Chapter 5

USING PROBLEM STRUCTURE FOR EFFICIENT CLAUSE LEARNING

Given the results about the strengths and limitations of clause learning in Chapter 4, it is natural to ask how the understanding we gain through this kind of analysis may lead to practical improvement in SAT solvers. The theoretical bounds tell us the potential power of clause learning; they don't give us a way of *finding* short solutions when they exist. In order to leverage their strength, clause learning algorithms must follow 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. The present chapter addresses this problem of moving from analytical results to practical improvement. The approach we take is the use of the problem structure for guiding SAT solvers in their branch decisions.

Both random CNF formulas and those encoding various real-world problems are quite hard for current SAT solvers. However, while DPLL based algorithms with lookahead but no learning (such as satz by Li and Anbulagan [80]) and those that try only one carefully chosen assignment without any backtracks (such as SurveyProp by Mézard and Zecchina [87]) 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 their inherent structure, such as dependence graphs in scheduling problems, causes and effects in planning, and algebraic structure in group theory.

Most theoretical and practical problem instances of satisfiability problems originate, not surprisingly, from a higher level description, such as a Planning Domain Description Language (PDDL) specification for planning [51], timed automata or logic description for model checking, task dependency graph for scheduling, circuit description for VLSI, algebraic structure for group theory, and processor specification for hardware. Typically, this description contains more structure of the original problem than is visible in the flat CNF representation in DIMACS format [68] 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.

Several ideas have been brought forward in the last decade for extracting structure after conversion into a CNF formula. These include the works of Giunchiglia et al. [56] and Ostrowski et al. [92] on exploiting variable dependency, Ostrowski et al. [92] on using constraint redundancy, Aloul et al. [6] and others on using symmetry, Brafman

[30] on exploiting binary clauses, and Amir and McIlraith [7] on using partitioning.

While all these approaches extract structure after conversion into a CNF formula, we argue that 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 by Shtrichman [101] and the separate consideration of cause variables and effect variables in planning by Kautz and Selman [71].

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* (recall Definition 4.2) 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 consider two families of formulas called the pebbling formulas and the GT_n formulas. The pebbling formulas, more commonly occurring in theoretical proof complexity literature such as in the works of Ben-Sasson et al. [22] and Beame et al. [15], can be thought of as representing precedence graphs in dependent task systems and scheduling scenarios. They can also be viewed as restricted planning problems. The GT_n formulas were introduced by Krishnamurthy [76] and have also been used frequently to obtain resolution lower bounds such as by Bonet and Galesi [28] and Alekhnovich et al. [3]. They represent a straightforward ordering principle on n elements. Although admitting a polynomial size solution, both pebbling and GT_n formulas are not so easy to solve in practice, as is indicated by our experimental results for unmodified zChaff.

We give an exact sequence generation algorithm for pebbling formulas, using the underlying pebbling graph as the high level description. We also give a much simpler but approximate branching sequence generation algorithm for GT_n formulas, utilizing their underlying ordering structure. Our sequence generators as presented work for the FirstUIP learning scheme (cf. Section 4.2.5), which is one of the best known. They can also be extended to other schemes, including FirstNewCut. Our empirical results are based on our extension of the SAT solver zChaff.

We show that the use of branching sequences produced by our generators leads to exponential empirical speedups for the class of grid and randomized pebbling formulas. We also report significant gains obtained for the class of GT_n formulas.

From a broader perspective, our results for pebbling and GT_n formulas serve 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.

5.1 Two Interesting Families of Formulas

We begin by describing in detail the two families of CNF formulas from the proof complexity literature mentioned above.

5.1.1 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 such as by Ben-Sasson et al. [22] and Beame et al. [15]. The version we will use in this chapter is known to be easy for regular resolution but hard for tree-like resolution [22], and hence for DPLL without learning. We use these formulas to show how one can utilize problem structure to allow clause learning algorithms to handle much bigger problems than they otherwise can.

