Dual Depth First Search for Binary Clause Reasoning

Sam Buss
Department of Mathematics, University of California, San Diego, La Jolla, California, USA

Oliver Kullmann
Department of Computer Science, University of Swansea, Swansea, Wales, UK

Virginia Vassilevska Williams
EECS and CSAIL, Massachusetts Institute of Technology, Boston, Massachusetts, USA

Abstract

We present a new algorithm called DualDFS which analyzes a set of binary clauses to determine the complete backbone of forced literals. DualDFS generalizes the failed literal algorithm by starting with a chain S of implications and using a dual depth-first search to find all literals that can be seen to be forced true or false via a literal in S. Experiments indicate that DualDFS performs comparably to, or better than, the state-of-the-art method KB3 of Frolysks, Yu, and Biere (2023) on sets of binary clauses arising in SAT competitions, and that it performs substantially better on many hard cases. The performance of DualDFS is analyzed on some crafted hard instances of binary clause reasoning. We give a reduction from the problem of detecting k-cycles in directed graphs to the problem of finding even a single forced literal in binary clause reasoning. Thus a sub-quadratic time algorithm for detecting backbone variables in binary clauses would improve on the best known algorithms for k-cycle detection. Due to known reductions from Max-k-SAT to cycle detection, a near-linear time algorithm for the 2-CNF backbone would imply $O((2-\delta)^n)$ time algorithms for $\delta > 0$ for Max-k-SAT for all constants k, resolving a major open problem.

2012 ACM Subject Classification Theory of computation → Theory and algorithms for application domains

Keywords and phrases satisfiability, binary clauses, forced literal, backbone

Latest revision May 6, 2024

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

The satisfiability problem (SAT) is to determine whether a Boolean formula in conjunctive normal form (CNF) has a satisfying assignment. A simple, but important, special case is 2-SAT, or "binary clause reasoning". There are well-known linear time algorithms for determining the satisfiability of instances of 2-SAT, the first by Even-Itai-Shamir [13]. Other linear time algorithms are given by Van Gelder [22], Del Val [10, 11] and Froleyks-Yu-Biere [14]. Aspvall-Plass-Tarjan [5] gave a linear time algorithm for quantified 2-SAT.

For satisfiable instances of 2-SAT, it is useful to identify the "backbone" variables. The *backbone* of a Boolean formula is the set of variables that have the same value $\tau(x)$ in all satisfying assignments τ. Janota, Lynes and Marques-Silva [17] defined the notion of backbone for general CNF formulas, and they and Biere-Froleyks-Wang [7] give algorithms for approximating the backbone of general CNF formulas, by finding a subset of the backbone. Froleyks-Yu-Biere [14] gave an algorithm KB3 that identifies backbone literals in a 2-CNF. Their algorithm is used to find the entire backbone in CadiBack [7]. See Section V of [14] for more discussion on prior methods for approximating the backbone.

A contribution of the present paper is that a subquadratic time algorithm for computing the entire backbone would have unexpected implications for algorithms for detecting k-cycles in graphs, and a linear time algorithm would imply a breakthrough for Max-k-SAT algorithms. Related hardness results for the backbone were given by Järvisalo-Korhonen [18] who proved that if the Strong Exponential Time Hypothesis (SETH) holds then there is no linear time algorithm for determining whether a Horn 3-SAT has a backbone literal, or even a failed literal. It is still open whether one can base the hardness of backbone computation for 2-CNFs on SETH. Section 4, however, uses a reduction by Lincoln, Vassilevska W. and Williams [19] from Max-k-SAT to k-cycle detection in directed graphs, to show that a linear time algorithm for the backbone of 2-CNFs would imply that for every constant $k \geq 3$, there is an $\varepsilon > 0$ and an $O((2-\varepsilon)^n)$ time algorithm for Max-k-SAT, resolving a major open problem in SAT algorithms. Even an $O(m^{4/3-\varepsilon})$ time algorithm for the backbone would have significant consequences as it would imply a faster algorithm for Max-3-SAT.

In comparison with finding the entire backbone, it is seemingly an easier task to find the sets of literals which are equivalent due to being in the same strongly connected components of the "binary implication graph" (this graph is defined below). We call such literals "SCC-equivalent".[1] There are many linear time algorithms for finding strongly connected components, notably a one-pass algorithm due to Tarjan [21] and a two-pass algorithm due to Kosaraju, see [20]. LInear time algorithms for finding SCC-equivalent literals are given by Van Gelder [22], Del Val [11] and Heule-Järvisalo-Biere [16].

Binary clause reasoning can be an important adjunct for SAT solving of general CNF formulas. Finding forced (backbone) literals is particularly useful, as they can be immediately eliminated. Many SAT solvers have incorporated binary clause reasoning, e.g., Bacchus's system 2CLS+EQ [6]. Heule-Järvisalo-Biere [16] introduced a number of in- and pre-processing techniques that depend on binary clause reasoning. For them, it was also useful to detect implied 2-clauses. The present paper does not address this problem, however.

A principal contribution of the present paper is the new DualDFS algorithm for finding all backbone literals. The idea behind DualDFS is a generalization of the technique of failed literals. The failed literal method is based on the fact that a literal ℓ is forced false exactly if setting ℓ true yields a contradiction by unit propagation. The DualDFS algorithm starts with a set S of literals instead of a single literal: in effect, it does unit propagation on every literal in S at once and finds not only finds all failed literals in S but other failed literals as well. The set S will consist of literals in an implication chain, and the DualDFS algorithm handles S in the same time that it would take to handle any one literal in S (modulo a small constant factor runtime overhead).

Sections 1 and 2 give preliminaries and describe the Dual DFS algorithm. Section 3 shows experimental results. On crafted examples, Dual DFS provides substantial improvements, and on examples from the SAT competitions, the Dual DFS is comparable to the KB3 algorithm. Section 4 shows that the existence of sufficiently good algorithms for the backbone of 2-SAT instances imply breakthroughs in algorithms for k-cycle detection and MAX-k-SAT.

Preliminaries We adopt the usual conventions for Boolean variables and literals, CNF formulas and sets of clauses, truth assignments, and satisfiability. The notations $C_{\restriction\tau}$ and $\Gamma_{\restriction\tau}$ indicate the result of applying a partial assignment to a clause C or a CNF formula Γ and simplifying. A *unit clause* is a clause of size 1. The closure of Γ under unit propagation can be carried out by the linear-time algorithm `UnitPropagate` shown in the appendix.

This paper is concerned exclusively with sets of 2-clauses. We may assume w.l.o.g. that Γ

[1] Note that the literals in the backbone that are forced true (or alternatively, false) are equivalent, but may not be SCC-equivalent.

Γ	
$\overline{a} \vee b$	$\overline{a} \vee \overline{b}$
$\overline{b} \vee c$	$\overline{a} \vee b$
$\overline{c} \vee d$	$\overline{a} \vee b$
$\overline{c} \vee e$	

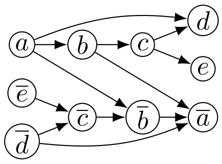

Figure 1 A set Γ of 2-clauses and its graph $\textsc{Big}(\Gamma)$. The literal a is a failed literal.

does not contain a unit clause, since otherwise we can invoke `UnitPropagate()` to obtain a truth assignment τ, and use $\Gamma_{\restriction\tau}$ instead of Γ. Let n be the number of distinct variables in Γ and m the number of clauses. Then the `UnitPropagate` algorithm runs in time $O(m)$.

A *failed literal* is a literal ℓ such that `UnitPropagate` $(\Gamma_{\restriction \ell\mapsto\top})$ yields a contradiction. (The notation "$\ell \mapsto \top$" denotes the minimal truth assignment that maps ℓ to $\top$.) When Γ is a set of 2-clauses, ℓ is a failed literal if and only $\Gamma \vDash \overline{\ell}$; i.e., if and only if ℓ is assigned the value false by every truth assignment satisfying Γ.

