is conflict then
 8:             Literal $fUIP \leftarrow$ FindUIP ($clause$);
 9:             return $fUIP$;
10:         else if $clause$ is unit then
11:             Literal $toLit \leftarrow$ undefined literal from $clause$.
12:             Literal $fromLit \leftarrow$ FindUIP ($clause \setminus \{toLit\}$);
13:             Add clause $(\overline{fromLit}, toLit)$
14:             Mark $roots(\overline{toLit}) \cup roots(fromLit)$ as weak
15:             $res \leftarrow$ BinaryWalk ($toLit$, $(\overline{fromLit}, toLit)$);
16:             if $res \neq undefined$ then
17:                 return $res$
18: return $undefined$;
```

```
BinaryWalk (Literal $t$, Antec clause C)
 1: if value($t$)=True then
 2:     return $undefined$;
 3: if value($t$)=False then
 4:     return $\overline{t}$;
 5: $value(t) \leftarrow TRUE$;
 6: $antecedent(var(t)) \leftarrow C$;
 7: Put $t$ on assignment stack;
 8: Put $t$ into nQueue;
 9: for each binary clause $(\overline{t}, u)$ do
10:     $res \leftarrow$ BinaryWalk ($u$, $(\overline{t}, u)$);
11:     if $res \neq undefined$ then
12:         return $res$;
13: return $undefined$;
```

```
FindUIP (Literal set $S$)
 1: mark all variables in $S$;
 2: $count \leftarrow |S|$;
 3: while $count > 1$ do
 4:     $v \leftarrow$ latest marked variable in the
            assignment stack
 5:     unmark $v$; $count - -$;
 6:     Let $(u, L)$ be antecedent clause of
            $v$, s.t. $var(L) = v$.
 7:     if $var(u)$ not marked then
 8:         mark $var(u)$; $count + +$;
 9: $res \leftarrow$ last marked literal in assign-
        ment stack.
10: unmark $var(res)$;
11: return $res$;
```

evaluated to FALSE and removing satisfied clauses. The simplification may result in shortening of some n-ary clauses to binary clauses, which change the Binary Implications Graph. In line 6 we perform a restricted version of what HYPRE does in such cases: while HYPRE marks as *weak* (i.e. nodes that should still be processed) all ancestor nodes, HYPERBINFAST only marks root ancestor nodes. Further, while HYPRE invokes this process every time an n-ary clause is being shortened, HYPERBINFAST only does so for clauses that become binary. The reduced overhead due to these changes is clear. In the second stage (line 8), we invoke FASTVISIT for some weak root node, a procedure that we will describe next. FASTVISIT can change the graph as well, so HYPERBINFAST iterates until convergence.

Computing the Binary Transitive Closure. Before describing FASTVISIT, we concentrate on the auxiliary function BINARYWALK, which FASTVISIT calls several times. The goal of BINARYWALK is to mark all literals that are in BTC(v) or return a failed literal, which can be either v itself or some descendant of v. It also updates a queue, called *nQueue* with those literals in BTC(v) for future processing by FASTVISIT. BINARYWALK performs DFS from a given literal on the Binary Implications Graph. In each recursive-call, if t is already set to FALSE (i.e. $\overline{t}$ is already set to TRUE in the current call to FASTVISIT), it means that there is a path in the binary implication graph from $\overline{t}$ to t, and hence $\overline{t}$ is a failed literal. This is a direct consequence of the following lemma:

Lemma 1. *In a DFS-traversal on a Binary Implications Graph from a literal u that marks all nodes it visits, if when visiting a node t another node $\overline{t}$ is already marked, and this is the* first time *such a 'collision' is detected, then $\overline{t} \xrightarrow{*} t$.*

When BINARYWALK detects such a failed literal it returns $\overline{t}$ all the way out (due to lines 11-12) and back to FASTVISIT and then to HYPERBINFAST.

The other case is when t does not have a value yet. In this case BINARYWALK sets it to TRUE and places it in *nQueue*, which is a queue of literals to be propagated later on by FASTVISIT. It also places t in the (global) assignment stack, and stores for $var(t)$ its antecedent clause (the clause that led to this assignment), both for later use in FINDUIP.