Pebbling formulas represent the constraints for sequencing a system of tasks that need to be completed, where each task can be accomplished in a number of alternative ways. The associated pebbling graph has a node for each task, labeled by a disjunction of variables representing the different ways of completing the task. Placing a pebble on a node in the graph represents accomplishing the corresponding task. Directed edges between nodes denote task precedence; a node is pebbled when all of its predecessors in the graph are pebbled. The pebbling process is initialized by placing pebbles on all indegree zero nodes. This corresponds to completing those tasks that do not depend on any other.

Formally, a *Pebbling formula* Pbl_G is an unsatisfiable CNF formula associated with a directed, acyclic *pebbling graph* G (see Figure 5.1). Nodes of G are labeled with disjunctions of variables, i.e. with clauses. A node labeled with clause C is thought of as *pebbled* under a (partial) variable assignment σ if $C|_\sigma = \text{TRUE}$. Pbl_G contains three kinds of clauses – precedence clauses, source clauses and target clauses. For instance, a node labeled $(x_1 \vee x_2)$ with three predecessors labeled $(p_1 \vee p_2 \vee p_3)$, q_1 and $(r_1 \vee r_2)$ generates six precedence 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 $k \in \{1,2\}$. The precedence clauses imply that if all predecessors of a node are pebbled, then the node itself must also be pebbled. For every indegree zero *source node* s of G, Pbl_G contains the clause labeling s as a source clause. Thus, Pbl_G implies that all source nodes are pebbled. For every outdegree zero *target node* of G labeled, say, $(t_1 \vee t_2)$, Pbl_G has target clauses $\neg t_1$ and $\neg t_2$. These imply that target nodes are not pebbled, and provide a contradiction.

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 Figure 5.1). *Randomized pebbling formulas* are more complicated and correspond to random pebbling graphs. We only consider pebbling graphs where no variable appears more than once in any node label. In general, random pebbling graphs allow multiple target nodes. However, the more the targets, the easier it is to produce a

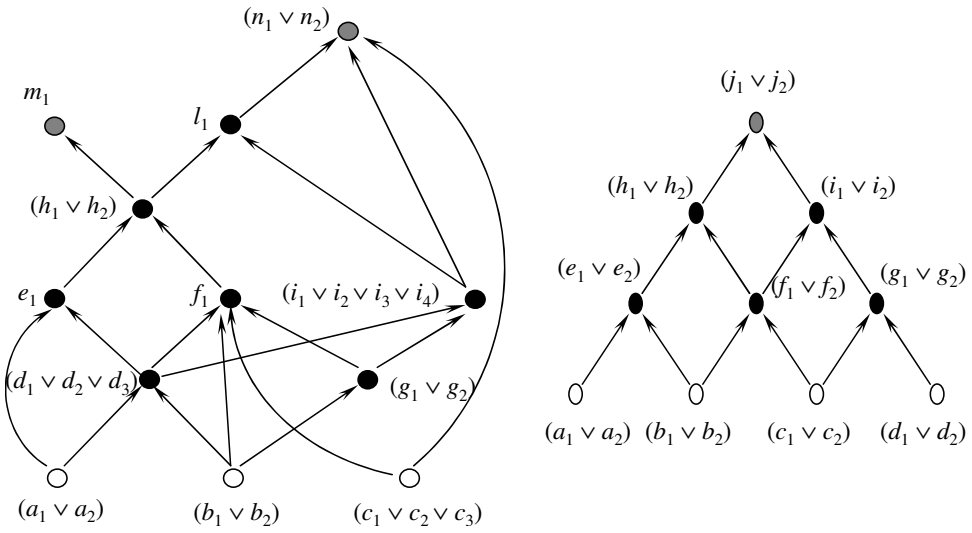

Figure 5.1: A general pebbling graph with distinct node labels, and a 4-layer grid pebbling graph

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 task system with a single fault, and finding a satisfying assignment for it corresponds to locating the fault.

