Using Problem Structure for Efficient Clause Learning

Ashish Sabharwal*, Paul Beame*, and Henry Kautz*

Computer Science and Engineering
University of Washington, Seattle WA 98195-2350
{ashish,beame,kautz}@cs.washington.edu

Abstract. DPLL based clause learning algorithms for satisfiability testing are known to work very well in practice. However, like most branch-and-bound techniques, their performance depends heavily on the variable order used in making branching decisions. We propose a novel way of exploiting the underlying problem structure to guide clause learning algorithms toward faster solutions. The key idea is to use a higher level problem description, such as a graph or a PDDL specification, to generate a good branching sequence as an aid to SAT solvers. The sequence captures hierarchical structure that is lost in the CNF translation. We show that this leads to exponential speedups on grid and randomized pebbling problems. The ideas we use originate from the analysis of problem structure recently used in [1] to study clause learning from a theoretical perspective.

1 Introduction

The NP-complete problem of determining whether or not a given CNF formula is satisfiable has been gaining wide interest in recent years as solutions to larger and larger instances from a variety of problem classes become feasible. With the continuing improvement in the performance of satisfiability (SAT) solvers, the field has moved from playing with toy problems to handling practical applications in a number of different areas. These include hardware verification [2, 3], group theory [4], circuit diagnosis, and experiment design [5, 6].

The Davis-Putnam-Logemann-Loveland procedure (DPLL) [7, 8] forms the backbone of most common complete SAT solvers that perform a recursive backtrack search, looking for a satisfying assignment to the variables of the given formula. The key idea behind it is the pruning of the search space based on falsified clauses. Various extensions to the basic DPLL procedure have been proposed, including smart branching selection heuristics [9], randomized restarts [10], and clause learning [11–15]. The last of these, which this paper attempts to exploit more effectively, originated from earlier work on explanation based learning (EBL) [16–19] and has resulted in tremendous improvement in performance on many useful problem classes. It works by adding a new clause to the set of given clauses ("learning" this clause) whenever the DPLL procedure fails on a partial assignment and needs to backtrack. This typically saves work later in the search process when another assignment fails for a similar reason.

Both random CNF formulas and those encoding various real-world problems are hard for current SAT solvers. However, while DPLL based algorithms with lookahead

* Research supported by NSF Grant ITR-0219468

but no learning (such as `satz` [20]) and those that try only one carefully chosen assignment without any backtracks (such as `SurveyProp` [21]) are our best tools for solving random formula instances, formulas arising from various real applications seem to require clause learning as a critical ingredient. The key thing that makes this second class of formulas different is the inherent structure, such as dependence graphs in scheduling problems, causes and effects in planning, and algebraic structure in group theory.

Trying to understanding clause learning from the theoretical point of view of proof complexity has lead to many useful insights. In [1], we showed that on certain classes of formulas, clause learning is provably exponential stronger than a proof system called regular resolution. This in turn implies an even larger exponential gap between the power of DPLL and that of clause learning, thus explaining the performance gains observed empirically. It also shows that for such structured formulas, our favorite non-learning SAT solvers for random formulas such as `satz` and `SurveyProp` are doomed to fail, whereas clause learning provides potential for small proofs. However, in order to leverage their strength, clause learning algorithms must use the "right" variable order for their branching decisions for the underlying DPLL procedure. While a good variable order may result in a polynomial time solution, a bad one can make the process as slow as basic DPLL without learning. This leads to a natural question: can such insights from theoretical analysis of problem structure help us further? For example, for the domains where we can deduce analytically that small solutions exist, can we guide clause learning algorithms to *find* these solutions efficiently?

As we mentioned previously, most theoretical and practical problem instances of satisfiability problems originate, not surprisingly, from a higher level description, such as a graph or Planning Domain Definition Language (PDDL) specification [22]. Typically, this description contains more structure of the original problem than is visible in the flat CNF representation in DIMACS format [23] to which it is converted before being fed into a SAT solver. This structure can potentially be used to gain efficiency in the solution process. While there has been work on extracting structure after conversion into a CNF formula by exploiting variable dependency [24], constraint redundancy [25], symmetry [26], binary clauses [27] and partitioning [28], using the original higher level description itself to generate structural information is likely to be more effective. The latter approach, despite its intuitive appeal, remains largely unexplored, except for suggested use in bounded model checking [29] and the separate consideration of cause variables and effect variables in planning [30].

In this paper, we further open this line of research by proposing an effective method for exploiting problem structure to guide the branching decision process of clause learning algorithms. Our approach uses the original high level problem description to generate not only a CNF encoding but also a *branching sequence* [1] that guides the SAT solver toward an efficient solution. This branching sequence serves as auxiliary structural information that was possibly lost in the process of encoding the problem as a CNF formula. It makes clause learning algorithms learn useful clauses instead of wasting time learning those that may not be reused in future at all. We give an exact sequence generation algorithm for pebbling formulas. The high level description used is a pebbling graph. Our sequence generator works for the 1UIP learning scheme [15], which is

one of the best known. Our empirical results are based on our extension of the popular SAT solver `zChaff` [14].