From Binary Transitive Closure to Propagation Closure. We now describe FASTVISIT. Recall that FASTVISIT is invoked for each root node in the Binary Implications Graph. FASTVISIT combines Unit Propagation with Binary Learning based on single assignments, i.e. learning of new clauses by propagating a single decision at a time. It relies on the simple observation that if $u \in UP(v)$ then $v \rightarrow u$. It is too costly to add an edge for every such pair v, u, because this corresponds to at least computing the transitive closure [1]. Since our stated goal is to form a binary graph in which $UP(v) = BTC(v)$ for each root node, it is enough to focus on a vertex u only if $u \in UP(v)$ but $u \notin BTC(v)$. Further, given such a vertex u, although adding the edge $v \rightarrow u$ achieves this goal, we rather find a vertex w, a descendant of v that also implies u, in the spirit of the First Unique-Implication-Point (UIP) scheme. The FINDUIP function called by FastVisit can in fact be seen as a variation of the standard algorithm for finding first UIPs [4]: unlike the standard usage of such a function in analyzing conflicts, here there are no decision levels and the clauses are binary. On the other hand it can receive as input an arbitrary set of assigned literals, and not just a conflict clause.

In line 4 FastVisit starts to process the literals in *nQueue*. For each literal p in this queue, it checks all the n-ary clauses ($n > 2$) watched by p. As usual, each such clause can be of interest if it is either conflicting or unit. If it is conflicting, then FastVisit calls FindUIP, which returns the first UIP causing this conflict. This UIP is a failed literal and is returned to HyperBinFast, which adds its negation as a unit clause in line 11. If the processed clause is a unit clause, the unassigned literal, denoted by *toLit*, is a literal implied by v that is not in $BTC(v)$ (otherwise it would be marked as TRUE in BinaryWalk). In other words, $toLit \in UP(v)$ and $toLit \notin BTC(v)$, which is exactly what we are looking for. At this point we can add a clause $(\overline{v}, toLit)$ but rather we call FindUIP, which returns a first UIP denoted by $fromLit$. The clause $(\overline{fromLit}, toLit)$ is stronger than $(\overline{v}, toLit)$ because the former also adds the information that $\overline{tolit} \rightarrow fromlit$. Note that this is an unusual use of this function, because *clause* is not conflicting. Because the addition of this clause changes the Binary Implications Graph, we need to mark as weak all the ancestor nodes of $fromLit$ and of $\overline{toLit}$, and to continue with BinaryWalk from *toLit*. This in effect continues to compute $BTC(v)$ with the added clause.

4 Experiments, Conclusions and Future Research

Table 1 shows experiments on an Intel 2.5Ghz computer with 1GB memory running Linux. The benchmark set is comprised of 165 industrial instances used in various SAT competitions. The number in brackets for each benchmark set denotes the number of instances. The global timeout for each instance was set to 3000 seconds. The timeout for HyperBinFast was set to 300 seconds, and for Hypre was set to 3000 seconds (Hypre is not implemented as anytime, and only full preprocessing is allowed). The timeout for the SAT solver was dynamically reduced to 3000 minus the time spent during preprocessing. All times in the table include preprocessing time when relevant. We count each failure as 3000 seconds as well. We used our experimental solver HaifaSat which participates in the SAT05 competition and Siege_v1 [3]. The full version of this article includes also a detailed comparison to zChaff 2004. Briefly, zChaff's total run-time is 212,508 sec. (56 fails) and with HyperBinFast it is 179,997 (44 fails).

The table shows that: 1) HyperBinFast helps each of the tested solvers to solve more instances in the given time bound on average 2) When the instance is solvable without HyperBinFast, still HyperBinFast typically reduces the overall run time 3) Whenever HyperBinFast does not help, its overhead in time is relatively small 4) It is very rare that an instance can be solved without HyperBinFast but cannot be solved with HyperBinFast. 5) On average, the total gain in time is about 20-25% relative to the pure configuration. 6) Some-

Table 1. Run-times (in seconds) and failures (denoted by 'F') for various SAT solvers with and without HyperBinFast. Times which are smaller by 10% than in competing configurations with the same SAT solver are bolded. Failures denoted by * are partially caused by bugs in the SAT solver

SAT solver →	HaifaSat						Siege_v1					
Preprocessor →	—		H-B-Fast		Hypre		—		H-B-Fast		Hypre	
	Time	F	Time	F	Time	F	Time	F	Time	F	Time	F
01_rule(20)	19,172	2	**7,379**	0	10,758	1	20,730	4	11,408	1	**5,318**	0