A set Γ of 2-clauses can be represented by a directed graph $\textsc{Big}(\Gamma)$, called the *Binary Implication Graph* [5]. The vertices of the binary implication graph are the literals of variables appearing in Γ. Thus, if Γ uses n distinct variables, $\textsc{Big}(\Gamma)$ has $2n$ vertices. For literals ℓ and p, there is an edge from ℓ to p in $\textsc{Big}(\Gamma)$ if only if the clause $\overline{\ell} \vee p$ is in Γ. Hence there is an edge from ℓ to p if and only if there is an edge from $\overline{p}$ to $\overline{\ell}$. It follows that if Γ has m clauses, then $\textsc{Big}(\Gamma)$ has $2m$ edges. An example is shown in Figure 1.

We write $\ell \to^* p$ to indicate there is a (directed) path in $\textsc{Big}(\Gamma)$ from ℓ to p. The length of the path is the number of edges on the path. If fhe length is zero, then $\ell = p$.

▶ **Proposition 1.** *Let Γ be a consistent set of 2-clauses. The following are equivalent for a literal ℓ: (a) $\Gamma \vDash \overline{\ell}$, (b) ℓ is a failed literal for Γ, (c) $\ell \to^* \overline{\ell}$ in $\textsc{Big}(\Gamma)$, and (d) There is a literal p such that $\ell \to^* p$ and $\ell \to^* \overline{p}$.*

Furthermore any literals ℓ and p, the following are equivalent: (f) $\Gamma \vDash (\ell \to p)$, (g) There is a path $\ell \to^* p$ in $\textsc{Big}(\Gamma)$, and (h) If $\sigma =$ `UnitPropagate`$(\Gamma_{\restriction\ell})$, then $\sigma(p) = \top$.

▶ **Definition 2.** *Let Γ be a set of 2-clauses. The literals ℓ and p are* SCC-equivalent *provided that there are paths $\ell \to^* p$ and $p \to^* \ell$.*

In other words, two literals are SCC-equivalent provided they are in the same strongly connected component of $\textsc{Big}(\Gamma)$. By duality, the literals ℓ and p are SCC-equivalent if and only if $\overline{\ell}$ and $\overline{p}$ are SCC-equivalent. Every literal is SCC-equivalent to itself.

As already discussed, there are efficient, linear time algorithms for determining the strongly connected components of a directed graph. Thus, we can just assume, without loss of much generality, that Γ has no non-trivial SCC-equivalences. Otherwise we can preprocess Γ to identify the strongly connected components and identify SCC-equivalent literals with a single literal. Nonetheless, our DualDFS algorithm presented in the next section is formulated to work in the presence of non-trivial SCC-equivalences, since there is very little overhead needed to accommodate non-trivial SCC-equivalences. Furthermore, in practice, one may wish to identify backbone literals without first finding all SCC-equivalences, e.g. if there are not very SCC-equivalences. (It would be possible to modify the DualDFS algorithm to identify SCC-equivalent literals on the fly; however this would be advantageous only in cases where there are not very many SCC-equivalent literals.)

2 The Dual-DFS Algorithm

We assume henceforth that Γ is a set of 2-clauses. The goal of the Dual-DFS algorithm is to identify the backbone, namely to identify all literals that are forced true ($\top$) by Γ and thereby all literals that are forced false ($\bot$).

The failed literal method can be used to determine if a literal ℓ is in the backbone by checking, firstly whether $\ell \to \overline{\ell}$ and secondly whether $\overline{\ell} \to \ell$. These two conditions can be checked by performing depth-first search (DFS) in $\textsc{Big}(\Gamma)$ starting from ℓ and a second DFS starting from $\overline{\ell}$. The Dual-DFS algorithm generalizes this idea by starting a depth-first search from a *set* of literals S. For this, we define:

▶ **Definition 3.** *Let S be a set of literals. We say that ℓ is* forced false via S *provided that there is a literal $p \in S$ such that*

$$\ell \to^* p \qquad \textit{and} \qquad p \to^* \overline{\ell}.$$

The notion of being forced true via S is defined dually, swapping ℓ and $\overline{\ell}$.

In other words, ℓ is forced false via S provided there is a path from ℓ to a member of S and continuing on to $\overline{\ell}$. It is allowed that p is equal to ℓ if $\ell \in S$.

▶ **Definition 4.** *A literal ℓ is* in the backbone via S *provided that ℓ is forced either true or false via S.*

▶ **Proposition 5.** *Suppose $\ell \in S$. Then ℓ is forced true (respectively, false) if and only ℓ is forced true (false) via S. Thus, ℓ is in the backbone if and only if ℓ is in the backbone via S.*

Proof. Suppose ℓ is forced true. Then there is a path $\overline{\ell} \to^* \ell$. Taking $p = \ell$, we have $p \to^* \ell$ by a path of length zero. It follows that ℓ is forced true via S. ◀

Clearly, if $S' \supset S$ and if ℓ is forced true (respectively, false) via S, then ℓ is also forced true (false) via S'. Also, if ℓ is forced true (respectively, false) via S, then there is singleton subset $\{p\} \subseteq S$ such that ℓ is forced true (or false) via $\{p\}$.

The notion of a *source* or *sink* (in $\textsc{Big}(\Gamma)$) is defined as usual. Thus ℓ is a source if there is no literal p such that $p \to \ell$. Similarly, it is a sink if there is no literal p such that $\ell \to p$.

▶ **Proposition 6.**
(a) *If ℓ is a source or a sink, then no literals forced true or false via $\{\ell\}$ other than possibly ℓ.*
(b) *If ℓ is forced true (false), then there is some non-sink/non-source literal p such that ℓ is forced true (resp., false) via $\{p\}$.*
(c) *A literal ℓ is forced true (false) via $\{p\}$ if and only if it is forced true (resp., false) via $\{\overline{p}\}$.*

Proof. Suppose p is not equal to ℓ and is forced true via $\{\ell\}$. By definition, this means $p \to^* \ell$ and $\ell \to^* p$. The former is impossible if ℓ is a source, and the latter is impossible if ℓ is a sink. This proves (a). For (b), suppose $\overline{\ell} \to \ell$, so there is a path from $\overline{\ell}$ to ℓ. Since Γ is a set of 2-clauses, there is no edge from $\overline{\ell}$ to ℓ in $\textsc{Big}(\Gamma)$ since this edge would be present only if the clause $\{\overline{\overline{\ell}}, \ell\} = \{\ell\}$ were in Γ and this is a unit clause. Therefore, the path from $\overline{\ell}$ to ℓ has a path of length at least two, and we take the literal p to be literal in the interior of the path. To prove (c), note that, by duality, $\overline{\ell} \to^* p \to^* \ell$ holds if and only if $\overline{\ell} \to^* \overline{p} \to^* \ell$, ◀

At a high level, the Dual-DFS algorithm operates as shown in `DualDFS_high_level()` by repeatedly choosing a new set S. From the above propositions, the suitable sets S can be chosen so that

Input: A set Γ of 2-clauses.
Return value: Whether Γ is consistent and, if so, the entire backbone.
Function `DualDFS_high_level(Γ)`
  loop
    Choose a suitable set S of literals.
    if *no suitable S is found* **then**
      halt: Γ is satisfiable; the entire backbone is discovered.
    Either discover Γ is unsatisfiable and halt; or, Identify all literals forced true or false via S.
    Let τ be the associated partial truth assignment.
    Apply τ, i.e. replace Γ with $\Gamma_{\upharpoonright\tau}$
    foreach *literal s_i in S* **do**
      Mark s_i and $\overline{s_i}$ as handled.

0. S is closed under complementation, i.e., for all literals $p \in S$ iff $\overline{p} \in S$.
1. S does not include any literals that have already been marked as “handled”.
2. S does not include any literal which is a source or a sink.

Items 0., 1. and 2. are justified by the earlier propositions. Furthermore, for our particular choices of S, there will never be a significant advantage to putting a source or sink literal in S, and there could be disadvantages. We impose also a fourth condition on the sets S:

3. The set S is a set of literals $S = \{s_1, \ldots, s_k\}$ so that $s_{i+1} \to s_i$ for all $i < k$. That is, Γ contains the clauses $s_i \vee \overline{s+1}$.

We will choose sets S that satisfy conditions 1.-3. as well as the condition 4. below. An example is shown in Figure 1. Condition 0. will not be satisfied, but instead is used to justify the fact that if a literal is handled, so is its complement.