5.1.2 The GT_n Formulas

The GT_n formulas are unsatisfiable CNF formulas based on the ordering principle that any partial order on the set $\{1, 2, \ldots, n\}$ must have a maximal element. They were first considered by Krishnamurthy [76] and later used by Bonet and Galesi [28] to show the optimality of the size-width relationship of resolution proofs. Recently, Alekhnovich et al. [3] used a variation, called GT'_n, to show an exponential separation between RES and regular resolution.

The variables of GT_n are $x_{i,j}$ for $i, j \in [n], i \neq j$, which should be thought of as the binary predicate $i \succ j$. Clauses $(\neg x_{i,j} \vee \neg x_{j,i})$ ensure that $\succ$ is anti-symmetric and $(\neg x_{i,j} \vee \neg x_{j,k} \vee x_{i,k})$ ensure that $\succ$ is transitive. This makes $\succ$ a partial order on $[n]$. *Successor clauses* $(\vee_{k \neq j} x_{k,j})$ provide the contradiction by saying that every element j has a successor in $[n] \setminus \{j\}$, which is clearly false for the maximal elements

of $[n]$ under the ordering $\succ$.

These formulas, although capturing a simple mathematical principle, are empirically difficult for many SAT solvers including zChaff. We employ our techniques to improve the performance of zChaff on these formulas. We use for our experiments the unsatisfiable version GT_n described above as well as a satisfiable version GT_n^{SAT} obtained by deleting a randomly chosen successor clause. The reason we consider these ordering formulas in addition to seemingly harder pebbling formulas is that the latter admit short tree-like proofs in certain extensions of RES whereas the former seem to critically require reuse of derived or learned clauses for short refutations. We elaborate on this in Section 5.2.2.

5.2 From Analysis to Practice

The complexity bounds established in the previous chapter indicate that clause learning is potentially quite powerful, especially when compared to ordinary DPLL. However, natural choices such as which conflict graph to choose, which cut in it to consider, in what order to branch on variables, and when to restart, make the process highly nondeterministic. These choices must be made deterministically (or randomly) when implementing a clause learning algorithm. To harness its full potential on a given problem domain, one must, in particular, implement a learning scheme and a branch decision process suited to that domain.

5.2.1 Solving Pebbling Formulas

As a first step toward our grand goal of translating theoretical understanding into effective implementations, we show, using pebbling problems as a concrete example, how one can utilize high level problem descriptions to generate effective branching strategies for clause learning algorithms. Specifically, we use insights from our theoretical analysis to give an efficient algorithm to generate an effective branching sequence for unsatisfiable as well as satisfiable pebbling formulas (see Section 5.1.1). This algorithm takes as input the underlying pebbling graph (which is the high level description of the pebbling problem), and not the CNF pebbling formula itself. As we will see in Section 5.2.3, 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 time to solve these problem instances (in fact, linear time in the size of the formula), 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 from

backtracking the right distance and proceeding further in an effective manner. To make this clear, consider the general pebbling graph of Figure 5.1. 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 and b_2 are both 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 will have learned the correct clause but will be *stuck* with two branches on d_1 and d_2. Unless we had branched on e_1 before branching on the variables of node d, we will not be able to learn e_1 as the clause corresponding to the next higher pebbling node.

Automatic Sequence Generation: `PebSeq1UIP`

Algorithm 5.1, `PebSeq1UIP`, 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.

`PebSeq1UIP` 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 a 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.

```
Input  : Pebbling graph G with no repeated labels
Output : Branching sequence for Pbl_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
        PebSubseq1UIPWrapper(u)
    // now add branching sequence for targets by increasing height
    Sort(G.targets, increasing order by height)
    foreach t ∈ G.targets do PebSubseq1UIPWrapper(t)
end
PebSubseq1UIPWrapper(node v) begin
    if |v.preds| > 0 then PebSubseq1UIP(v, |v.preds|)
end
PebSubseq1UIP(node v, int i) begin
    u ← v.preds[i]
    if i = 1 then
        // this is the lowest predecessor
        if !u.visited and u ∉ G.sources then
            u.visited ← TRUE
            PebSubseq1UIPWrapper(u)
        return
    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
            PebSubseq1UIPWrapper(u)
    PebSubseq1UIP(v, i − 1)
    for j ← (|u.labels| − 2) downto 1 do
        Output u.labels[1], ..., u.labels[j]
        PebSubseq1UIP(v, i − 1)
    PebSubseq1UIP(v, i − 1)
end
```