11_rule_2(20)	22,975	6	**7,491**	0	21,247	0	29,303	8	**17,733**	2	20,178	1
22_rule(20)	27,597	8	22,226	5	**16,825**	2	31,839	10	29,044	9	**17,510**	3
bmc2(6)	1,262	0	**81**	0	163	0	3,335	1	**85**	0	156	0
CheckerI-C(4)	**682**	0	902	0	989	0	4,114	0	3,541	0	**2,492**	0
comb(3)	4,131	1	4,171	1	4,056	1	5,679	1	6,027	1	**4,439**	1
f2clk(3)	4,059	1	4,060	1	**3,448**	1	6,105	2	6,063	2	**5,078**	1
fifo8(4)	1,833	0	**554**	0	1,756	0	5,555	1	2,420	0	**1,159**	0
fvp2(22)	1,995	0	2,117	0	3,288	0	1,860	0	2,009	0	2,431	0
hanoi(5)	131	0	285	0	119	0	**357**	0	1,231	0	802	0
hanoi03(4)	**427**	0	533	0	979	0	6,026	2	6,028	2	6,014	2
IBM02(9)	**3,876**	0	5,070	0	11,072	3	10,442	4	**7,881**	0	12,653	3
ip(4)	203	0	**172**	0	365	0	630	0	548	0	**349**	0
pipe03(3)	1,339	0	1,266	0	1,809	0	2,006	0	**1,275**	0	1,822	0
pipe-sat-1-1(10)	**3,310**	0	5,147	0	27,130	10	**2,445**	0	5,249	0	30,029	10
sat02(9)	17,330	4	**14,797**	4	16,669	4	24,182	7	18,843	5	17,662	4
vis-bmc(8)	13,768	3	10,717	2	10,139	2	10,449	2	6,989	1	**5,715**	0
vliw_unsat_2.0(8)	19,425	5	19,862	6	20,421	6	16,983	6	17,891	6	20,375	6
w08(3)	2,681	0	**1,421**	0	2,899	0	4,387	1	**1,711**	0	2,726	0
Total(165)	146,194	30	**108,251**	19	154,132	30	186,426	49	145,978	29	156,910	31

times HyperBinFast is weaker than Hypre (it does not simplify the formula enough), so the SAT solver fails on the corresponding instance but succeeds after applying Hypre. 7) With HaifaSat, Hypre is not cost-effective, neither in the total number of failures or the total run time, while HyperBinFast reduces HaifaSat's failures by 35% and reduces its total solving time by 25%.

Preprocessing Vs. SAT: For the above benchmark, it took HaifaSat 97,909 seconds after HyperBinFast and only 54,567 seconds after Hypre, which indicates that indeed the quality of the CNF generated by Hypre is better, as expected. But these numbers may mislead because, recall, the timeouts for the two preprocessors are different, which, in turn, is because HyperBinFast is an anytime algorithm. In Table 2 we list several benchmarks for which both preprocessors terminated before their respective timeouts, together with the time it took the preprocessor and then HaifaSat to solve them. Therefore, this table shows performance of HyperBinFast comparing to Hypre without taking into consideration the anytime aspect. To the extent that these instances are representative, it can be seen that typically the SAT solving time is longer after

Table 2. Few representative single instances for which both HYPRE and HYPERBINFAST terminated before their respective (different) timeouts. The SAT times refer to HaifaSat's solving time. It can be seen that typically the solving time is longer after HYPERBINFAST, but together with the SAT solver time it is more cost effective than HYPRE

Benchmark:	HYPRE	SAT	HYPERBINFAST	SAT
01_rule.k95.cnf	377	1679	4	1504
11_rule2.k70.cnf	1387	47	71	285
22_rule.k70.cnf	671	251	51	1410
fifo8_400.cnf	164	1226	12	309
7pipe.cnf	651	258	147	416
ip50.cnf	109	82	6	79
w08_14.cnf	1231	5	267	298
Total:	4590	3548	558	4301

HYPERBINFAST, but together with the SAT solver time it is more cost effective than HYPRE.

Conclusions and future research. Our preprocessor HYPERBINFAST is a compromise between the original HYPRE and a pure SAT solver: it tries to benefit from the preprocessing when possible while reducing the overhead when it is not effective. With HYPERBINFAST, preprocessing is generally cost-effective when combined with modern SAT solvers, as is evident from our experiments with 165 industrial CNF instances from previous SAT competitions. We pointed to two directions for future research: develop more efficient dynamic strategies for determining the amount of time spent for preprocessing, and make preprocessing *decide* on the set of variables from which it begins its traversal of the Binary Implications Graph (and not just choose all the root nodes as we do now). This concept can be generalized to preprocessing in general: while SAT solvers focus on the *semantics* of the formula, that is, they attempt to find the 'important' variables, preprocessors focus on the *syntactical* characteristics of the formula, and are therefore much more sensitive to its size. Hence, attempting to build a *semantic preprocessor* seems like a worthwhile direction to pursue next.

References

1. F. Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *SAT2003*, volume 2919 of *Lect. Notes in Comp. Sci.*, pages 341–355, 2003.
2. Roman Gershman and Ofer Strichman. Cost-effective hyper-resolution for preprocessing cnf formulas(full version), 2005. www.cs.technion.ac.il/~gershman/papers/sat05_full.pdf.
3. L.Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.
4. J.P.M. Silva and K.A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical Report TR-CSE-292996, Univerisity of Michigen, 1996.