▶ **Theorem 7.** *Let S be a set of literals satisfying 1.-3. A literal ℓ is forced true via S if and only if there are $i \geq j$ such that there is a path in* Big(Γ) *from $\overline{\ell}$ to s_i and a path from s_j to ℓ. When this holds, $i \geq j$ can be chosen so that there is a path from $\overline{\ell}$ to s_i that does not pass through any $s_{i'}$ with $i' > i$, and there is a path from s_j to ℓ that does not pass through any $s_{j'}$ with $j' < j$.*

Proof. Let i be the maximum value such that there is a path from $\overline{\ell}$ to s_i. Also, let j be the minimum value such that there is a path from s_j to ℓ. Suppose i and j exist and $i \geq j$. Then $\overline{\ell} \to^* s_i \to^* s_j \to^* \ell$, and therefore ℓ is forced true via S.

Conversely, suppose that ℓ is forced true via S, so there is a path in Big(Γ) from $\overline{\ell}$ to ℓ that contains at least one member of S. Let i be maximum such that s_i is on the path, and let j be minimum such that s_j is on the path. Then clearly, i and j satisfy the desired conditions of the theorem. ◀

We further impose a maximality condition on S:

4. Let S be as in condition 3. Every literal s_0 such that $s_1 \to s_0$ is an edge in Big(Γ) is a sink. Every literal s_{i+k} such that $s_{k+1} \to s_k$ is an edge in Big(Γ) is a source.

The additional Condition 4. can be satisfied without loss of generality. This is because if s_1 implies some non-sink s_0, then S can be extended by adding s_0. Similarly, if s_k is implied by some non-source s_{k+1}, S can be extended by adding s_{k+1}.

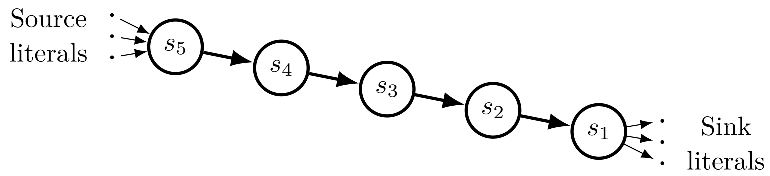

Figure 2 A set $S = \{s_1, \cdots, s_5\}$ satisfying conditions 1.-4. The boldface circles and edges are used here and in later figures to indicate that literals are in the set S. The literals can have other incoming and outgoing edges. In the preferred implementation, s_1 has only sinks as children and s_5 has only sources as parents.

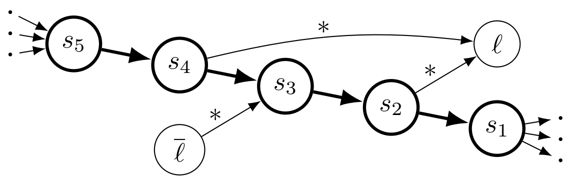

Figure 3 Showing how a failed literal is detected by the Dual-DFS algorithm. The edges labeled with *'s indicate paths of length ≥ 0 that do not involve any other s_i's. During the first phase of the Dual-DFS algorithm, the node ℓ is reached via the depth-first search from s_2. It is not re-traversed when later continuing the depth-first search from s_4. The second phase of the Dual-DFS algorithm discovers the path from $\overline{\ell}$ to s_3 (or, rather, a path from $\overline{s}_3$ to ℓ). At this point, since $3 \geq 2$ and thus $s_3 \to^* s_2$, the literal $\overline{\ell}$ is discovered to be a failed literal, so ℓ is forced true.

The advantage of making S bigger to satisfy Condition 4. is that more literals can be marked as handled. Making S bigger by (repeatedly) adding such s_0 or s_k would not be expected to worsen the runtime of the DualDFS algorithm; because it does not cause the DualDFS algorithm to traverse any additional portion of the binary implication graph (although it may traverse it in a different order).

2.1 Overview of the Dual DFS algorithm

For simplicity, suppose that the set S conditions 1.-4. has been fixed. (In actuality, as we discuss later, our preferred implementation dynamically chooses members of S.) The DualDFS algorithm has two phases: the first phase uses a depth-first search to find all literals ℓ implied by members of S. It initially does a depth-first search (DFS) starting at s_1 to find all literals implied by s_1. It then continues the DFS from s_2, but does not revisit literals already found to be implied by s_1. It next does a DFS for all literals implied by s_3, etc., for all s_i. The literals ℓ found during the first phase are marked with a time value i indicating that s_i was the first member of S found to imply ℓ. If this first phase, encounters a literal ℓ and later encounters its complement $\overline{\ell}$, then it must be that $\ell \to^* \overline{\ell}$. (This fact requires proof, see Theorem 8.) Figure 5 shows this situation. Thus $\overline{\ell}$ is forced true and ℓ is forced false. Unit propagation is used to set true all literals p such that $\overline{\ell} \to^* p$. If this reveals a contradiction by setting some literal both true and false, Γ is inconsistent and the algorithm terminates.

Another way that the first phase can find a forced literal is shown in Figure 4. This is the situation where the depth-first search below s_i encounters $\overline{s}_j$ for some $j > i$. In this case, s_j is forced false and it can be unit propagated immediately upon recognizing it as forced.

The first phase of DualDFS terminates when it runs out of s_i's in S or if it reaches an s_i that has been set false by unit propagation. However, our preferred implementation, shown

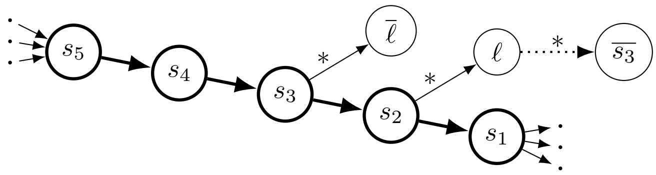

Figure 4 The situation when both ℓ and $\overline{\ell}$ are implied by literals s_i and s_j in S. A path $s_j \to^* \overline{\ell}$ and duality imply the existence of a path $\ell \to^* \overline{s_j}$. This means that also some s_i implies some $\overline{s}_j$, for $j \geq i$. Therefore, s_j is forced false. ($i = 2$ and $j = 3$ are shown in the figure.)

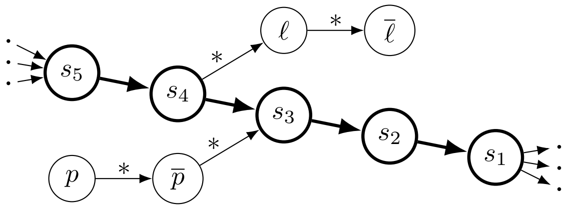

Figure 5 Examples of how a failed literal ℓ or p can be discovered even though it is not forced via S. As shown, ℓ and hence also s_4 are forced false. Likewise, p is forced false, and s_3 is forced true.

below, attempts to recreate the set S in some situations where an s_i is found to be set false.

The second phase of the DualDFS algorithm does a "reverse depth-first search"; namely, it does a depth-first search for literals implied by the $\overline{s}_i$'s, the negations of literals in S. Of course, by duality, if $\overline{s}_i \to^* \ell$, then also $\overline{\ell} \to^* s_i$. The reverse DFS starts from $\overline{s}_k$, then continues from $\overline{s}_{k-1}$, etc. In this way, each literal ℓ found in the second phase knows the largest value of j such that $\ell \to^* s_j$. There are two ways that literals get forced true or false during the second phase. The first way is pictured in Figure 3. Here $\overline{\ell} \to s_3$ and $s_2 \to^* \ell$. Since also $s_3 \to^* s_2$, we have that ℓ is forced true. When this is discovered, the literal ℓ is immediately set true, and unit propagation is performed so that every literal implied by ℓ is set true. If this yields a contradiction, Γ is inconsistent and the algorithm terminates.

The second way that a literal can be forced during the second phase is shown in Figure 5 for the literal p. Here p is forced false, but not via S. When this situation arises during the second phase, $\overline{p}$ is set false and unit propagation is carried out to set every literal implied by $\overline{p}$. Again, this may discover a contradiction, in which case Γ is inconsistent.

The second phase terminates if it reaches an s_i which has been forced true, otherwise it terminates after processing s_1.