We show that the use of branching sequences produced by our generator leads to exponential speedups for the class of grid and randomized pebbling formulas. These formulas, more commonly occurring in theoretical proof complexity literature [31–34], can be thought of as representing precedence graphs in dependent task systems and scheduling scenarios. They can also be viewed as restricted planning problems. Although admitting a polynomial size solution, both grid and randomized pebbling problems are not so easy to solve deterministically, as is indicated by our experimental results for unmodified `zChaff`. From a broader perspective, our result for pebbling formulas serves as a proof of concept that analysis of problem structure can be used to achieve dramatic improvements even in the current best clause learning based SAT solvers.

2 Preliminaries

A Conjunctive Normal Form (CNF) formula F is an AND ($\wedge$) of *clauses*, where each clause is an OR ($\vee$) of *literals*, and a literal is a variable or its negation ($\neg$). The Davis-Putnam-Logemann-Loveland (DPLL) procedure [7, 8] for testing satisfiability of such formulas works by *branching* on variables, setting them to TRUE or FALSE, until either an initial clause is *violated* (*i.e.* has all literals set to FALSE) or all variables have been set. In the former case, it backtracks to the last branching decision whose other branch has not been tried yet, reverses the decision, and proceeds recursively. In the latter, it terminates with a satisfying assignment. If all possible branches have been unsuccessfully tried, the formula is declared unsatisfiable. To increase efficiency, *pure literals* (those whose negation does not appear) and *unit clauses* (those with only one unset literal) are immediately set to true. In this paper, by DPLL we will mean this basic procedure along with additions such as randomized restarts [10] and local or global branching heuristics [9], but no learning.

Clause learning (see *e.g.* [11]) can be thought of as an extension of the DPLL procedure that caches causes of assignment failures in the form of learned clauses. It proceeds by following the normal branching process of DPLL until there is a "conflict" after unit propagation. If this conflict occurs without any branches, the it declares the formula unsatisfiable. Otherwise, it analyzes the "conflict graph" and learns the cause of the conflict in the form of a "conflict clause" (see Fig. 1). It now backtracks and continues as in ordinary DPLL, treating the learned clause just like initial ones. One expects that such cached causes of conflicts will save computation later in the process when an unsatisfiable branch due to fail for a similar reason is explored.

Different implementations of clause learning algorithms vary in the strategy they use to choose the clause to learn from a given conflict [15]. For instance, `grasp` [12] uses the *Decision scheme* among others, `zChaff` [14] uses the *1UIP* scheme, and we proposed in [1] a new learning scheme called *FirstNewCut*. The results in this paper are for the 1UIP scheme, but can be obtained for certain other schemes as well, including FirstNewCut.

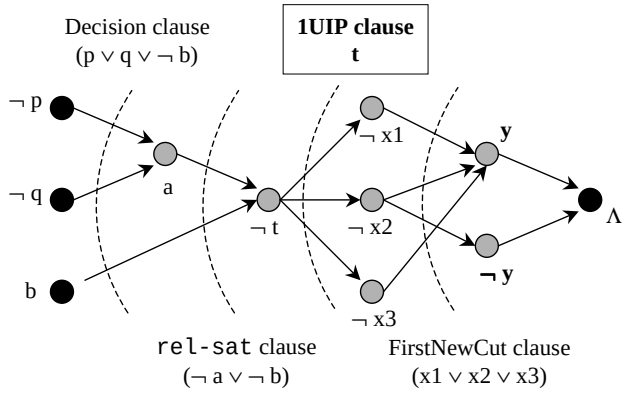

Fig. 1. A conflict graph with various conflict clauses that can potentially be learned

2.1 Branching Sequence

The notion of branching sequence was used in [1] to prove exponential separation between DPLL and clause learning. It generalizes the idea of a static *variable order* by letting it differ from branch to branch in the underlying DPLL procedure. In addition, it also specifies which branch (TRUE or FALSE) to explore first. This can clearly be useful for satisfiable formulas, and can also help on unsatisfiable ones by making the algorithm learn useful clauses earlier in the process.

Definition 1 ([1]). *A* branching sequence *for a CNF formula F is a sequence $\sigma = (l_1, l_2, \ldots, l_k)$ of literals of F, possibly with repetitions. A DPLL based algorithm $\mathcal{A}$ on F* branches according to *σ if it always picks the next variable v to branch on in the literal order given by σ, skips it if v is currently assigned a value, and otherwise branches further by setting the chosen literal to* FALSE *and deleting it from σ. When σ becomes empty, $\mathcal{A}$ reverts back to its default branching scheme.*

Definition 2. *A branching sequence σ is* complete *for F under an algorithm $\mathcal{A}$ if $\mathcal{A}$ branching according to σ terminates before or as soon as σ becomes empty.*

Clearly, how well a branching sequence works for a formula depends on the specifics of the clause learning algorithm used, such as its learning scheme and backtracking process. One needs to keep these in mind when generating the sequence. It is also important to note that while the size of a variable order is always the same as the number of variables in the formula, that of an effective branching sequence is typically much more. In fact, the size of a branching sequence complete for an unsatisfiable formula F is equal to the size of an unsatisfiability proof of F, and when F is satisfiable, it is proportional to the time needed to find a satisfying assignment.

