Trade-offs Between Time and Memory in a Tighter Model of CDCL SAT Solvers

Jan Elffers
KTH Royal Institute of Technology

Jan Johannsen
Ludwig-Maximilians-Universität München

Massimo Lauria
Universitat Politècnica de Catalunya

Thomas Magnard
École Normale Supérieure

Jakob Nordström
KTH Royal Institute of Technology

Marc Vinyals
KTH Royal Institute of Technology

August 23, 2016

Abstract

A long line of research has studied the power of conflict-driven clause learning (CDCL) and how it compares to the resolution proof system in which it searches for proofs. It has been shown that CDCL can polynomially simulate resolution even with an adversarially chosen learning scheme as long as it is asserting. However, the simulation only works under the assumption that no learned clauses are ever forgotten, and the polynomial blow-up is significant. Moreover, the simulation requires very frequent restarts, whereas the power of CDCL with less frequent or entirely without restarts remains poorly understood. With a view towards obtaining results with tighter relations between CDCL and resolution, we introduce a more fine-grained model of CDCL that captures not only time but also memory usage and number of restarts. We show how previously established strong size-space trade-offs for resolution can be transformed into equally strong trade-offs between time and memory usage for CDCL, where the upper bounds hold for CDCL without any restarts using the standard 1UIP clause learning scheme, and the (in some cases tightly matching) lower bounds hold for arbitrarily frequent restarts and arbitrary clause learning schemes.

1 Introduction

For two decades the dominant strategy for solving the Boolean satisfiability problem (SAT) in practice has been *conflict-driven clause learning (CDCL)* [BS97, MS99, MMZ+01]. Although SAT is an NP-complete problem, and is hence widely believed to be intractable in the worst case, CDCL SAT solvers have turned out to be immensely successful over a wide range of application areas. An important problem is to understand how such SAT solvers can be so efficient and what theoretical limits exist on their performance.

1.1 Previous Work

At the core, CDCL searches for proofs in the proof system *resolution* [Bla37]. While pre- and inprocessing techniques can, and sometimes do, go significantly beyond resolution (incorporating, e.g., solving of linear equations $\mod 2$ and reasoning with cardinality constraints), understanding the power of even just the fundamental CDCL search algorithm seems like an interesting and challenging problem in its own right. Three crucial aspects of CDCL solvers, which are the focus of our work, are running time, memory usage, and restart policy.

In resolution, time is modelled by the size/length complexity measure, in that lower bounds on proof size yield lower bounds on the running time of CDCL solvers. Resolution proof size is a well-studied

measure. It is not hard to show that it need never be larger than exponential in the formula size, and such exponential lower bounds were shown already in, e.g., [Hak85, Urq87, CS88].

Another more recently studied measure is *(clause) space*, measured as the number of clauses one needs to keep in memory while verifying the correctness of a resolution proof.[1] We remark that although the study of space was originally motivated by SAT solving concerns, it is not a priori clear to what extent this abstract space measure corresponds to CDCL memory usage. Space need never be more than linear in the worst case [ET01], even though such proofs might have exponential size, and optimal linear lower bounds on space were obtained in [ABRW02, BG03, ET01].

More interesting than such space bounds is perhaps what can be said regarding simultaneous optimization of time and space, which is the setting in which SAT solvers operate. There are strong trade-offs [BN11, BBI12, BNT13] showing that this is not possible in general. What this means is that one can find formulas for which (a) there are short proofs and (b) also space-efficient proofs but (c) no proof can get close to being simultaneously both size- and space-efficient.

Regarding restarts, such a concept does not quite make sense for resolution proofs and so has not been studied in that context as far as we are aware.

It is natural to ask to what extent upper and lower bounds for resolution apply to CDCL. By comparison, it is well understood that the *DPLL* method [DP60, DLL62] searches for proofs in *tree-like resolution*, which incurs an exponential loss in performance as compared to general resolution. There has been a long line of research investigating how CDCL compares to general resolution, e.g., [BKS04, Van05, BHJ08, HBPV08], culminating in the result by [PD11] that CDCL viewed as a proof system polynomially simulates resolution with respect to size/time. The nonconstructive part of this result is that variable decisions are not done according to some concrete heuristic but are provided as helpful advice to the solver. This limitation is probably inherent, since a fully algorithmic result would have unexpected implications in complexity theory [AR08]. It is worth noting, however, that in independent work [AFT11] showed that for resolution proofs where all clauses have constant size, using a *random* variable selection heuristic will yield a constructive polynomial-time simulation.

One strength of the results in [AFT11, PD11] is that they hold regardless of the specific learning scheme used, as long as it is *asserting* (an assumption that anyway lies at the heart of the CDCL algorithm). The results also have a few less desirable aspects, however:

- The simulations require very frequent restarts. Only the first conflict after each restart is useful, and after that one has to wait for the next restart to make any further progress.
- There is also a large polynomial blow-up in the simulations, which means that for practical purposes these simulations are far too inefficient to yield really concrete insights into CDCL performance as compared to resolution.
- Finally, and most seriously, the results crucially rely on the assumption that no learned clause is ever forgotten. This is unrealistic, as typically around 90–95% of learned clauses are erased during CDCL search and this is absolutely essential for performance.

It would be desirable to obtain results relating CDCL and resolution that also take the above aspects into account.

Addressing one of these concerns, a more fine-grained study of the power of CDCL without restarts has been conducted in, e.g., [BHJ08, BBJ14, BK14, BS14]. One problematic aspect here is that the models studied appear to be quite far from actual CDCL behaviour. Some papers assume non-standard and rather artificial preprocessing steps. Others study CDCL models that do not enforce that unit clauses are propagated or that do not trigger conflict analysis as soon as a clause is falsified. In the latter case, as a result one gets very limited restrictions on what the clause learning schemes are, and it is hard even to talk about what "conflict analysis" is supposed to mean in this context. This is not an issue for

[1] We mention for completeness that there is also a *total space* measure counting the number of literals in memory, which has been studied in, e.g., [ABRW02, BGT14, BBG+15, Bon16], but for our purposes clause space seems like a more relevant measure to focus on.

results establishing lower bounds limiting what CDCL can do—here a stronger model of CDCL only makes the results stronger—but for upper bounds the results become too optimistic, indicating that the theoretical CDCL model can do much better than what seems possible in practice. As a case in point, there are currently no known separations between general resolution and CDCL without restarts, but part of the reason for this appears to be that the models of CDCL without restarts are clearly too strong to be realistic.

We are not aware of any work on models measuring not only time but also memory consumption in a proof system formalizing CDCL. As discussed above, one can define a space measure for resolution proofs, but it is not clear what relation, if any, there is between this space measure and the size of the clause database during CDCL execution.

1.2 Our Contributions

In this work, we present a proof system that tightly models running time, memory usage, and restarts in CDCL. The model draws heavily on [AFT11, PD11], combined with ideas from [BHJ08] to capture memory and restarts. Indeed, we do not claim any key new technical insights for this part of our work, but rather it is more a matter of carefully studying previous models and painstakingly putting the pieces together to get as clean and simple a proof system as possible that is nevertheless significantly "closer to the metal" than in previous papers.

Our CDCL proof system enforces unit propagation and triggers conflict analysis directly at a conflict. It can incorporate any asserting learning scheme (as long as it is based on resolution derivations from the current conflict and reason clauses), and this scheme is specified explicitly as a parameter. Right from the definitions one obtains natural measures of time, memory usage, and restarts. Variable decisions are still provided externally, just as in [AFT11, PD11], but in principle one could also plug in, say, the most commonly used *VSIDS (variable state independent decaying sum)* decision scheme with *phase saving* and analyse what proofs can be generated using these heuristics (though this is not the focus of our current work). Since we are now managing the database of learned clauses explicitly, we also have to specify a clause database reduction policy. In this paper, the decisions about which clauses to delete are also provided to the solver, but the model allows to plug in a concrete reduction policy as well.

We argue that the proof system we present faithfully models possible execution traces during CDCL search. Some interesting questions to study in this model are as follows:

1. Do upper and lower bounds on resolution size and space transfer to this CDCL proof system?

2. How does CDCL compare to general resolution if we want efficient simulations with respect to *both* time and space, and in addition aim for at most constant-factor blow-ups rather than arbitrary polynomial blow-ups?

3. What is the power of CDCL without restarts compared to the subsystems of tree-like resolution or so-called *regular resolution*? (Briefly, regularity is the somewhat SAT solver-like restriction on resolution that along each path in the proof any variable is branched over only once.)

The worst-case upper bounds on size and space in resolution carry over to time and memory usage in CDCL, and it turns out that this can in fact be read off from [NOT06], although that paper uses quite a different language (and so we present a self-contained proof in this paper for completeness). More interestingly, we show that there is a straightforward translation from CDCL to resolution that preserves both time and space, and so we obtain that all size and space lower bounds previously established for resolution apply also to CDCL (which, in particular, was not at all obvious for space).

This means that the lower bounds on time-space trade-offs in [BN11, BBI12, BNT13] also hold for CDCL. But this does not yet yield true trade-offs, since for such results we also want *upper bounds*. That is, we want to show that CDCL can find time- or space-efficient proofs optimizing just one of these measures in isolation. It is known how to construct such proofs in resolution, but these proofs are not obviously CDCL-like. Since SAT solving was mentioned as a motivation for [BN11, BBI12, BNT13]

it is a relevant question whether the size-space trade-offs shown in these papers correspond to anything one could expect to see in practice, or whether the size- and space-efficient proofs have such peculiar structure that nothing similar can be found by CDCL proof search.

The main contribution of our work is to address the question of whether true time-space trade-offs can be established for CDCL. Finding an answer turns out to be surprisingly technically challenging, and we are not able to prove the known trade-offs for exactly the same formulas as in [BN11, BBI12, BNT13] However, for many of the formulas it is possible to modify them slightly to obtain CDCL trade-offs with essentially the same parameters. An additional feature of these trade-offs is that all our upper bounds hold for CDCL *without any restarts* using the standard *1UIP (first unique implication point)* learning scheme, while the (often tightly matching) lower bounds hold for arbitrarily frequent restarts and arbitrarily chosen clause learning schemes (even non-asserting ones).

We leave as open problems whether CDCL with 1UIP clause learning and with or without restarts can simulate or be separated from general or regular resolution, respectively. While those problems still look quite challenging, we hope and believe that it should be possible to make progress by investigating them in a model that more closely resembles what happens during CDCL proof search in practice, such as the model presented in this paper.

1.3 Organization of This Paper

In Section 2 we define our model of CDCL and relate it to the resolution proof system, and an overview of our results is then given in Section 3. A more detailed discussion of CDCL worst-case bounds is presented in Section 4. We establish our time-space trade-off results for CDCL in Sections 5 and 6. Finally, Section 7 contains some concluding remarks.

2 Modelling CDCL as a Proof System

We start by describing our model of CDCL and how it is formalized as a proof system. As already mentioned, this is very much inspired by [AFT11, PD11], but with ideas added from [BHJ08]. We want to remark right away that we describe the model at a level of detail that might seem excessive to SAT practitioners familiar with CDCL. We do so precisely because a serious issue with many contributions on the theoretical side has been that they fail to get crucial details of the model right, as discussed in the introduction.[2]

MASSIMO'S COMMENT 1: *I think that the footnote is required. We should also acknowledge and praise the workshop for the correction.*

2.1 Preliminaries

Let us first fix some standard notation and terminology. A *literal* a over a Boolean variable x is either x itself or its negation $\overline{x}$ (a *positive* or *negative* literal, respectively). A *clause* $C = a_1 \vee \cdots \vee a_k$ is a disjunction of literals, where the clause is *unit* if it contains only one literal. A *CNF formula* F is a conjunction of clauses $F = C_1 \wedge \cdots \wedge C_m$. We think of clauses and CNF formulas as sets, so that the order of elements is irrelevant and there are no repetitions.

A *resolution derivation* of C from F is a sequence of clauses $(C_1, C_2, \ldots, C_\tau)$ such that $C_\tau = C$ and every C_i is either a clause in F (an *axiom*) or is derived from clauses C_j, C_k with $j, k < i$, by the *resolution rule*

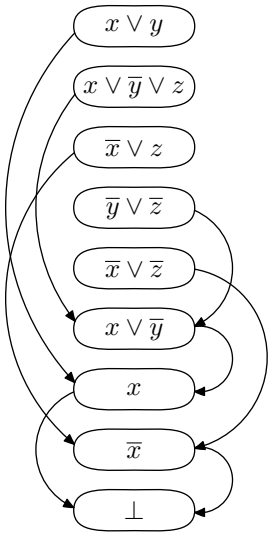
$$\frac{C \vee x \qquad D \vee \overline{x}}{C \vee D} \ , \tag{2.1}$$

where we say that $C \vee x$ and $D \vee \overline{x}$ are *resolved* over x. A derivation is *trivial* if all variables resolved over are distinct and each C_i either is an axiom or is derived from a resolution rule application where one

[2] Indeed there were issues with the model we presented in the conference version of this paper as well. Our description of the behaviour of the solver after a restart was not matching exactly what actual solvers seem to do in practice. Our trade-offs deal with CDCL proofs without restarts, therefore the correctness of the results was not compromised.

1.	$x \vee y$	Axiom
2.	$x \vee \overline{y} \vee z$	Axiom
3.	$\overline{x} \vee z$	Axiom
4.	$\overline{y} \vee \overline{z}$	Axiom
5.	$\overline{x} \vee \overline{z}$	Axiom
6.	$x \vee \overline{y}$	Res$(2, 4)$
7.	x	Res$(1, 6)$
8.	$\overline{x}$	Res$(3, 5)$
9.	$\bot$	Res$(7, 8)$

(a) Annotated list of clauses in refutation.



(b) DAG representation of refutation.

Figure 1: Resolution refutation represented as list of clauses and directed acyclic graph.

of the resolved clauses is an axiom. A *resolution refutation* of, or *resolution proof* for, an unsatisfiable formula F is a derivation of the empty clause $\bot$ (containing no literals) from the axioms in F. We can representent a refutation either as an annotated list of clauses as in Figure 1a or as a directed acyclic graph (DAG) as in Figure 1b. We say that a refutation is *tree-like* if this DAG is a tree (which is the case in Figure 1b).

The *length* or *size* of a proof is the number of clauses in it counted with repetitions. The space of a proof at step t is the number of clauses at steps $\leq t$ that are used in applications of the resolution rule at steps $\geq t$. Looking at the example in Figure 1, the space usage at step 7 is 5 (the clauses in memory at this point are clauses 1, 3, 5, 6, and 7). The space of a proof is obtained by measuring the space at each step and taking the maximum.

2.2 A Formal Description of Conflict-Driven Clause Learning

A CDCL solver running on a formula F decides variable assignments and propagates values that follow from such assignments until a clause is falsified, at which point a learned clause is added to the clause database $\mathcal{D}$ (where we always have $F \subseteq \mathcal{D}$) and the search backtracks. A key concept is the current partial assignment maintained by the solver together with some book-keeping why variables were set this way, which we refer to as the *trail*. This is a sequence $s = (x_1 = b_1/*, x_2 = b_2/*, \ldots, x_\ell = b_\ell/*)$ where all variables are distinct and where $* = \mathsf{d}$ indicates that the assignment is a decision and $* = C$ that it was propagated by the clause C. We write $s_{\leq j}$ and $s_{<j}$ to denote the subsequences that are the prefixes of length j and $j - 1$ of s, respectively. We denote the empty trail by ϵ.

The *decision level* of an assignment $x_j = b_j/*$ is the number of decision assignments in $s_{\leq j}$. The decision level of a (non-empty) trail is that of its last assignment. Identifying a trail s with the partial assignment it defines, we write $C{\restriction_s}$ to denote the clause C *restricted by* s, which is the trivially true clause if s satisfies C and otherwise C with all literals falsified by s removed, and this notation is extended to sets of clauses by taking unions. If a trail s falsifies a clause C, we say that C is *asserting* if it has a unique variable at the maximum decision level of s. If so, the second largest decision level represented in C is the *assertion level* of C.

A trail $s = (x_1 = b_1/*, \ldots, x_\ell = b_\ell/*)$ is *legal* with respect to a formula F and clause database $\mathcal{D} \supseteq F$ if the following holds:

- $\mathcal{D}{\restriction_{s_{<\ell}}}$ does not contain the empty clause;

- if the jth element of s is $x_j = b_j/\mathsf{d}$, then $\mathcal{D}\restriction_{s_{<j}}$ does not contain a unit clause;
- if the jth element of s is $x_j = b_j/C$, then C is contained in $\mathcal{D}$ and has the property that $C\restriction_{s_{<j}}$ is unit and is satisfied by setting $x_j = b_j$.

This captures properties that must hold during CDCL search, and so in what follows trails are implicitly required to be legal unless otherwise specified.

At each point in time, the solver is in a *CDCL state* $(F, \mathcal{D}, s)$, where at the beginning $\mathcal{D} = F$ and $s = \epsilon$. It is convenient to describe the solver as being in one of the four modes **DEFAULT** (where it starts), **UNIT**, **CONFLICT**, or **DECISION**, where transitions are performed as described below (guided by plug-in components that specify the detailed behaviour; also to be discussed in what follows):

DEFAULT If s falsifies a clause in $\mathcal{D}$, the solver moves to **CONFLICT**, otherwise it checks that all variables in F have been assigned, and in that case the solver halts and outputs `SAT` together with the assignment s. Otherwise, if $\mathcal{D}\restriction_s$ contains a unit clause, the solver transits to **UNIT** mode. If none of the previous cases applies, the solver uses its *restart policy* to decide whether to restart, i.e., to set $s = \epsilon$ and to move to **DEFAULT**. At last, if none of the others cases applies, solver uses its *clause database reduction policy* to decide whether to shrink $\mathcal{D}$ to $\mathcal{D}' \subsetneq \mathcal{D}$, where $\mathcal{D}'$ must still contain F and all clauses mentioned in the current trail s, after which it moves to **DECISION**.

CONFLICT If s has decision level 0, the solver outputs `UNSAT`. Otherwise it applies the *learning scheme* to derive an asserting clause C and then *backjumps* by updating the state to $(F, \mathcal{D}\cup\{C\}, s')$ (where s' is the prefix of s that contains all assignments with decision level less than or equal to the assertion level of C), and shifts to **UNIT** mode.

UNIT The solver uses the *unit propagation scheme* to pick a clause C in $\mathcal{D}$ such that $C\restriction_s$ is unit, extends s with the assignment $x = b/C$ that satisfies $C\restriction_s$, and moves to **DEFAULT** mode.

DECISION The solver uses the *decision scheme* to determine an assignment $x = b/\mathsf{d}$ with which to extend the trail and moves to **DEFAULT** mode.

MASSIMO'S COMMENT 2: *The definition of stable state was unsatisfactory (I can say that because I believe wrote it). After we fixed the model, it was at best confusing and at worst incorrect. I just decided to define it more explicitly.*

We say that a CDCL state $(F, \mathcal{D}, s)$ is *stable* if, when solver is in **DEFAULT** mode, it causes neither a conflict, a unit propagation nor to output `SAT`. We say it is a *conflict state* if it causes a move from **DEFAULT** to **CONFLICT**. We remark that CDCL solvers typically apply restarts and database reductions only in the first stable state after a conflict. However, it is not hard to see that from a proof complexity point of view the solver does not get any stronger by allowing these steps to be performed at any stable state, and since this simplifies the description we have done so above.

In order to obtain a concrete CDCL implementation, one needs to instantiate the components referred to above. Let us briefly discuss how this can be done.

For the *clause learning scheme* the assumption is that the clause is derivable in resolution from the clause falsified (the *conflict clause*) and the clauses causing unit propagations (the *reason clauses*) and that the learned clause is always asserting. For our upper bounds we use the 1UIP learning scheme from [ZMMM01], which is simply a trivial resolution derivation from the conflict clause and the reason clauses processed in reverse order up to the first point when there remains only one variable of maximal decision level in the clause.[3]

The *restart policy* determines when the solver should clear the trail and start over from the beginning (but keeping the clause database as it is). From a theoretical point of view adding more frequent restarts can only make the solver more powerful. Hence, in order to obtain the strongest possible result we want to prove our upper bounds on CDCL with a strict no-restarts policy and our lower bounds in a setting with no restrictions on restarts.

[3] In fact, our results hold for any UIP scheme, but for simplicity we focus on 1UIP, which is anyway dominant in practice.

If there is more than one unit clause that can propagate in **UNIT** mode, the *unit propagation scheme* determines in which order the clauses are chosen. Typically this will depend somewhat randomly on low-level implementation details, and therefore we try to prove our upper and lower bounds for the settings when the order chosen is maximally unhelpful and maximally helpful, respectively.

The *decision scheme* is used to choose the next variable to assign when there are no unit propagations. The dominant heuristic in practice is VSIDS [MMZ+01], but for our theoretical analysis we follow [AFT11, PD11] by allowing the decisions to be chosen externally by a helpful oracle and fed to the solver.

The *database reduction policy*, finally, regulates when and how to forget learned clauses. Making this aspect explicit is the main difference between our work and [AFT11, PD11]—the latter papers crucially need the unrealistic assumption that no learned clauses may ever be erased. In principle, here one could plug in, say, the *literal block distance (LBD)* heuristic in [AS09] to decide which clauses to throw away or keep, but in this work we will let this, too, be part of the external input provided to construct a CDCL proof.

2.3 Formalizing CDCL as a Proof System

In order to construct a proof system corresponding to CDCL, we will simply let the proofs be execution traces that contain enough information to allow efficient verification that they are consistent with the detailed description of the CDCL model above. More formally, we say that a *CDCL trace* π is an ordered sequence of the following types of elements:

- decisions $x_i = b/\mathsf{d}$;
- unit propagations $x_i = b/C$ (with reason C);
- learned clauses $\mathsf{add}\, C/\sigma_C$ (with conflict analysis σ_C);
- deletions of clauses $\mathsf{del}\, C$;
- restarts R.

Given a CDCL model with components as above partially or fully specified, a trace π is *legal*, or is a *CDCL proof*, if it is consistent with an execution of the CDCL model as described in Section 2.2.

To verify a trace we start the CDCL solver in state (F, F, ϵ) in **DEFAULT** mode and then traverse the sequence π, updating the state as we go along. If the next element is a decision $x_i = b/\mathsf{d}$, then the solver should be in **DECISION** mode and the assignment should be consistent with the decision scheme, or should at least lead to a legal trail if the decision scheme is not specified. For a unit propagation $x_i = b/C$ the solver should be in **UNIT** mode and $x_i = b$ should be propagated by C under the current trail, and should be the propagation chosen by the propagation scheme if specified. For $\mathsf{add}\, C/\sigma_C$ the solver should be in **CONFLICT** mode and C should be an asserting clause obtained by the resolution derivation σ_C from the conflict and reason clauses in the current trail, where σ_C complies with the requirements of the learning scheme if specified. Deletions of clauses $\mathsf{del}\, C$ and restarts R should only happen in **DEFAULT** mode and should be consistent with their respective policies if provided (where, at a minimum, no clause on the trail may be forgotten).

We say that a CDCL trace is a *CDCL proof of unsatisfiablity* or *CDCL refutation* of F if it is legal and makes the CDCL solver output `UNSAT`, and that it is a *CDCL proof of satisfiablity* if the output is `SAT`. It should be clear that if the components specified are efficiently computable, then CDCL traces are efficiently verifiable and constitute a proof system in the sense of [CR79] (and since all traces we construct will be legal, we will sometimes use the words "trace" and "proof" as synonyms).