▶ **Theorem 8.** *Suppose that ℓ and later $\overline{\ell}$ are encountered during the first phase of DualDFS and that $\ell, \overline{\ell} \notin S$. Then:*

(a) *For the least value i such that $s_i \to^* \ell$,*

$$s_i \to^* \ell \to^* \overline{\ell} \to^* \overline{s}_i \tag{1}$$

so s_i and ℓ are failed literals. In addition, i is the least value such that $s_i \to^* \overline{\ell}$.

(b) *It is possible that $\overline{\ell} \to^* s_{i'}$ for some $i' < i$. In this case $s_{i''}$ will be set true for all $i'' \leq i'$ while unit propagating after setting ℓ false.*

(c) *$\overline{\ell} \to^* \overline{s}_j$ for all $j \geq i$.*

(d) *$\overline{\ell} \to^* \overline{s_{i'}}$ does not hold for any $i' < i$.*

(e) *If unit propagation after setting ℓ false yields a contradiction, then Γ is unsatisfiable.*

Proof. To prove (a), let j be the least value such that $s_j \to^* \overline{\ell}$. Since ℓ was encountered before $\overline{\ell}$, $j \geq i$. Duality gives $\overline{\ell} \to^* \overline{s}_j$; thus $s_j \to^* s_i \to^* \ell \to^* \overline{s}_j$, So s_j is forced false. If $j > i$, the first phase never carries out the depth-first search starting from s_j, since s_j will be recognized as forced false by virtue of having been encountered during the depth-first search below s_i. This contradicts the hypothesis that $\overline{\ell}$ is encountered in the first phase. Therefore, $j = i$. That is, $s_i \to^* \overline{\ell}$. By duality, $\ell \to^* \overline{s_i}$. That is, we have $s_i \to^* \ell \to^* \overline{s}_i$ The depth-first search from s_i reaches ℓ, and continues to (possibly) eventually reach $\overline{s_i}$. If it reaches $\overline{\ell}$ first, then (1) holds. Otherwise, it reaches $\overline{s_i}$ and sets s_i false and the first phase terminates without reaching $\overline{\ell}$, contradicting the hypothesis that both ℓ and $\overline{\ell}$ were encountered.

Part (b) is obvious. Part (c) follows from (a) since $\overline{s_i} \to^* \overline{s_j}$ for $i \geq i$. To prove (d), note that if $\overline{\ell} \to^* \overline{s_{i'}}$ then by duality, $s_{i'} \to^* \ell$. If $i' < i$, this contradicts the choice of i. Part (e) is immediate from the fact that ℓ is forced false. ◀

A dual theorem holds for the second phase of the Dual DFS algorithm:

▶ **Theorem 9.** *Suppose that the depth-search during the second phase of DualDFS encounters p and then $\overline{p}$, and that $p, \overline{p} \notin S$. Let i be maximum such that $\overline{s}_i \to^* p$.*

(a) *We have $\overline{s_i} \to^* p \to^* \overline{p} \to^* s_i$. Therefore $\overline{s_i}$ and p are failed literals. In addition, i is the maximum value such that $\overline{s_i} \to^* p$. (This is shown in Figure 5 with $i = 3$.)*

(b) *It is not possible that $\overline{p} \to^* \overline{s_{i'}}$ for $i' > i$.*

(c) *$\overline{p} \to^* s_j$ for all $j \leq i$.*

(d) *$\overline{p} \to^* s_{i'}$ does not hold for any $i' > i$.*

(e) *If unit propagation after setting p false yields a contradiction, then Γ is unsatisfiable.*

Proof. Part (a) is proved using the construction of the proof of Theorem 9 to establish that $p \to^* \overline{p} \to^* \overline{s_i}$ holds. For part (b), if $\overline{p} \to^* \overline{s_i}$, then by duality, $s_i \to^* p \to^* \overline{p}$ and then p would have been forced false during the first phase of the depth-first search. The proofs of (c), (d), and (e) are very similar to the proof of Theorem 8. ◀

▶ **Theorem 10.** *Suppose the DualDFS algorithm discovers i as the first (and thus minimum) value such that $s_i \to^* \ell$, and j as the first (and thus maximum) value such that $\overline{s_j} \to^* \ell$ (equivalently, $\overline{\ell} \to s_j$). Also suppose $j \geq i$. (See Figure 3.) Then $\overline{\ell}$ is a failed literal, and ℓ is forced true.*

(a) *It is possible that $\ell \to^* s_{i'}$ for some $i' < j$. (This allows $i' > i$ in which case ℓ, s_i and $s_{i'}$ are SCC-equivalent.) In this case, $s_{i''}$ is forced true for all $i'' \leq i'$.*

(b) *It is not the case that $\ell \to^* s_{j'}$ for any $j' > j$.*

(c) *It is not possible that $\ell \to^* \overline{s_{i'}}$ for any i'.*

(d) *If unit propagation after setting ℓ true yields a contradiction, then Γ is unsatisfiable.*

Proof. The fact that ℓ is a failed literal is from Theorem 7. Parts (a) and (d) are obvious. To prove (b) note that that if $\ell \to^* s_{j'}$, then $\overline{s_{j'}} \to \overline{\ell}$ and this contradicts the choice of j. To prove (c), note that if $\ell \to^* \overline{s_{i'}}$, then $s_{i'} \to^* \overline{\ell}$, and by Theorem 8, ℓ would have already been set either false or true during the first phase. ◀

2.2 The Dual DFS Algorithm

We now describe our preferred implementation of the Dual DFS algorithm for finding all of the forced literals, that is, the entire backbone.

All literals in Γ are initialized as "not handled". We define a *generalized sink* to be a not-handled literal such that every child is labeled as handled. Each iteration of the Dual

DFS algorithm starts by identifying a set $S = \{s_1, \ldots, s_k\}$ that satisfies Conditions 1.-3. and such that every child of s_1 is a generalized sink.

The DualDFS algorithm first invokes a routine `InitializeS()` that initializes "half" of the set S. It starts with a literal t_0 and produces a set S equal to $\{s_1, \ldots, s_k\}$ so that $s_k = t_0$, $s_{k-i} = t_i$, and $s_i \to s_{i+1}$ for each i. `InitializeS()` starts at t_0, and greedily chooses each t_{i+1} as the first child of t_i that is not a generalized sink. The other "half" of S, that is the part above s_k, will be dynamically generated during the first phase of DualDFS as described below. It is possible that `InitializeS()` finds a contradiction in Γ. It is also possible that `InitializeS()` discovers that t_i is in the backbone, and so forced true or false. In this case, there may be no set S created. A detailed description of `InitializeS()` is in the appendix.

The notation $\mathsf{var}(\ell)$ denotes the variable underlying a literal ℓ. That is, $\mathsf{var}(x) = \mathsf{var}(\overline{x}) = x$ for x a variable. Each variable x has associated values $x.\mathsf{sign}$ and $x.\mathsf{time}$, along with flags indicating whether it has been handled and whether it has been assigned a value. If x is assigned a value, then $x.\mathsf{sign} \in \{+, -\}$ indicates whether it is has been assigned the value true $(+)$ or false $(-)$. The sign of a literal ℓ, denoted $\mathsf{sign}(\ell)$, equals either $+$ or $-$ depending on whether ℓ is a variable or a negated variable. $x.\mathsf{time}$ indicates the least value i such that s_i implies either x or $\overline{x}$ as discovered during the first phase of DualDFS.

The routine `ForceImmediate` sets a literal true and unit propagates as much as possible. Variables are marked as handled when they are forced either true or false. A stack is used to hold literals for unit propagation. The detailed algorithm is shown in the appendix.

The first phase of the DualDFS algorithm carries out the depth-first search from the variables s_i in the (partially formed) set S. The code for the first phase is in Algorithm `DualDFS_Phase1`(ℓ). The values $x.\mathsf{time}$ are initialized to equal ∞, indicating that the variable x has not been encountered yet. At the end of the first phase, the set S is finalized, and k is updated to equal the (possibly new) size of S.