2.2 Pebbling Formulas

Pebbling formulas are unsatisfiable CNF formulas whose variations have been used repeatedly in proof complexity to obtain theoretical separation results between different

proof systems [31–34]. The version we will use in this paper is known to be easy for regular resolution but hard for tree-like resolution (and hence for DPLL without learning) [33].

A *Pebbling formula* pbl_G is an unsatisfiable CNF formula associated with a directed, acyclic *pebbling graph* G (see Fig. 2). Nodes of G are labeled with disjunctions of variables. A node labeled $(x_1 \vee x_2)$ with, say, three predecessors labeled $(p_1 \vee p_2 \vee p_3)$, q_1 and $(r_1 \vee r_2)$ generates six clauses $(\neg p_i \vee \neg q_j \vee \neg r_k \vee x_1 \vee x_2)$, where $i \in \{1,2,3\}, j \in \{1\}$ and $k \in \{1,2\}$. Intuitively, a node labeled $(x_1 \vee x_2)$ is thought of as *pebbled* under a (partial) variable assignment σ if $(x_1 \vee x_2)$ = TRUE under σ. The clauses mentioned above state that if all predecessors of a node are pebbled, then the node itself must also be pebbled. For every indegree zero *source node* of G labeled $(s_1 \vee s_2)$, pbl_G has a clause $(s_1 \vee s_2)$, stating that all source nodes are pebbled. For every outdegree zero *target node* of G labeled $(t_1 \vee t_2)$, pbl_G has clauses $\neg t_1$ and $\neg t_2$, saying that target nodes are not pebbled, and thus providing a contradiction.

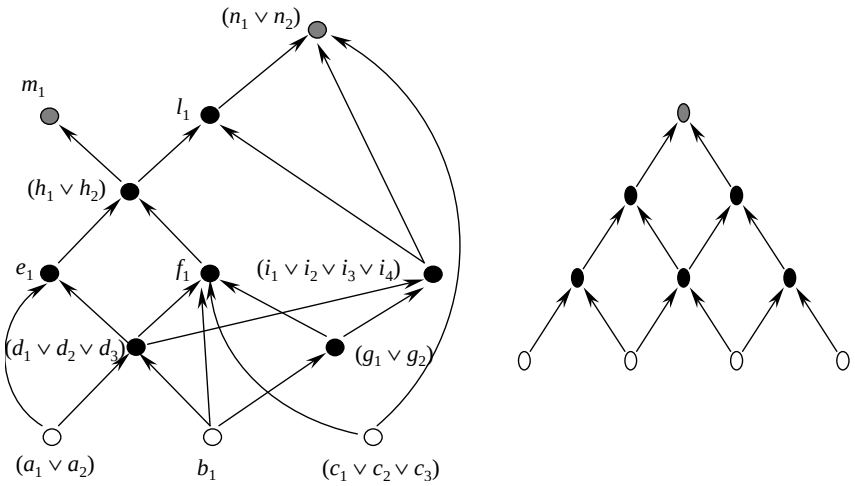

Fig. 2. A general pebbling graph with distinct node labels, and a grid pebbling graph with 4 layers

Grid pebbling formulas are based on simple pyramid shaped layered pebbling graphs with distinct variable labels, 2 predecessors per node, and disjunctions of size 2 (see Fig. 2). *Randomized pebbling formulas* are more complicated and correspond to random pebbling graphs. In general, they allow multiple target nodes. However, the more the targets, the easier it is to produce a contradiction because we can focus only on the (relatively smaller) subgraph under the lowest target. Hence, for our experiments, we add a simple grid structure at the top of randomly generated pebbling formulas to make them have exactly one target.

All pebbling formulas with a single target are minimally unsatisfiable, *i.e.* any strict subset of their clauses admits a satisfying assignment. For each formula Pbl_G we use for our experiments, we also use a satisfiable version of it, called Pbl_G^{SAT}, obtained by randomly choosing a clause of Pbl_G and deleting it. When G is viewed as a task graph,

Pbl_G^{SAT} corresponds to a single fault task system, and finding a satisfying assignment for it corresponds to locating the fault.

3 Branching Sequence for Pebbling Formulas

In this section, we will give an efficient algorithm to generate an effective branching sequence for pebbling formulas. This algorithm will take as input the underlying pebbling graph (which is the high level description of the pebbling problem), and not the pebbling formula itself. As we will see in Section 4, the generated branching sequence gives exponential empirical speedup over `zChaff` for both grid and randomized pebbling formulas.