The *time* of a CDCL proof π is the number of elements in the sequence plus the sum of the length of all conflict analysis resolution derivations σ_C, i.e., the total number of variable decisions, propagations, and steps in conflict analysis. The *space* of the proof at a given point in time is the number of learned

clauses $|\mathcal{D} \setminus F|$, i.e., the number of statements add C/σ_C minus the number of statements del C up to that point, and the space of a proof is obtained by taking the maximum over all time steps in it.

These measures are intended to capture the execution time and memory usage of a CDCL solver execution described by the trace π, and in addition we want them to translate to length and space bounds for resolution. This is indeed the case, but in order to make a precise, formal statement we first need to discuss clause learning schemes.

2.4 Conflict Graph and Learning Schemes

To define the learning schemes that we will discuss in the following sections we need to introduce some concepts. The interested reader can find an extended discussion about conflict analysis in [BKS04].

The *conflict graph* of a conflict state is defined iteratively as follows. We begin with two vertices labelled by the conflict literal and its negation. For each literal a in the graph that is unit propagated with reason C, we add an edge from each literal in $\overline{C \setminus \{a\}}$ (and possibly a new vertex) to a. For this construction we assume that the conflicting clause propagates the opposite of the conflict literal.

For each cut in this directed graph that separates all decision literals from the two conflict literals, we can collect the vertices from which the cut edges emanate and negate all these literals to obtain a clause that is falsified by the current assignment and is therefore a conflict clause that can be learned. The *reason side* contains the decisions and the *conflict side* contains the conflicts.

JAKOB'S COMMENT 1: *Is this an **example** of a conflict clause or a **definition** of a conflict clause? I would have thought the former but it sounds like the latter. I think AFT11 and PD11 are more generous with conflict clause definitions, but I am not sure.*

MARC'S COMMENT 1: *It is an example, rewording so it sounds more like one.*

A *unique implication point* (UIP) is a vertex that belongs to all paths from the last decision literal to the two conflict literals. The clause induced by the cut whose conflict side are the successors of a UIP is asserting. The *1UIP* learning scheme learns the clause induced by the UIP closest to the conflict literals. It is well-defined because there is always at least one UIP: the last decision literal.

We will refer to a learning scheme that learns asserting clauses that come from a cut in the conflict graph as a *cutting* learning scheme. 1UIP is clearly a cutting scheme. The *decision* learning scheme, which learns the opposite of all decision literals in the current assignment, is asserting but not necessarily cutting.

We say that a learning scheme is *trivial* if it produces clauses that can be derived from the clause database using trivial resolution. All cutting learning schemes are trivial [BKS04].

2.5 From CDCL Proofs to Resolution Proofs

We now show that a CDCL refutation with a learning scheme that uses trivial resolution to derive conflict clauses can be translated to a resolution refutation in essentially the same time and space.

Theorem 2.1. *If there is a CDCL proof with some trivial learning scheme refuting a CNF formula F in time τ and space s, then F has a resolution refutation of length at most τ and space at most $s + 3$.*

Proof. Given a legal CDCL trace refuting F, we construct a resolution proof in the following way. We list all the trivial resolution derivations of learned clauses in order, but we omit occurrences of learned clauses, so that we use the last occurrence of a learned clause at the end of a derivation as an input clause for the current derivation (see Figure 2). At the end we add a derivation of the empty clause from the last conflict.

We can prove that it is a legal refutation by forward induction over the learned clauses in the derivation. Each partial derivation uses clauses in the database, all legally derived by induction hypothesis.

JAKOB'S COMMENT 2: *"Induction over the number of learned clauses" or just "forward induction over learned clauses in the derivation"?*

MASSIMO'S COMMENT 3: *Changed.*

The length of the resolution refutation is the sum of lengths of each derivation, which we account for in the time of CDCL refutation, plus the length of the derivation of the empty clause. Since every derivation uses reason clauses, the length of the last derivation is at most the number of unit propagations.

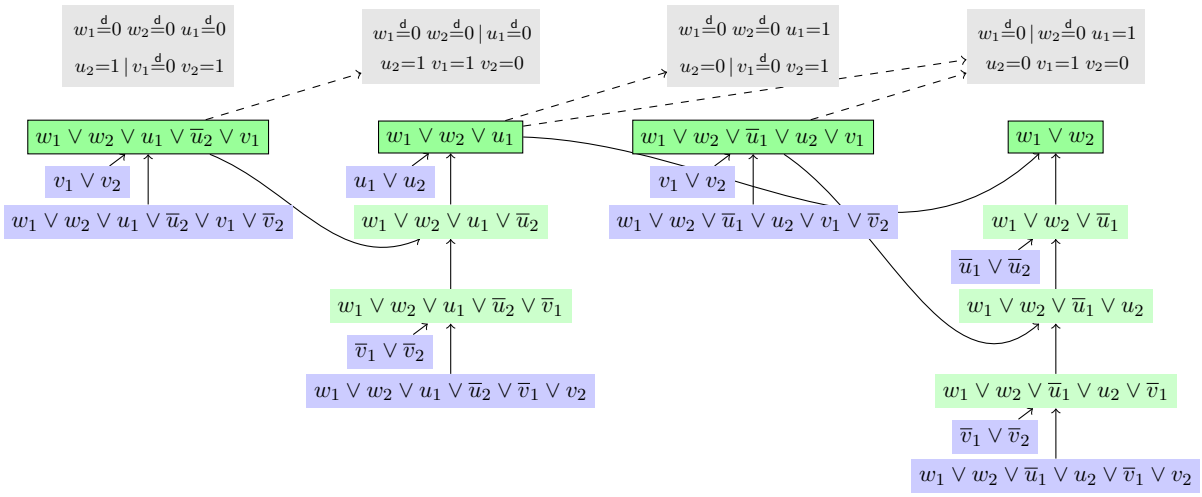


Figure 2: Fragment of a CDCL proof translated into resolution. Dark blue clauses are axioms, dark green clauses with a border are learned and added to the clause database, and light green clauses are auxiliary. Upper boxes contain the trail at each conflict state. Edges show resolution steps, and dashed edges are reasons for unit propagation.

To estimate the space of the refutation, we consider the partial derivation of the clause learned in a conflict state $(F, \mathcal{D}, s)$. At each resolution step, one of the clauses was in the clause database, and the other is either in the database or it was derived in the previous step and will not be used again. The same applies for later derivations. Therefore, for each resolution step t, all the clauses derived before step t that are used strictly later are in the clause database. Furthermore, axioms are always restated immediately before a derivation and we should not count them, so the upper bound is actually $|\mathcal{D} \setminus F|$. It only remains to count the clause derived at step t, and the two clauses used to derive it. Thus, the space at step t is at most $|\mathcal{D} \setminus F| + 3$. Taking the maximum over all steps gives precisely $s + 3$. □

JAKOB'S COMMENT 3: *I didn't quite get the space argument above.*

JAKOB'S COMMENT 4: *My spontaneous reaction is that I would prefer to have a short, sweet caption for Figure 2 and then discuss the details in running text. But perhaps we already discussed this?*

3 Overview of Time-Space Trade-off Results

In this section we survey the kind of CDCL time-space trade-offs obtained in this paper, and discuss some of the challenges that have to be overcome when establishing such results.

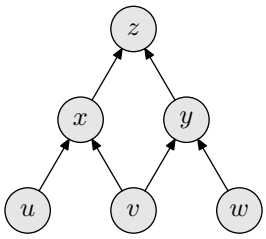
3.1 Statement of Trade-off Theorems

Our first set of trade-off results are for formulas defined in terms of pebble games as described in [BW01]. Given a directed acyclic graph (DAG) G with source vertices S and a unique sink vertex z, and with all non-sources having fan-in 2, we let every vertex in G correspond to a variable and define the *pebbling formula* over G, denoted Peb_G, to consist of the following clauses:

- for all $s \in S$, the unit clause s (*source axioms*),
- for all non-sources w with immediate predecessors u, v, the clause $\overline{u} \vee \overline{v} \vee w$ (*pebbling axioms*),
- for the sink z, the unit clause $\overline{z}$ (*sink axiom*).
  JAKOB'S COMMENT 5: *Here we define the DAGs to have fan-in exactly 2.*

See the formula in Figure 3b defined in terms of the pyramid graph in Figure 3a for an illustration.

These formulas are not too interesting, since it is easy to see that they are solved immediately by unit propagation, but if we replace each variable by an exclusive or of two new, fresh variables, and then



(a) Pyramid graph Π_2 of height 2.

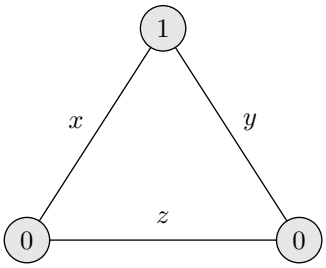
$$
\begin{aligned}
& u \\
\wedge\ & v \\
\wedge\ & w \\
\wedge\ & (\overline{u} \vee \overline{v} \vee x) \\
\wedge\ & (\overline{v} \vee \overline{w} \vee y) \\
\wedge\ & (\overline{x} \vee \overline{y} \vee z) \\
\wedge\ & \overline{z}
\end{aligned}
$$

(b) Pebbling contradiction Peb_{Π_2}.

$$
\begin{aligned}
& (u_1 \vee u_2) \\
\wedge\ & (\overline{u}_1 \vee \overline{u}_2) \\
\wedge\ & (v_1 \vee v_2) \\
\wedge\ & (\overline{v}_1 \vee \overline{v}_2) \\
\wedge\ & (w_1 \vee w_2) \\
\wedge\ & (\overline{w}_1 \vee \overline{w}_2) \\
\wedge\ & (u_1 \vee \overline{u}_2 \vee v_1 \vee \overline{v}_2 \vee x_1 \vee x_2) \\
\wedge\ & (u_1 \vee \overline{u}_2 \vee v_1 \vee \overline{v}_2 \vee \overline{x}_1 \vee \overline{x}_2) \\
\wedge\ & (u_1 \vee \overline{u}_2 \vee \overline{v}_1 \vee v_2 \vee x_1 \vee x_2) \\
\wedge\ & (u_1 \vee \overline{u}_2 \vee \overline{v}_1 \vee v_2 \vee \overline{x}_1 \vee \overline{x}_2) \\
\wedge\ & (\overline{u}_1 \vee u_2 \vee v_1 \vee \overline{v}_2 \vee x_1 \vee x_2) \\
\wedge\ & (\overline{u}_1 \vee u_2 \vee v_1 \vee \overline{v}_2 \vee \overline{x}_1 \vee \overline{x}_2) \\
\wedge\ & (\overline{u}_1 \vee u_2 \vee \overline{v}_1 \vee v_2 \vee x_1 \vee x_2) \\
\wedge\ & (\overline{u}_1 \vee u_2 \vee \overline{v}_1 \vee v_2 \vee \overline{x}_1 \vee \overline{x}_2) \\
\wedge\ & (v_1 \vee \overline{v}_2 \vee w_1 \vee \overline{w}_2 \vee y_1 \vee y_2) \\
\wedge\ & (v_1 \vee \overline{v}_2 \vee w_1 \vee \overline{w}_2 \vee \overline{y}_1 \vee \overline{y}_2) \\
\wedge\ & (v_1 \vee \overline{v}_2 \vee \overline{w}_1 \vee w_2 \vee y_1 \vee y_2) \\
\wedge\ & (v_1 \vee \overline{v}_2 \vee \overline{w}_1 \vee w_2 \vee \overline{y}_1 \vee \overline{y}_2) \\
\wedge\ & (\overline{v}_1 \vee v_2 \vee w_1 \vee \overline{w}_2 \vee y_1 \vee y_2) \\
\wedge\ & (\overline{v}_1 \vee v_2 \vee w_1 \vee \overline{w}_2 \vee \overline{y}_1 \vee \overline{y}_2) \\
\wedge\ & (\overline{v}_1 \vee v_2 \vee \overline{w}_1 \vee w_2 \vee y_1 \vee y_2) \\
\wedge\ & (\overline{v}_1 \vee v_2 \vee \overline{w}_1 \vee w_2 \vee \overline{y}_1 \vee \overline{y}_2) \\
\wedge\ & (x_1 \vee \overline{x}_2 \vee y_1 \vee \overline{y}_2 \vee z_1 \vee z_2) \\
\wedge\ & (x_1 \vee \overline{x}_2 \vee y_1 \vee \overline{y}_2 \vee \overline{z}_1 \vee \overline{z}_2) \\
\wedge\ & (x_1 \vee \overline{x}_2 \vee \overline{y}_1 \vee y_2 \vee z_1 \vee z_2) \\
\wedge\ & (x_1 \vee \overline{x}_2 \vee \overline{y}_1 \vee y_2 \vee \overline{z}_1 \vee \overline{z}_2) \\
\wedge\ & (\overline{x}_1 \vee x_2 \vee y_1 \vee \overline{y}_2 \vee z_1 \vee z_2) \\
\wedge\ & (\overline{x}_1 \vee x_2 \vee y_1 \vee \overline{y}_2 \vee \overline{z}_1 \vee \overline{z}_2) \\
\wedge\ & (\overline{x}_1 \vee x_2 \vee \overline{y}_1 \vee y_2 \vee z_1 \vee z_2) \\
\wedge\ & (\overline{x}_1 \vee x_2 \vee \overline{y}_1 \vee y_2 \vee \overline{z}_1 \vee \overline{z}_2) \\
\wedge\ & (z_1 \vee \overline{z}_2) \\
\wedge\ & (\overline{z}_1 \vee z_2)
\end{aligned}
$$

(c) XORified pebbling formula $Peb^{\oplus}_{\Pi_2}$.

Figure 3: Example pebbling formula for the pyramid of height 2.



(a) Labelled triangle graph.

$$
\begin{aligned}
& (x \vee y) \\
\wedge\ & (\overline{x} \vee \overline{y}) \\
\wedge\ & (x \vee \overline{z}) \\
\wedge\ & (\overline{x} \vee z) \\
\wedge\ & (y \vee \overline{z}) \\
\wedge\ & (\overline{y} \vee z)
\end{aligned}
$$

(b) Corresponding Tseitin formula.

Figure 4: Example Tseitin formula.

expand out to CNF we obtain a *XORified pebbling formula* $Peb_G^{\oplus}$ as in Figure 3c. Given the right kind of graphs, [BN11] showed that such formulas have strong trade-offs between length and space in resolution, and we are able to lift most of these results to CDCL. We give two examples of such results below.

Theorem 3.1 (Robust trade-offs (informal)). *There are XORified pebbling formulas F_n of size $\Theta(n)$ such that:*

- *CDCL with 1UIP learning and no restarts can refute F_n in time $\mathrm{O}(n)$ and space $\mathrm{O}(n/\log n)$ simultaneously.*
- *CDCL with 1UIP learning and no restarts can refute F_n in space $\mathrm{O}\big((\log n)^2\big)$ and time $n^{\mathrm{O}(\log n)}$ simultaneously.*
- *Any CDCL refutation of F_n in space $\mathrm{o}(n/\log n)$ requires time at least $n^{\Omega(\log\log n)}$ regardless of learning scheme and restart policy.*

Theorem 3.2 (Exponential trade-offs (informal)). *There are XORified pebbling formulas F_n of size $\Theta(n)$ such that:*

- *CDCL with 1UIP learning and no restarts can refute F_n in time $\mathrm{O}(n)$ and space $\mathrm{O}\big(\sqrt[4]{n}\big)$ simultaneously.*
- *CDCL with 1UIP learning and no restarts can refute F_n in space $\mathrm{O}\big(\sqrt[8]{n}\big)$ and time $n^{\mathrm{O}(\sqrt[8]{n})}$ simultaneously.*
- *Any CDCL refutation of F_n in space $\mathrm{O}\big(n^{1/4-\epsilon}\big)$ for $\epsilon > 0$ requires exponential time regardless of learning scheme and restart policy.*

The other formula family considered in this paper are *Tseitin formulas*, which are defined in terms of undirected graphs with vertices labelled $0/1$ in such a way that the total sum of all vertex labels is odd. The variables of the formula are the edges of the graph. For every vertex we add a constraint saying that the parity of the number of true edges incident to the vertex is equal to the vertex label. Summing over all vertices, each edge is counted exactly twice and hence the total number of true edges must be even. But this contradicts that the sum of the labels is odd, and thus the formulas are unsatisfiable. Figure 4b gives an example Tseitin formula generated from the labelled graph in Figure 4a.

Using Tseitin formulas over long, skinny grids, we can build on [BNT13] to obtain the following trade-off, which applies even for superlinear space.

Theorem 3.3 (Superlinear space trade-offs (informal)). *For a Tseitin formula $F_{w,\ell}$ over a grid graph with w rows and ℓ columns, $1 \leq w \leq \ell^{1/4}$, and with double edges between every two vertices at horizontal distance one or vertical distance one, it holds that*

- *CDCL with 1UIP learning and no restarts can refute $F_{w,\ell}$ in time $\mathrm{O}(2^{5w}\ell)$ and space $\mathrm{O}(2^{2w})$.*
- *CDCL with 1UIP learning and no restarts can refute $F_{w,\ell}$ in space $\mathrm{O}(w\log(\ell))$ and time $\mathrm{O}\big(\ell^{\mathrm{O}(w)}\big)$.*
- *For any CDCL refutation in time τ and space s, regardless of learning scheme and restart policy, it holds that*

$$\tau = \left(\frac{2^{\Omega(w)}}{s}\right)^{\Omega\left(\frac{\log\log\ell}{\log\log\log\ell}\right)}.$$

3.2 Proof Techniques and Technical Challenges

All the trade-offs stated in Theorems 3.1, 3.2, and 3.3 are known to hold for resolution, and so by Theorem 2.1 we immediately obtain that the lower bounds carry over to CDCL. What we need to show in order to establish these theorems is that CDCL can find proofs that match the upper bounds in resolution.

The general idea how we would like to do this is clear: given a resolution proof $\pi = (C_1, C_2, \ldots, C_\tau)$, we should force the CDCL solver to efficiently learn the clauses C_i one by one, making sure at all times that the clause database size is comparable to the space complexity of the resolution proof. This seems hard to do, however, and somewhat ironically what causes trouble for us are the unit propagations that otherwise make CDCL so efficient. To illustrate the problem, suppose that we have learned $C \vee x$ and $D \vee \overline{x}$ and now want to learn their resolvent $C \vee D$. It would be nice to decide on all literals in $C \vee D$ being false, after which we could get a conflict on x. But there might be other clauses in the database that propagate literals to "wrong values" before we manage to falsify all of $C \vee D$, and if so the CDCL search will veer off in another direction and we will not be able to learn this resolvent.

This highlights two technical difficulties that we need to be able to deal with:

- Not only do we have to decide on variables in the right order, but we have to make sure that no other unexpected (and unwanted) propagations occur.
- In contrast to resolution, where having more clauses at your disposal never hurts, keeping too many learned clauses in the clause database can actually hinder the CDCL search. This is also a striking contrast to [AFT11, PD11], where a key technical lemma is precisely that having more clauses in the database can only be helpful.

We do not know how to simulate general resolution efficiently with respect to length and space simultaneously, even using ever so frequent restarts. And an additional problem is that we want to know—in order to better understand basic CDCL reasoning with unit propagation and conflict analysis—whether for the pebbling and Tseitin formulas presented above CDCL can find efficient proofs *even without restarts*. This makes our task substantially more complicated.

If we allow suitably frequent restarts, however, it is not too hard to show that CDCL can efficiently simulate the "canonical" resolution proofs for these formulas. To give at least some flavour of the technical arguments that will follow later in the paper when we reason about the CDCL proof system, we conclude this section with a description of how this result can be proven for pebbling formulas.

3.3 Pebbling Formula Upper Bound for CDCL with Restarts

A pebbling formula encodes the *black pebble game* played on a DAG G, where we start with G being empty and want to finish with a pebble on the sink z. A vertex can be pebbled if its predecessors have pebbles (vacuously true for sources), and pebbles can always be removed. The *time* of a pebbling is the number of moves before z is reached and the *space* is the maximum number of pebbles on vertices of G at any point.

Resolution can simulate such pebblings by deriving, whenever a vertex w is pebbled, the two *pebble clauses* $w_1 \vee w_2$ and $\overline{w}_1 \vee \overline{w}_2$ saying that the exclusive or $w_1 \oplus w_2$ is true, and by erasing these clauses whenever a pebble is removed. For a source vertex the pebble clauses are already available as source axioms in the formula (see Figure 3c), and it is not hard to show that resolution can efficiently propagate exclusive ors from predecessors to successors. Once pebble clauses have been derived for the sink z, contradiction immediately follows from the sink axioms.

We want to mimic this in CDCL as described in the algorithm `Pebble` in Figure 5 producing a CDCL trace. For a pebble placement, we want to learn first $w_1 \vee w_2$, corresponding to "half of the pebble" on w, and then $\overline{w}_1 \vee \overline{w}_2$. How to do this is described in the procedure `HalfPebble` in Figure 6, where the notation x^b for $b \in \{0, 1\}$ is used as a compact way of denoting $x^1 = x$ and $x^0 = \overline{x}$.

When a pebble is placed on w in the pebbling, we let the CDCL solver make the decisions $(w_1 = 0/\mathsf{d}, w_2 = 0/\mathsf{d})$ with the goal of learning $w_1 \vee w_2$. Then we decide values for the variables of the predecessors u and v

```
Input: a black pebbling 𝒫
foreach (move,w) in 𝒫 where w is not a source or the sink do
    if move is Add then
        HalfPebble (w, 0)
        HalfPebble (w, 1)
    else
        del w_1 ∨ w_2
        del w̄_1 ∨ w̄_2
PebbleSink
```

Figure 5: Procedure `Pebble` ($\mathcal{P}$)

```
Input: A vertex w, a Boolean b
Decide w_1 = b/d
Decide w_2 = b/d
FindConflicts (w, b)
Learn w_1^{1-b} ∨ w_2^{1-b} and assert w_2 = 1 − b/u
Restart R
foreach clause C ∈ 𝒟 \ F such that |C| > 2 do
    del C
```

Figure 6: Procedure `HalfPebble` (w, b)

of w, and since there are clauses in memory encoding $u_1 \oplus u_2$ and $v_1 \oplus v_2$ this will provoke repeated conflicts until finally the clause $w_1 \vee w_2$ is learned. Since this only involves a constant number of variables, the time and space required for this is constant, and our goal can be achieved, e.g., as described in `FindConflicts` in Figure 7.

But now we run into problems. At this point the CDCL solver will backjump to the decision $w_1 = 0$, where the learned clause $w_1 \vee w_2$ asserts $w_2 = 1$. As the next step, we want to generate conflicts that lead to the "second half" of the pebble $\overline{w}_1 \vee \overline{w}_2$ being learned, but there is no way this can happen since the decision $w_1 = 0$ is on the trail and the clause $\overline{w}_1 \vee \overline{w}_2$ is thus satisfied. Moreover, if the solver is not allowed to restart, then this satisfying assignment is fixed on the trail, and no new conflict could possibly cause a backjump to before this assignment. Therefore, the solver is forced to continue the proof search elsewhere. This turns out to be a major obstacle, which we are able to circumvent only by substantial extra work involving reordering the pebbling and using a different algorithm. This will be a large part of the work in the rest of this paper as the technical arguments are rather intricate.