Algorithm 5.1: `PebSeq1UIP`, generating branching sequence for pebbling formulas

`PebSeq1UIP` 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 the

formula is made satisfiable by deleting enough clauses. The subroutine `PebSubseq1UIP` 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 previously 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 $visitedAsHigh$), `PebSubseq1UIP` outputs all but one variables labeling u. If u is not a source and has not previously 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, `PebSubseq1UIP` 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.

The Grid Case. It is insightful to look at the simplified version of the sequence generation algorithm that works only for grid pebbling formulas. This is described below as Algorithm 5.2, `GridPebSeq1UIP`. Note that both predecessors of any node are at the same level for grid pebbling graphs and need not be sorted by height. There are no nodes labeled with unit clauses and there is exactly one target node t, simplifying the whole algorithm to a single call to `PebSubseq1UIP(t,2)` in the notation of Algorithm 5.1. The last *for* loop of this procedure and the recursive call that follows it are now redundant. We combine the original wrapper method and the calls to `PebSubseq1UIP` with parameters $(v, 2)$ and $(v, 1)$ into a single method `GridPebSubseq1UIP` with parameter v.

The resulting branching sequence can actually be generated by a simple depth first traversal (DFS) of the grid pebbling graph, printing no labels for the nodes on the rightmost path (including the target node), both labels for internal nodes, and one arbitrarily chosen label for source nodes. However, this resemblance to DFS is a somewhat misleading coincidence. The resulting sequence diverges substantially from DFS order as soon as label size or indegree of some nodes is changed. For the 10 node depth 4 grid pebbling graph shown in Figure 5.1, the branching sequence generated by the algorithm is $h_1, h_2, e_1, e_2, a_1, b_1, f_1, f_2, c_1$. Here, for instance, b_1 is generated after a_1 not because it labels the right (second) predecessor of node e but because it labels the left (first) predecessor of node f. Similarly, f_1 and f_2 appear after the subtree rooted at h as left predecessors of node i rather than as right predecessors of node h.

Example 5.1. To clarify the algorithm for the general case, we describe its execution on a small example. Let G be the pebbling graph in Figure 5.2. 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

```
Input  : Grid pebbling graph G with target node t
Output : Branching sequence for Pbl_G for the 1UIP learning scheme
begin
  GridPebSubseq1UIP(t)
end

GridPebSubseq1UIP(node v) begin
  if v ∈ G.sources then return
  u ← v.preds.left
  Output u.firstLabel
  if !u.visitedAsLeft and u ∉ G.sources then
    u.visitedAsLeft ← TRUE
    Output u.secondLabel
    if !u.visited then
      u.visited ← TRUE
      GridPebSubseq1UIP(u)
  u ← v.preds.right
  if !u.visited and u ∉ G.sources then
    u.visited ← TRUE
    GridPebSubseq1UIP(u)
end
```

Algorithm 5.2: `GridPebSeq1UIP`, generating branching sequence for grid pebbling formulas

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 `PebSubseq1UIPWrapper` on a, and then repeat this process for b. This is followed by a recursive call to `PebSubseq1UIPWrapper` on the target node t.