The inner while loop of `DualDFS_Phase1` performs the depth-first search from s_i. Literals encountered during the DFS are checked for being handled when they are popped from the stack (see line 9), since it is possible that a literal becomes handled (by virtue of being assigned a value true or false) after it is pushed onto the stack. Lines 10 and 11 check whether the literal ℓ has just been encountered for the first time, and if so the time-stamp value for ℓ is to indicate that i is the least value such that $s_i \to^* \ell$. Lines 13 and 14 unit propagate a literal ℓ that is found to be forced true in the fashion pictured in Figure 5 (with ℓ and $\overline{\ell}$ interchanged). Note that it is possible that ℓ is equal to $\overline{s_i}$. In any event, unit propagating ℓ true, sets $\overline{\ell}$ false and thereby sets s_i false. This also sets s_j false for every $j > i$. For this reason the `DualDFS_Phase1` algorithm halts in this event. When starting with a new S-variable s_i, line 4 checks whether the literal $\overline{s_i}$ has already been encountered with the opposite sign. If so, $s_i \to^* \overline{s_i}$, so s_i is a failed literal. The last part of the outer while loop checks whether s_{i+1}, the next member of S, has been handled. This can happen due to s_{i+1} being set false while unit propagating a literal ℓ. If so, an alternative literal is chosen for s_{i+1}. This is strictly speaking not necessary, as it would be acceptable for `DualDFS_Phase1` to just remove all literals s_j with $j \geq i$ from S; but we include it in our preferred implementation to try to speed up the process of finding forced literals from a larger set S.

The second phase of the DualDFS algorithm looks for variables that are forced either via S (as shown in Figure 3) or not via S (such shown by the literal p in Figure 5). Every literal visited during Phase 2 has its time value, $\mathsf{var}(\cdot).\mathsf{time}$, set to k^*: this controls the depth-first searches. When $\mathsf{var}(\ell).\mathsf{time}$ is not equal to k^* and is $\leq i$, lines 10 and 11 handle the case where ℓ is a failed literal that is forced via S as shown in Figure 3 (with the roles of ℓ and $\overline{\ell}$ interchanged again). When $\mathsf{var}(\ell).\mathsf{time} = k^*$ the same lines handle the case where ℓ is

Input: $s_1, \ldots, s_k$ as chosen by `InitializeS`
Effect: Variables encountered during the DFS have their sign and time set. Some variables may be forced true or false.
Return value: *true* if no contradiction is found, otherwise *false*.

```
Function DualDFS_Phase1(ℓ)
    i := 1
    while s_i exists do
        if var(s_i).time ≠ ∞ and sign(s_i) ≠ var(s_i).sign then
            return ForceImmediate(¬s_i)
        Push s_i onto the DFS stack as its only member
        while the DFS stack is not empty do
            Pop literal ℓ from the DFS stack
            if ℓ is not handled then
                if var(ℓ).time equals ∞ then
                    var(ℓ).time := i;  var(ℓ).sign := sign(ℓ)
                    Push each child p of ℓ onto the DFS stack
                else if sign(ℓ) ≠ var(ℓ).sign then
                    return ForceImmediate(¬s_i)
        if s_{i+1} does not exist or is handled then
            Unassign the values s_j for all j > i (if any)
            if s_i has a parent p that is not handled and not a generalized source then
                s_{i+1} := p
        i := i + 1
    k := i − 1, so S = {s_1, ..., s_k}
    return true
```

found to be forced false as shown in Figure 5 (with $\overline{p}$ and p playing the roles of ℓ and $\overline{\ell}$). In the latter case, the call to `ForceImmediate`($\overline{\ell}$) will always set s_i true. If there are SCC-equivalent literals, it is also possible that s_i is set true in the former case as well. If s_i is set true, then s_j is set true for all $j < i$; therefore, `DualDFS_Phase2` stops when this happens.

The overall DualDFS algorithm is shown in the algorithm on page 11. Every literal that becomes part of the set S formed by `InitializeS`(p) is handled by the calls to `DualDFS_Phase1`() or `DualDFS_Phase2`(), either by being forced true or false or by remaining in S until the end of the second phase. Since handled literals never become unhandled, the while loop of line 3 needs to consider each potential p only once.

2.3 Examples and runtime analyses

Figure 6 shows four examples of CNF formulas. *vglayers* and *fyb_rakes* were identified by [14] as hard cases for their solver. The CNFs *fyb_rakes* turn out to be very simple for Dual DFS: the only possible sets S consist of all the P-variables, or dually the $\neg P$ variables, except the first one. Once the set S is processed (in linear time), no further work is needed. For similar reasons, the CNFs are very easy for Dual DFS, as only one set S needs to be considered.

The *vglayers* CNF is based on [22]. There are r many groups of literals ($r = 4$ in the figure), each with p many variables. The literals in the first half of the groups are negated. For each x in the i-th group and y in the $(i+1)$st group, there is a clause $\overline{x} \vee y$. Thus there

Input: $s_1, \ldots, s_k$ as updated by `DualDFS_Phase1`.
Effect: Variables encountered during the DFS have their time set to $k{+}1$. Some variables may be forced true or false.
Return value: *true* if no contradiction is found, otherwise *false*.

Function `DualDFS_Phase2`()
    $k^* := 1 + (\max k$ value used during Phase 1)
    for $i := k,\ k{-}1, \ldots, 2,\ 1$ **do**
        Set var(s_i).time $:= k^*$
        Push $\overline{s_i}$ onto the DFS stack as its only member
        while *the DFS stack is non-empty* **do**
            Pop literal ℓ from the DFS stack
            if *ℓ is not handled* **then**
                if var(ℓ).time $= k^*$ *or* var(ℓ).time $\leq i$ **then**
                    $r :=$ `ForceImmediate`($\overline{\ell}$)
                    if *(not r) or s_i is handled* **then return** r
                else if var(ℓ).time $\neq k^*$ **then**
                    var(ℓ).time $:= k^*$
                    var(ℓ).sign $:=$ sign(ℓ)
                    foreach *child p of ℓ* **do** push p onto the DFS stack
        Mark s_i and $\overline{s_i}$ as handled
    return true

Input: A set Γ of 2-clauses
Effect: All literals in the backbone are set to their forced values
Return value: *true* if Γ is satisfiable, otherwise *false*

Function `DualDFS_high_level`()
    foreach *variable x* **do** x.time $:= \infty$
    while *there is a literal p which is not handled and not a generalized source or sink* **do**
        `InitializeS`(p)
        if *not* `DualDFS_Phase1`() **then return** false
        if *not* `DualDFS_Phase2`() **then return** false
        foreach *variable x encountered* **do** x.time $:= \infty$
    return true   `// Backbone literals all set to their forced values`

are $n = r \cdot p$ variables and $m = (r-1)p^2$ clauses. The Dual DFS algorithm identifies p many sets S: each S is a chain of implications of literals from the first to the last group. The i-th S has to traverse the entire non-handled portion of the BIG graph, so requires runtime $O((p-i)pr) = O(p^2r)$. The runtime for the Dual DFS algorithm is thus $O(p^3r)$. When $p = r$, the runtime of the Dual DFS algorithm is thus $O(p^4) = O(r^4) = O(m^{4/3})$. If the number of groups is instead set constant, the Dual DFS algorithm requires time $O(p^3) = O(m^{3/2})$. A straightforward failed literal algorithm would take time $O(nm) = O(p^3r^2)$. The Dual DFS algorithm does substantially better in the case $r = p$. The two algorithms have similar asymptotic times when $r = 4$; however, the Dual DFS algorithm would be expected to have an advantage since it handles $r{-}2 = 2$ variables at a time.