If we instead give the solver the option to restart at this point, it can clear the trail and also forget all unnecessary clauses. This means that the decisions $(w_1 = 1/\mathsf{d}, w_2 = 1/\mathsf{d})$ can be made, after which the clause $\overline{w}_1 \vee \overline{w}_2$ is learned in the same way as above. To conclude, we again trigger a restart and erase all auxiliary clauses that are no longer needed.

JAKOB'S COMMENT 6: *(a) I would remove the shorthand "Learn C and assert $x = b/\mathsf{u}$" and write out the full trace instead in the procedure figure. (b) I would simply remove: "Other text annotations are for readability and do not appear in the formal proof."*

MARC'S COMMENT 2: *Removed from the introduction. We should probably keep the shorthand for the full version, though, otherwise procedures in the no-restarts section would become too long.*

JAKOB'S COMMENT 7: *OK.*

Pebble removals are very straightforward to simulate: the only condition that could stop us from erasing the clauses $w_1 \vee w_2$ and $\overline{w}_1 \vee \overline{w}_2$ is if they are reasons for propagated literals on the trail, but since we have just made a restart the trail is empty. Formalizing the arguments above, we obtain the following lemma.

Lemma 3.4. *If $\mathcal{P}$ is a black pebbling of G in space s and time τ, then there is a CDCL proof of $Peb_G^{\oplus}$ with restarts using the 1UIP learning scheme and any unit propagation scheme in space at most $2s + 3$*

Input: A vertex w with predecessors u and v, a Boolean b

1 Decide $u_1 = 0/\mathsf{d}$
2 Propagate $u_2 = 1/u_1 \vee u_2$
3 Decide $v_1 = 0/\mathsf{d}$
4 Learn $w_1^{1-b} \vee w_2^{1-b} \vee u_1 \vee \overline{u}_2 \vee v_1$
5 Assert $v_1 = 1/w_1^{1-b} \vee w_2^{1-b} \vee u_1 \vee \overline{u}_2 \vee v_1$
6 Learn $w_1^{1-b} \vee w_2^{1-b} \vee u_1$
7 Assert $u_1 = 1/w_1^{1-b} \vee w_2^{1-b} \vee u_1$
8 Propagate $u_2 = 0/\overline{u}_1 \vee \overline{u}_2$
9 Decide $v_1 = 0/\mathsf{d}$
10 Learn $w_1^{1-b} \vee w_2^{1-b} \vee \overline{u}_1 \vee u_2 \vee v_1$
11 Assert $v_1 = 1/w_1^{1-b} \vee w_2^{1-b} \vee \overline{u}_1 \vee u_2 \vee v_1$

Figure 7: Procedure `FindConflicts` (w, b)

and time $\mathrm{O}(\tau)$.

JAKOB'S COMMENT 8: *For FindConflict we now also have vertices with just one predecessor (cf. metacomment above). Should we just ignore this? Or add some comment somewhere?*

MARC'S COMMENT 3: *IIRC we only use vertices of indegree one in permutation graphs, and we do not talk about them in the overview, so let us not say anything here.*

JAKOB'S COMMENT 9: *Don't we miss a unit propagation for u_2 in case 1? Or we don't show the final unit propagation before conflict? (Is this consistent with the proof system description?)*

MARC'S COMMENT 4: *Technically we do. But then we cannot tell whether we should propagate u_2 or $\overline{u}_2$, so we hid this detail under the rug.*

JAKOB'S COMMENT 10: *OK, but then we should mention explicitly somewhere (at some point—not necessarily before SAT submission) that we omit the final pairs of unit propagations on $x, \overline{x}$ leading to a conflict*

Proof. Given a pebbling $\mathcal{P}$ we generate a trace as described by the procedure `Pebble`. Note that this procedure maintains the invariant that the pebble clauses for a non-source vertex w are in the clause database if and only if there is a pebble on w. No other clauses are in memory. The space bound follows from this invariant, and the time bound holds by construction.

It remains to check that the trace thus generated is legal. Observe that the clauses in memory only propagate if at least one variable from each vertex they mention is set. Since the decision sequence mentions at most three vertices at the same time, we only need to reason about clauses that mention these vertices.

The correctness of `FindConflicts` is straightforward to verify, since the order of unit propagations can be seen to be uniquely determined. At the end of `FindConflicts`, the assignments to w_1 and w_2 are decisions and all predecessor variables u_1, u_2, v_1, v_2 are set by unit propagation. Since one of the conflicting clauses, a pebbling axiom, contains the decision variables w_1 and w_2, they have to appear in any cut and therefore any conflict clause. The remaining variables in the conflict graph have maximal decision level, so they cannot appear in an asserting clause because w_2 already appears. Therefore, we learn the clause $w_1^{1-b} \vee w_2^{1-b}$ and assert $w_2 = 1 - b/\mathsf{u}$. We only erase clauses after a restart, so no erased clauses can be reasons for unit propagations.

JAKOB'S COMMENT 11: *The text below is referring to other learning schemes than 1UIP, but the lemma in this section is stated only for 1UIP.*

JANE'S COMMENT 1: *One could erase starting from "To conclude the proof"*

This concludes the proof.

□

Figure 8 is the result of running `Pebble` on a specific pebbling strategy on the example formula from Figure 3c, with resolution steps omitted.

1 $x_1 = 0/\mathsf{d}$
2 $x_2 = 0/\mathsf{d}$
3 $u_1 = 0/\mathsf{d}$
4 $u_2 = 1/u_1 \vee u_2$
5 $v_1 = 0/\mathsf{d}$
6 $\mathsf{add}\, x_1 \vee x_2 \vee u_1 \vee \overline{u}_2 \vee v_1/\sigma$
7 $v_1 = 1/x_1 \vee x_2 \vee u_1 \vee \overline{u}_2 \vee v_1$
8 $\mathsf{add}\, x_1 \vee x_2 \vee u_1/\sigma$
9 $u_1 = 1/x_1 \vee x_2 \vee u_1$
10 $u_2 = 0/\overline{u}_1 \vee \overline{u}_2$
11 $v_1 = 0/\mathsf{d}$
12 $\mathsf{add}\, x_1 \vee x_2 \vee \overline{u}_1 \vee u_2 \vee v_1/\sigma$
13 $v_1 = 1/x_1 \vee x_2 \vee \overline{u}_1 \vee u_2 \vee v_1$
14 $\mathsf{add}\, x_1 \vee x_2/\sigma$
15 $x_2 = 1/x_1 \vee x_2$
16 R
17 $\mathsf{del}\, x_1 \vee x_2 \vee u_1 \vee \overline{u}_2 \vee v_1$
18 $\mathsf{del}\, x_1 \vee x_2 \vee u_1$
19 $\mathsf{del}\, x_1 \vee x_2 \vee \overline{u}_1 \vee u_2 \vee v_1$
20 $x_1 = 1/\mathsf{d}$
21 $x_2 = 1/\mathsf{d}$
22 $u_1 = 0/\mathsf{d}$
23 $u_2 = 1/u_1 \vee u_2$
24 $v_1 = 0/\mathsf{d}$
25 $\mathsf{add}\, \overline{x}_1 \vee \overline{x}_2 \vee u_1 \vee \overline{u}_2 \vee v_1/\sigma$
26 $v_1 = 1/\overline{x}_1 \vee \overline{x}_2 \vee u_1 \vee \overline{u}_2 \vee v_1$
27 $\mathsf{add}\, \overline{x}_1 \vee \overline{x}_2 \vee u_1/\sigma$
28 $u_1 = 1/\overline{x}_1 \vee \overline{x}_2 \vee u_1$
29 $u_2 = 0/\overline{u}_1 \vee \overline{u}_2$
30 $v_1 = 0/\mathsf{d}$
31 $\mathsf{add}\, \overline{x}_1 \vee \overline{x}_2 \vee \overline{u}_1 \vee u_2 \vee v_1/\sigma$
32 $v_1 = 1/\overline{x}_1 \vee \overline{x}_2 \vee \overline{u}_1 \vee u_2 \vee v_1$
33 $\mathsf{add}\, \overline{x}_1 \vee \overline{x}_2/\sigma$
34 $x_2 = 0/\overline{x}_1 \vee \overline{x}_2$
35 R
36 $\mathsf{del}\, \overline{x}_1 \vee \overline{x}_2 \vee u_1 \vee \overline{u}_2 \vee v_1$
37 $\mathsf{del}\, \overline{x}_1 \vee \overline{x}_2 \vee u_1$
38 $\mathsf{del}\, \overline{x}_1 \vee \overline{x}_2 \vee \overline{u}_1 \vee u_2 \vee v_1$
39 $y_1 = 0/\mathsf{d}$
40 $y_2 = 0/\mathsf{d}$
41 $v_1 = 0/\mathsf{d}$
42 $v_2 = 1/v_1 \vee v_2$
43 $w_1 = 0/\mathsf{d}$
44 $\mathsf{add}\, y_1 \vee y_2 \vee v_1 \vee \overline{v}_2 \vee w_1/\sigma$
45 $w_1 = 1/y_1 \vee y_2 \vee v_1 \vee \overline{v}_2 \vee w_1$
46 $\mathsf{add}\, y_1 \vee y_2 \vee v_1/\sigma$
47 $v_1 = 1/y_1 \vee y_2 \vee v_1$
48 $v_2 = 0/\overline{v}_1 \vee \overline{v}_2$
49 $w_1 = 0/\mathsf{d}$
50 $\mathsf{add}\, y_1 \vee y_2 \vee \overline{v}_1 \vee v_2 \vee w_1/\sigma$
51 $w_1 = 1/y_1 \vee y_2 \vee \overline{v}_1 \vee v_2 \vee w_1$
52 $\mathsf{add}\, y_1 \vee y_2/\sigma$
53 $y_2 = 1/y_1 \vee y_2$
54 R
55 $\mathsf{del}\, y_1 \vee y_2 \vee v_1 \vee \overline{v}_2 \vee w_1$
56 $\mathsf{del}\, y_1 \vee y_2 \vee v_1$
57 $\mathsf{del}\, y_1 \vee y_2 \vee \overline{v}_1 \vee v_2 \vee w_1$
58 $y_1 = 1/\mathsf{d}$
59 $y_2 = 1/\mathsf{d}$
60 $v_1 = 0/\mathsf{d}$
61 $v_2 = 1/v_1 \vee v_2$
62 $w_1 = 0/\mathsf{d}$
63 $\mathsf{add}\, \overline{y}_1 \vee \overline{y}_2 \vee v_1 \vee \overline{v}_2 \vee w_1/\sigma$
64 $w_1 = 1/\overline{y}_1 \vee \overline{y}_2 \vee v_1 \vee \overline{v}_2 \vee w_1$
65 $\mathsf{add}\, \overline{y}_1 \vee \overline{y}_2 \vee v_1/\sigma$
66 $v_1 = 1/\overline{y}_1 \vee \overline{y}_2 \vee v_1$
67 $v_2 = 0/\overline{v}_1 \vee \overline{v}_2$
68 $w_1 = 0/\mathsf{d}$
69 $\mathsf{add}\, \overline{y}_1 \vee \overline{y}_2 \vee \overline{v}_1 \vee v_2 \vee w_1/\sigma$
70 $w_1 = 1/\overline{y}_1 \vee \overline{y}_2 \vee \overline{v}_1 \vee v_2 \vee w_1$
71 $\mathsf{add}\, \overline{y}_1 \vee \overline{y}_2/\sigma$
72 $y_2 = 0/\overline{y}_1 \vee \overline{y}_2$
73 R
74 $\mathsf{del}\, \overline{y}_1 \vee \overline{y}_2 \vee v_1 \vee \overline{v}_2 \vee w_1$
75 $\mathsf{del}\, \overline{y}_1 \vee \overline{y}_2 \vee v_1$
76 $\mathsf{del}\, \overline{y}_1 \vee \overline{y}_2 \vee \overline{v}_1 \vee v_2 \vee w_1$
77 $z_1 = 0/\mathsf{d}$
78 $z_2 = 0/z_1 \vee \overline{z}_2$
79 $x_1 = 0/\mathsf{d}$
80 $x_2 = 1/x_1 \vee x_2$
81 $y_1 = 0/\mathsf{d}$
82 $\mathsf{add}\, z_1 \vee z_2 \vee x_1 \vee \overline{x}_2 \vee y_1/\sigma$
83 $y_1 = 1/z_1 \vee z_2 \vee x_1 \vee \overline{x}_2 \vee y_1$
84 $\mathsf{add}\, z_1 \vee z_2 \vee x_1/\sigma$
85 $x_1 = 1/z_1 \vee z_2 \vee x_1$
86 $x_2 = 0/\overline{x}_1 \vee \overline{x}_2$
87 $y_1 = 0/\mathsf{d}$
88 $\mathsf{add}\, z_1 \vee z_2 \vee \overline{x}_1 \vee x_2 \vee y_1/\sigma$
89 $y_1 = 1/z_1 \vee z_2 \vee \overline{x}_1 \vee x_2 \vee y_1$
90 $\mathsf{add}\, z_1/\sigma$
91 $z_1 = 1/z_1$
92 $z_2 = 1/\overline{z}_1 \vee z_2$
93 $x_1 = 0/\mathsf{d}$
94 $x_2 = 1/x_1 \vee x_2$
95 $y_1 = 0/\mathsf{d}$
96 $\mathsf{add}\, \overline{z}_1 \vee \overline{z}_2 \vee x_1 \vee \overline{x}_2 \vee y_1/\sigma$
97 $y_1 = 1/\overline{z}_1 \vee \overline{z}_2 \vee x_1 \vee \overline{x}_2 \vee y_1$
98 $\mathsf{add}\, \overline{z}_1 \vee \overline{z}_2 \vee x_1/\sigma$
99 $x_1 = 1/\overline{z}_1 \vee \overline{z}_2 \vee x_1$
100 $x_2 = 0/\overline{x}_1 \vee \overline{x}_2$
101 $y_1 = 0/\mathsf{d}$
102 $\mathsf{add}\, \overline{z}_1 \vee \overline{z}_2 \vee \overline{x}_1 \vee x_2 \vee y_1/\sigma$
103 $y_1 = 1/\overline{z}_1 \vee \overline{z}_2 \vee \overline{x}_1 \vee x_2 \vee y_1$
104 UNSAT

Figure 8: Proof of the pebbling formula $Peb^{\oplus}_{\Pi_2}$

4 Worst-case Upper Bound

MASSIMO'S COMMENT 4: *STATUS: Section 3 is fine, but Section 3.1 need to be integrated with the rest of the paper. Does the theorem in Section 3.1 make obsolete the theorem in section 3?*

JAKOB'S COMMENT 12: *STATUS UPDATE needed here... Seems like it is time to add back the simulation of tree-like resolution and provide a polished proof for that.*

The simulation by [PD11] shows that CDCL with unrestricted restarts can polynomially simulate resolution. The authors only analyze time and assume that no learnt clause is ever forgotten. However, it is not hard to come up with a clause deletion strategy that gives an $\mathrm{O}(s \cdot poly(n))$ space bound, where s is the space of the resolution refutation we simulate and n is the number of variables. Because the proof length is always at least linear in n, the $\mathrm{O}(poly(n))$ blowup is polynomial with respect to length. Space, however, can be $\mathrm{O}(\log n)$ or even $\mathrm{O}(1)$, so that the blowup is exponential. Therefore, the simulation does not resolve the question of space-efficient simulation of resolution with restarts.

To start acquainting ourselves with the CDCL proof system, let us demonstrate that the worst-case behaviour for CDCL is the same as for resolution: any unsatisfiable CNF formula can be refuted in exponential time and linear space simultaneously. We remark that this result was already shown in [NOT06]. Since the language used there is quite different, however, we present a self-contained proof below for completeness.

Theorem 4.1. *Let F be an unsatisfiable CNF formula on n variables. There is a CDCL refutation of F without restarts and with any asserting learning scheme, unit propagation scheme, and decision scheme in time $\mathrm{O}(n2^n)$ and space at most $n-1$.*

Proof. We will let the CDCL solver use a very aggressive database reduction policy, namely to erase as many clauses as possible from the database. This means that when the database is reduced we delete all clauses except those that are reason clauses in the current trail, and so in formal notation the clause database after a reduction step in state $(F, \mathcal{D}, s)$ is

$$\mathcal{D}' = F \cup \{C \mid x{=}b/C \in s\} \ . \tag{4.1}$$

We first argue that the clause database contains at most $n-1$ learned clauses at every step. The computation begins with unit propagations at decision level 0. If there is a conflict then the trace ends, otherwise the solver reaches a stable state. In this phase we do not learn any clauses. Afterwards the clause database is pruned at each stable state and keeps growing until the solver reaches the next stable state (or terminates).

Let $(F, \mathcal{D}, s)$ be the state of the solver right after an erasure step, and let d be the decision level of s. By construction $\mathcal{D} \setminus F$ has at most one clause for each unit propagation in the sequence s, therefore $d + |\mathcal{D} \setminus F| \leq |s|$. Observe that if $|s| > n-2$ then $F|_s$ would have either a unit or an empty clause. Since we are in a stable state, $|s| \leq n-2$.

Now consider an arbitrary state $(F, \mathcal{D}', s')$, and consider the state right after the most recent erasure. Call such state $(F, \mathcal{D}, s)$, and let d be the decision level of s. After an erasure, the solver does a decision step that increases the decision level to $d+1$. Each time the solver reaches **CONFLICT** mode the decision level decreases, and does not increase anymore until the next stable state. So there can be at most $d+1$ conflicts which result in learning a clause (a conflict at level zero would just end the computation, without changing the clause database). Thus $|\mathcal{D}' \setminus F| \leq d + 1 + |\mathcal{D} \setminus F| \leq n-1$ as we wanted.

Finally we prove that the solver terminates in time $\mathrm{O}(n2^n)$. Given a trail s we define the string $b(s) \in \{0,1\}^{|s|}$ by letting the coordinates be

$$b(s)_i = \begin{cases} 0 & \text{if } s_i \text{ is a decision} \\ 1 & \text{if } s_i \text{ is a unit propagation} \end{cases} \tag{4.2}$$

for $1 \leq i \leq |s|$.

We establish the upper bound on time by exhibiting a subsequence of states such that the corresponding subsequence $b(s)$ is strictly increasing with respect to the lexicographic order, and there are a

constant number of states between any two states in the subsequence. The length of a string $b(s)$ is at most n, so the subsequence has length at most $2^{n+1} - 1$ and the time upper bound follows by recalling that a trivial resolution has length at most n.

Consider a stable state $(F, \mathcal{D}, s)$. The solver applies an erasure and a decision, ending in some state $(F, \mathcal{D}', s')$ where $s' = s \cup (x = v/\mathsf{d})$. The string increases since $b(s') = b(s)0 > b(s)$. If the state $(F, \mathcal{D}, s)$ causes a unit propagation, then the new state is $(F, \mathcal{D}, s')$ with $b(s') = b(s)1 > b(s)$. In the case of a conflict state $(F, \mathcal{D}, s)$ of decision level d, the solver learns a clause C of assertion level $d' < d$ (unless $d = 0$, in which case the trace ends). The solver gets to state $(F, \mathcal{D}', s')$ where $\mathcal{D}' = \mathcal{D} \cup \{C\}$ and s' is the prefix of s of decision level d', and then immediately unit propagates the asserting literal. Thus the trail of the next state s'' has $b(s'') = b(s')1$, while the prefix of length $|s'| + 1$ of $b(s)$ is equal to $b(s')0$ by construction, and so $b(s'') > b(s)$. □

Theorem 4.1 shows that CDCL without restarts can solve any formula in exponential time and linear space. The simulation works with any decision heuristic and clause learning scheme. It doesn't give much insight in the power of CDCL compared to resolution, though. Could CDCL without restarts simulate tree-like resolution, or even regular resolution? Could it also simulate these with respect to space? These questions are much more difficult, as we need to adapt the strategy to concrete refutations, and show that the simulation is efficient compared to the refutation. Therefore, we consider formula families instead of arbitrary CNFs and show simulation results of CDCL without restarts for these families.

5 Trade-offs for Pebbling Formulas

In this section we discuss trade-offs between space and running time for CDCL refutations of the formulas based on pebble games that we introduced in Section 3. Let us start with a brief review of those pebble games.

The *black-white pebble game* [CS76] is played by a single player on a DAG G with all vertices having indegree at most 2 and with a single sink (i.e., a vertex with no outgoing edges). At each step the player can either place or remove a pebble. A black pebble can be placed on a vertex if all its predecessors have pebbles, and it can be removed at any time. A white pebble can be placed at any time, but it can be removed from a vertex only if all its predecessors contain pebbles. The game starts with the DAG being empty, and the goal is to reach a position where there is a black pebble on the sink and the rest of the graph contains no pebbles. We call a sequence of moves that reaches the goal a *complete pebbling*. The *time* of a pebbling is the number of steps, and the *space* is the maximum number of pebbles in G at any point in time during the pebbling. The *black pebble game* [PH70] that we introduced in Section 3 is a restricted version where only black pebbles are allowed. The interested reader can find more information about pebble games and a comparison of the various flavours of pebbling in the surveys [Pip80, Nor16].

Recall that, given a DAG G, the XORified pebbling formula $Peb_G^{\oplus}$ is defined as follows. For every vertex $v \in V(G)$ there are two variables v_1 and v_2. There are clauses encoding the following: for each source vertex of G, the parity of its variable pair is odd, for each internal vertex, odd parity on all predecessors of the vertex implies odd parity on the vertex itself, and the parity on the sink is even.

More precisely, if we denote as $pred(v)$ the set of all predecessors of v (i.e., vertices that have an outgoing edge toward v), the pebbling formula consists of the set of constraints

$$v_1 \oplus v_2 = 1 \qquad \text{for each source } v, \tag{5.1a}$$

$$\bigwedge_{u \in pred(v)} (u_1 \oplus u_2 = 1) \rightarrow v_1 \oplus v_2 = 1 \qquad \text{for each internal vertex } v, \tag{5.1b}$$

$$z_1 \oplus z_2 = 0 \qquad \text{for the sink } z, \tag{5.1c}$$

expressed in CNF form. We refer to (5.1a) as *source axioms*, to (5.1b) as *pebbling axioms*, and to (5.1c) as *sink axioms*.

It is straightforward to transform a black pebbling of G into a resolution proof for $Peb_G^{\oplus}$. For a vertex v, we call $v_1 \vee v_2$ and $\overline{v}_1 \vee \overline{v}_2$ the *pebble clauses* for v. We maintain the invariant that in the

Procedure Pebble($\mathcal{P}$)

Input: a black pebbling $\mathcal{P}$
foreach *(move,v) in* $\mathcal{P}$ *where* v *is not a source or the sink* **do**
  if *move is Add* **then**
    `HalfPebble` $(v, 0)$
    `HalfPebble` $(v, 1)$
  else
    del $v_1 \vee v_2$
    del $\overline{v}_1 \vee \overline{v}_2$
`PebbleSink`

resolution proof, we have in memory the pebble clauses for all pebbled vertices. Each time a pebble is removed, we erase the corresponding pebble clauses. Correctness of the simulation follows because we can derive the pebble clauses for v from the pebble clauses for all its predecessors, and in a black pebbling, we can only pebble a vertex when all its predecessors are pebbled. This simulation yields the following lemma.