The call `PebSubseq1UIPWrapper` on a in turn invokes `PebSubseq1UIP` 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 `PebSubseq1UIP` 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, `PebSubseq1UIPWrapper` on b, proceeds similarly, setting predecessor order as (d, f, e), with d as the lowest predecessor. Procedure `PebSubseq1UIP` 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,

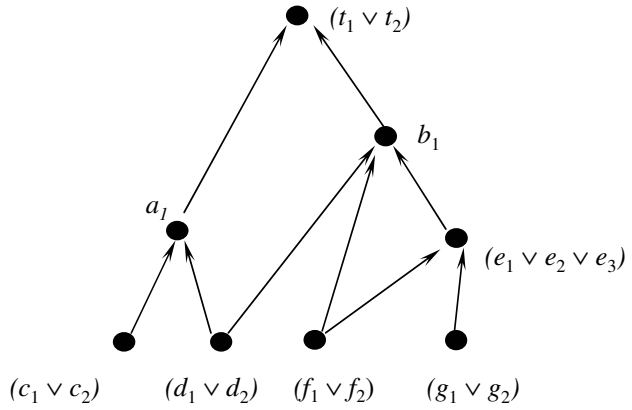

Figure 5.2: A simple pebbling graph to illustrate branch sequence generation

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 `PebSubseq1UIPWrapper(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 `PebSubseq1UIP` 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 `PebSubseq1UIPWrapper` 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.

Complexity of Sequence Generation

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 `PebSeq1UIP` 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 roughly nl^d. Taking clause sizes into account, the size of the formula, $|Pbl_G|$, is roughly $n(l+d)l^d$. Note that the 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 5.2.3). Since we construct Pbl_G^{SAT} by deleting exactly one randomly chosen clause from Pbl_G (see Section 5.1.1), the size $|Pbl_G^{SAT}|$ of the satisfiable version is also essentially the same.

Let us now compute the running time of `PebSeq1UIP`. Initial computation of heights and predecessor sorting takes time $\Theta(nd \log d)$. Assuming n_u unit clause labeled nodes and n_t target nodes, the remaining node sorting time is $\Theta(n_u \log n_u + n_t \log n_t)$. Since `PebSubseq1UIPWrapper` is called at most once for each node, the total running time of `PebSeq1UIP` 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 `PebSubseq1UIP-Wrapper` without taking into account 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 $\Theta(nd \log d + nT_{wrapper})$. If $T(v,i)$ denotes the running time of `PebSubseq1UIP(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 `PebSeq1UIP` 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 `PebSubseq1UIP`. Let $Q(v,i)$ denote this contribution. $Q(v,i) = (l-2)(Q(v,i-1) + \Theta(l))$, 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 5.1. *Given a pebbling graph G with label size at most l and indegree of non-source nodes at most d, Algorithm 5.1,* `PebSeq1UIP`*, 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. □

5.2.2 Solving GT_n Formulas

We now consider automatic sequence generation for the ordering formulas introduced in Section 5.1.2. Since these formulas, like pebbling formulas, also originate in the proof complexity literature and in fact represent a problem that is structurally simpler to state and reason about than the pebbling problem, one may wonder what this section adds to the chapter. The answer lies in two key motivations. First, as we

will see, the automatically generated sequence for these formulas, unlike pebbling formulas, is extremely simple in nature and incomplete as a branching sequence. Nonetheless, it provides dramatic improvement in performance. Second, there is reason to believe that pebbling formulas may be easier than the GT_n formulas for resolution type proof systems. We formalize the intuition behind this in the next few paragraphs.

While pebbling formulas are not so easy to solve by popular SAT solvers, they may not inherently be too difficult for clause learning algorithms. In fact, even without any learning, they admit tree-like proofs under a somewhat stronger related proof system called RES(k) for large enough k as shown by Esteban et al. [50]:

Proposition 5.1 ([50]). *Pbl_G has a tree-like* RES(k) *refutation of size $O(|G|)$, where k is the maximum width of a clause labeling a node of G. In particular, when G is a grid graph with n nodes, Pbl_G has a tree-like* RES(2) *refutation of size $O(n)$.*

Here RES(k) denotes the extension of RES defined by Krajíček [75] that allows resolving, instead of clauses, disjunctions of conjunctions of up to k literals. Recall that clauses are disjunctions of literals, i.e., RES(1) is simply RES. Atserias and Bonet [10] discuss how a tree-like RES(k) proof of a formula F can be converted into a not-too-large tree-like RES proof of a related formula $F(k)$ over a few extra variables. More precisely, their result and Proposition 5.1 together imply that the addition of natural extension variables corresponding to k-conjunctions of variables of Pbl_G leads to a tree-like RES proof of size $O(|G| \cdot k)$ of a related pebbling formulas $Pbl_G(k)$.