`zChaff`, despite being one of the current best clause learners, by default does not perform very well on seemingly simple pebbling formulas, even on the uniform grid version. Although clause learning should ideally need only polynomial (in fact, linear in the size of the formula) time to solve these problem instances, choosing a good branching order is critical for this to happen. Since nodes are intuitively pebbled in a bottom up fashion, we must also learn the right clauses (*i.e.* clauses labeling the nodes) in a bottom up order. However, branching on variables labeling lower nodes before those labeling higher ones prevents any DPLL based learning algorithm to backtrack the right distance and proceed further. To make this clear, consider the general pebbling graph of Fig. 2. Suppose we branch on and set d_1, d_2, d_3 and a_1 to FALSE. This will lead to a contradiction through unit propagation by implying a_2 is TRUE and b_1 is FALSE. We will learn $(d_1 \vee d_2 \vee d_3 \vee \neg a_2)$ as the associated 1UIP conflict clause and backtrack. There will still be a contradiction without any further branches, making us learn $(d_1 \vee d_2 \vee d_3)$ and backtrack. At this stage, we have learned the correct clause but are *stuck* with the two branches on d_1 and d_2. Unless we already branched on e_1, there is no way we can now learn it as a clause corresponding to the next higher node.

3.1 Sequence Generation: `GenSeq1UIP`

Algorithm 1, `GenSeq1UIP`, describes a way of generating a good branching sequence for pebbling formulas. It works on any pebbling graph G with distinct label variables as input and produces a branching sequence linear in the size of the associated pebbling formula. In particular, the sequence size is linear in the number of variables as well when the indegree as well as label size are bounded by a constant.

`GenSeq1UIP` starts off by handling the set U of all nodes labeled with unit clauses. Their outgoing edges are deleted and they are treated as pseudo sources. The procedure first generates branching sequence for non-target nodes in U in increasing order of height. The key here is that when `zChaff` learns a unit clause, it fast-backtracks to decision level zero, effectively restarting at that point. We make use of this fact to learn these unit clauses in a bottom up fashion, unlike the rest of the process which proceeds top down in a depth-first way.

`GenSeq1UIP` now adds branching sequences for the targets. Note that for an unsatisfiability proof, we only need the sequence corresponding to the first (lowest) target. However, we process all targets so that this same sequence can also be used when

```
Input  : Pebbling graph G with no repeated labels
Output : Branching sequence for G for the 1UIP learning scheme
begin
    foreach v in BottomUpTraversal(G) do
        v.height ← 1 + max_{u∈v.preds}{u.height}
        Sort(v.preds, increasing order by height)

    // First handle unit clause labeled nodes and generate their sequence
    U ← {v ∈ G.nodes : |v.labels| = 1}
    G.edges ← G.edges \ {(u, v) ∈ G.edges : u ∈ U}
    Add to G.sources any new nodes with now 0 preds
    Sort(U, increasing order by height)
    foreach u ∈ U \ G.targets do
        Output u.label
        GenSubseq1UIPWrapper(u)

    // Now add branching sequence for targets by increasing height
    Sort(G.targets, increasing order by height)
    foreach t ∈ G.targets do
        GenSubseq1UIPWrapper(t)
end
GenSubseq1UIPWrapper(node v) begin
    if |v.preds| > 0 then
        GenSubseq1UIP(v, |v.preds|)
end
GenSubseq1UIP(node v, int i) begin
    u ← v.preds[i]

    // If this is the lowest predecessor ...
    if i = 1 then
        if !u.visited and u ∉ G.sources then
            u.visited ← TRUE
            GenSubseq1UIPWrapper(u)
        return

    // If this is not the lowest one ...
    Output u.labels \ {u.lastLabel}
    if !u.visitedAsHigh and u ∉ G.sources then
        u.visitedAsHigh ← TRUE
        Output u.lastLabel
        if !u.visited then
            u.visited ← TRUE
            GenSubseq1UIPWrapper(u)
    GenSubseq1UIP(v, i − 1)

    for j ← (|u.labels| − 2) downto 1 do
        Output u.labels[1], ..., u.labels[j]
        GenSubseq1UIP(v, i − 1)
    GenSubseq1UIP(v, i − 1)
end
```

Algorithm 1: GenSeq1UIP

the formula is made satisfiable by deleting enough clauses. The subroutine `GenSubseq1UIP` runs on a node v, looking at its i^{th} predecessor u in increasing order by height. No labels are output if u is the lowest predecessor; the negations of these variables will be indirectly implied during clause learning. However, it is recursed upon if not already visited. This recursive sequence results in learning something close to the clause labeling this lowest node, but not quite that exact clause. If u is a higher predecessor (it will be marked as $visitedAsHigh$), `GenSubseq1UIP` outputs all but one variables labeling u. If u is not a source and has not already been visited as high, the last label is output as well, and u recursed upon if necessary. This recursive sequence results in learning the clause labeling u. Finally, `GenSubseq1UIP` generates a recursive pattern, calling the subroutine with the next lower predecessor of v. The precise structure of this pattern is dictated by the 1UIP learning scheme and fast backtracking used in `zChaff`. Its size is exponential in the degree of v with label size as the base.