Lemma 5.1 ([BN11]). *If G has a black pebbling of time τ and space s, then $Peb_G^{\oplus}$ has a resolution refutation of length $\mathrm{O}(\tau)$ and clause space $\mathrm{O}(s)$.*

In the other direction, it can be shown (although it is substantially more complicated) that any resolution refutation in length L and space s of $Peb_G^{\oplus}$ can be converted to a pebbling of G with asymptotically the same bounds on time and space, but this requires the full black-white pebble game.

Theorem 5.2 ([BN11]). *From a resolution refutation of $Peb_G^{\oplus}$ of length L and clause space s we can extract a black-white pebbling for G with time $\mathrm{O}(L)$ and space $\mathrm{O}(s)$.*

5.1 Trade-offs with Restarts

MARC'S COMMENT 5: *Assuming we already discussed why we want to do the translation.*

JAKOB'S COMMENT 13: *Now we know what the preceding sections will look like, and so it might be that the text below needs revising in light of this.*

As a warm-up, let us finish showing how to translate black pebblings into CDCL proofs with unrestricted restarts. Recall that our plan in Section 3 was to proceed in the same way that we translate pebblings into resolution proofs. This is, for each pebbled vertex v we keep the pebble clauses in memory, except if v is a source, in which case these clauses are already source axioms of the pebbling formula.

JAKOB'S COMMENT 14: *Writing "we learn the pebble clauses as follows (`HalfPebble`)" sounds strange to me. Do we mean "we learn the pebble clauses as follows (see the pseudo-code in `HalfPebble`)" or "please refer to Figure blah for the details" or something?*

Formally, we generate a CDCL trace using the following procedure. The trace is legal for the 1UIP learning scheme, and we discuss how to use a more general scheme later on.

`HalfPebble` sets the parity of a vertex to even and then learns the clause forbidding that assignment. The example in Figure 2 is actually the translation into resolution of the trace generated by `HalfPebble` $(w, 0)$.

`FindConflicts` generates a conflict between an even vertex and its predecessors. We need to distinguish two cases depending on whether the vertex we are learning has one or two predecessors.

`PebbleSink` is a tweak of `HalfPebble` that accounts for propagations caused by sink axioms.

Using the decision and clause erasure strategies in this procedure, we can establish an analogue of Lemma 5.1 as follows.

Lemma 5.3. *If $\mathcal{P}$ is a black pebbling of G in space s and time τ, then there is a CDCL proof of $Peb_G^{\oplus}$ with restarts and any cutting learning scheme and unit propagation scheme in space at most $2s + 3$ and time $\mathrm{O}(\tau)$.*

Procedure HalfPebble(v, b)

Input: A vertex v, a boolean b

Decide $v_1 = b/\mathsf{d}$
Decide $v_2 = b/\mathsf{d}$
`FindConflicts` (v, b)
Learn $v_1^{1-b} \vee v_2^{1-b}$ and assert $v_2 = 1 - b/\mathsf{u}$
Restart R
foreach *clause* $C \in \mathcal{D} \setminus F$ *such that* $|C| > 2$ **do**
    $\mathsf{del}\, C$

Procedure FindConflict(Case 1, w, b)

Input: A vertex w with predecessor u, a boolean b

Decide $u_1 = 0/\mathsf{d}$
Learn $w_1^{1-b} \vee w_2^{1-b} \vee u_1$ and assert $u_1 = 1/\mathsf{u}$

Procedure FindConflict(Case 2, w, b)

Input: A vertex w with predecessors u and v, a boolean b

Decide $u_1 = 0/\mathsf{d}$
Propagate $u_2 = 1/u_1 \vee u_2$
Decide $v_1 = 0/\mathsf{d}$
Learn $w_1^{1-b} \vee w_2^{1-b} \vee u_1 \vee \overline{u}_2 \vee v_1$ and assert $v_1 = 1/\mathsf{u}$
Learn $w_1^{1-b} \vee w_2^{1-b} \vee u_1$ and assert $u_1 = 1/\mathsf{u}$
Propagate $u_2 = 0/\overline{u}_1 \vee \overline{u}_2$
Decide $v_1 = 0/\mathsf{d}$
Learn $w_1^{1-b} \vee w_2^{1-b} \vee \overline{u}_1 \vee u_2 \vee v_1$ and assert $v_1 = 1/\mathsf{u}$

Procedure PebbleSink

Decide $z_1 = 0/\mathsf{d}$
Propagate $y_z = 0/z_1 \vee \overline{z}_2$
`FindConflicts` (z, 0)
Learn z_1 and assert $z_1 = 1/\mathsf{u}$
Propagate $y_z = 1/z_1 \vee \overline{z}_2$
`FindConflicts` (z, 1)

Proof. The procedure `Pebble` maintains the following invariant. A non-source vertex v has a pebble if and only if the pebbling clauses of v are in the clause database. No other clauses are in memory. The bound on space follows from this invariant, and the bound on time is by construction.

It remains to check that the trace is legal. Observe that the clauses we have in memory only propagate if at least one variable from each vertex they mention is set. Since the decision sequence mentions at most 3 vertices at the same time, we only need to reason about clauses that mention only these vertices.

The correctness of `FindConflicts` is easy to verify. There is no choice of which variable to propagate.

At the end of `FindConflicts` w_1 and w_2 are decisions, and all variables below are set by unit propagation. Since one of the conflicting clauses, a pebbling axiom, contains the decision variables w_1 and w_2, they appear in any cut and therefore any conflict clause. The remaining variables in the conflict graph are of last decision level, so they cannot appear in an asserting clause because w_2 already appears. Therefore we learn the clause $w_1^{1-b} \vee w_2^{1-b}$ and assert $w_2 = 1 - b/\mathsf{u}$.

We only erase clauses after a restart, so we are not erasing clauses involved in unit propagation.

Finally, observe that `PebbleSink` is doing essentially the same as two calls to `HalfPebble`, except that at line 4 there is exactly one decision, z_1, so this is the clause that we learn, and that at the end there is a conflict at level 0, so the proof ends.

To conclude the proof we need to argue that we can replace 1UIP by any cutting learning scheme. The argument to determine that at the end of `FindConflicts` we learn a pebble clause still works.

The remaining clauses that we learn are only used to unit propagate once, so learning any clause that asserts the same literal will serve the same purpose. Since the decision variable appears in a conflicting clause, it is the only possible asserting literal. Therefore we can replace all lines of the form "Learn C and assert $x = b/\mathsf{u}$" by "Learn $L(F, \mathcal{D}, s)$ and assert $x = b/\mathsf{u}$", where L is a given learning scheme and x is the same variable. □

5.2 Trade-offs without Restarts

In Section 5.1, it was shown that CDCL with restarts can do a time and space efficient simulation of any black pebbling. The upper bounds we claim in Section 3 are for the more challenging setting where CDCL is not allowed to restart, however. To establish such results, we need to modify the proofs substantially.

Recall the state at line 4 of `HalfPebble` after learning the first pebble clause of a vertex v. The vertex is set to odd, so there cannot be conflicts among v and its predecessors, and we cannot learn the second pebble clause. Since we are not allowed to restart, the search has to proceed elsewhere. It turns out that we can systematically control the process of learning a clause, moving to another part of the graph, and then learning the remaining clause. However, we have to settle for a restricted class of pebblings and we have to impose some additional structure on them.

We do the conversion from standard pebblings to CDCL refutations in three steps. First we convert the pebbling into a recursive form that suits our arguments better. Unfortunately this conversion needs to be tailored for every graph. The next step is to modify the pebbling to make it easier to simulate. This step is universal, but we need to modify the underlying graph, therefore the formulas that we prove trade-offs for differ slightly from those in [BN11]. The last step is to actually generate a CDCL trace from the pebbling.

We are going to show the specific form of black pebbling that has an efficient translation in the CDCL proof system in Section 5.3. In Section 5.4 we are going to explain the simulation itself. Finally in Section 5.5 we will revisit graphs with known time/space trade-offs and recast the time- and space-efficient pebblings of these graphs into our new framework.

5.3 Binary Tree Pebbling

MASSIMO'S COMMENT 5: *STATUS: This section is mostly about terminology. I would like someone else to read it and edit them.*

We can express our conditions more easily if we think of a pebbling recursively. Therefore we first introduce the notion of binary tree pebbling as a useful language to represent black pebbling strategies with a recursive structure.

In this section we denote by $\mathcal{T}$ a rooted binary tree with children ordered left and right. We use the convention that a single child is the left child. We also assume that $G = (V, E)$ is a DAG with exactly one sink and such that each vertex has at most two incoming edges. To avoid confusion between a DAG G and a tree $\mathcal{T}$ describing a pebbling of G, we say that G has *vertices*, which we usually denote by u, v, w, and $\mathcal{T}$ has *nodes*, which we denote by p, q, l, r.

The *left postfix order* of the nodes of a tree $\mathcal{T}$ is the total order $(p_1, p_2, \ldots, p_{|\mathcal{T}|})$ induced by a traversal of $\mathcal{T}$ that first visits the left subtree, then the right subtree (if any), and then the root. We say that p_i *comes at time* i in the traversal and we denote as $p < p'$ the fact that a node p comes at an earlier time than node p'.

Definition 5.4 (Binary tree pebbling). A *binary tree pebbling* of G is a binary tree $\mathcal{T}$ whose nodes are labelled by vertices of G according to a label function ℓ and such that the following holds:

1. if p is an *internal* node of $\mathcal{T}$ the label function ℓ forms a bijection between the children of p and the predecessors of $\ell(p)$ in G;

2. if p is a leaf of $\mathcal{T}$, then either $\ell(p)$ is a source vertex of G or there is an internal node $q < p$ such that $\ell(p) = \ell(q)$; we call the latest such q the *cache node* for p.

The time of a binary tree pebbling is its order.

A binary tree pebbling corresponds to a generalized depth first search of G that starts at the sink and that can mark a vertex as "visited" only if all its predecessors are maked as "visited" and that, unlike standard DFS, can remove the "visited" mark at any stage of the process. If the process reaches any such vertex again, it has to visit it again, as if it had never reached it before.

The converse also holds: a standard depth first traversal corresponds to a pebbling strategy where we remember all "visited" nodes. Indeed, consider the spanning tree produced by a depth first traversal of a graph, except that instead of discarding a forward edge, we add a new leaf labelled as the already visited node the edge is pointing to. It is easy to verify that such binary tree is a pebbling and that its order is at most $|V| + |E|$.

Proposition 5.5. *Every DAG has a binary tree pebbling of time at most* $|V| + |E|$.

Although we are more interested in the reverse direction, let us first show that that binary tree pebblings are essentially black pebblings. We extract a pebbling from a binary tree pebbling by traversing the tree in postorder and keeping a pebble on a vertex as long as it is useful.

Definition 5.6 (Node lifespan). The *lifespan* of a non-root internal node p_i is the maximal interval $[i, j]$ such that p_j is the parent of either p_i or a non-source leaf whose cache node is p_i.

The lifespan of a leaf node p_i is only interesting when $\ell(p_i)$ is a source of G, and it is $[i, j]$ where p_j is the parent node of p_i. If a leaf node p_i is labelled by a non source vertex of G we say by convention that its lifespan is the empty interval. The lifespan of the root node is simply $[|\mathcal{T}|, |\mathcal{T}|]$.

We say that some node p_i is *alive at time* t if it has lifespan $[i, j]$ with $i \leq t \leq j$.

Definition 5.7. The space of a binary pebbling tree $\mathcal{T}$, denoted as $space(\mathcal{T})$, is the maximum number of nodes alive at time t for $1 \leq t \leq |\mathcal{T}|$.

Proposition 5.8. *If a DAG* G *has a binary tree pebbling* $\mathcal{T}$*, then* G *has a black pebbling with space* $space(\mathcal{T})$ *and time at most* $2|\mathcal{T}|$.

Proof. Consider a binary tree pebbling $\mathcal{T}$ and the corresponding left postfix order $(p_1, p_2, \ldots p_{|\mathcal{T}|})$. While we scan this sequence we build a pebbling of G as follows. At time t we consider whether

node p_t has a lifespan $[t, j]$ for some $j \geq t$. In this case we place a pebble on $\ell(p_t)$. Then, for each node p_i with lifespan $[i, t]$, we remove the pebble on $\ell(p_i)$.

This pebbling has space exactly $space(\mathcal{T})$ and achieves the goal to place a pebble on the sink of G. At each node we place at most one pebble and we remove as many pebbles as we place, so the number of moves is at most $2|\mathcal{T}|$. Now we show that it is legal, i.e., for every node p_t with parent $p_{t'}$ there is a pebble on $\ell(p_t)$ at least until time t'. This guarantees that every pebble placement is legal. If p_t has a lifespan $[t, s']$ then by definition $s' \geq t'$. If p_t has no lifespan it means that there is a previous internal node p_s with $\ell(p_t) = \ell(p_s)$ with lifespan $[s, s']$ and $s' \geq t'$. □

Finally we discuss the strategy opposite to that in Proposition 5.5: discard pebbles as soon as possible.

JANE'S COMMENT 2: *Changed $\max\{d+1, 3\}$ to $d+2$ below because I think that is the maximum space: for example if $d = 2$, you need to use 3 pebbles in the right subtree while there's also one waiting on the root of the left subtree.*

Proposition 5.9. *Every DAG with depth d has a binary tree pebbling of time at most $2^{d+1} - 1$ and space $d + 2$, in which all leaves are labelled by source vertices.*

Proof. If the graph has depth 0 it consists of a single vertex and we are done. Otherwise consider the sink vertex u and assume without loss of generality that it has two children, v and w. The binary tree pebbling has the root node labelled by u, and two subtrees of order $2^d - 1$ and space $d+1$ each, obtained by considering the binary tree pebbling of the subgraph of the ancestors of v and w, respectively. The total order of the tree is $2^{d+1} - 1$ and the total space is $d + 2$ since the node at the root of the left subtree is the only node in the left subtree which is alive during the whole visit of the right subtree. When the root is visited, both its left and right children are alive, so the space must be at least 3, but $d + 2 \geq 3$ if $d \geq 1$. □

MASSIMO'S COMMENT 6: *STATUS: as in Section 5.3. Maybe this subsubsection could be shortened with some editing effort?*

In order to do the translation from a pebbling strategy to a CDCL refutation we need the binary tree pebbling to fulfill some further properties.

Definition 5.10 (Robustness). We say that a binary tree pebbling $\mathcal{T}$ of graph G is *robust* if satisfies the following additional properties.

1. Each internal node has exactly two children;
2. if p is an internal node of $\mathcal{T}$, and l and r its left and right children, then for every node $q \in \mathcal{T}_r$ it holds that $\ell(l) \notin \{\ell(q)\} \cup pred_G(\ell(q))$; and
3. if an internal node q is the right child of l, then q is not the cache node of any non-source leaf in the subtree rooted in r.

MARC'S COMMENT 6: *I had to hack condition 2*

Enforcing the robustness property is necessary but luckily not very problematic. We will argue that is is possible to modify any graph G into a new graph $r(G)$ so that any binary tree pebbling $\mathcal{T}$ for G can be turned into a binary tree pebbling $r(\mathcal{T})$ for $r(G)$ with negligible penalty in space and time.

We split every edge of the original graph in two, by adding a new vertex, and then we add a new source vertex as a new predecessors of all vertices of incoming degree one (this includes the vertices we just added to split the edges). Formally, the robust graph is the following.

Definition 5.11. We define $r(G) = (V', E' \cup E'')$ where $V' = V \cup E \cup \{*\}$ and

$$E' = \{(v, e), (e, v') \mid e \in E \text{ and } e = (v, v')\}$$
$$E'' = \{(*, u) \mid u \in V \text{ and } |pred(u)| = 1 \text{ or } u \in E\} \ .$$

Proposition 5.12.

MASSIMO'S COMMENT 7: *Update it to the new indegree.*

MARC'S COMMENT 7: *Is this comment still relevant?*

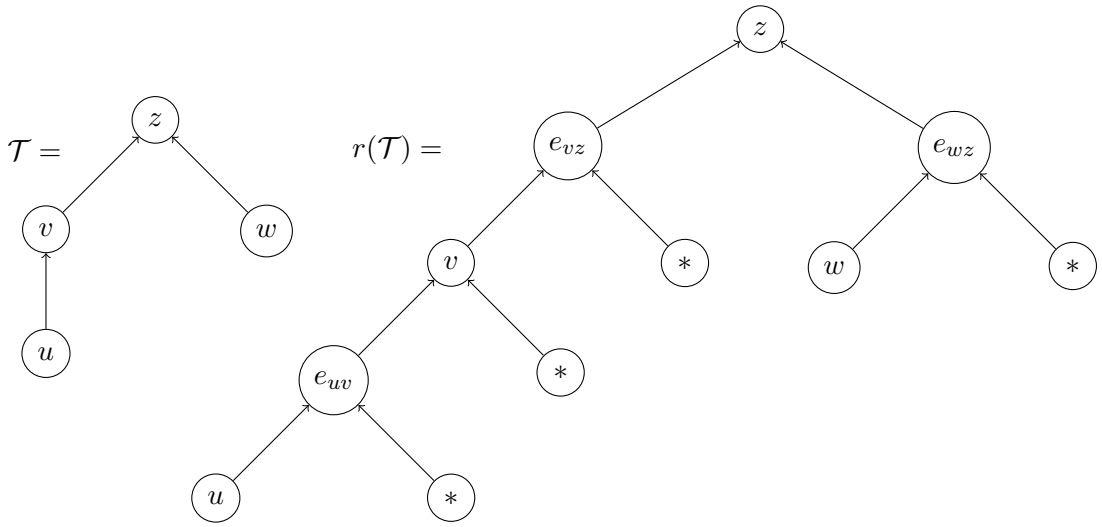


Figure 9: Robust transform of a binary pebbling

For every black (resp. black-white) pebbling of G of time τ and space s there is a black (resp. black-white) pebbling of $r(G)$ of time $\mathrm{O}(\tau)$ and space $s+2$. For every black (resp. black-white) pebbling of $r(G)$ of time τ and space s there is a black (resp. black-white) pebbling of G of time $\mathrm{O}(\tau)$ and space s.

Proof. The transformation of a pebbling for G into a pebbling for $r(G)$ is trivial. For the opposite direction consider a pebbling for $r(G)$, either black or black-white. We will build a pebbling of G of the claimed length and space, translating pebbling moves in $r(G)$ into moves for G. First we remove vertex $*$ from $r(G)$ to get a new graph $r(G)'$. A vertex removal can only decrease pebbling length and space.

Now we transform the pebbling of $r(G)'$ into a pebbling of G, and we maintain the invariants that: (a) whenever some vertex $e = (v, v')$ has a pebble in the original pebbling, then v must be pebbled in the new pebbling; (b) if a vertex v, coming from the original graph G, has a pebble, then it has a pebble also in the new pebbling; and (c) the pebble on v is white only if there is a white pebble on v in the original pebbling.

When a white pebble is placed on vertex e we instead place a pebble on v, unless it is already pebbled. When a black pebble is placed on vertex e we don't do anything since v must have been pebbled at that point. If a white pebble is placed on v in $r(G)$ we do the same in G unless there is already a black pebble there. If a black pebble is placed on v in $r(G)$ then in G we have a pebble on each predecessor of v. We can then black pebble v in G as well (after removing any white pebble on the same vertex). If a pebble is removed from v, we remove it from G only if in $r(G)$ the vertices representing all the outgoing edges of v in G are empty. If the pebble was white then it means that predecessors of v in $r(G)$ (and then in G) are pebbled. So we turn the pebble black. If e is unpebbled and neither v nor any other e' successors of v are pebbled in the original pebbling, then in our pebbling we remove the pebble over v, which is black because v does not have a pebble in the original pebbling.

This pebbling of G has no more pebbles than the one of $r(G)$, and we always turn one move over $r(G)$ into $\mathrm{O}(1)$ moves on G. □

Lemma 5.13. *For any binary tree pebbling $\mathcal{T}$ of G it is possible to efficiently build a robust binary tree pebbling $r(\mathcal{T})$ of $r(G)$ with space $\mathrm{O}(space(\mathcal{T}))$ and time $\mathrm{O}(|\mathcal{T}|)$.*

Proof. We build a robust binary tree pebbling $r(\mathcal{T})$ from $\mathcal{T}$ by first splitting edges in two and then adding a node labelled by $*$ as the right child of every node of incoming degree 1, in a way that matches what we did for G in Definition 5.11.

More specifically each edge $e = (s, t)$ of $\mathcal{T}$ is substituted by the path of three nodes s, p_e, t, where s and t are the original nodes of $\mathcal{T}$ and p_e is a new one, labelled by $(\ell(s), \ell(t))$. If s was the left (resp. right) child of t then p_e is the new left (resp. right) child of t. In any case s is always the left child of p_e.

Procedure Cleanup(p_j)

foreach *clause* $C \in \mathcal{D} \setminus F \setminus \bigcup_{p \text{ alive at time } j} \mathit{scaffolding}(p, j)$ **do**
  del C

To complete the construction we add a new right child node $p_{v,*}$, labelled by $*$, to any internal node v of $\mathcal{T}$ without a right child. The order of $r(\mathcal{T})$ is at most $\mathrm{O}(|\mathcal{T}|)$ by construction.

To analyze the space of $r(\mathcal{T})$ we observe a few simple facts from the construction: (a) each of the labels of the new nodes of type p_e occur only once; (b) the left postfix order of $\mathcal{T}$ is the projection of $r(\mathcal{T})$ over the nodes in $V(\mathcal{T})$. These facts, together with Definition 5.6, imply that the lifespan of each $p \in V(\mathcal{T})$ in the visit of $r(\mathcal{T})$ contains the projection of the lifespan of the same node p in the visit of $\mathcal{T}$. Therefore no more than $\mathit{space}(\mathcal{T})$ nodes in $V(\mathcal{T})$ can be simultaneously alive.

The new nodes $p_{v,*}$ are alive just until the next step. The lifespan of a node p_e with $e = (p, q)$ ends no later than when q is visited. For comparison, in the left postfix order of $\mathcal{T}$ the lifespan of p ends no earlier than at the visit of q. Thus we can charge each p_e to the lifespan of p in the left postfix order of $\mathcal{T}$. Since the mapping from p_e to p is injective, the total contribution of these nodes is $\mathit{space}(\mathcal{T})$.

In the end we get that $\mathit{space}(r(\mathcal{T})) = 2\mathit{space}(\mathcal{T}) + \mathrm{O}(1)$.

To conclude we need to show that $r(\mathcal{T})$ is indeed robust. It is easy to see that the new tree respects all conditions of Definition 5.4 and that every internal node has two children, but the other conditions in 5.10 require an explicit proof.

Condition 2 is verified immediately for nodes for types p_e and $p_{u,*}$ that have trivial (or null) right subtrees. For an arbitrary node p labelled by a vertex u, its left child is labelled by some vertex e which corresponds to an edge in G, and such label occur only once in $r(\mathcal{T})$, and in particular cannot occur in the right subtree of p. Furthermore, the only node that has e as a predecessor is u, so e does not appear as a predecessor of a node in the right subtree of p either.

With respect to condition 3, the only nodes that appear as right children are labelled by $*$, which is not a cache node because it is a source, and by edges of G, which are not cache nodes either because leaves are labelled after vertices in G. □

5.4 CDCL Trace from a Tree Pebbling

MASSIMO'S COMMENT 8: *STATUS: I am quite confident of the soundness of this Subsection, but I would like someone else to read the proofs and edit them. This is the most important technical part of this Section (and maybe of the paper).*

MARC'S COMMENT 8: *I am also quite confident that the result holds and the algorithms are correct, but I find it very likely that the proof has bugs and omissions.*

In this section we consider a DAG G and a robust binary tree pebbling $\mathcal{T}$ for it. We want to describe a CDCL proof that refutes $\mathit{Peb}_G^{\oplus}$ in time and space proportional to the time and space of $\mathcal{T}$, respectively.