The *randlayers* example was picked to be hard for Dual DFS. This CNF family has three

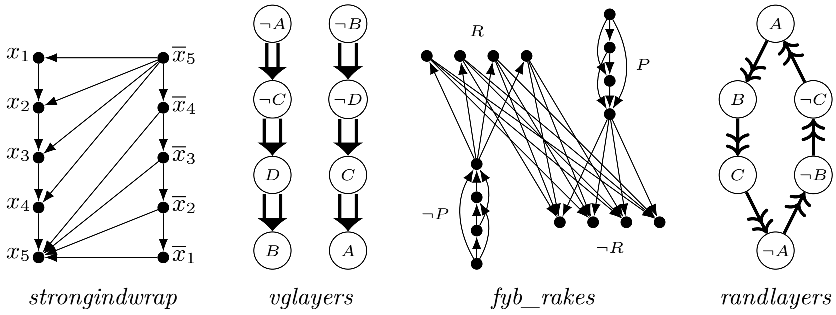

Figure 6 The BIGs for *strongindwrap*, *vglayers*, *randlayers* and *fyb_rakes*. The doubled arrows in *vglayers* indicate all possible (directed) edges are present; e.g., for every $a \in A$ and $c \in C$, the clause $a \vee \bar{c}$ is present. The doubled arrows in *randlayers* indicate that f-many randomly selected outgoing edges are chosen; e.g., for each $a \in A$ the clause $\overline{a} \vee \overline{b}$ is present for f randomly selected $b \in B$. In *fyb_rakes*, the variables $r \in R$ have edges to the literals $\overline{r'}$ for all $r \neq r' \in R$. The figures for *vglayers* and *fyb_rakes* are adapted from [14].

parameters: the number r of groups of variables (equal to $r = 3$ in the figure), the number p of variables per group, and the fanout f. For each variable x in the i-th group, there are f randomly chosen y's in the $(i+1)$st group so that $\overline{x} \vee y$ is a clause. For each x in the last group, there are f randomly chosen y's in the first group so that $\overline{x} \vee \overline{y}$ is a clause. There are $n = p \cdot r = p \log p$ many variables, and $m = p \cdot f \cdot r = f \cdot p \log p$ many clauses.

In the case where p, r and f satisfy $f^r = p$, so some constant fraction of the literals is in the backbone, a *rough* estimate of the runtime shows there are about p many sets S used, each requiring time f^r, so the total runtime is $O(p \cdot f^r) = O(n^2/\log n) = O(nm/\log m)$.

3 Experimental results

We show experimental results comparing DUALDFS to the leading 2-CNF backbone extraction algorithm KB3 from [14], as implemented in CADIBACK-(version 0.2.1). We consider SAT-competition benchmarks and crafted instances. The code for KB3 was obtained from `https://github.com/arminbiere/cadiback`, modified to exit early in case no none-binary clauses are present (so no call to CADICAL is performed). The single option used is `-big-no-els`. Times shown are in seconds and mean user-cpu time, run as a single running process on a multicore machine. Times for DUALDFS are denoted by **tDD**, and times for KB3 within CADIBACK as **tCBB** (for "cadiback BIG"). We used additionally a program `TotalPermutation` that computes a hash-value from an input-DIMACS-file, using it as the seed to the C++ 64-bit Mersenne Twister, and then randomly flips all signs of literals and randomly permutes the variables, the order of the clauses, and the order of literals in the clauses. These "permuted runtimes" are given as **PtDD** and **PtCBB**. The times `(P)tCBB` ignore reading of the input and building the implication graph, but give the pure search time. Similarly, the times `(P)tDD` ignore reading of the input, and additionally the following analytical steps are not counted in the runtime: To obtain a more stable runtime, the input is sorted. To obtain an insight into trivial forms of forced literals x, subsumption-resolution pairs $(a \vee x) \wedge (\neg a \vee x)$ are eliminated and subsequent unit-clause propagation (UCP) run. Then we also always check satisfiability (via a linear-time computation of strong connected components), where a few forced literals may be detected sporadically. For the crafted benchmarks, these analytical steps do not affect `(P)tDD` (besides the sorting of clauses), while for the SAT competition benchmarks we comment on the effects. For all instances, we used a third simple algorithm to verify the correctness of DualDFS and KB3.

SAT competition benchmarks We started with the benchmark set `sc04to22sat`: Sampled and Normalized Satisfiable Instances from the main track of the SAT Competition 2004 to 2022 of [14]. This contains 1798 instances (general CNFs), from which we extracted 1650 2-CNF instances (to be made available online) as follows: UCP was performed, and then the 2-CNF part was extracted, with gaps in the variable-numbering removed. The resulting 145 empty 2-CNFs were removed, and also the three instances with $n \leq 5$, resulting in 148 removed instances. For these 1650 instances: the average number n of variables is $164'017$, with maximum $11'992'725$, while the average number m of clauses is $799'134$, with maximum $134'145'273$ (the average density, that is, $\frac{m}{n}$, is 17.92). 810 instances have no forced literals, while the average number of forced literals is 1839.1, with maximum $104'464$; the average percentage of forced literals (relative to the number of variables) is 1.95%, with maximum 82.56%. We now come to the runtimes. We use a Linux machine with two Intel Xeon Platinum 8168 processors (base frequency 2.7GHs, max frequency 3.7GHz) and 376GiB memory, with gcc and g++ in version 11.4. To give a basic impression, we first report total user-time, which here includes the complete run of the programs, plus the checking of the forced literals (for correctness and completeness): DualDFS took 8m47s, with 15m20s for the permuted versions, while K3B in CadiBack took 15m50s, with 42m52s for the permuted versions. We see that DualDFS is considerably faster, and that both algorithms suffer from the permutation of the inputs, with DualDFS being more stable. The averages for `tDD` and `PtDD` are 0.0942 resp. 0.09416 (hardly any difference), while the averages for `tCBB` and `PtCBB` are 0.3765 resp. 1.045. So for the core runtimes DualDFS is roughly 4x faster on the original instances, and 10x faster on the permuted instances. There are 229 instances with subsumption-resolutions; for these instances the average number of such clause-pairs is 1309 (maximum 94518), where the resulting number of eliminated variables (including UCP) has the average 1999 (maximum 95568). Now the averages for `tDD` and `PtDD` are 0.4177 resp. 0.2667, while the averages for `tCBB` and `PtCBB` are 0.131 resp. 0.4266. (Currently we do not have an explanation for these anomalies.) For the remaining $1650 - 229 = 1421$ instances, then the averages for `tDD` and `PtDD` are 0.04207 resp. 0.06635, while the averages for `tCBB` and `PtCBB` are 0.4161 resp. 1.144.

Crafted benchmarks The table on the next page shows experiments on the crafted instances. Here we use a Linux machine with one AMD EPYC 7443P 24-Core Processor (base frequency 2.85GHs, max frequency 4.0GHz) and 995GiB memory, with gcc and g++ in version 12.3. Missing data means the computation could not be performed due to either missing memory or an exception thrown. `fyb_rakes` uses the same size p for the sets R and P: `tDD` $\approx \Theta(p^2) = \Theta(m)$, `tCBB` $\approx \Theta(p^3) = \Theta(m^{3/2})$. `vglayers` uses $p = r$: For DualDfs the instances are too easy to make a meaningful evaluation, while `tCBB` $\approx \Theta(p^3) = \Theta(m)$. For `randlayers` we use p for the number of groups of variables, 2^p for the number of variables per group, while the fanout is 2. The sizes of the inputs are still too small to show the real growth, but `tCBB` $\approx 7 \cdot$ `tDD`. Finally for `strongindwrap` with $p = n$, clearly `tDD` is linear, while `tCBB` is exponential. This case shows a strong dependency on the chosen order: while KB3 in CadiBack following the given order performs very badly, using a random order also yields here ("on average") linear runtimes, but roughly by a factor of 4 slower than DualDfs.

p	n	m	tDD	PtDD	tCBB	PtCBB
fyb_rakes p p						
2500	5'000	15'621'250	0.032	0.036	10.30	10.51
5000	10'000	62'492'500	0.122	0.144	84.96	89.96
10000	20'000	249'985'000	0.485	0.605	658.69	690.54
20000	40'000	999'970'000	1.945	3.387	5204.24	5484.90
40000	80'000	3'999'940'000	7.725	15.855		
80000	160'000	15'999'880'000	30.764	38.869		
vglayers p p						
200	40'000	7'960'000	0.000	0.000	0.01	0.02
400	160'000	63'840'000	0.000	0.001	0.09	0.13
800	640'000	511'360'000	0.001	0.002	0.67	1.53
1000	1'000'000	999'000'000	0.001	0.002	1.29	2.95
1200	1'440'000	1'726'560'000	0.000	0.006		
randlayers p 2**p 2						
16	1'048'576	2'097'152	0.010	0.027	0.07	0.09
17	2'228'224	4'456'448	0.025	0.057	0.19	0.26
18	4'718'592	9'437'184	0.059	0.122	0.45	0.74
19	9'961'472	19'922'944	0.166	0.260	0.97	1.87
20	20'971'520	41'943'040	0.352	0.533	2.40	4.61
strongindwrap p						
10000	10'000	19'997	0.000	0.000	0.34	0.01
50000	50'000	99'997	0.001	0.001	8.45	0.03
200000	200'000	399'997	0.002	0.019	134.28	0.17
1000000	1'000'000	1'999'997	0.021	0.447	3353.63	0.447
10000000	10'000'000	19'999'997	0.177	4.369		16.12
20000000	20'000'000	39'999'997	0.365	10.679		34.90
30000000	30'000'000	59'999'997	0.553	9.786		52.78
40000000	40'000'000	79'999'997	0.732	11.867		69.81
100000000	100'000'000	199'999'997	1.764	49.311		190.07

4 Reducing k-cycle detection to 2-SAT backbones

We now discuss reductions from the problems of finding triangles or k-cycles in directed graphs to the problem of detecting backbone literals in instances of 2-SAT. It turns out that if there are algorithms for 2-SAT that run in linear time $O(m)$ or even sufficiently better than $O(nm)$, then we immediately obtain improvements to the best known algorithms for detecting k-cycles and the best known algorithms for Max-k-SAT.