Example To clarify the algorithm, we describe its execution on a small example. Let G be the pebbling graph in Fig. 3. Denote by t the node labeled $(t_1 \vee t_2)$, and likewise for other nodes. Nodes c, d, f and g are at height 1, nodes a and e at height 2, node b at height 3, and node t at height 4. $U = \{a, b\}$. The edges (a, t) and (b, t) originating from these unit clause labeled nodes are removed, and t, with no predecessors anymore, is added to the list of sources. We output the label of the non-target unit nodes in U in increasing order of height, and recurse on each of them in order, *i.e.* we output a_1, setting $B = (a_1)$, call `GenSubseq1UIPWrapper` on a, and then repeat this process for b. This is followed by a recursive call to `GenSubseq1UIPWrapper` on the target node t.

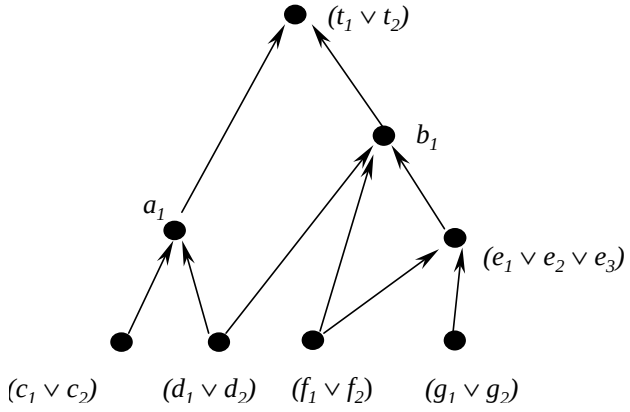

Fig. 3. A simple pebbling graph to illustrate branch sequence generation

The call `GenSubseq1UIPWrapper` on a in turn invokes `GenSubseq1UIP` with parameters $(a, 2)$. This sorts the predecessors of a in increasing order of height to, say, d, c, with d being the lowest predecessor. v is set to a and u is set to the second predecessor c. We output all but the last label of u, *i.e.* of c, making the current branching sequence $B = (a_1, c_1)$. Since u is a source, nothing more needs to be done

for it and we make a recursive call to `GenSubseq1UIP` with parameters $(a, 1)$. This sets u to d, which is the lowest predecessor and requires nothing to be done because it is also a source. This finishes the sequence generation for a, ending at $B = (a_1, c_1)$. After processing this part of the sequence, `zChaff` will have a as a learned clause.

We now output b_1, the label of the unit clause b. The call, `GenSubseq1UIPWrapper` on b, proceeds similarly, setting predecessor order as (d, f, e), with d as the lowest predecessor. Procedure `GenSubseq1UIP` is called first with parameters $(b, 3)$, setting u to e. This adds all but the last label of e to the branching sequence, making it $B = (a_1, c_1, b_1, e_1, e_2)$. Since this is the first time e is being visited as high, its last label is also added, making $B = (a_1, c_1, b_1, e_1, e_2, e_3)$, and it is recursed upon with `GenSubseq1UIPWrapper(e)`. This recursion extends the sequence to $B = (a_1, c_1, b_1, e_1, e_2, e_3, f_1)$. After processing this part of B, `zChaff` will have both a and $(e_1 \vee e_2 \vee e_3)$ as learned clauses. Getting to the second highest predecessor f of b, which happens to be a source, we simply add another f_1 to B. Finally, we get to the third highest predecessor d of b, which happens to be the lowest as well as a source, thus requiring nothing to be done. Coming out of the recursion, back to $u = f$, we generate the pattern given by the last `for` loop, which is empty because the label size of f is only 2. Coming out once more of the recursion to $u = e$, the `for` loop pattern generates e_1, f_1 and is followed by a call to `GenSubseq1UIP` with the next lower predecessor f as the second parameter, which generates f_1. This makes the current sequence $B = (a_1, c_1, b_1, e_1, e_2, e_3, f_1, f_1, e_1, f_1, f_1)$. After processing this, `zChaff` will also have b as a learned clause.

The final call to `GenSubseq1UIPWrapper` with parameter t doesn't do anything because both predecessors of t were removed in the beginning. Since both a and b have been learned, `zChaff` will have an immediate contradiction at decision level zero. This gives us the complete branching sequence $B = (a_1, c_1, b_1, e_1, e_2, e_3, f_1, f_1, e_1, f_1, f_1)$ for the pebbling formula Pbl_G.

3.2 Complexity Analysis

Let graph G have n nodes, indegree of non-source nodes between d_{min} and d_{max}, and label size between l_{min} and l_{max}. For simplicity of analysis, we will assume that $l_{min} = l_{max} = l$ and $d_{min} = d_{max} = d$ ($l = d = 2$ for a grid graph).

Let us first compute the size of the pebbling formula associated with G. The running time of `GenSeq1UIP` and the size of the branching sequence generated will be given in terms of this size. The number of clauses in the pebbling formula Pbl_G is nl^d, ignoring a slight counting error for the source and target nodes. Taking clause sizes into account, the size of the formula, $|Pbl_G|$, is $n(l + d)l^d$. Note that size of the CNF formula itself grows exponentially with the indegree and gets worse as label size increases. The best case is when G is the grid graph, where $|Pbl_G| = \Theta(n)$. This explains the degradation in performance of `zChaff`, both original and modified, as we move from grid graphs to random graphs (see section 4). Since we construct Pbl_G^{SAT} by deleting exactly one randomly chosen clause from Pbl_G (assuming G has only one target node), the size $|Pbl_G^{SAT}|$ of the satisfiable version is also essentially the same.