We build the proof recursively by traversing the binary tree pebbling according to the left postfix order. Ideally we would like to learn the pebble clauses for the label of each visited node and keep such clauses for the lifespan of the node itself. As we argued, this is not possible without restarts, but we roughly achieve this goal by learning the clauses corresponding to some ancestors of the visited node within a fixed constant distance. This results in a constant number of binary clauses per visited node.

More precisely, we define the *scaffolding* of a node p alive at time j as the set of descendants of p that have a pebble at time j and are reachable from p by a path of nodes that do not have a pebble. For instance, the scaffolding of a pebbled node is the node itself, and the scaffolding of an unpebbled node whose two predecessors are pebbled are its predecessors. We will ensure that the size of any scaffolding never exceeds 3.

Our deletion strategy is, after processing p_j, to keep the scaffolding of each node alive at time j and to erase every other clause.

It is important to delete clauses in a timely manner, not only for keeping the size of the clause database in shape but to avoid spurious unit propagations.

Procedure QuickVisit(Case 1, b, $\mathcal{T}$)

Input: a robust binary tree pebbling $\mathcal{T}$ with root p.

1 Decide $\ell(l)_1 = 0/\mathsf{d}$
2 Propagate $\ell(l)_2 = 1/\ell(l)_1 \vee \ell(l)_2$
3 Decide $\ell(r)_1 = 0/\mathsf{d}$
4 Learn $\ell(p)_1^{1-b} \vee \ell(p)_2^{1-b} \vee \ell(l)_1 \vee \overline{\ell(l)}_2 \vee \ell(r)_1$ and assert $\ell(r)_1 = 1/\mathsf{u}$
5 Learn $\ell(p)_1^{1-b} \vee \ell(p)_2^{1-b} \vee \ell(l)_1$ and assert $\ell(l)_1 = 1/\mathsf{u}$
6 Propagate $\ell(l)_2 = 0/\overline{\ell(l)}_1 \vee \overline{\ell(l)}_2$
7 Decide $\ell(r)_1 = 0/\mathsf{d}$
8 Learn $\ell(p)_1^{1-b} \vee \ell(p)_2^{1-b} \vee \ell(l)_1 \vee \overline{\ell(l)}_2 \vee \ell(r)_1$ and assert $\ell(r)_1 = 1/\mathsf{u}$

Procedure QuickVisit(Case 2, b, $\mathcal{T}$)

Input: a robust binary tree pebbling $\mathcal{T}$ with root p.

1 Decide $\ell(l)_1 = 0/\mathsf{d}$
2 Decide $\ell(l)_2 = 0/\mathsf{d}$
3 `QuickVisit` (Case 1, $b = 0$, $\mathcal{T}_l$)
4 Learn $\ell(l)_1 \vee \ell(l)_2$ and assert $\ell(l)_2 = 1/\mathsf{u}$
5 Decide $\ell(r)_1 = 0/\mathsf{d}$
6 Learn $\ell(p)_1 \vee \ell(p)_2 \vee \ell(l)_1 \vee \overline{\ell(l)}_2 \vee \ell(r)_1$ and assert $\ell(r)_1 = 1/\mathsf{u}$
7 Learn $\ell(p)_1 \vee \ell(p)_2 \vee \ell(l)_1$ and assert $\ell(l)_1 = 1/\mathsf{u}$
8 Decide $\ell(l)_2 = 1/\mathsf{d}$
9 `QuickVisit` (Case 1, $b = 1$, $\mathcal{T}_l$)
10 Learn $\overline{\ell(l)}_1 \vee \overline{\ell(l)}_2$ and assert $\ell(l)_2 = 0/\mathsf{u}$
11 Decide $\ell(r)_1 = 0/\mathsf{d}$
12 Learn $\ell(p)_1 \vee \ell(p)_2 \vee \ell(l)_1 \vee \overline{\ell(l)}_2 \vee \ell(r)_1$ and assert $\ell(r)_1 = 1/\mathsf{u}$

In the following text, we assume that $\mathcal{T}'$ is a subtree of $\mathcal{T}$, with root p and children l and q. We say that a node p is pebbled if the pebble clauses of $\ell(p)$ are in the clause database. We also assume the learning scheme to be 1UIP, and we appeal to the argument in Section 5.1 to show that this is without loss of generality.

To learn pebble clauses we will use two auxiliary procedures, `DeepVisit` and `QuickVisit`. The former is defined recursively and it is the main component of our CDCL refutation. The latter is the base case and only works for some specific trees of constant size. A visit procedure starts with p set to even, sets some variables in $\mathcal{T}'$, and ends with a conflict at the level of $\ell(p)_2$ or earlier. It is the responsibility of the caller to learn the appropriate clause.

We use `QuickVisit` in the following three cases, where we assume that p is not pebbled.

1. l and r have a pebble.
2. r has a pebble, l does not, but it has two children, l_1 and l_2, that are pebbled.
3. l has a pebble, r does not, but it has two children, r_1 and r_2, that are pebbled.

Case 1 of `QuickVisit` is essentially `FindConflicts` from Section 5.1 with the notation adapted to binary pebbling.

In Case 2 we also generate a conflict with the predecessors of a node and we pebble the left node as a side-effect. First we learn the clause $\ell(l)_1 \vee \ell(l)_2$ using Case 1 `QuickVisit`, analogously to `HalfPebble`. Then instead of restarting we generate a conflict earlier in the trail, with the same goal of clearing the left subtree. Now we can apply Case 1 again and learn $\overline{\ell(l)}_1 \vee \overline{\ell(l)}_2$. Finally we generate another conflict and backjump.

Finally, Case 3 is essentially the same as Case 2, flipping the left and the right subtrees.

The recursive procedure for large trees is `DeepVisit`. In order to learn the pebble needed at some node p of the binary tree pebbling, we need to visit both subtrees multiple times. The first time we visit

each subtree we use `DeepVisit` which, as a side effect, learns the pebbles of the nodes at a constant distance from p. Afterwards, since a pebbled node behaves as a source, the subtrees become effectively of constant size. We can use `QuickVisit` where appropriate to generate local conflicts and end with a backjump.

Procedure DeepVisit($\mathcal{T}$)

Input: a robust binary tree pebbling $\mathcal{T}$ with root p.

If $\mathcal{T}$ meets the requirements use `QuickVisit` instead, otherwise go ahead.
Decide $\ell(l)_1 = 0/\mathsf{d}$
if $\ell(l)$ *is pebbled* **then**
    Propagate $\ell(l)_2 = 1/\ell(l)_1 \vee \ell(l)_2$
else
    Decide $\ell(l)_2 = 0/\mathsf{d}$
    `DeepVisit` ($\mathcal{T}_l$)
    Learn $\ell(l)_1 \vee \ell(l)_2$ and assert $\ell(l)_2 = 1/\mathsf{u}$
    `Cleanup` (l)
Decide $\ell(r)_1 = 0/\mathsf{d}$
if $\ell(r)$ *is not pebbled* **then**
    Propagate $\ell(r)_2 = 0/\ell(p)_1 \vee \ell(p)_2 \vee \ell(l)_1 \vee \overline{\ell(l)}_2 \vee \ell(r)_1 \vee \overline{\ell(r)}_2$
    `DeepVisit` ($\mathcal{T}_r$)
Learn $\ell(p)_1 \vee \ell(p)_2 \vee \ell(l)_1 \vee \overline{\ell(l)}_2 \vee \ell(r)_1$ and assert $\ell(r)_1 = 1/u$
if $\ell(r)$ *is not pebbled* **then**
    Propagate $\ell(r)_2 = 1/\ell(p)_1 \vee \ell(p)_2 \vee \ell(l)_1 \vee \overline{\ell(l)}_2 \vee \ell(r)_1 \vee \overline{\ell(r)}_2$
    `Cleanup` (r)
    `QuickVisit` ($\mathcal{T}_r, b = 1$)
Learn $\ell(p)_1 \vee \ell(p)_2 \vee \ell(l)_1$ and assert $\ell(l)_1 = 1/\mathsf{u}$
if $\ell(l)$ *is pebbled* **then**
    Propagate $\ell(l)_2 = 1/\overline{\ell(l)}_1 \vee \overline{\ell(l)}_2$
else
    Decide $\ell(l)_2 = 1/\mathsf{d}$
    `QuickVisit` ($\mathcal{T}_l, b = 1$)
    Learn $\overline{\ell(l)}_1 \vee \overline{\ell(l)}_2$ and assert $\ell(l)_2 = 0/u$
Decide $\ell(r)_1 = 0/\mathsf{d}$
if $\ell(r)$ *is not pebbled* **then**
    Propagate $\ell(r)_2 = 0/\ell(p)_1 \vee \ell(p)_2 \vee \overline{\ell(l)}_1 \vee \ell(l)_2 \vee \ell(r)_1 \vee \overline{\ell(r)}_2$
    `QuickVisit` ($\mathcal{T}_r, b = 0$)
Learn $\ell(p)_1 \vee \ell(p)_2 \vee \ell(l)_2 \vee \overline{\ell(l)}_1 \vee \ell(r)_1$ and assert $\ell(r)_1 = 1/\mathsf{u}$
if $\ell(r)$ *is not pebbled* **then**
    Propagate $\ell(r)_2 = 1/\ell(p)_1 \vee \ell(p)_2 \vee \overline{\ell(l)}_1 \vee \ell(l)_2 \vee \ell(r)_1 \vee \overline{\ell(r)}_2$
    `QuickVisit` ($\mathcal{T}_r, b = 1$)

Finally we pebble the root and start the recursive procedure in `BinaryPebble`.

We can now establish the technical lemma that is the main goal of this section.

Lemma 5.14 (CDCL refutation). *If $\mathcal{T}$ is a robust binary tree pebbling of a graph G, then* `BinaryPebble` *($\mathcal{T}$) is a CDCL proof of $Peb_G^{\oplus}$ without restarts and with any cutting learning scheme and unit propagation scheme in time $\mathrm{O}(|\mathcal{T}|)$ and space at most $\mathrm{O}(space(\mathcal{T}))$*

The number of steps in the trace is $\mathrm{O}(|\mathcal{T}|)$ because `QuickVisit` always runs in constant time and `DeepVisit` calls itself recursively at most once on each subtree. Furthermore, all conflicts involve a constant number of variables, so all derivations are of constant size as well.

To measure the space we need the following invariant.

Claim 5.15. After we run `DeepVisit` on a subtree rooted in p, with left and right children l and r, we store a pebble on l and on either r or its two children.

Procedure BinaryPebble($\mathcal{T}$)

Input: a robust binary tree pebbling $\mathcal{T}$ with root p.

1 Decide $\ell(p)_1 = 0/\mathsf{d}$
2 Propagate $\ell(p)_2 = 0/\ell(p)_1 \vee \overline{\ell(p)}_2$
3 `DeepVisit` ($\mathcal{T}$)
4 Learn $\ell(p)_1$ and assert $\ell(p)_1 = 1/\mathsf{u}$
5 `QuickVisit` ($\mathcal{T}, b = 1$)

Claim 5.16. The scaffolding of every node alive at time j has size at most 3.

Proof. Observe that the order in which `DeepVisit` processes nodes is the left postfix order of $\mathcal{T}$, except that some of the calls to `DeepVisit` actually result in calls to `QuickVisit` on some constant size subtrees. Therefore every node alive at time j has been visited. □

This proves that there are at most 6 times more pebbling clauses than alive nodes in memory. The remaining learned clauses in the database are used for unit propagation. `QuickVisit` involves a constant number of clauses, and propagations in `DeepVisit` only occur because of pebble clauses, which we already accounted for, and pebbling axioms, which are not learned. Putting all these observations together we have that during the CDCL procedure we remember $\mathrm{O}(1)$ clauses per alive node, plus an additional $\mathrm{O}(1)$ clauses. This amounts to space $\mathrm{O}(\mathit{space}(\mathcal{T}))$.

To conclude the proof of Lemma 5.14 we have to show that the trace is legal. Before a call to a visit procedure of $\mathcal{T}'$, the following invariants hold.

Claim 5.17. The trail assigns all nodes in the path from the root of $\mathcal{T}$ to p to even, and possibly their left siblings to odd. No other node is assigned.

Claim 5.18. No other vertex labelled in $\mathcal{T}'$ has a variable assigned in the trail.

Proof. The nodes that are assigned in the trail do not appear in $\mathcal{T}'$ by conditions 2 and 3 of Definition 5.10. □

It is straightforward to check the propagations and conflicts listed in `QuickVisit`, and in Claim 5.20 we will show that these are the only that happen.

At the last conflict all variables below $\ell(p)_2$ are set by unit propagation. If $\ell(p)_2 = b/\mathsf{d}$ is a decision, since the conflicting clause $\ell(p)_1^{1-b} \vee \ell(p)_2^{1-b} \vee \overline{\ell(l)}_1 \vee \ell(l)_2 \vee \overline{\ell(r)}_1 \vee \ell(r)_2$ contains the decision variables $\ell(p)_1$ and $\ell(p)_2$, they appear in any cut and therefore any conflict clause. The remaining variables in the conflict graph are of last decision level, so they cannot appear in an asserting clause because $\ell(p)_2$ already appears. Therefore we learn the clause $\ell(p)_1^{1-b} \vee \ell(p)_2^{1-b}$ and assert $\ell(p)_2 = 1 - b/\mathsf{u}$. If $\ell(p)_2$ is a propagation we backjump to an earlier point.

In Case 2 we also learn clauses $\overline{\ell(l)}_1 \vee \overline{\ell(l)}_2$ and $\ell(l)_1 \vee \ell(l)_2$, so at the end of `QuickVisit` the vertex $\ell(l)$ is pebbled. Analogously, the vertex $\ell(r)$ is pebbled in Case 3.

It is also easy to verify that the propagations and conflicts in `DeepVisit` are correct. After line 14 both $\mathcal{T}_l$ and $\mathcal{T}_r$ have been visited, and both l and r are alive nodes when we visit p. This means that we store pebble clauses for their four immediate predecessors in the database, by induction hypothesis, therefore the subsequent calls to `QuickVisit` are legal. At the end of the procedure we have the pebble clauses that we claim in 5.15 in the database. We learn the pebble clauses for $\ell(l)$ in lines 8 and 25, and we learn the pebble clauses for the children of $\ell(r)$ in the call to `QuickVisit` at line 18.

There is a detail that we have swept under the carpet up to this point. We claimed that the base case of `DeepVisit` is `QuickVisit`, but this is only the case if leaves are pebbled; otherwise `DeepVisit` would try to access an empty tree.

Claim 5.19. When we query whether a leaf is pebbled, the answer is yes.

Proof. Leaves labelled by source vertices are always pebbled. For a non-source leaf, we only query its state when its cache node is alive, but there is some time during which a node is alive and not pebbled. Consider a cache node q. We need to distinguish three cases.

1. If q is the left child of a node p_j, then q will have a pebble at time j. During that time we visit the right subtree of p_j, but by condition 2 of Definition 5.10 $\ell(q)$ does not appear at all in that subtree.

2. If q is the right child of a node l, which is the left child of a node p_j, then q will have a pebble at time j. During that time we visit the right subtree of p_j, but by condition 3 of Definition 5.10 $\ell(q)$ does not appear as a leaf in that subtree.

3. If q is the right child of a node r, which is the right child of a node p_j, then q will have a pebble at time j and we will not visit any subtree during that time.

□

Finally we show that no other unit propagation or conflict occurs.

Claim 5.20. All possible unit propagations and conflicts are listed in `QuickVisit` and `DeepVisit`.

Proof. First we claim that all propagations and conflicts caused by pebble clauses, both learned and source axioms, are accounted for. Indeed, whenever a variable x_u is set, the next step depends on whether u has a pebble. The remaining learned clauses are erased as soon as they stop unit propagating, so they do not pose a problem either.

Regarding pebbling axioms, we show that when we visit a node p, only those axioms with support in p and its predecessors cause conflicts. Consider an axiom in $x_u \oplus y_u = 1 \wedge x_v \oplus y_v = 1 \rightarrow x_w \oplus y_w = 1$ such that two of its nodes are set and do not satisfy the axiom. We distinguish two cases.

1. u and v are set to odd.

2. u (or v) is set to odd and w to even.

In Case 1, by Claim 5.17 and condition 2 of Definition 5.10, u and v are not the predecessors of any node in the current subtree. Therefore, w is not set and no propagation happens.

In Case 2, since w is set to even, it is not pebbled, so by Claim 5.19 it is an internal node. By Claim 5.17, the only reason v is not set is that it is the node that we are currently visiting. Therefore, we account for propagations and conflicts. □

5.5 CDCL Trade-offs for Pebbling Formulas

MASSIMO'S COMMENT 9: *This section is done. The time efficient upper bounds for CS graph got a little bit too long. Please read it and fix any minor issue.*

We obtain trade-offs for CDCL refutations from trade-offs between pebbling time and pebbling space for DAGs. In particular our CDCL trade-offs will come from pebbling trade-offs for (variants of) Carlson-Savage graphs [CS80, CS82], stacks of superconcentrators [LT82] and bit reversal graphs [LT82]. The upper bounds in this section apply to all CDCL systems regardless of the learning scheme, as long as it is cutting, and without restarts. The lower bounds apply to any CDCL system, regardless of the learning scheme and of the restart policy.

Lemma 5.21. *Let G be some directed acyclic graph with indegree at most 2.*

1. *If G has a binary tree pebbling $\mathcal{T}$ then $Peb^{\oplus}_{r(G)}$ has a CDCL refutation without restarts, with any cutting learning scheme of time $\mathrm{O}(|\mathcal{T}|)$ and space $\mathrm{O}(space(\mathcal{T}))$*

2. *Given any CDCL refutation of $Peb^{\oplus}_{r(G)}$ of time τ and space s, we can extract a black-white pebbling for G of time $\mathrm{O}(\tau)$ and space $\mathrm{O}(s)$.*

Proof. To prove item 1 we use Lemma 5.13 to get a robust binary tree pebbling $r(\mathcal{T})$ for $r(G)$ of time $\mathrm{O}(|\mathcal{T}|)$ and space $\mathrm{O}(space(\mathcal{T}))$. Then Lemma 5.14 immediately gives the CDCL refutation $Peb^{\oplus}_{r(G)}$ of time $\mathrm{O}(|\mathcal{T}|)$ and space $\mathrm{O}(space(\mathcal{T}))$. To prove item 2 we observe that a CDCL refutation is indeed a resolution refutation, therefore we can use Theorem 5.2 to get a black-white pebbling for graph $r(G)$ of time $\mathrm{O}(\tau)$ and space $\mathrm{O}(s)$. Using Proposition 5.12 we obtain a black-white pebbling of G of time $\mathrm{O}(\tau)$ and space $\mathrm{O}(s)$ as well. □

The first trade-off comes from *stacks of superconcentrators*, a graph family originally defined in [LT82].

A DAG is a *superconcentrator* if it has m sources $S = \{s_1, \ldots, s_m\}$, m sinks $Z = \{z_1, \ldots, z_m\}$, and for any subsets $S' \subseteq S$ and $Z' \subseteq Z$ with $|S'| = |Z'|$ there are $|S'|$ vertex disjoint paths, each connecting one source in S' to some sink in Z'.

In literature there are efficient constructions of superconcentrators of indegree 2 with m sources and sinks, $\mathrm{O}(m)$ edges and depth $\mathrm{O}(\log m)$, for every $m = C \cdot 2^k$ where C is a fixed value depending of the construction and k is a free parameter (see for example [AC03]).

Let $\mathrm{SC}^{(1)}_m, \ldots, \mathrm{SC}^{(r)}_m$ denote r copies of an efficiently constructible superconcentrator with m sources and sinks, with $m = \Theta(2^k)$ for some $k > 0$, indegree 2, $\mathrm{O}(m)$ edges and depth $\mathrm{O}(\log m)$. The graph $\Phi(m, r)$ is constructed by placing these r copies on top of one another, i.e., with the sinks of $z^j_1, z^j_2, \ldots, z^j_m$ of $\mathrm{SC}^{(j)}_m$ connected to the sources $s^{j+1}_1, s^{j+1}_2, \ldots, s^{j+1}_m$ of $\mathrm{SC}^{(j+1)}_m$ with edges (z^j_i, s^{j+1}_i) for $i = 1, \ldots, m$ and $j = 1, \ldots, r-1$.

Theorem 5.22 (Trade-off for stack of superconcentrators). *There exists an efficiently constructible family of 3-CNF formulas F_n of size $\Theta(n)$ that have*

- *a CDCL refutation without restarts and with any cutting learning scheme in time $\mathrm{O}(n)$ and space $\mathrm{O}(n/\log n)$ simultaneously;*
- *a CDCL refutation without restarts and with any cutting learning scheme in space $\mathrm{O}((\log n)^2)$ and time $n^{\mathrm{O}(\log n)}$ simultaneously.*

Moreover there exists a constant $K > 0$ such that any CDCL refutation in space $s \leq Kn/\log n$ satisfies that time τ is at least $n^{\Omega(\log\log n)}$, regardless of the learning scheme and of the restart policy.

Proof. The formula family F_n for which we prove this theorem is $Peb^{\oplus}_{G_n}$, where we build the graph G_n as follows. We first start with a stack of super concentrators $\Phi(m, r)$, where we set the parameters $m = \Theta(n/\log n)$ and $r = \Theta(\log n)$, and then we add an additional binary tree on top to make the graph have a single sink. More specifically we put on top a complete binary tree of depth $\log m$ where the sources are denoted $s^{r+1}_1, s^{r+1}_2, \ldots, s^{r+1}_m$ and the sink as z^{r+1}_1. The sinks of the topmost layer of the superconcentrator are connected to this tree through edges (z^r_i, s^{r+1}_i). Let us denote this graph as $\widehat{\Phi}(m, r)$ for the rest of the proof.

In [LT82] it is shown that any black-white pebbling for $\Phi(m, r)$ that has space $s \leq m/20$ requires time $\tau \geq m \cdot \left(\frac{rm}{64s}\right)^r$. Since any black-white pebbling for $\widehat{\Phi}(m, r)$ induces a black-white pebbling for $\Phi(m, r)$ the result holds for $\widehat{\Phi}(m, r)$ too. In particular this means that there is a constant K such that any black-white pebbling with space at most $Kn/\log n$ takes time at least $n^{\Omega(\log\log n)}$.

We show two binary tree pebblings for $\widehat{\Phi}(m, r)$: a time efficient one, with simultaneous time $\mathrm{O}(mr) = \mathrm{O}(n)$ and space $\mathrm{O}(m) = \mathrm{O}(n/\log n)$, and a space efficient one, with simultaneous time $m^{\mathrm{O}(r)} = n^{\mathrm{O}(\log n)}$ and space $\mathrm{O}(r \log m) = \mathrm{O}((\log n)^2)$. After building these binary tree pebblings we get the theorem by setting G_n to be $r(\widehat{\Phi}(m, r))$ and by using Lemma 5.21 on graph $\widehat{\Phi}(m, r)$.

