Generalizing CDCL with Graph Backtracking

Robin Coutelier
TU Wien, Vienna, Austria

Thomas Hader
TU Wien, Vienna, Austria

Laura Kovács
TU Wien, Vienna, Austria

Abstract

We present *graph backtracking*, a novel, fine-grained backtracking scheme for CDCL-based SAT solving, parametrized by a user-defined weight function. For conflict repair, we challenge the decision level abstraction and use the implication graph as a precise guiding structure to minimize the weight of literals that are unassigned. Graph backtracking is sound, complete, and terminating. We show that it is a generalization of chronological and non-chronological backtracking by simulating them with specific weight functions. Our approach is implemented in the experimental solver NapSAT. Empirical results show that graph backtracking requires fewer literal propagations than standard approaches, leading to improved solver runtime.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Automated reasoning

Keywords and phrases SAT Solving, Backtracking, Conflict Analysis, CDCL

Digital Object Identifier 10.4230/LIPIcs.SAT.2026.1

Funding The authors acknowledge support from the ERC Consolidator Grant ARTIST 101002685; the TU Wien Doctoral Colleges TrustACPS and SecInt; the FWF SpyCoDe SFB projects F8504; and the WWTF Grant ForSmart 10.47379/ICT22007.

Acknowledgements We would like to thank Mathias Fleury, Michael Rawson, Clemens Eisenhofer, Márton Hajdu, and Pascal Fontaine for fruitful discussions. We also thank the anonymous reviewers for their feedback.

1 Introduction

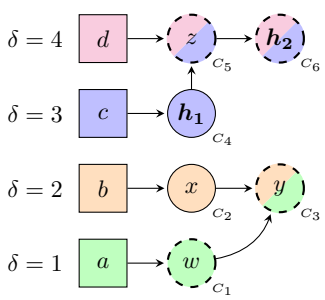
The CDCL algorithm [23] is the dominant approach in propositional satisfiability (SAT) solving [6]. CDCL solvers typically employ a rather aggressive backtracking scheme for conflict repair, referred to as non-chronological backtracking (NCB). More recently, chronological backtracking (CB) has been shown to complement NCB in SAT solving [20, 17], with dedicated CB-based algorithms improving the state of the art in model counting [17, 24] and AllSAT [27], with further applications to MaxSAT [19]. However, existing backtracking schemes maintain a strict top-down order: undoing the most recent decisions and their consequences first. This approach provides strong invariants and enables efficient implementations, but restricts the locality of the search and may lead to unnecessary backtracking. Orthogonal to backtracking, user propagators [7, 14] give users more control over the behavior of the SAT solver by allowing them to extract partial assignments, raise application-specific conflicts, assign new literals, and provide new clauses during proof search.

Inspired by developments in user propagators and backtracking strategies, we introduce *graph backtracking* (GB), a more general backtracking scheme for CDCL-based SAT solving. GB enables a more flexible and fine-grained mechanism for repairing conflicts. It determines precisely which literals can be unassigned and performs minimal backtracking. Furthermore, instead of always undoing the latest assigned variables, a user can supply a weight function,

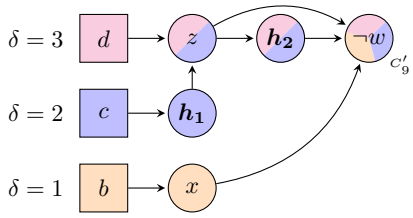
allowing more control over backtracking. In particular, the user may prefer some literals to remain assigned longer. Associating them with a higher weight discourages the solver from unassigning them, thus achieving the desired behavior.

Motivating example. We motivate graph backtracking via a SAT-based search. Let h_1 and h_2 denote comparatively heavy literals, and the following clause set F:

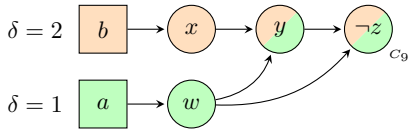
$$C_1 = w \vee \neg a \qquad C_3 = y \vee \neg w \vee \neg x \qquad C_5 = z \vee \neg d \vee \neg h_1 \qquad C_7 = a \vee \neg c \vee \neg d$$
$$C_2 = x \vee \neg b \qquad C_4 = h_1 \vee \neg c \qquad C_6 = h_2 \vee \neg z \qquad C_8 = \neg w \vee \neg y \vee \neg z \vee \neg h_2$$



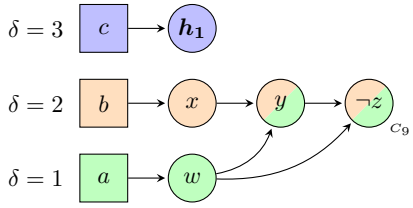
(a) Conflict detected in clause C_8.