Let us now compute the running time of `GenSeq1UIP`. Initial computation of heights and predecessor sorting takes time $\Theta(nd \log d)$. Assuming n_u unit clause la-

beled nodes and n_t target nodes, the remaining node sorting time is $\Theta(n_u \log n_u + n_t \log n_t)$. Since `GenSubseq1UIPWrapper` is called at most once for each node, the total running time of `GenSeq1UIP` is $\Theta(nd \log d + n_u \log n_u + n_t \log n_t + nT_{wrapper})$, where $T_{wrapper}$ denotes the running time of `GenSubseq1UIPWrapper` without including recursive calls to itself. When n_u and n_t are much smaller than n, which we will assume as the typical case, this simplifies to $\Theta(nd \log d + nT_{wrapper})$. Let $T(v, i)$ denote the running time of `GenSubseq1UIP(v,i)`, again without including recursive calls to the wrapper method. Then $T_{wrapper} = T(v, d)$. However, $T(v, d) = lT(v, d-1) + \Theta(l)$, which gives $T_{wrapper} = T(v, d) = \Theta(l^{d+1})$. Substituting this back, we get that the running time of `GenSeq1UIP` is $\Theta(nl^{d+1})$, which is about the same as $|Pbl_G|$.

Finally, we consider the size of the branching sequence generated. Note that for each node, most of its contribution to the sequence is from the recursive pattern generated near the end of `GenSubseq1UIP`. Let us denote that by $Q(v, i)$. Then $Q(v, i) = (l-2)(Q(v, i-1) + \Theta(l))$, which gives $Q(v, i) = \Theta(l^{d+2})$. Hence, the size of the sequence generated is $\Theta(nl^{d+2})$, which again is about the same as $|Pbl_G|$.

Theorem 1. *Given a pebbling graph G with label size at most l and indegree of non-source nodes at most d, algorithm* `GenSeq1UIP` *produces a branching sequence σ of size at most S in time $\Theta(dS)$, where $S = |Pbl_G| \approx |Pbl_G^{SAT}|$. Moreover, the sequence σ is complete for Pbl_G as well as for Pbl_G^{SAT} under any clause learning algorithm using fast backtracking and 1UIP learning scheme (such as* `zChaff`*).*

Proof. The size and running time bounds follow from the previous discussion in this section. That this sequence is complete can be verified by a simple hand calculation simulating clause learning with fast backtracking and 1UIP learning scheme.

4 Experimental Results

We conducted experiments on a Linux machine with a 1600 MHz AMD Athelon processor, 256 KB cache and 1024MB RAM. Time limit was set to 1 day and memory limit to 512MB; the program was set to abort as soon as either of these was exceeded. We took the base code of `zChaff` [14] and modified it to incorporate branching sequence given as part of the input, along with a CNF formula. When a branching sequence is specified but gets exhausted before a satisfying assignment is found or the formula is proved to be unsatisfiable, the code reverts to the default variable selection scheme of `zChaff`. We analyzed the performance with random restarts turned off. For all other parameters, we used the default values of `zChaff`.

Table 1 shows the performance on grid pebbling formulas. Results are reported for `zChaff` with no learning or specified branching sequence (DPLL), with specified branching sequence only, with clause learning only (original `zChaff`), and both. Table 2 shows similar results for randomized pebbling formulas. In both cases, the branching sequence used was generated according to Algorithm 1, `GenSeq1UIP`. Note that randomized pebbling graphs typically have a more complex structure. In addition, higher indegree and larger disjunction labels make both the CNF formula size as well

Table 1. `zChaff` on grid pebbling formulas. Note that problem size substantially increases as we move down the table. ‡ denotes out of memory

	Grid formula		*Runtime in seconds*	
Solver	*Layers*	*Variables*	*Unsatisfiable*	*Satisfiable*
DPLL	5	30	0.24	0.12
	6	42	110	0.02
	7	56	$>$ 24 hrs	0.07
	8	72	$>$ 24 hrs	$>$ 24 hrs
Branching	5	30	0.20	0.00
sequence	6	42	105	0.00
only	7	56	$>$ 24 hrs	0.00
	9	90	$>$ 24 hrs	$>$ 24 hrs
Clause	20	420	0.12	0.05
learning	40	1,640	59	36
only (original	65	4,290	‡	47
`zChaff`)	70	4,970	‡	‡
Clause	100	10,100	0.59	0.62
learning +	500	250,500	254	288
branching	1,000	1,001,000	4,251	5,335
sequence	1,500	2,551,500	21,097	‡

Table 2. `zChaff` on random pebbling formulas with distinct labels, indegree ≤ 5, and disjunction label size ≤ 6. Note that problem size increases as we move down the table. ‡ denotes out of memory