The space efficient construction is immediate since the depth of $\widehat{\Phi}(m, r)$ is $\mathrm{O}(r \log m)$ and therefore the graph has a binary tree pebbling of the same depth by Proposition 5.9. The time efficient binary tree pebbling $\mathcal{T}$ is the depth-first pebbling from Proposition 5.5, which has time $\mathrm{O}(mr)$. The pebbling is induced by a depth first search, so there is at most one internal node labelled by each vertex. The rest of the proof consists of showing that this binary tree pebbling has space $\mathrm{O}(m)$.

Consider, in the left postfix order of $\mathcal{T}$, the interval between the first occurrence of a sink of layer j, and a sink of layer $j+1$, say z^{j+1}_i. Since all sources of layer j have already been visited, no node with a

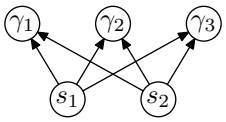


Figure 10: Base case $\Gamma(3,1)$ for Carlson-Savage graph with 3 sinks.

label in layer $j-1$ is alive. Since z_i^{j+1} is the first node in layer $j+1$, no node with label in layer $j+2$ or above has been visited and therefore cannot be alive. Therefore the number of alive nodes is bounded by the number of nodes in the binary tree with labels in layers j and $j+1$, that is $\mathrm{O}(m)$. □

Before moving to the next graph family we need to describe one of its components, the pyramid graph, for which we also need some properties.

JAKOB'S COMMENT 15: *Do we really need theorem-like structures for what follows below? Can we just inline in running text?*

JANE'S COMMENT 3: *Because we refer to pyramid in the definition of Carlson-Savage graph, I left it as definition.*

Definition 5.23. The pyramid graph of height h (denoted as Π_h) is a graph over $(h+1)(h+2)/2$ vertices, indexed as (i,j) for $0 \le i \le j \le h$. For $i > 0$ each vertex (i,j) has two incoming edges from vertices $(i-1,j-1)$ and $(i-1,j)$. The sink vertex is (h,h).

The black-white pebbling price of a pyramid of height h is $h/2 + \Theta(1)$ [Kla85].

JANE'S COMMENT 4: *In the proof below, I changed some indices that seemed like typos ((j,i) instead of (i,j) mainly).*

Proposition 5.24. *There is a binary tree pebbling for the pyramid of height $h \ge 1$ of time $\mathrm{O}(h^2)$ and space $h+2$.*

Proof. We build the tree by induction over h. The case $h = 0$ is immediate. For $h > 0$ consider the subgraph of the pyramid induced by the vertices (i,j) with $j \neq h$. This is a pyramid graph of height $h-1$ with sink $(h-1,h-1)$. Consider the tree $\mathcal{T}'$ of this pyramid. The tree $\mathcal{T}$ for Π_h has root node p_h labelled by (h,h), its left subtree is a copy of $\mathcal{T}'$ and its right subtree is made by nodes $p_{h-1},\ldots,p_0$ and leaf nodes $q_{h-2},\ldots,q_0$, where p_i is labelled by (i,h) and q_i is labelled by $(i,h-1)$, and for $i > 0$ the left and right child of p_i are, respectively, q_{i-1} and p_{i-1}.

There is a bijection between the edges of $\mathcal{T}$ and the edges of Π_h, therefore the size of $\mathcal{T}$ is $\mathrm{O}(h^2)$. To see that the cost of the tree is $h+2$ observe that in the left postfix order visit of $\mathcal{T}$, whenever the (unique) internal node labelled by (i,j) is visited it will be a cache node only until the visit of $(i+1,j+1)$, therefore the worst case is at the visit of the (unique) node of $\mathcal{T}$ labelled by $(1,h)$. In that moment the alive nodes are internal nodes with labels $(1,h)$, $(i,h-1)$ for $0 \le i \le h-1$ and the leaf labelled by $(0,h)$. □

The next trade-off result is based on Carlson and Savage graphs [CS80, CS82]. We use a slight adaptation of this construction, more suitable for proof complexity results, due to [Nor12]. The definition of this family of graphs is rather intricate, so we suggest that the reader refers to the illustrations in Figures 10 and 11.

Definition 5.25 (Carlson-Savage graphs [CS80, CS82]). For positive integers c, r, the graph family $\Gamma(c,r)$, is defined by induction over r. The base case $\Gamma(c,1)$ is a DAG consisting of two sources s_1, s_2 and c sinks $\gamma_1,\ldots,\gamma_c$, with edges from both sources to all sinks. The graph $\Gamma(c,r+1)$ has c sinks and is built from the following components:

- c disjoint copies $\Pi_{2r}^{(1)},\ldots,\Pi_{2r}^{(c)}$ of a pyramid (Definition 5.23) of height $2r$, where we let $z_1,\ldots,z_c$ denote the pyramid sinks;
- one copy of $\Gamma(c,r)$, for which we denote the sinks by $\gamma_1,\ldots,\gamma_c$;
- c disjoint and identical *spines* of length $\lambda = 2c^2r$. Each spine is a sequence of λ vertices, denoted as $v_1,\ldots,v_\lambda$, connected by edges $(v_\ell, v_{\ell+1})$ for $\ell = 1,\ldots,\lambda-1$.

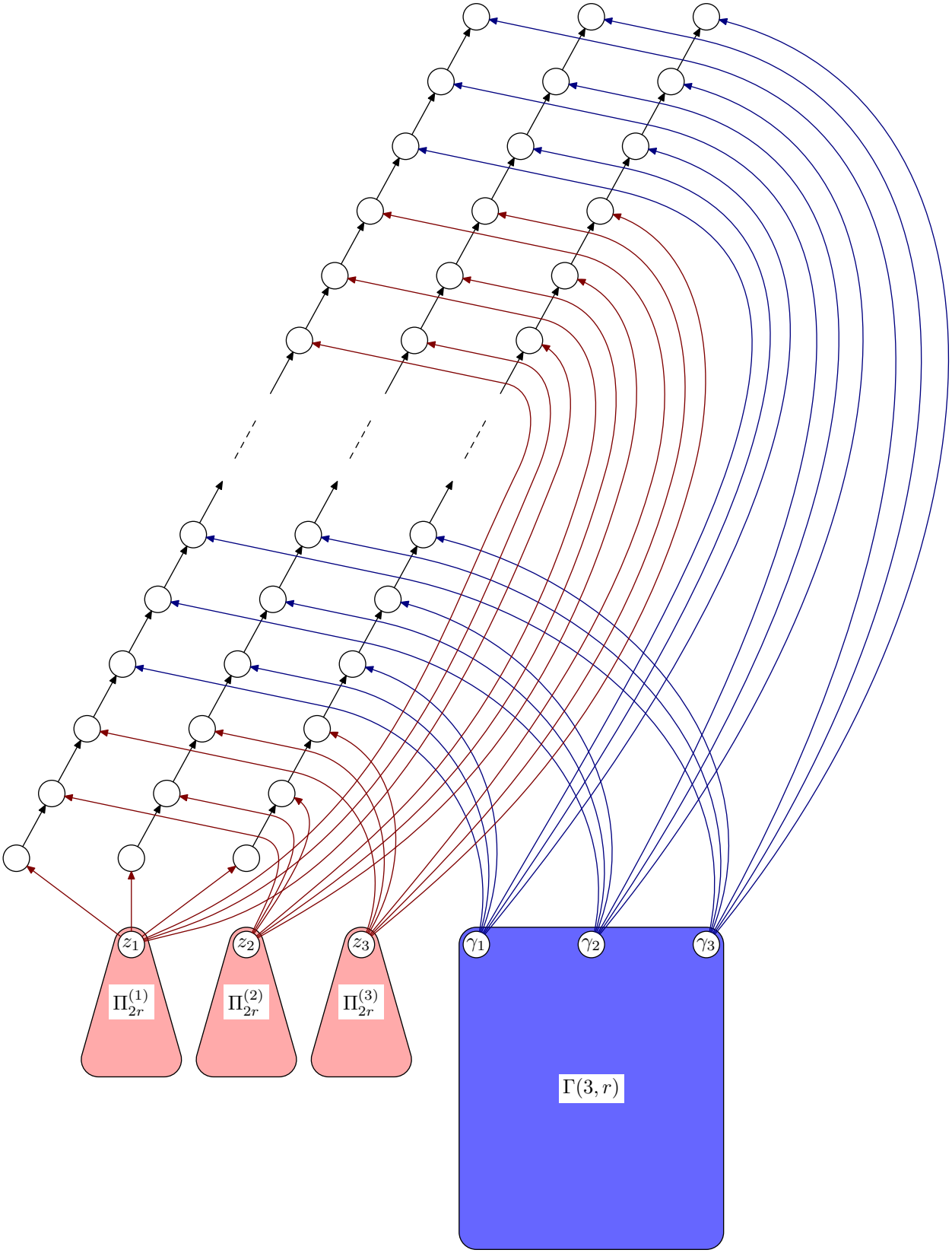


Figure 11: Inductive definition of Carlson-Savage graph $\Gamma(3, r+1)$ with 3 spines and sinks.

These components are connected through edges as follows.

- There are edges (z_j, v_ℓ) to any vertex v_ℓ where $\ell = 2c\ell' + j$ for some integer ℓ'.
- There are edges (γ_j, v_ℓ) to any vertex v_ℓ where $\ell = 2c\ell' + c + j$ for some integer ℓ'.

Pebbling games on the Carlson-Savage graphs have strong time-space tradeoffs. Specifically, in [Nor12], it was shown that for any black-white pebbling of $\Gamma(c, r)$ in space s and time τ, if $s \leq r + k$ for some $0 \leq k \leq c/8$, then

$$\tau \geq \left(\frac{c - 2k}{4k + 4}\right)^r r! \ . \tag{5.2}$$

Theorem 5.26 (Trade-off for arbitrarily slowly growing space). *Let $g(n) = \omega(1)$ be any arbitrarily slowly growing monotone function such that $g(n) = \mathrm{O}(n^{1/7})$ and fix any $\epsilon > 0$. There exists an efficiently constructible family of 3-CNF formulas F_n of size $\Theta(n)$ that have*

- *a CDCL refutation without restarts and with any cutting learning scheme in time $\mathrm{O}(n)$ and space $\mathrm{O}\big(\sqrt[3]{n/g^2(n)}\big)$ simultaneously;*
- *a CDCL refutation without restarts and with any cutting learning scheme in space $\mathrm{O}(g(n))$ and time $n^{\mathrm{O}(g(n))}$ simultaneously.*

Moreover any CDCL refutation of space $\mathrm{O}\big((n/g^2(n)\big)^{1/3-\epsilon}\big)$ has superpolynomial time, regardless of the learning scheme and of the restart policy.

Picking an appropriate function g, in this case $g(n) = \sqrt[8]{n}$, the trade-off is even exponential.

Corollary 5.27. *There exists an efficiently constructible family of 3-CNF formulas F_n of size $\Theta(n)$ that have*

- *a CDCL refutation without restarts and with any cutting learning scheme in time $\mathrm{O}(n)$ and space $\mathrm{O}\big(\sqrt[4]{n}\big)$ simultaneously;*
- *a CDCL refutation without restarts and with any cutting learning scheme in space $\mathrm{O}\big(\sqrt[8]{n}\big)$ and time $n^{\mathrm{O}(\sqrt[8]{n})}$ simultaneously.*

Moreover, for any $\epsilon > 0$, any CDCL refutation of space $\mathrm{O}\big(n^{1/4-\epsilon}\big)$ has exponential time, regardless of the learning scheme and of the restart policy.

Proof of Theorem 5.26. The formula is based on the single sink version of the graph $\Gamma(c, r)$ from Definition 5.25, which is the graph with a maximally unbalanced binary tree on top. This graph is denoted as $\widehat{\Gamma}(c, r)$ and it is formally defined as follows. Let $\gamma_1, \ldots, \gamma_c$ be the sinks of $\Gamma(c, r)$; we add new vertices $\eta_2, \ldots, \eta_c$, two edges (γ_1, η_2), (γ_2, η_2) and then we add edge pairs (η_{j-1}, η_j), (γ_j, η_j) for j from 3 to c. Hence $\widehat{\Gamma}(c, r)$ has a unique sink vertex η_c, it contains $\Gamma(c, r)$ as an induced subgraph, and it has $\Theta(cr^3 + c^3r^2)$ vertices. The main part of the proof is to show that

- there is a binary tree pebbling for $\widehat{\Gamma}(c, r)$ of space $2r + 1 + 3c$ and time $\Theta(cr^3 + c^3r^2)$;
- there is a binary tree pebbling for $\widehat{\Gamma}(c, r)$ of space $2r + 2$;

and that the inequality claimed in [Nor12] (see Equation (5.2)) holds for $\widehat{\Gamma}(c, r)$ as well. Regarding (5.2), we can just observe that any black-white pebbling of η_c induces a black-white pebbling of $\Gamma(c, r)$ within at most the same time and space.

To get the statement of the theorem we are going to apply Lemma 5.21 to $\widehat{\Gamma}(c, r)$, where we set r as $r(n) = g(n)$ and c as $c(n) = \sqrt[3]{n/g^2(n)}$, and we are going to set G_n to be $r(\widehat{\Gamma}(c(n), r(n)))$. The size of the resulting formula $Peb^{\oplus}_{G_n} = Peb^{\oplus}_{r(\widehat{\Gamma}(c(n),r(n)))}$ is $\Theta(cr^3 + c^3r^2) = \Theta(n)$, the space efficient CDCL refutation has space $\mathrm{O}(g(n))$ and the time efficient CDCL refutation has linear time and space

$\mathrm{O}\big(\sqrt[3]{n/g^2(n)}\big)$, for $g(n) = \mathrm{O}(n^{1/7})$. Now let's fix $k = c^{1-\epsilon}$ in (5.2) for some $\epsilon > 0$ so that the space of the black-white pebbling extracted by the refutation of $Peb^{\oplus}_{r(\widehat{\Gamma}(c(n),r(n)))}$, which is within a linear factor from k, is less than $c/8$ for n large enough. Equation (5.2) and Lemma 5.21 imply that any CDCL refutation with space $\mathrm{O}\big(\big(\sqrt[3]{n/g^2(n)}\big)^{1-\epsilon}\big)$ has time $\big(n/g^2(n)\big)^{\Omega(g(n))}$, which is superpolynomial since $g(n) = \omega(1)$.

Now it remains to build the space efficient tree $\widehat{\mathcal{T}}_{(\mathrm{s},r)}$ and the time efficient tree $\widehat{\mathcal{T}}_{(\tau,r)}$ for $\widehat{\Gamma}(c,r)$. As intermediate step we describe, by induction on r, two binary tree pebblings for the subgraph of $\Gamma(c,r)$ constituted by all vertices from which a sink γ_j is reachable. The space efficient version of this intermediate construction is denoted as $\mathcal{T}_{(\mathrm{s},r)}$, and the time efficient one as $\mathcal{T}_{(\tau,r)}$. Since $\Gamma(c,r)$ is symmetric with respect to the permutation of its sink we will discuss the construction of $\mathcal{T}_{(\mathrm{s},r)}$ and $\mathcal{T}_{(\tau,r)}$ with respect of a generic sink.

The two base cases $\mathcal{T}_{(\mathrm{s},1)}$ and $\mathcal{T}_{(\tau,1)}$ are the same. They have the root labelled by γ' and its two children labelled by sources s_1 and s_2.

For the inductive case we consider an arbitrary sink γ' of $\Gamma(c,r+1)$, and we build the two trees $\mathcal{T}_{(\mathrm{s},r+1)}$ and $\mathcal{T}_{(\tau,r+1)}$ for the subgraph of all vertices of $\Gamma(c,r+1)$ that can reach γ'. We suggest to refer to Figure 11 to follow the argument. We denote as $(v_1,\ldots,v_\lambda)$ the sequence of the vertices in the spine that ends with sink γ', so that z_1 is a predecessor of v_1 and v_λ is γ' itself.

We define the tree $\mathcal{T}_{(*,r+1)}$ so that its leftmost path backward from the root has length λ and is labelled by the sequence of vertices $(v_\lambda,\ldots,v_1)$. The only child of the vertex labelled by v_1 is a leaf vertex labelled by z_1. For $1 < \ell \leq \lambda$ the right child of the node labelled by v_ℓ is a leaf node labelled by the corresponding predecessor of v_ℓ among $\{z_1,\ldots,z_c,\gamma_1,\ldots,\gamma_c\}$. For the sake of uniformity we give the definition of $\mathcal{T}_{(*,1)}$ too, which is identical to $\mathcal{T}_{(\tau,1)}$ and to $\mathcal{T}_{(\mathrm{s},1)}$. For $1 \leq j \leq c$ we denote as $\mathcal{T}^{\Pi}_j$ the binary tree pebbling for the pyramid subgraph $\Pi^{(j)}_{2r}$ given in Proposition 5.24, which has time $\mathrm{O}(r^2)$ and space $2r+2$. Both the $\mathcal{T}_{(\mathrm{s},r)}$ and $\mathcal{T}_{(\tau,r)}$ constructions have $\mathcal{T}_{(*,r)}$ as a starting point, and the difference in the two constructions is in the subtrees that we attach to their leaves.[4]

JAKOB'S COMMENT 16: *Why do we have \paragraph environments here? It creates a bit of a strange impression when we mix different types of formatting. I would skip \paragraph formatting...*

JANE'S COMMENT 5: *changed it to \textbf.*

The space efficient construction $\mathcal{T}_{(\mathrm{s},r+1)}$ from $\mathcal{T}_{(\mathrm{s},r)}$. We start with $\mathcal{T}_{(*,r+1)}$ and we attach to each leaf labelled z_j a copy of $\mathcal{T}^{\Pi}_j$. For each leaf labelled γ_j we attach a copy of the space efficient binary tree pebbling $\mathcal{T}_{(\mathrm{s},r)}$ obtained by inductive hypothesis, namely the space efficient tree for the subgraph of $\Gamma(c,r)$ constituted by all vertices that can reach γ_j. We claim by induction on r that this construction has space at most $2r+1$. For $r=1$ the space is 3 so we are within the limits. For $r > 1$ observe that when we visit the right subtree of a node labelled by some vertex v_ℓ, the only node alive outside of that subtree is the one labelled by $v_{\ell-1}$. The space used by the total construction then is one plus the maximum space among all subtrees attached to $\mathcal{T}_{(*,r+1)}$. The space for each copy of $\mathcal{T}^{\Pi}_j$ is $2r+2$, and by induction the space for each copy of $\mathcal{T}_{(\mathrm{s},r)}$ is $2r+1$. Therefore the space for $\mathcal{T}_{(\mathrm{s},r+1)}$ is $2r+3$, and the inductive claim is proved.

The space efficient construction $\widehat{\mathcal{T}}_{(\mathrm{s},r)}$ from $\mathcal{T}_{(\mathrm{s},r)}$. Now we can build the space efficient binary tree of $\widehat{\Gamma}(c,r)$. The top part of the tree is an isomorphic copy of the tree in $\widehat{\Gamma}(c,r)$ that connects the sinks $\gamma_1,\ldots,\gamma_c$ of $\Gamma(c,r)$ to the new sink η_c, where nodes are labelled by the corresponding vertices. To each node labelled by γ_j we attach a new copy of the space efficient tree $\mathcal{T}_{(\mathrm{s},r)}$ for the corresponding sink. The final tree is a binary tree pebbling for $\widehat{\Gamma}(c,r)$, and it has space $2r+2$, since one space unit is used for the top part and each copy of $\mathcal{T}_{(\mathrm{s},r)}$ has space $2r+1$.

The time efficient construction $\mathcal{T}_{(\tau,r+1)}$ from $\mathcal{T}_{(\tau,r)}$. In the time efficient construction we only attach subtrees to just the first few leaves in $\mathcal{T}_{(*,r+1)}$ so that later leaves remain leaves in the final construction too, and use them as cache nodes. This makes the tree smaller, but increases the space of the construction.

[4]Here we use the verb *to attach* in a very specific way. Given a leaf node p labelled by some vertex v and a binary tree pebbling $\mathcal{T}$ with the root labelled by the same vertex v, we say that we attach $\mathcal{T}$ to the node p by substituting p with $\mathcal{T}$ itself.

We consider the sequence of nodes $(p'_1, \ldots, p'_c, q_1, \ldots, q_c, p_1, \ldots, p_c)$ from $\mathcal{T}_{(*,r+1)}$ so that, for each $j \in [c]$, node p'_j is the child labelled by z_j of the node labelled by v_j, node q_j is the child labelled by γ_j of the node labelled by v_{c+j} and node p_j is the child labelled by z_j of the node labelled by v_{2c+j}.

We will attach a subtree only to those $3c$ nodes. Any later occurrence of a leaf node labelled by z_j will use p_j as cache node, and any later occurrence of a leaf labelled by γ_j will use q_j as cache node.

We need to stress that $p'_1, \ldots, p'_c$ are not going to be cache nodes. This choice is made because the left postorder needs to do a full visit of $\mathcal{T}_{(\tau,r)}$ between the visit of $p'_1, \ldots, p'_c$ and the visit of any later node labelled by some z_j. If such nodes were cached during the recursive descent, the space of the tree would explode.

For this reason we are going to visit each pyramid $\Pi^{(j)}_{2r}$ twice, namely for $j \in [c]$ we attach the tree $\mathcal{T}^{\Pi}_j$ to both p'_j and p_j.

JANE'S COMMENT 6: *Changed "latest" to "last" in next sentence.*

In this way p_j is the last internal node to have label z_j, and it is going to be the cache node for all later nodes with that label. To node q_1 instead we attach a copy of the tree $\mathcal{T}_{(\tau,r)}$ built by induction, and for $2 \leq j \leq c$ we attach to node q_j a copy of $\mathcal{T}_{(*,r)}$, built by induction for the sink γ_j.

The tree built so far is the final tree $\mathcal{T}_{(\tau,r+1)}$, and since it is more complex than the space efficient one we will explicitly show its correctness.

To do that we need notation to refer to vertices in the subgraph $\Gamma(c,r)$ contained in $\Gamma(c,r+1)$, and notation to refer to nodes in the subtree $\mathcal{T}_{(\tau,r)}$ inside $\mathcal{T}_{(\tau,r+1)}$. We recall that γ' denotes our target sink in the graph $\Gamma(c,r+1)$, and that we denoted as z_j the sinks of the pyramids and as γ_j the sinks of the copy of $\Gamma(c,r)$ contained in the recursive construction $\Gamma(c,r+1)$. When $r > 1$ we denote as $z^\dagger_1, \ldots, z^\dagger_c$ the sinks of the pyramids (of height $2r-2$) inside the subgraph of $\Gamma(c,r)$, and as $\gamma^\dagger_1, \ldots, \gamma^\dagger_c$ the sinks of its internal copy of $\Gamma(c,r-1)$. Furthermore we denote as $p^\dagger_j$ the (unique) cache node in $\mathcal{T}_{(\tau,r)}$ labelled by $z^\dagger_j$, and as $q^\dagger_j$ the (unique) cache node in $\mathcal{T}_{(\tau,r)}$ labelled by $\gamma^\dagger_j$.

Let us now argue correctness.

Claim 5.28. For each leaf node in $\mathcal{T}_{(\tau,r)}$ which is not labelled by a source vertex of $\Gamma(c,r)$ there is a corresponding cache node earlier in the tree.