We assume $G = (V, E)$ is a directed graph; we let $n = |V|$ be the number of vertices and $m = |E|$ be the number of edges. For instances of SAT, we continue to let n and m be the numbers of variables and clauses. The triangle detection (k-cycle detection) problem is the problem of deciding whether G contains a triangle (respectively a k-cycle).

We define a deterministic polynomial time reduction from the triangle detection problem to the problem of finding a backbone literal in a 2-CNF. Given a graph G, we create an instance of 2-SAT $\Gamma_{G,3}$. The variables of $\Gamma_{G,3}$ has three variable v^1, v^2, and v^3 for each vertex v of G. The clauses of $\Gamma_{G,3}$ are obtained by including, for each (directed) edge $e = \langle u, v\rangle$ in E, the two clauses $u^1 \to v^2$ and $u^2 \to v^3$ and the clause $u^3 \to \neg v^1$. Note that $\Gamma_{G,3}$ has $3 \cdot |V|$ many variables and $3 \cdot |E|$ many edges.

▶ **Theorem 11.** *G has a triangle if and only if $\Gamma_{G,3}$ has a failed literal. Furthermore, G has a triangle involving vertex v if and only if $\Gamma_{G,3}$ forces v false.*

The theorem is immediate from the construction. A consequence of the theorem is that an algorithm for the backbone literals of 2-SAT instance that runs in time $O(m^c)$ for any c can be converted into an $O(m^c)$-time algorithm that finds all vertices in G that are part of a triangle (a 3-cycle). Similarly an $O(m^c)$ time algorithm to find even a single failed literal in an instance of 2-SAT would give an $O(m^c)$ time algorithm for triangle detection.

The best known triangle detection algorithm [3] in m-edge graphs utilizes fast matrix multiplication. For the current value of the matrix multiplication exponent $\omega < 2.372$ [12, 23], it runs in $O(m^{1.407})$ time, and even if $\omega = 2$ (i.e. if there's a linear time matrix multiplication algorithm), the triangle detection runtime would still only be $O(m^{4/3})$, far from linear. It is in fact conjectured that $m^{4/3-o(1)}$ time is needed for triangle detection (e.g. [1, 2]); this conjecture also implies the same running time lower bound for finding a single failed literal.

The following hypothesis from Fine-Grained complexity concerns the complexity of detecting k-cycles (for constant $k \geq 3$) in directed graphs:

▶ **Hypothesis 1** (k-Cycle Hypothesis [8, 15, 4, 19]). *For every $\varepsilon > 0$ there is an integer $k \geq 3$ such that no $O(m^{2-\varepsilon})$ time algorithm can detect a k-cycle in an m-edge directed graph, in the word-RAM model with $O(\log m)$ bit words.*

The main motivation behind this hypothesis is that despite decades of research, the best algorithms for k-Cycle run in time $O(m^{2-c/k})$ for various fixed constants c, independent of k.

If $\omega > 2$ and k large enough, the algorithms of Alon, Yuster and Zwick [3] are the best for k-cycle detection; they run in time $O(m^{2-2/k})$ if k is even and in time $O(m^{2-2/(k+1)})$ if k is odd. Yuster and Zwick [25] use matrix multiplication to obtain improved algorithms and analyze them for small k. Dalirooyfard, Vuong and Vassilevska Williams [9] complete the analysis for all k for the exponent c_k for which the Yuster-Zwick algorithm detects k-cycles in $\tilde{\Theta}(m^{c_k})$ time. Under the (unproven) assumption that $\omega = 2$, the values for c_k satisfy $c_k = 2(k+1)/(k+3) > 2 - 5/k$ if k is odd. For k even, still assuming $\omega = 2$, $c_4 > 7/4$ and $c_6 > 17/11$, and for general even k, $c_k = (2k-4/k)/(k+2-4/k) > 2-5/k$. As $k \to \infty$, the values $c_k \to 2$. Further motivation for the hypothesis was provided by [19].

Our lower bound construction for failed literals can be generalized to k-cycles instead of triangles. Fix $k \geq 3$. Define the 2-SAT instance $\Gamma_{G,k}$ similarly to $\Gamma_{G,3}$ but with k vertices per vertex of G instead of 3 vertices. $\Gamma_{G,k}$ has $k \cdot |V|$ many variables and $k \cdot |E|$ many edges. Since k is constant, this is still $O(|V|)$ many variables and $O(|E|)$ many edges.

▶ **Theorem 12.** *G has a k-cycle if and only if $\Gamma_{G,k}$ has a failed literal. Furthermore, G has a k-cycle involving vertex v if and only if $\Gamma_{G,k}$ forces v false.*

Thus, any algorithm for detecting backbone literals in an instance of 2-SAT that runs in time $O(m)$ or even time $O((mn)^{1-\epsilon})$ would refute the k-Cycle Hypothesis and would be a substantial breakthrough for k-cycle detection and other problems (as shown by [19]).

Implications for Max-k-SAT. Williams [24] obtained the first algorithm for Max-2-SAT on n variables that substantially beat the $\sim 2^n$ brute-force algorithm, obtaining an $O^*(2^{\omega n/3}) \leq O(1.73^n)$ time algorithm. His algorithm was a reduction from Max-2-SAT to finding a triangle in a graph on $O^*(2^{n/3})$ vertices. A generalization of [24] shows that for every $k \geq 2$, Max-k-SAT on n variables can be reduced to the problem of finding a $(k+1)$-*hyperclique* in a $N = O(2^{n/(k+1)})$-node k-uniform hypergraph, so that an $O(N^{k+1-\varepsilon})$ time algorithm for the latter problem for any $\varepsilon > 0$ would imply a $O((2-\delta)^n)$ time algorithm for Max-k-SAT for some $\delta > 0$, resolving a big open problem in SAT algorithms.

Lincoln et al. [19] showed that the $(k+1)$-hyperclique problem in N-node k-uniform hypergraphs can be reduced to finding a $(k+1)$-cycle in a directed graph with $M = O(N^k)$ edges. Thus, a linear time algorithm for $(k+1)$-cycle give a $O(N^k)$ time algorithm for $(k+1)$-hyperclique, in turn implying an $O^*(2^{nk/(k+1)}) = O^*(2^{n(1-1/(k+1))})$ time algorithm for Max-k-SAT. Obtaining such an algorithm for *any* $k > 2$ is a major open problem.

We summarize the above reasoning in the corollary below.

▶ **Corollary 13.** *If for some $\varepsilon > 0$ and some integer $k \geq 3$, the backbone of a 2-CNF on m clauses can be computed in $O(m^{1+1/k-\varepsilon})$ time, then Max-k-SAT can be solved in $O((2-\delta)^n)$ time for some $\delta > 0$.*

Thus, if Max-3-SAT does not admit $O((2-\delta)^n)$ time algorithms for any $\delta > 0$, then computing the 2-CNF backbone cannot be done in $O(m^{4/3-\varepsilon})$ time for any $\varepsilon > 0$.

If the 2-CNF backbone can be computed in $O(m^{1+\varepsilon})$ time for all $\varepsilon > 0$, then for every $k \geq 3$ there is a $\delta > 0$ and an $O((2-\delta)^n)$ time algorithm for Max-k-SAT.