	Random pebbling formula			*Runtime in seconds*	
Solver	*Nodes*	*Variables*	*Clauses*	*Unsatisfiable*	*Satisfiable*
	9	33	300	0.00	0.00
DPLL	10	29	228	0.58	0.00
	10	48	604	$>$ 24 hrs	$>$ 24 hrs
Branching	10	29	228	0.09	0.00
sequence	10	48	604	115	0.02
only	12	42	2,835	$>$ 24 hrs	8.88
	12	43	1,899	$>$ 24 hrs	$>$ 24 hrs
Clause	50	154	3,266	0.91	0.03
learning	87	296	9,850	‡	65
only (original	109	354	11,106	584	0.78
`zChaff`)	110	354	18,467	‡	‡
Clause	110	354	18,467	0.28	0.29
learning +	4,427	14,374	530,224	48	49
branching	7,792	25,105	944,846	181	‡
sequence	13,324	43,254	1,730,952	669	‡

as the required branching sequence larger. This explains the difference between the performance of `zChaff`, both original and modified, on grid and randomized pebbling instances. For all instances considered, the time taken to generate the branching sequence from an input graph was substantially less than that for generating the pebbling formula itself.

5 Discussion and Future Work

This paper has developed the idea of using a high level description of a satisfiability problem for generating auxiliary information that can guide a SAT algorithm trying to solve it. Our preliminary experimental results show a clear exponential improvement in performance when such information is used to solve both grid and randomized pebbling problems. Although somewhat artificial, these problems are interesting in their own right and provide hard instances for some of the best existing SAT solvers like `zChaff`. This bolsters our belief that high level structure can be recovered and exploited to make clause learning more efficient.

Extending our results to more practical problems such as generalized STRIPS planning [35] is an obvious direction for future work. These problems induce a natural layered graph structure similar to but more complicated than pebbling graphs. A similar layered structure is seen in bounded model checking problems [36]. We hope that some of the ideas mentioned in this paper will help relate pebbling with planning and bounded model checking, and allow one to use our solution for the former to create an effective strategy for the latter. On another front, there has been a lot of work on generating variable orders for BDD (Binary Decision Diagram) based algorithms (see *e.g.* [37, 38]), where using a good order is perhaps even more critical. Some of the ideas there extend to the BED (Boolean Expression Diagram) model [39] which combines BDDs with propositional satisfiability for model checking. There has also been work on using BDD variable orders for DPLL algorithms without learning [40]. It would be interesting to see if any of these variable ordering strategies provide new ways of capturing structural information in our context.

The form in which we extract and use problem structure is a branching sequence. Although capable of capturing more information than a static variable order, branching sequences suffer from a natural drawback. The exactness they seem to require for pebbling formulas might pose problems when we try to generate branching sequences for harder problems where we know that a polynomial size sequence is unlikely to exist. The usefulness of an incomplete or approximately perfect branching sequence is still unclear. It is not unlikely that we get substantial (but not exponential) improvement as long as the approximate sequence makes correct decisions most of the time, especially near the top of the underlying DPLL tree. However, this needs to be tested experimentally.

Finally, the entire approach of generating auxiliary information by analyzing the problem domain has the inherent disadvantage of requiring the knowledge of higher level problem description. This makes it different from blackbox approaches that try to extract structure after the problem has been translated into a CNF formula. This precluded us, for example, from testing the approach on most of the standard CNF bench-

marks for which such a description is not available. However, given that the source of non-random formulas encoding real-world problems is always a high level description, this, we believe, is not a real drawback.

References

1. Beame, P., Kautz, H., Sabharwal, A.: Understanding the power of clause learning. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence, Acapulco, Mexico (2003) 1194–1201
2. Velev, M., Bryant, R.: Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In: Proceedings of the 38th Design Automation Conference. (2001) 226–231
3. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic model checking using SAT procedures instead of BDDs. In: Proceedings of the 36th Design Automation Conference, New Orleans, LA (1999) 317–320
4. Zhang, H., Hsiang, J.: Solving open quasigroup problems by propositional reasoning. In: Proceedings of the International Computer Symp., Hsinchu, Taiwan (1994)
5. Konuk, H., Larrabee, T.: Explorations of sequential ATPG using boolean satisfiability. In: 11th VLSI Test Symposium. (1993) 85–90
6. Gomes, C.P., Selman, B., McAloon, K., Tretkoff, C.: Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In: Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems, Pittsburgh, PA (1998)
7. Davis, M., Putnam, H.: A computing procedure for quantification theory. Communications of the ACM **7** (1960) 201–215
8. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM **5** (1962) 394–397
9. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: IJCAI (1). (1997) 366–371
10. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: Proceedings, AAAI-98: 15th National Conference on Artificial Intelligence, Madison, WI (1998) 431–437
11. Bayardo Jr., R.J., Schrag, R.C.: Using CST look-back techniques to solve real-world SAT instances. In: Proceedings, AAAI-97: 14th National Conference on Artificial Intelligence. (1997) 203–208
12. Marques-Silva, J.P., Sakallah, K.A.: GRASP – a new search algorithm for satisfiability. In: Proceedings of the International Conference on Computer Aided Design, San Jose, CA, ACM/IEEE (1996) 220–227
13. Zhang, H.: SATO: An efficient propositional prover. In: Proceedings of the 14th International Conference on Automated Deduction. Volume 1249 of Lecture Notes in Computer Science., Townsville, Australia (1997) 272–275
14. 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, Las Vegas, NV, ACM/IEEE (2001) 530–535
15. Zhang, L., Madigan, C.F., Moskewicz, M.H., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: Proceedings of the International Conference on Computer Aided Design, San Jose, CA, ACM/IEEE (2001) 279–285
16. de Kleer, J., Williams, B.C.: Diagnosing multiple faults. Artificial Intelligence **32** (1987) 97–130