Proof. Our proof is by induction on r. For $r = 1$ it is immediate. Let us assume that it is true for $r \geq 1$, we want to prove that each leaf node in $\mathcal{T}_{(\tau,r+1)}$ which is not labelled by a source vertex of $\Gamma(c,r+1)$ there is a corresponding cache node earlier in the tree. Subtrees $\mathcal{T}^{\Pi}_j$ are correct by construction, and the correctness of subtree $\mathcal{T}_{(\tau,r)}$ holds by induction. The remaining non-source leaves in $\mathcal{T}_{(\tau,r+1)}$ are either the ones coming from $\mathcal{T}_{(*,r+1)}$ to which we did not attach a subtree, or they are non-source leaf nodes inside the $(c-1)$ copies of $\mathcal{T}_{(*,r)}$ attached to $p_2, \ldots, p_c$. In the first case the leaf is either labelled by γ_j or by z_j and it comes later in the left postfix order than nodes $q_1, \ldots, q_c, p_1, \ldots, p_c$, one of them being its cache node. For $r = 1$ the leaves of all attached copies of $\mathcal{T}_{(*,r)}$ are all labelled by sources s_1 and s_2. For $r > 1$ the leaves of all attached copies of $\mathcal{T}_{(*,r)}$ are all labelled by vertices among $\gamma^\dagger_1, \ldots, \gamma^\dagger_c, z^\dagger_1, \ldots, z^\dagger_c$, and earlier in the left postfix order, $\mathcal{T}_{(\tau,r+1)}$ contains the subtree $\mathcal{T}_{(\tau,r)}$, which contains nodes $q^\dagger_1, \ldots, q^\dagger_c, p^\dagger_1, \ldots, p^\dagger_c$ with the same labels. □

The size of the tree is estimated in the following claim.

Claim 5.29. The number of internal vertices in $\mathcal{T}_{(\tau,r)}$ plus $(c-1)$ copies of $\mathcal{T}_{(*,r)}$ is at most C times the number of edges in $\Gamma(c,r)$, for some universal constant $C > 0$.

Proof. Our proof is again by induction on r. It is immediate for $r = 1$. Assume that the claim holds for $r \geq 1$, we prove it for $r+1$. Observe that $\mathcal{T}_{(\tau,r+1)}$ contains a copy of $\mathcal{T}_{(\tau,r)}$ ending in γ_1 plus $(c-1)$ copies of $\mathcal{T}_{(*,r)}$ ending in $\gamma_2, \ldots, \gamma_c$, respectively. By inductive hypothesis this accounts to at most C times the size of $\Gamma(c,r)$. Furthermore $\mathcal{T}_{(\tau,r+1)}$ contains one isomorphic copy of one spine from $\Gamma(c,r+1)$, while the other $(c-1)$ spines in $\Gamma(c,r+1)$ are accounted to the edges of the $(c-1)$ copies of $\mathcal{T}_{(*,r+1)}$ as in the claim. Tree $\mathcal{T}_{(\tau,r+1)}$ also contains $2c$ binary tree pebblings for pyramid graphs of

height $2r$, each of size at most C' times the size of the pyramid graph itself, for a universal constant C' (see Proposition 5.24). Let C be a universal constant larger than $\max\{2C', 1\}$ and then the induction step follows. □

We claim that the space of $\mathcal{T}_{(\tau,r)}$ is at most $2r + 1 + 3c$. The space is 3 for $r = 1$ so we are within the bound. For the general claim we use the following inductive statement.

Claim 5.30. During each moment of the visit of $\mathcal{T}_{(\tau,r+1)}$, the set of alive nodes union the set of nodes in $\{q_1, \ldots, q_c, p_1, \ldots, p_c\}$ already visited at that moment has size at most $2r + 3 + 3c$.

Proof. We assume that either $r = 1$ or that the claim holds for $\mathcal{T}_{(\tau,r)}$ if $r > 1$.

Observe that the left postfix order visits the sequence of nodes labelled v_ℓ in order. We divide the sequence in four segments: the first three have length c each, and the last one is made by the rest of the sequence.

First segment. At the beginning we visit a copy of $\mathcal{T}_1^{\Pi}$ using space $2r + 2$, then the node labelled by v_1. At this point no node in this copy of $\mathcal{T}_1^{\Pi}$ is alive anymore. The next step is to visit the copy of $\mathcal{T}_2^{\Pi}$ rooted in p'_2, and then the node labelled by v_2, and so on up to $\mathcal{T}_c^{\Pi}$. During the visit of some $\mathcal{T}_j^{\Pi}$, the only other alive node is the one labelled by v_{j-1}. At some point we reach the node labelled by v_c using the maximum space $2r + 3$, and that is the only alive node at that moment.

Second segment (I). If $r = 1$ then the visit of the subtree $\mathcal{T}_{(\tau,r)}$ is done in space 3. Otherwise we visit the subtree $\mathcal{T}_{(\tau,r)}$ in space $2r + 1 + 3c$, passing through nodes $\{q_1^\dagger, \ldots, q_c^\dagger, p_1^\dagger, \ldots, p_c^\dagger\}$, that stay alive. The root of this tree is q_1, which is a cache node for later nodes. During the visit of the subtree $\mathcal{T}_{(\tau,r)}$ the node labelled by v_c is alive, therefore we have maximum space so far equal to $2r + 2 + 3c$, and the alive nodes are: the node labelled by v_c, node q_1 and, when $r > 1$, $\{q_1^\dagger, \ldots, q_c^\dagger, p_1^\dagger, \ldots, p_c^\dagger\}$.

Second segment (II). The visit now proceeds up to the node labelled by v_{2c}: we visit each of the $(c - 1)$ copies of $\mathcal{T}_{(*,r)}$ in order. If $r = 1$ then $\mathcal{T}_{(*,1)}$ is a node with two leaf children, so three additional alive nodes are necessary. If $r > 1$ each of them is a maximally unbalanced tree with leaves labelled by $\{\gamma_1^\dagger, \ldots, \gamma_c^\dagger, z_1^\dagger, \ldots, z_c^\dagger\}$ for which we already have alive cache nodes, therefore only two additional alive nodes are necessary to complete this phase. During the visit of the copy of $\mathcal{T}_{(*,r)}$ rooted in q_j, the alives nodes are $q_1, \ldots, q_{j-1}$ from previous copies; the nodes $\{q_1^\dagger, \ldots, q_c^\dagger, p_1^\dagger, \ldots, p_c^\dagger\}$ from recursion when $r > 1$; the node labelled by v_{c+j-1}; and at most two additional nodes during the visit itself (three if $r = 1$). The maximum space in this specific phase is at most $3 + 3c$, and after its conclusion the alive nodes are: the node labelled by v_{2c} and $q_1, \ldots, q_c$. In particular after this moment we will not visit any node for which any of $\{q_1^\dagger, \ldots, q_c^\dagger, p_1^\dagger, \ldots, p_c^\dagger\}$ is a cache node.

Third segment. In the next phase we visit another series of the binary tree pebbling for the pyramids, but this time their roots $p_1, \ldots, p_c$ are going to be cache nodes for later non-source leaf nodes. For each $j \in [c]$ we visit the copy of $\mathcal{T}_j^{\Pi}$ rooted in node p_j while nodes $q_1, \ldots, q_c$, nodes $p_1, \ldots, p_{j-1}$ and the node labelled by v_{2c+j-1} are alive. The total space in this phase is at most $2r + 3 + 2c$.

Fourth segment. For each $\ell > 3c$, we just visit v_ℓ in order, using cache nodes $\{q_1, \ldots, q_c, p_1, \ldots, p_c\}$, with two additional alive nodes simultaneously.

In total we have that the number of alive nodes is always at most $2r + 3 + 3c$, and furthermore the nodes $\{q_1, \ldots, q_c, p_1, \ldots, p_c\}$ are counted as alive from the moment they are visited, until the end of the visit, as requested by the claim. □

The time efficient construction $\widehat{\mathcal{T}}_{(\tau,r)}$ from $\mathcal{T}_{(\tau,r)}$. Now we can build the time efficient binary tree of $\widehat{\Gamma}(c, r)$. The top part of the tree is an isomorphic copy of the tree in $\widehat{\Gamma}(c, r)$ that connects the sinks $\gamma_1, \ldots, \gamma_c$ of $\Gamma(c, r)$ to the new sink η_c, where nodes are labelled by the corresponding vertices. To the node labelled by γ_1 we attach a copy of the time efficient tree $\mathcal{T}_{(\tau,r)}$, while to each other node labelled by γ_j with $j > 1$ we attach a copy of $\mathcal{T}_{(*,r)}$. By Claim 5.29 the time of $\widehat{\mathcal{T}}_{(\tau,r)}$ is $\mathrm{O}(c)$ plus a constant times the size of $\Gamma(c, r)$, which is $\mathrm{O}(cr^3 + c^3r^2)$ as desired. The left postfix visit of $\widehat{\mathcal{T}}_{(\tau,r)}$ is the composition of the visit of $\mathcal{T}_{(\tau,r)}$ plus the visit of the rest of the tree, which uses the $2c$ cache nodes q_j and p_j from $\mathcal{T}_{(\tau,r)}$.

JANE'S COMMENT 7: *I changed $q_j^\dagger$ and $p_j^\dagger$ to q_j and p_j because it's $\mathcal{T}_{(\tau,r)}$ within $\widehat{\mathcal{T}}_{(\tau,r)}$ here, not $\mathcal{T}_{(\tau,r)}$ within $\mathcal{T}_{(\tau,r+1)}$.*

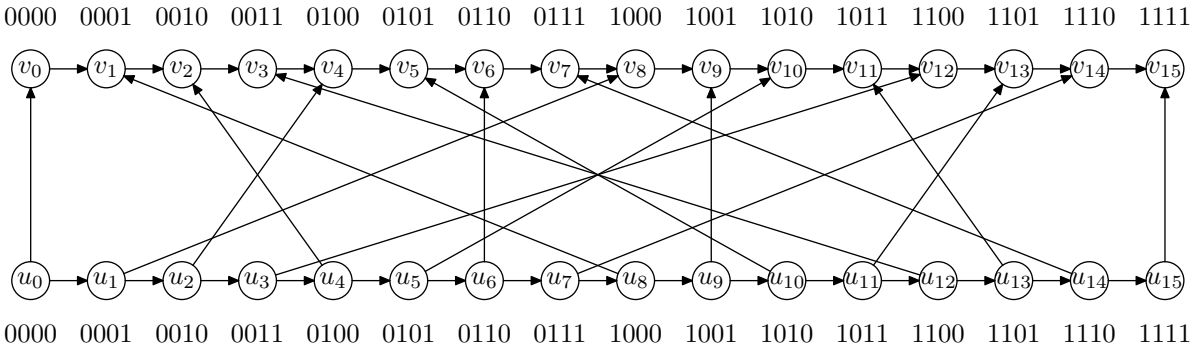


Figure 12: Bit reversal graph with $k = 4$.

The rest of the visit can be done with a constant number (i.e., 4) of additional alive nodes, so the space of $\widehat{\mathcal{T}}_{(\tau,r)}$ is dominated by the one for the visit of $\mathcal{T}_{(\tau,r)}$, and is $2r + 1 + 3c$. □

The last graph family we consider is the bit reversal graph from [LT82], which is made by two paths of vertices, where each vertex in the bottom path is connected to a vertex in the top path, according to a specific bijection (see Figure 12).

Definition 5.31 (Bit reversal graph). Fix $k > 0$ and let π be the permutation that maps each integer $0 \leq i < 2^k$ written as its k bit binary representation into the number that corresponds to the reverse of that binary representation. The *bit-reversal graph* has $2 \cdot 2^k$ vertices, divided into two paths. The lower path is $u_0, \ldots, u_{2^k-1}$ where u_{i-1} is a predecessor of u_i for $1 \leq i < 2^k$. The upper path is $v_0, \ldots, v_{2^k-1}$ where v_{i-1} is a predecessor of v_i for $1 \leq i < 2^k$. For $i > 0$, each v_i has two predecessors, namely v_{i-1} and $u_{\pi(i)}$. The only predecessor of v_0 is u_0.

Theorem 5.32 ([LT82]). *Fix $n = 2 \cdot 2^k$. Consider the bit reversal graph with n vertices and any black-white pebbling of time t and $s > 3$. Then it holds that $t \geq \frac{n^2}{18s^2} + 2n$.*

Theorem 5.33 (Trade-off for pebbling formulas over bit reversal graphs). *There exists an efficiently constructible family of 3-CNF formulas F_k of size $n = \Theta(2^k)$ that have a CDCL refutation without restarts and with any cutting learning scheme in space $\mathrm{O}(s)$ and time $\mathrm{O}(n^2/s)$ simultaneously, for every $s = \mathrm{O}(\sqrt{n})$. Moreover, any CDCL refutation simultaneously in space $s \leq \sqrt{n}$ and time τ satisfies*

$$\tau = \Omega\left(\left(\frac{n}{s}\right)^2\right) , \tag{5.3}$$

regardless of the learning scheme and of the restart policy.

Proof. The formula family F_k for which we prove this theorem is $Peb^{\oplus}_{G_k}$, where the graph G_k is the robust version of the bit reversal graph in Definition 5.31 with $2 \cdot 2^k$ vertices.

We will show that the bit reversal graph has a binary tree pebbling of space $\mathrm{O}(s)$ and time $\mathrm{O}(n^2/s)$. Since the appropriate trade-off for black-white pebbling also holds by Theorem 5.32, the result then follows by Lemma 5.21.

We start with a chain of nodes $p_0, \ldots, p_{2^k-1}$ labelled respectively $v_0, \ldots, v_{2^k-1}$, and connected so that for $0 < i < 2^k$, p_{i-1} is the left child of p_i. Each p_i has a right child q_i labelled by vertex $u_{\pi(i)}$. Let us further denote $\mathcal{T}_{i,j}$ for $0 \leq i \leq j < 2^k$ as a tree made by a single chain of $j - i + 1$ nodes labelled from the leaf to the root with vertices $u_i, \ldots, u_j$ respectively. Notice that $\mathcal{T}_{i,i}$ is a tree containing a single node.

A trivial construction would be to attach to each node q_i a tree $\mathcal{T}_{0,\pi(i)}$. This produces a binary tree pebbling for the bit reversal graph of time $\mathrm{O}(n^2)$ and constant space. Another simple construction would be to attach at node q_i the tree $\mathcal{T}_{j,\pi(i)}$ where j is the index closest to $\pi(i)$ so that label u_j occur in one of the trees attached to some earlier $p_{i'}$ with $i' < i$. In this construction instead all internal nodes labelled by some u_j are indeed cache nodes. The space of this tree is $\mathrm{O}(n)$ and its time is $\mathrm{O}(n)$ as well.

JANE'S COMMENT 8: *I rewrote most of the part below.*

For our construction we do something in between of these two extremes: we divide the sequence $0, \ldots, 2^k - 1$ into s intervals of length $2^k/s$. Let L be the set of lower endpoints of the intervals. We use the nodes in L as cache nodes and go back to the closest cache node each time. Formally, we attach the tree $\mathcal{T}_{0,0}$ to p_0, and for each $0 < i < 2^k$, we attach $\mathcal{T}_{s_i,\pi(i)}$ to node p_i, where s_i is the closest cache node: if $t_i = \max_{j<i} \pi(j)$ is the rightmost vertex in the lower path visited so far, then $s_i = \max\{l \in L \mid l \leq t_i\}$. Therefore, the total number of cache nodes is $\mathrm{O}(s)$. For the time bound, observe that a length ℓ tree $\mathcal{T}_{i,i+\ell-1}$ adds $\ell/(2^k/s) - \mathrm{O}(1)$ new cache nodes. Because each cache node is added only once, $(s/2^k)\sum \ell - \mathrm{O}(2^k) \leq s$, so the total path length is $\mathrm{O}((2^k)^2/s + 2^k)$. Therefore the total time of the whole construction is $\mathrm{O}(n^2/s + n)$, which is $\mathrm{O}(n^2/s)$ when $s = \mathrm{O}(\sqrt{n})$. □

6 Trade-offs for Tseitin formulas

MASSIMO'S COMMENT 10: *STATUS: This section requires editing and prof reading. I still need to edit the two proofs. The one with small space should be editable by anyone. For the one with short length I may need to do a pass first, before anybody else puts his eyes on it.*

JAKOB'S COMMENT 17: *In a future version it would be nice to actually put a small grid here (with single edges, perhaps) instead of a triangle.*

In this section we show time-space trade-offs for CDCL refutations of Tseitin formulas, which are formulas encoding a particular form of unsatisfiable linear systems of equations $\bmod 2$. Let us give a formal definition of these formulas.

Definition 6.1 (Tseitin formula). Let $G = (V, E)$ be an undirected graph and $\chi : V \to \{0, 1\}$ be a vertex *charge* function. Identify every edge $e \in E$ with a variable x_e that gets value in $\{0, 1\}$ and let $PARITY_{v,\chi}$ denote the CNF encoding of the parity constraint $\sum_{e \ni v} x_e = \chi(v) \pmod 2$ for a vertex $v \in V$. Then the *Tseitin formula* over G with respect to χ is $T_{G,\chi} = \bigwedge_{v \in V} PARITY_{v,\chi}$.

When the degree of G is bounded by d, $PARITY_{v,\chi}$ has at most 2^{d-1} clauses, all of width at most d, and hence $T_{G,\chi}$ is a CNF formula with at most $2^{d-1}|V|$ clauses. Again, we refer the reader to Figure 4 for a small example of a Tseitin formula.

We say that a set of vertices U has *odd (even) charge* if $\sum_{u \in U} \chi(u)$ is odd (even). By a simple counting argument one sees that $T_{G,\chi}$ is unsatisfiable if $V(G)$ has odd charge.

In this section we are interested in grid (multi-)graphs with double edges. A grid graph of size $w \times \ell$ has vertex set $\{(i, j) \mid 0 \leq i < w, 0 \leq j < \ell\}$. A grid graph wit double edges has the following set of edges: for each vertex (y, x), if $y < w - 1$, there are two vertical edges to $(y + 1, x)$ labelled $ev_{y,x}$, $ev'_{y,x}$, and if $x < \ell - 1$, there are two horizontal edges to $(y, x + 1)$ labelled $eh_{y,x}$, $eh'_{y,x}$.

For Tseitin formulas on grid graphs with double edges, trade-off results are know for resolution refutations.

Theorem 6.2 ([BNT13]). *Consider a Tseitin formula over a $w \times \ell$ grid graph with double edges, with odd charge function. It holds that*

- *the formula has a resolution refutation of length $\mathrm{O}(2^{\mathrm{O}(w)}\ell)$ and clause space $\mathrm{O}(2^{\mathrm{O}(w)})$,*
- *the formula has a tree-like resolution refutation of length $\mathrm{O}(\ell^{\mathrm{O}(w)})$ and clause space $\mathrm{O}(w \log(\ell))$,*
- *if $1 \leq w \leq \ell^{1/4}$, then if a resolution refutation has length L and clause space s,*

$$L = \left(\frac{2^{\Omega(w)}}{s}\right)^{\Omega\left(\frac{\log\log\ell}{\log\log\log\ell}\right)}$$

The short resolution refutation follows a dynamic programming approach. During the proof we maintain a subset of vertices and we keep in memory the clauses that represent the sum of charges over that subset—equivalently, the sum over the border. The proof ends when we show that an empty border

has odd charge. We update the subset by adding vertices ordered by column, so that the border has size at most $w+1$ and we can represent its sum in at most 2^w clauses. Adding a vertex means resolving a constant number of clauses with the clauses in memory, which gives a total length of $\mathrm{O}(2^w w\ell)$.

The small resolution refutation is the tree-like proof induced by a search tree. The search tree performs a binary search along the columns, at each step querying the w variables in the middle column and then recursively exploring the subgraph of odd charge. The depth of this search tree is at most $w(\log(\ell)+1)$, which yields a proof of length $\mathrm{O}(\ell^w)$ and space $\mathrm{O}(w\log(\ell))$.

Our goal is to find CDCL equivalents of these resolution proofs. In the next two subsections, we show CDCL simulations of both proofs. We assume the 1UIP learning scheme.

Theorem 6.3 (CDCL simulation results for Tseitin). *Consider the Tseitin formula over a $w \times \ell$ grid graph with double edges, with charge 1 in vertex $(0,0)$ and charge 0 in all other vertices. It holds that*

- *the formula has a time-efficient CDCL refutation with the 1UIP learning scheme of time $\mathrm{O}(2^{5w}\ell)$ and space $\mathrm{O}(2^{2w})$ simultaneously,*
- *the formula has a space-efficient CDCL refutation with the 1UIP learning scheme of space $\mathrm{O}(w\log\ell)$ and time $\mathrm{O}(\ell^{\mathrm{O}(w)})$ simultaneously.*

Moreover, if $1 \le w \le \ell^{1/4}$, any CDCL refutation of space s has time

$$\tau = \left(\frac{2^{\Omega(w)}}{s}\right)^{\Omega\left(\frac{\log\log\ell}{\log\log\log\ell}\right)},$$

regardless of the learning scheme and of the restart policy.

In the proof of both simulations, we use the following notation. We denote the variable corresponding to edge e also by e. We denote the set of vertices between $x=l$ and $x=r$ by $V_{[l,r]}$ and the set of edges in the induced subgraph of $V_{[l,r]}$ by $E_{[l,r]}$. Let $EH_j = \bigcup_{0\le i<w}\{eh_{i,j}, eh'_{i,j}\}$ denote the set of $2w$ horizontal edges between columns j and $j+1$, for $0 \le j < w-1$. Let $EV_j = \bigcup_{0\le i<w-1}\{ev_{i,j}, ev'_{i,j}\}$ denote the set of $2w-2$ vertical edges in column j, for $0 \le j < w$. Let $\mathrm{charge}(S) = \bigoplus_{e\in S} v(e)$ denote the parity of the set of variables S under the current assignment v, where $v(e)=1$ corresponds to assignment TRUE to variable e and $v(e)=0$ corresponds to FALSE.

In the dynamic programming proof, we will learn the set of clauses saying that $\mathrm{charge}(EH_j)=1$ for all $0 \le j < w-1$. This linear equation is encoded using 2^{2w-1} clauses. Each of these clauses rules out a single even charge assignment to EH_j. Let L_j denote this set of clauses for EH_j.

The main lemma we use is the following.

Lemma 6.4. *Consider a subgrid $V_{[l,r]}$ ($0 \le l \le r \le w-1$). Suppose $E_{[l,r]}$ is unassigned and both EH_{l-1} and EH_r are assigned, unless $l=0$ or $r=\ell-1$. Let $(\ell_0,\ell_1,\dots,\ell_{2w-1})$ denote the last $2w$ assignments. Suppose $\{\mathrm{var}(\ell_i) \mid 0 \le i < 2w\} \in \{EH_{l-1}, EH_r\}$, and let B be the set of literals assigned* TRUE *from $(EH_{l-1}\cup EH_r)\setminus\{\mathrm{var}(\ell_i)\mid 0\le i<2w\}$ (B is empty if $l=0$ or $r=\ell-1$).*

Suppose that the only learnt clauses not being used as a reason are from the L_j sets, and that the restricted problem on $V_{[l,r]}$ would be satisfiable if any of the assignments to $EH_{l-1}\cup EH_r$ was inverted.

For each ℓ_i that is not a decision, denote its reason by C_i. Let $R_i = C_i \setminus (\{\neg\ell_j \mid 0\le j<i\}\cup\{\ell_i\})$ and suppose $R_i \cap V_{[l,r]} = \emptyset$. Suppose at least one ℓ_i is a decision and let k be the largest index such that ℓ_k is a decision. Assume the only possible learnt clauses involving variables from $V_{[l,r]}$ are clauses from one of the L_j.