For GT_n formulas, however, no such short tree-like proofs are known in RES(k) for any k. Reusing derived clauses (equivalently, learning clauses with DPLL) seems to be the key to finding short proofs of GT_n. This makes them a good candidate for testing clause learning based SAT solvers. Our experiments indicate that GT_n formulas, despite their simplicity, are quite hard for zChaff with its default parameter settings. Using a good branching sequence based on the ordering structure underlying these formulas leads to significant performance gains.

Automatic Sequence Generation: `GTnSeq1UIP`

Since there is exactly one, well defined, unsatisfiable GT formula for a fixed parameter n, it is not surprising that the approximate branching sequence given in Figure 5.3 that we use for it is straightforward. However, the fact that the same branching sequence works well for the satisfiable version of the GT_n formulas, obtained by deleting a randomly chosen successor clause, is worth noting.

Recall that `PebSeq1UIP` was a fairly complex algorithm that generated a perfect branching sequence for randomized pebbling graphs. In contrast, Algorithm 5.3, `GTnSeq1UIP`, for generating the branching sequence in Figure 5.3 is nearly trivial. As remarked earlier, it produces an incomplete sequence (see Definition 4.3) that nonetheless boosts performance in practice.

$$
\begin{array}{cccccc}
- & x_{2,1} & x_{3,1} & x_{4,1} & \dots & x_{n-1,1} \\
x_{1,2} & - & x_{3,2} & x_{4,2} & \dots & x_{n-1,2} \\
x_{1,3} & x_{2,3} & - & x_{4,3} & \dots & x_{n-1,3} \\
x_{1,4} & x_{2,4} & x_{3,4} & - & \dots & x_{n-1,4} \\
\vdots & & & & & \\
x_{1,n} & x_{2,n} & x_{3,n} & x_{4,n} & \dots & - \\
x_{1,n} & x_{2,n} & x_{3,n} & x_{4,n} & \dots & x_{n-1,n}
\end{array}
$$

Figure 5.3: Approximate branching sequence for GT_n formulas. The sequence goes top-down, and left to right within each row. '$-$' corresponds to a non-existent variable $x_{i,i}$.

Input : A natural number n
Output : Branching sequence for GT_n for the 1UIP learning scheme
begin
  for $i = 1$ *to* n **do**
    for $j = 1$ *to* $(n-1)$ **do**
      if $i \neq j$ **then** Output $x_{j,i}$
end

Algorithm 5.3: `GTnSeq1UIP`, generating branching sequence for GT_n formulas

5.2.3 Experimental Results

We conducted experiments on a Linux machine with a 1600 MHz AMD Athelon processor, 256 KB cache and 1024 MB RAM. Time limit was set to 6 hours and memory limit to 512 MB; the program was set to abort as soon as either of these was exceeded. We took the base code of zChaff [88], version 2001.6.15, and modified it to incorporate a branching sequence given as part of the input, along with a CNF formula. When an incomplete branching sequence is specified that 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 VSIDS of zChaff (cf. Section 2.3.2).

For consistency, we analyzed the performance with random restarts turned off. For all other parameters, we used the default values of zChaff. For all formulas, results are reported for DPLL (zChaff with clause learning disabled), for CL (unmodified zChaff), and for CL with a specified branching sequence (modified zChaff).

Tables 5.1 and 5.2 show the performance on grid pebbling and randomized pebbling formulas, respectively, using the branching sequence generated by Algorithm 5.1, `PebSeq1UIP`. Table 5.3 shows the performance on the GT_n formulas using the branching sequence generated by Algorithm 5.3, `GTnSeq1UIP`.

Table 5.1: zChaff on *grid pebbling* formulas. ‡ denotes out of memory.