17. Stallman, R., Sussman, G.J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artificial Intelligence **9** (1977) 135–196
18. Genesereth, R.: The use of design descriptions in automated diagnosis. Artificial Intelligence **24** (1984) 411–436
19. Davis, R.: Diagnostic reasoning based on structure and behavior. Artificial Intelligence **24** (1984) 347–410
20. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence, Nagoya, Japan (1997) 366–371
21. Mézard, M., Zecchina, R.: Random k-satisfiability problem: From an analytic solution to an efficient algorithm. Physical Review E **66** (2002) 056126
22. Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL – the planning domain definition language. Technical report, Yale University, New Haven, CT (1998)
23. Johnson, D.S., Trick, M.A., eds.: Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge. Volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. AMS (1996)
24. Giunchiglia, E., Maratea, M., Tacchella, A.: Dependent and independent variables in propositional satisfiability. In: Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA). Volume 2424 of Lecture Notes in Computer Science., Cosenza, Italy, Springer-Verlag (2002) 296–307
25. Ostrowski, R., Grégoire, E., Mazure, B., Sais, L.: Recovering and exploiting structural knowledge from cnf formulas. In: 8th Principles and Practice of Constraint Programming. Volume 2470 of Lecture Notes in Computer Science., Ithaca, NY, Springer-Verlag (2002) 185–199
26. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult SAT instances in presence of symmetry. In: Proceedings of the 39th Design Automation Conference, New Orleans, LA (2002) 731–736
27. Brafman, R.I.: A simplifier for propositional formulas with many binary clauses. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence, Seattle, WA (2001) 515–522
28. Amir, E., McIlraith, S.A.: Partition-based logical reasoning. In: Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning, Breckenridge, CO (2000) 389–400
29. Shtrichman, O.: Tuning SAT checkers for bounded model checking. In: Proceedings of the 12th International Conference on Computer Aided Verification, Chicago, IL (2000) 480–494
30. Kautz, H.A., Selman, B.: Pushing the envelope: Planning, propositional logic, and stochastic search. In: Proceedings, AAAI-96: 13th National Conference on Artificial Intelligence, Portland, OR (1996) 1194–1201
31. Bonet, M.L., Esteban, J.L., Galesi, N., Johansen, J.: On the relative complexity of resolution refinements and cutting planes proof systems. SIAM Journal on Computing **30** (2000) 1462–1484
32. Bonet, M.L., Galesi, N.: A study of proof search algorithms for resolution and polynomial calculus. In: Proceedings 40th Annual Symposium on Foundations of Computer Science, New York, NY, IEEE (1999) 422–432
33. Ben-Sasson, E., Impagliazzo, R., Wigderson, A.: Near-optimal separation of treelike and general resolution. Technical Report TR00-005, Electronic Colloquium in Computation Complexity, `http://www.eccc.uni-trier.de/eccc/` (2000)
34. Beame, P., Impagliazzo, R., Pitassi, T., Segerlind, N.: Memoization and DPLL: Formula caching proof systems. In: Proceedings 18th Annual IEEE Conference on Computational Complexity, Aarhus, Denmark (2003) 225–236

35. Kautz, H.A., Selman, B.: Planning as satisfiability. In: Proceedings of the 10th European Conference on Artificial Intelligence, Vienna, Austria (1992) 359–363
36. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Amsterdam, the Netherlands (1999) 193–207
37. Aziz, A., Tasiran, S., Brayton, R.K.: BDD variable orderings for interacting finite state machines. In: Proceedings of the 31th Design Automation Conference, San Diego, CA (1994) 283–288
38. Harlow, J.E., Brglez, F.: Design of experiments in BDD variable ordering: Lessons learned. In: Proceedings of the International Conference on Computer Aided Design, San Jose, CA (1998) 646–652
39. Hulgaard, H., Williams, P.F., Andersen, H.R.: Equivalence checking of combinational circuits using boolean expression diagrams. IEEE Transactions on Computer-Aided Design of Integrated Circuits **18** (1999) 903–917
40. Reda, S., Drechsler, R., Orailoglu, A.: On the relation between SAT and BDDs for equivalence checking. In: Proceedings of the International Symposium on Quality Electronic Design, San Jose, CA (2002) 394–399