Consider any CDCL trace setting only variables within $E_{[l,r]}$ until the first conflict that causes the CDCL refutation to backtrack from ℓ_k (and possibly further). At this conflict, CDCL with 1UIP learns the clause $(\neg\ell_0\vee\ldots\vee\neg\ell_k)\vee\bigvee_{i>k}R_i\vee\bigvee_{\ell\in B}\neg\ell$ with asserting literal $\neg\ell_k$.

Proof. Consider the resolution derivation of the conflict clause derived, where the clauses from the L_j that were learnt before the trace began are axioms, and we expand the derivation of clauses learnt during

the trace. Observe that we never resolve over any of the variables in EH_{l-1} or EH_r, because these are part of the trail during the whole trace.

Suppose the asserting variable is one of the edges in $E_{[l,r]}$. Then ℓ_k was not used in the derivation; however, the conclusion does not hold if we invert ℓ_k: in that case, the restricted problem is satisfiable, and we can flip any two variables for each double edge in $E_{[l,r]}$ (note that this doesn't falsify the clauses from the L_j). But the derivation should also be true for the restricted formula with ℓ_k inverted, so we obtain a contradiction.

Therefore the asserting variable must be one of $\ell_k, \ell_{k+1}, \ldots, \ell_{2w-1}$. Because the restricted formula $V_{[l,r]}$ would be satisfiable if any of the assignments to $EH_{l-1} \cup EH_r$ was inverted, for all border vertices of $V_{[l,r]}$ adjacent to the ℓ_i, we use at least one clause from the charge axioms for these vertices, involving each ℓ_i. So 1UIP resolves over $\ell_{k+1}, \ldots, \ell_{2w-1}$ and ends up with asserting variable ℓ_k. □

6.1 Time-efficient proof

In this section, we show a CDCL simulation of the time-efficient proof. We learn all the L_j clause sets and try to do tree-like resolution otherwise. The simulation strategy is given as a recursive procedure `SolveDP`(j) which produces a sequence of decisions and forget commands. The initial call to generate this sequence is `SolveDP`$(\ell - 2)$. The clause deletion strategy will be to forget a learnt clause part of an L_j when a forget command for this clause is encountered in the sequence; all other learnt clauses are forgotten at the earliest stable state in which they do not occur as a reason anymore. We use a procedure `Bruteforce`(L, cb) that takes as input a list of variables L and a callback function cb, and tries all possibilities for the variables in L (generating a full binary tree of depth $|L|$; variables in L are decided on in the order they appear in L), calling function cb at each leaf in the tree (that is, for each assignment of L).

The strategy first assigns the variables in the order from $x = w$ to $x = 1$ and then learns the L_j from $x = 1$ to $x = w$. In order to learn that $\text{charge}(EH_j) = 1$, we try all possible assignments such that $\text{charge}(EH_j) = 0$ and derive UNSAT for the subproblem on $V_{[0,j]}$. For the first such assignment, we have to solve the whole subgrid $V_{[0,j]}$, but as a result we learn L_{j-1}, so that proving all other assignments UNSAT can be done without assignments past column $j - 1$. In order to learn a clause from L_j, the last assignment to EH_j must be a decision; if it is a propagation, variables from EH_{j+1} (used to refute an assignment to EH_j with odd charge) could be included in the conflict clause. Therefore, we process the assignments such that for assignments with even charge, the last assignment a decision. This is implemented by deciding the last variable such that the total charge is 0. For example, if $w = 3$, the bitstrings for EH_j are processed in the following order. The first bit denotes the top-level decision in the tree; blue-colored bold numbers denote implications, the others decisions. `000`, `00`**`1`**, `0`**`11`**, `0`**`10`**, **`1`**`01`, **`100`**, **`110`**, **`111`**. Note that the last variable is a decision in every other step, and that in these steps, the parity of the bitstring is even. If the last variable is a decision, we learn a clause from L_j; otherwise, we use the assignment to $EH_j \cup EH_{j+1}$ to prove UNSAT in EV_{j+1}.

In the pseudocode, we denote by $\#\,\text{trailingzeroes}(i, n)$ the number of trailing zeroes of the binary representation of i ($0 \leq i < 2^n$; it returns n for $i = 0$).

Lemma 6.5. *Suppose $0 \leq j \leq \ell - 2$. If certain preconditions hold, calling `SolveDP`(j) has time complexity (length of the trace) $\mathrm{O}(2^{5w} \cdot (j + 1))$, space complexity $\mathrm{O}(2^{2w})$ and results in a set of post-conditions.*

Preconditions:

- *$E_{[0,j+1]}$ is unassigned and there are no learnt clauses involving variables from $E_{[0,j+1]}$.*
- *If $j = \ell - 2$, the trail is empty; otherwise, the last assignments on the trail are the assignments to EH_{j+1} (all zeroes, and all decisions).*

Postconditions:

- *$E_{[0,j+1]}$ is unassigned.*

- *If $j = \ell - 2$, UNSAT is determined; otherwise, the clause $\bigvee_{x \in EH_{j+1}} x$ is learnt, and the only change to the trail is that the lowest decision is replaced by an implication of $\bigvee_{x \in EH_{j+1}} x$.*
- *All clauses in L_j are learnt and remembered.*

Proof. The Tseitin formula restricted to the current assignment is unsatisfiable on $V_{[0,j+1]}$ because, if $j < \ell - 1$, then $\mathrm{charge}(EH_{j+1}) = 0$ so the restricted formula on $V_{[0,j+1]}$ has odd charge.

We bruteforce over EH_j. This means creating a full binary tree of depth $2w - 1$ over the variables in EH_j, except for the last one $eh_{w-1,j}$. We set the variables in order of increasing y. At each leaf of the tree, we call $\texttt{MainLoopDP}(j)$. Because we want to learn that $\mathrm{charge}(EH_j) = 1$, as explained above, we decide the last variable $eh'_{w-1,j}$ such that $\mathrm{charge}(EH_j) = 0$ and learn a clause from L_j by solving the inconsistent restricted formula on $V_{[0,j]}$. Then the last variable flips, $\mathrm{charge}(EH_j) = 1$ and we solve the inconsistent restricted formula on $V_{[j+1,j+1]}$, that is, for EV_{j+1}.

If $j = 0$, in the first part of $\texttt{MainLoopDP}(0)$ we bruteforce over EV_0. We state without proof that only at the end of this bruteforce, we get a conflict beyond the assignments to EV_0, and then by Lemma 6.4 with $l = 0$ and $r = 0$, we learn each clause from L_0 (because B is empty and $k = 2w - 1$, that is, the last variable set in EH_0 was a decision). In the second part, we bruteforce over EV_1. Again we state without proof that only at the end of this bruteforce, we get a conflict beyond the assignments to EV_1, and then by Lemma 6.4 with $l = r = 1$, backtracking on the EH_0 set works as usual. At the end of the last call to $\texttt{MainLoopDP}(0)$, all literals in EH_0 were implied, so if $\ell = 1$ we have a top-level conflict and conclude UNSAT; otherwise, by Lemma 6.4 with $l = 0$ and $r = 1$ we learn $\bigvee_{x \in EH_1} x$ and the lowest decision is negated.

Now assume $j > 0$. For the first leaf, $\texttt{MainLoopDP}(j)$ recurses to $\texttt{SolveDP}(j - 1)$. The result of this recursive call, by induction, is to learn L_{j-1} and to learn the first clause of L_j as well. For all other leafs, $\texttt{MainLoopDP}(j)$ bruteforces over EH_{j-1} except the last edge, plus half the edges in EV_j, one edge per edge pair. This is enough to discover UNSAT because EH_{j-1} was learnt. Note that we skip the last edge of both EH_{j-1} and EV_j because these will be unit propagated (for EH_{j-1}, this is because L_{j-1} is learnt; for EV_j, it is because all other edges incident to the same vertex were already assigned). We state without proof that only at the end of this bruteforce, we get a relative top-level conflict. By Lemma 6.4 with $l = 0$ and $r = j$, we learn a clause from L_j. In the second part of $\texttt{MainLoopDP}(j)$, we have $\mathrm{charge}(EH_j) = 1$, so EH_j and EH_{j+1} (or the empty set of edges adjacent to the last column if $j = \ell - 2$) have opposite parity and we discover the conflict by bruteforcing over EV_{j+1}. We state without proof that only at the end of this bruteforce, we get a conflict beyond these assignments. Except for the last leaf, by Lemma 6.4, we backtrack in the intended way. For the last leaf, all assignments to EH_j are implications, by Lemma 6.4 with $l = 0$ and $r = j + 1$, we learn $\bigvee_{x \in EH_{j+1}} x$, and the lowest decision is negated, except if $j = \ell - 2$, in which case we have a top-level conflict and UNSAT is determined.

The time complexity: the non-recursive part of $\texttt{SolveDP}(j)$ creates a search tree of depth $2w$, and calls $\texttt{MainLoopDP}(j)$ at each leaf, generating another search tree of depth at most $3w$, so the time complexity is $\mathrm{O}(2^{5w} \cdot (j + 1))$. It can be verified that at most one edge propagates at each node in all "brute force" trees, so the total cost of the unit propagations is constant.

The space complexity: no clauses are learnt before the recursive call to $\texttt{SolveDP}(j - 1)$. Except for the L_j, all clauses learnt during the "brute force" parts are used in a tree-like manner and removed when we are in a stable state and they are not used as a reason anymore. Except for the recursive call, we use $\mathrm{O}(w)$ variables, so the additional space of other learnt clauses is $\mathrm{O}(w)$. So the space complexity is $\mathrm{O}(2^{2w})$. □

6.2 Space-efficient proof

In this section, we show a CDCL simulation of the space-efficient proof. The simulation strategy is given as a recursive procedure $\texttt{SolveTreelike}(l, r)$ which produces a sequence of decisions.

Procedure Bruteforce(L,`callback`)

Input: A list $L_0, L_1, \ldots, L_{n-1}$, and a callback function `callback`

```
for i ← 0, 1, ..., 2^n − 1 do
    for j ← #trailingzeroes(i, n) − 1, ..., 0 do
        Decide L_{n−1−j} = 0/d
    callback()
```

Procedure MainLoopDP(j)

```
Decide eh'_{w−1,j} = charge(EH_j \ {eh'_{w−1,j}})/d
if j = 0 then
    Bruteforce ([ev_{0,0}, ..., ev_{w−2,0}],None)
else
    if all variables in EH_j are assigned zero then
        SolveDP (j − 1)
    else
        Bruteforce
        ([eh_{0,j−1}, eh'_{0,j−1}, ..., eh_{w−2,j−1}, eh'_{w−2,j−1}, eh_{w−1,j−1}, ev_{0,j}, ..., ev_{w−2,j}],None)
Learn a clause from L_j
Assert eh'_{w−1,j} = 1 − charge(EH_j \ {eh'_{w−1,j}})/u
Bruteforce ([ev_{0,j+1}, ..., ev_{w−2,j+1}],None)
```

The initial call to generate this sequence is $\texttt{SolveTreelike}(0, w-1)$. The clause deletion strategy will be to forget a learnt clause at the earliest stable state in which they do not occur as a reason anymore. We use the same procedure $\texttt{Bruteforce}(L, cb)$ as described in the time-efficient proof.

The strategy bruteforces over the middle column $EH_{\lfloor (w-1)/2 \rfloor}$, and then recursively solves either $V_{[0,\lfloor (w-1)/2 \rfloor]}$ or $V_{[1+\lfloor (w-1)/2 \rfloor, w-1]}$ depending on which part has an odd total charge.

Lemma 6.6. *Consider a subgrid $V_{[l,r]}$ ($0 \leq l \leq r \leq w-1$). Suppose $V_{[l,r]}$ is unassigned and there are no learnt clauses containing variables within $E_{[l,r]}$. Suppose we assign EH_{l-1} (if $l > 0$) and EH_r (if $r < \ell - 1$) such that, in the resulting restricted Tseitin formula on $V_{[l,r]}$, the sum of charges is odd in column l, and even in columns $l+1, \ldots, r$.*

Then $\texttt{SolveTreelike}(l, r)$ is a proof of UNSAT for the restricted problem and backtracks beyond assignments to $E_{[l,r]}$.

Proof. If $l = r$, we state (without proof) that brute forcing over every other edge of EV_l constitutes an UNSAT proof with backtracking beyond assignments to EV_l after the last specified conflict. Otherwise, we first brute force over EH_m. At each iteration of $\texttt{MainLoopTreelike}(l, r)$, we set the last edge in EH_m such that $\text{charge}(EH_m) = 0$. Then we recursively call $\texttt{SolveTreelike}(l, m)$. After executing $\texttt{SolveTreelike}(l, m)$, we get a conflict at the decision level of $eh'_{w-1,m}$. By Lemma 6.4, CDCL flips $eh'_{w-1,m}$ and doesn't backtrack further. Then $\text{charge}(EH_m) = 1$ and we recursively call $\texttt{SolveTreelike}(m+1, r)$. Note that the restricted sum of charges in the vertices of column $m+1$ is now odd, so the preconditions for the inductive step on $[m+1, r]$ hold. After executing $\texttt{SolveTreelike}(m+1, r)$, by Lemma 6.4, CDCL backtracks in the intended way; for the last leaf, all edges in EH_m are implications, so backtracking to the assignment before the recursive call occurs. □

Theorem 6.3 follows from the two previous refutations together with Theorem 6.2.

Procedure SolveDP(j)

```
Bruteforce ([eh_{0,j}, eh'_{0,j}, ..., eh_{w−2,j}, eh'_{w−2,j}, eh_{w−1,j}],MainLoopDP(j))
if j > 0 then Forget all clauses in L_{j−1}
```

Procedure MainLoopTreelike(l,r)

$m \leftarrow \lfloor (l+r)/2 \rfloor$
Decide $eh'_{w-1,m} = \text{charge}(EH_j \setminus \{eh'_{w-1,m}\})/\mathsf{d}$
`SolveTreelike` (l, m)
Assert $eh'_{w-1,m} = 1 - \text{charge}(EH_j \setminus \{eh'_{w-1,m}\})/\mathsf{u}$
`SolveTreelike` $(m+1, r)$

Procedure SolveTreelike(l,r)

if $l = r$ **then**
  `Bruteforce` $([ev_{0,l}, ev_{1,l}, \ldots, ev_{w-2,l}]$,None)
else
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  `Bruteforce` $([eh_{0,m}, eh'_{0,m}, \ldots, eh_{w-2,m}, eh'_{w-2,m}, eh_{w-1,m}]$,`MainLoopTreelike` $(l,r))$

7 Concluding Remarks

In this paper, we present a proof system that closely models conflict-driven clause learning (CDCL) and yields natural measures not only of running time but also of memory usage and number of restarts. To the best of our knowledge, previous papers considered either zero restarts or very frequent restarts, and none of the models captured space. We show that lower bounds on proof size and space in resolution carry over to this CDCL proof system. Furthermore, we establish that currently known trade-offs between size and space in resolution can be transformed into essentially equally strong trade-offs between time and memory usage for CDCL, where the upper bounds are achieved by CDCL without any restarts using the standard 1UIP clause learning scheme, and the lower bounds apply even for arbitrarily frequent restarts and arbitrary clause learning schemes.

The focus of our work is theoretical, namely to see if CDCL proof search is in principle subject to the kind of trade-offs shown previously for the resolution proof system in which it searches for proofs. Since the answer turns out to be yes, an interesting direction for future work would be to investigate experimentally whether anything like these time-space trade-offs show up also in practice (when variable decisions have to be made constructively, typically using the VSIDS decision scheme).

Two other interesting problems are whether CDCL with 1UIP clause learning can simulate general resolution efficiently with respect to both time and space (measuring time only, a polynomial simulation follows from [PD11]), and whether CDCL with 1UIP and without restarts can simulate or be separated from regular resolution. If one believes that a separation should be more likely, a first step could be to revisit the formulas in [BBJ14, BK14] and study them in our more restrictive setting, which more closely models actual CDCL search and hence might make proving lower bounds easier. It should be said, though, that both of these problems still look like formidable challenges, but one could hope that it would be possible to shed new light on them by focusing on a model that better describes how CDCL proof search works.

A more specialized question along the same lines, but still quite intriguing, is what can be said if VSIDS and phase saving is plugged into our CDCL model. The VSIDS heuristic seems like an important part of what makes CDCL SAT solvers so successful in practice, and yet there are also theoretical combinatorial formulas where it seems to be less useful. It would be interesting if one could find explicit examples of formulas where VSIDS in combination with phase saving goes provably wrong compared to the best possible resolution proof, causing a large polynomial or even superpolynomial blow-up in proof size.

Acknowledgements

We are grateful to the anonymous *SAT* conference reviewers for detailed comments that helped improve the exposition in this paper.

The third author performed this work while at KTH Royal Institute of Technology, and most of the work of the second and fourth author was done while visiting KTH. The first, third, fifth, and sixth author were funded by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement no. 279611 as well as by Swedish Research Council grant 621-2012-5645. The third author was also supported by the European Research Council under the European Union's Horizon 2020 Research and Innovation Programme / ERC grant agreement no. 648276.

References

[ABRW02] Michael Alekhnovich, Eli Ben-Sasson, Alexander A. Razborov, and Avi Wigderson. Space complexity in propositional calculus. *SIAM Journal on Computing*, 31(4):1184–1211, 2002. Preliminary version in *STOC '00*.

[AC03] Noga Alon and Michael Capalbo. Smaller explicit superconcentrators. *Internet Mathematics*, 1(2):151–163, 2003.

[AFT11] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, 40:353–373, January 2011. Preliminary version in *SAT '09*.

[AR08] Michael Alekhnovich and Alexander A. Razborov. Resolution is not automatizable unless W[P] is tractable. *SIAM Journal on Computing*, 38(4):1347–1363, October 2008. Preliminary version in *FOCS '01*.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI '09)*, pages 399–404, July 2009.

[BBG+15] Patrick Bennett, Ilario Bonacina, Nicola Galesi, Tony Huynh, Mike Molloy, and Paul Wollan. Space proof complexity for random 3-CNFs. Technical Report 1503.01613, arXiv.org, April 2015.

[BBI12] Paul Beame, Chris Beck, and Russell Impagliazzo. Time-space tradeoffs in resolution: Superpolynomial lower bounds for superlinear space. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC '12)*, pages 213–232, May 2012.

[BBJ14] Maria Luisa Bonet, Sam Buss, and Jan Johannsen. Improved separations of regular resolution from clause learning proof systems. *Journal of Artificial Intelligence Research*, 49:669–703, 2014.

[BG03] Eli Ben-Sasson and Nicola Galesi. Space complexity of random formulae in resolution. *Random Structures and Algorithms*, 23(1):92–109, August 2003. Preliminary version in *CCC '01*.

[BGT14] Ilario Bonacina, Nicola Galesi, and Neil Thapen. Total space in resolution. In *Proceedings of the 55th Annual IEEE Symposium on Foundations of Computer Science (FOCS '14)*, pages 641–650, October 2014.

[BHJ08] Samuel R. Buss, Jan Hoffmann, and Jan Johannsen. Resolution trees with lemmas: Resolution refinements that characterize DLL-algorithms with clause learning. *Logical Methods in Computer Science*, 4(4:13), December 2008.

[BK14] Samuel R. Buss and Leszek Kołodziejczyk. Small stone in pool. *Logical Methods in Computer Science*, 10, June 2014.

[BKS04] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, December 2004. Preliminary version in *IJCAI '03*.

[Bla37] Archie Blake. *Canonical Expressions in Boolean Algebra*. PhD thesis, University of Chicago, 1937.

[BN11] Eli Ben-Sasson and Jakob Nordström. Understanding space in proof complexity: Separations and trade-offs via substitutions. In *Proceedings of the 2nd Symposium on Innovations in Computer Science (ICS '11)*, pages 401–416, January 2011.

[BNT13] Chris Beck, Jakob Nordström, and Bangsheng Tang. Some trade-off results for polynomial calculus. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC '13)*, pages 813–822, May 2013.

[Bon16] Ilario Bonacina. Total space in resolution is at least width squared. In *Proceedings of the 43rd International Colloquium on Automata, Languages and Programming (ICALP '16)*, July 2016. To appear.

[BS97] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97)*, pages 203–208, July 1997.

[BS14] Paul Beame and Ashish Sabharwal. Non-restarting SAT solvers with simple preprocessing can efficiently simulate resolution. In *Proceedings of the 28th National Conference on Artificial Intelligence (AAAI '14)*, pages 2608–2615. AAAI Press, July 2014.

[BW01] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow—resolution made simple. *Journal of the ACM*, 48(2):149–169, March 2001. Preliminary version in *STOC '99*.

[CR79] Stephen A. Cook and Robert Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, March 1979.

[CS76] Stephen A. Cook and Ravi Sethi. Storage requirements for deterministic polynomial time recognizable languages. *Journal of Computer and System Sciences*, 13(1):25–37, 1976. Preliminary version in *STOC '74*.

[CS80] David A. Carlson and John E. Savage. Graph pebbling with many free pebbles can be difficult. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC '80)*, pages 326–332, 1980.

[CS82] David A. Carlson and John E. Savage. Extreme time-space tradeoffs for graphs with small space requirements. *Information Processing Letters*, 14(5):223–227, 1982.

[CS88] Vašek Chvátal and Endre Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, October 1988.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[ET01] Juan Luis Esteban and Jacobo Torán. Space bounds for resolution. *Information and Computation*, 171(1):84–97, 2001. Preliminary versions of these results appeared in *STACS '99* and *CSL '99*.

[Hak85] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2-3):297–308, August 1985.

[HBPV08] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively P-simulate general propositional resolution. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI '08)*, pages 283–290, July 2008.

[Kla85] Maria M. Klawe. A tight bound for black and white pebbles on the pyramid. *Journal of the ACM*, 32(1):218–228, January 1985. Preliminary version in *FOCS '83*.

[LT82] Thomas Lengauer and Robert Endre Tarjan. Asymptotically tight bounds on time-space trade-offs in a pebble game. *Journal of the ACM*, 29(4):1087–1130, October 1982. Preliminary version in *STOC '79*.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 530–535, June 2001.

[MS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. Preliminary version in *ICCAD '96*.

[Nor12] Jakob Nordström. On the relative strength of pebbling and resolution. *ACM Transactions on Computational Logic*, 13(2):16:1–16:43, April 2012. Preliminary version in *CCC '10*.

[Nor16] Jakob Nordström. New wine into old wineskins: A survey of some pebbling classics with supplemental results. Manuscript in preparation. To appear in *Foundations and Trends in Theoretical Computer Science*. Current draft version available at `http://www.csc.kth.se/~jakobn/research/`, 2016.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[PD11] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175:512–525, February 2011. Preliminary version in *CP '09*.

[PH70] Michael S. Paterson and Carl E. Hewitt. Comparative schematology. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 119–127, 1970.

[Pip80] Nicholas Pippenger. Pebbling. Technical Report RC8258, IBM Watson Research Center, 1980. in Proceedings of the 5th IBM Symposium on Mathematical Foundations of Computer Science, Japan.

[Urq87] Alasdair Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, January 1987.

[Van05] Allen Van Gelder. Pool resolution and its relation to regular resolution and DPLL with clause learning. In *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '05)*, volume 3835 of *Lecture Notes in Computer Science*, pages 580–594. Springer, 2005.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '01)*, pages 279–285, November 2001.