References

1. Amir Abboud, Karl Bringmann, Seri Khoury, and Or Zamir. Hardness of approximation in p via short cycle removal: cycle detection, distance oracles, and beyond. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 1487–1500. ACM, 2022.
2. Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443. IEEE Computer Society, 2014.
3. N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.
4. Bertie Ancona, Monika Henzinger, Liam Roditty, Virginia Vassilevska Williams, and Nicole Wein. Algorithms and hardness for diameter in dynamic graphs. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 13:1–13:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.
5. Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3), 1979.
6. Fahiem Bacchus. Enhancing Davis Putnam with extended binary clause learning. In *Proc. 18th Annual Conference on Artificial Intelligence and 15th Conference on Innovative Applications of Artificial Intelligence*, pages 613–619. AAAI Press and MIT Press, 2002.
7. Armin Biere, Nils Froleyks, and Wenzi Wang. CadiBack: Extracting backbones with CaDiCal. In *Proc. 26th Intl. Conf. on Theory and Applications of Satisfiability Testing SAT*, pages 3:1–12. Leibniz-Zentrum für Informatik, 2023.
8. Mina Dalirrooyfard and Virginia Vassilevska Williams. Conditionally optimal approximation algorithms for the girth of a directed graph. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 35:1–35:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
9. Mina Dalirrooyfard, Thuy-Duong Vuong, and Virginia Vassilevska Williams. Graph pattern detection: Hardness for all induced patterns and faster noninduced cycles. *SIAM Journal on Computing*, 50(5):1627–1662, 2021.
10. Alvaro del Val. On 2-SAT and renamable Horn. In *Proc. 17th Conf. on Artifical Intelligence and 21th Conf. on Innovative Applications of Aritficial Intelligence*, pages 279–284. AAAI Press and MIT Press, 2000.
11. Alvaro del Val. Simplifying binary propositional theories into concected components twice as fast. In *Logic of Programming, Artificial Intelligence, and Reasoning (LPAR)*, Lecture Notes in Artificial Intelligence 2250, pages 392–406. Springer Verlag, 2011.
12. Ran Duan, Hongxun Wu, and Renfei Zhou. Faster matrix multiplication via asymmetric hashing. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 2129–2138. IEEE, 2023.
13. A. Even, A. Atai, and A. Shamir. On the complexity of the timetable and multicommodity flow problems. *SIAM Journal on Computation*, 5(4):691–703, 1976.

14 Nils Froleyks, Emily Yu, and Armin Biere. BIG backbones. In *Proc. 23rd Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 162–167. TU Wien Academic Press, 2023.

15 Maximilian Probst Gutenberg, Virginia Vassilevska Williams, and Nicole Wein. New algorithms and hardness for incremental single-source shortest paths in directed graphs. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 153–166. ACM, 2020.

16 Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Efficient CNF simplification based on binary implication graphs. In *Theory and Applications of Satisfiability Testing (SAT)*, Lecture Notes in Computer Sceience 6695, pages 201–215. Springer Verlag, 2011.

17 Mikolás Janota, Inês Lynce, and João Marques-Silva. Algorithms for computing backbones of propositional formulae. *AI Communications*, 28(2):161–177, 2015.

18 Matti Järvisalo and Janne H. Korhonen. Conditional lower bounds for failed literals and related techniques. In *Proc. 17th Intl. Conf. Theory and Applications of Satisfiability Testing (SAT)*, Lecture Notes in Computer Science 8561, pages 75–84. Springer, 2014.

19 Andrea Lincoln, Virginia Vassilevska Williams, and R. Ryan Williams. Tight hardness for shortest cycles and paths in sparse graphs. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1236–1252. SIAM, 2018.

20 Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7:67–72, 1981.

21 Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

22 Allen Van Gelder. Towards leaner binary-clause reasoning in a satisfiability solver. *Annals of Mathematics and Artificial Intelligence*, 43(1):239–253, 2005.

23 Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proc. SODA 2024*, page to appear, 2024.

24 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005.

25 Raphael Yuster and Uri Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 254–260, 2004.

A Appendix

For space reasons, and for completeness of the exposition, we include in the appendix detailed descriptions of the simpler algorithms used as subalgorithms by DualDFS.

A.1 UnitPropagate

Input: A set Γ of clauses
Return value: The unit propagation assignment $\tau^{\mathrm{UP}}(\Gamma)$ or "Unsatisfiable"

Function `UnitPropagate(`Γ`)`
  $\tau = \emptyset$; // The empty truth assignment
  while Γ *contains a unit clause* $\{\ell\}$ **do**
    Extend τ so that $\tau(\ell) = \top$
    $\Gamma := \Gamma_{\restriction\tau}$
    if $\bot \in \Gamma$ **then** **return** "Unsatisfiable";
  return τ

`UnitPropagate()` is written using nondeterministic choices for literals ℓ to unit propagate; however, the end result of the algorithm is independent of the order in which the literals ℓ are propagated. The unit propagation algorithm is also very efficient: in a suitable random access model (RAM) of computation, `UnitPropagate()` can have runtime linearly bounded in terms of $|\Gamma|$, where $|\Gamma|$ is the number of occurrences of literals in clauses in Γ. Clearly, if $\tau =$ `UnitPropagate`(Γ), then any truth assignment satisfying Γ extends τ.

A.2 ForceImmediate

Input: A literal ℓ
Effect: ℓ is set true and unit propagation is carried out
Return value: *true* if no contradiction is found, otherwise *false*.

Function `ForceImmediate(`ℓ`)`
  Push ℓ onto the UP stack as its only member
  while *the UP stack is nonempty* **do**
    Pop p from the UP stack
    if $\mathsf{var}(p)$ *has not been assigned a truth value* **then**
      Set p true Mark p and $\overline{p}$ as handled
      foreach *child* t *of* p **do** push t onto the UP stack
    else if p *is assigned the value false* **then**
      return false // Conflict!
  return true

A.3 InitializeS

`InitializeS` initializes "half" of the set S. It starts with a literal t_0 and produces a set S equal to $\{s_1, \ldots, s_k\}$ so that $s_k = t_0$ and $s_i \to s_{i+1}$ for each i. `InitializeS()` acts by starting at t_0, and greedily choosing each t_{i+1} as the first child of t_i which is not a generalized sink. The other "half" of S, that is, the part above s_k, will be dynamically generated during the first phase of DualDFS. It is possible that `InitializeS()` finds a contradiction in Γ. It is also possible that `InitializeS()` discovers that t_i is in the backbone, and so forced true or false. In this case, there may be no set S created.

Input: A literal t_0 that is not a generalized sink.
Return value: A set $S = \{s_1, \ldots, s_k\}$ with $s_k = t_0$, or "abort" if S is a subset of the backbone, or "false" if Γ is unsatisfiable.

1 **Function** `InitializeS`(t_0)
2     $k := 0$
3   **while** *t_k has a child t_{k+1} that is not handled, not a generalized sink, and not equal to any t_i for $i < k$* **do**
4   Set t_{k+1} to be such a child.
5   **if** *t_{k+1} equals $\bar{t}_i$ for some $i \leq k$* **then**
6   **if** *not* `ForceImmediate` *(t_{k+1})* **then** **return** false
7   **while** *t_k is set true* **do**
8   **if** $k = 0$ **then abort**
9                 $k := k - 1$ `// Discard` t_k
10   **if** *t_k is set false* **then abort**
11   **else** $k := k + 1$
12   **return** $t_0, \ldots, t_{k-1}$ as $s_k, \ldots, s_1$. (Discard t_k.)

If `InitializeS()` aborts then every t_i has been forced true or false. The test in line 5 is carried out in constant time by maintaining for each variable x a flag indicating whether x or $\overline{x}$ is in the set S. The pseudocode does not show it explicitly, but this flag is updated whenever a literal is added to or removed from S. In line 3, a potential t_{k+1} might be equal to an already chosen t_i. This can happen only if there are non-trivial SCC-equivalences, but the test in line 3 prevents the same literal being added again to S. Another possibility is that t_{k+1} is equal to $\overline{t_i}$; in this case, $\overline{t_i}$ is discovered to be a failed literal and thus t_{k+1} is set true. It is possible that some t_j's are SCC-equivalent to t_{k+1} and are also set false. The while loop starting on line 7 removes these from S. Then, if the remaining t_k's are all false, it aborts.