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

Table 5.2: zChaff on *randomized pebbling* formulas with distinct labels, indegree ≤ 5, and disjunction label size ≤ 6. ‡ denotes out of memory.

Solver	*Randomized pebbling formula* *Nodes*	*Variables*	*Clauses*	*Runtime in seconds* *Unsatisfiable*	*Satisfiable*
DPLL	9	33	300	0.00	0.00
	10	29	228	0.58	0.00
	10	48	604	> 6 hrs	> 6 hrs
CL (unmodified zChaff)	50	154	3,266	0.91	0.03
	87	296	9,850	‡	65
	109	354	11,106	584	0.78
	110	354	18,467	‡	‡
CL + **branching sequence**	110	354	18,467	0.28	0.29
	4,427	14,374	530,224	48	49
	7,792	25,105	944,846	181	> 6 hrs
	13,324	43,254	1,730,952	669	249

For both grid and randomized pebbling formulas, the size of problems that can be solved increases substantially as we move down the respective tables. Note that randomized pebbling graphs typically have a more complex structure than grid pebbling graphs. In addition, higher indegree and larger disjunction labels make both the CNF formula size as well as the required branching sequence larger. This explains

Table 5.3: zChaff on GT_n formulas. ‡ denotes out of memory.

Solver	GT_n *formula* n	*Variables*	*Clauses*	*Runtime in seconds* *Unsatisfiable*	*Satisfiable*
DPLL	8	62	372	1.05	0.34
	9	79	549	48.2	0.82
	10	98	775	3395	248
	11	119	1,056	$>$ 6 hrs	743
CL (unmodified zChaff)	10	98	775	0.20	0.00
	13	167	1,807	93.7	7.14
	15	223	2,850	1492	0.01
	18	322	5,067	‡	‡
CL + **branching sequence**	18	322	5,067	0.52	0.13
	27	727	17,928	701	0.17
	35	1,223	39,900	3.6	0.15
	45	2,023	86,175	‡	0.81

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 the input graph was significantly less than that for generating the pebbling formula itself.

For the GT_n formulas, since the branching used was incomplete, the solver had to revert back to zChaff's VSIDS heuristic to choose variables to branch on after using the given branching sequence as a guide for the first few decisions. Nevertheless, the sizes of problems that could be handled increased significantly. The satisfiable versions proved to be relatively easier, with or without a specified branching sequence.

5.3 Discussion

This chapter 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 experimental results show a clear exponential improvement in performance when such information is used to solve both grid and randomized pebbling problems, as well as the GT_n ordering 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. Pebbling graphs are structurally similar to the layered graphs induced naturally by problems involving unwinding of state space over time, such as CNF formulations of planning by Kautz and Selman [70] and bounded model checking by Biere et al. [25]. This bolsters our belief that high level structure can be recovered and exploited to

make clause learning more efficient.

In practice, a solver must employ good branching heuristics as well as implement a powerful proof system. Our result that pebbling formulas have short CL proofs depends critically upon the solver choosing a branching sequence that solves the formula in a "bottom-up" fashion, so that the learned clauses have maximal reuse. Nevertheless, we were able to automatically generate such sequences for grid and randomized pebbling formulas. For the GT_n formulas, we used a different approach and instead provided a very simple but imperfect automatically generated branching sequence that boosted performance significantly in practice.

Our approach of exploiting high level problem description to generate auxiliary information for SAT solvers, of course, requires the knowledge of this high level description to begin with. The standard CNF benchmarks such as those in the online collection at SATLIB [63], unfortunately, do not come with such a description and thus do not allow an extended evaluation of our technique on several interesting formulas routinely used by researchers. We regard this not as a drawback of our approach but instead as an easily avoidable limitation of the currently prevalent notion of SAT solvers as blackboxes taking only a pure CNF formula as input. There is no good reason for the high level problem description to be unavailable to generate auxiliary structural information since CNF formulas for practically all interesting problems, from theory and practice, are created from a more abstract specification. We continue to build upon this philosophy in the next chapter.