(b) Graph backtracking (GB) [this work].



(c) Non-chronological backtracking (NCB) [18].



(d) Chronological backtracking (CB) [20].

Figure 1 Implication graphs of F for different backtracking strategies. Decisions are represented as squares and applied in lexicographic order. When relevant, implied literals are annotated with their justifications. Colors indicate dependencies on decision literals. Decision levels are annotated with δ. The literals of the conflicting clause C_8 are highlighted with thicker dashed borders.

Figure 1a displays a possible implication graph resulting from F. An implication graph denotes logical implication relations between literals (for preliminaries, see Section 2). A conflict is detected in clause C_8 during the propagation of z. To repair it, the traditional conflict analysis [23] algorithm learns the clause $C_9 = \neg w \vee \neg y \vee \neg z$ and non-chronological backtracking [18] backjumps to the decision level of b (Figure 1c). NCB notably unassigns the heavy literals h_1 and h_2 against the user's preference. Chronological backtracking [20] learns the same clause C_9 but realizes that the decision c does not need to be undone. Instead, it would only undo d, z and h_2 (Figure 1d). This is an improvement, as h_1 remains assigned.

Our graph backtracking algorithm allows unassigning *any*[1] of the decisions involved in the conflict and their subsequent implications, i.e., any of the colors in Figure 1a. Considering the user-provided weights, it unassigns the green literals a, w and y, preserving both heavy literals h_1 and h_2, as illustrated in Figure 1b. Our modified analysis (see Section 4.2) learns[2] the clause $C'_9 = \neg w \vee \neg x \vee \neg z \vee \neg h_2$ and GB now implies $\neg w$ because of C'_9 (Figure 1b).

[1] Some safeguards are needed to ensure termination (Section 4.1).
[2] Shrinking could also be used to shorten the clause to $C'_9 = \neg w \vee \neg x \vee \neg z$. We omit this optimization.

Contributions. Our main contributions are as follows:

- We extend the interface of SAT solvers with a weight function for literals (Section 4.1).
- We present *graph backtracking*, a precise and user-guided backtracking algorithm (Algorithm 4).
- We show that graph backtracking is sound and terminating (Theorem 9).
- We show that graph backtracking is a generalization of (N)CB (Theorems 10 and 11).
- We suggest several practical optimizations of graph backtracking (Section 6).
- We implement graph backtracking and evaluate its performance empirically (Section 7).

2 Preliminaries

We assume familiarity with propositional logic and CDCL [6] and use the standard logical connectives $\neg$, $\wedge$, $\vee$, and $\Rightarrow$. We denote a finite set of propositional variables by $\mathcal{V}$. A *literal* is either a variable or its negation; that is, a literal is v or $\neg v$, for some $v \in \mathcal{V}$. The set of literals is denoted by $\mathcal{L} = \{v, \neg v : v \in \mathcal{V}\}$. Note that $\neg\neg\ell = \ell$. A *clause* is a (disjunctive) set of literals $C = \{c_1, c_2, \ldots, c_n\}$ where c_i is a literal. We denote the empty clause as $\square$ and the undefined clause $\blacksquare$ is used for the absence of a clause in partial functions. Without loss of generality, we only consider a propositional formula F in *conjunctive normal form (CNF)* as a (conjunctive) set of clauses $\{C_1, C_2, \ldots, C_m\}$ over $\mathcal{V}$.

An *ordered set* is a set with a total positional order on its elements. Ordered sets support the concatenation operator $\cdot$ and are stable under set operations, i.e., set intersection and difference on ordered sets maintain the order of the left operand.

A (partial) assignment π is an ordered set of literals. A literal and its negation cannot appear simultaneously in π. In an assignment π, a literal is *satisfied* if $\ell \in \pi$; *falsified* if $\neg\ell \in \pi$; and *unassigned* otherwise. We say that a clause is *satisfied* if $C \cap \pi \neq \emptyset$; *falsified* if $C \setminus \{\neg\ell : \ell \in \pi\} = \square$; and *unit* in literal ℓ if $C \setminus \{\neg\ell' : \ell' \in \pi\} = \{\ell\}$.

A literal ℓ is *implied* by a clause C under a partial assignment π if C is unit in ℓ under π and ℓ is satisfied by π. The *justification* of ℓ, also known as the *reason* of ℓ, is the unit clause $\rho(\ell) = C$ that explains the implication of ℓ on π. A *missed implication* is a clause C that implies a literal ℓ under π but is not the justification of ℓ. A *missed lower implication* is a missed implication C of ℓ such that ℓ is implied by C at a lower decision level than $\rho(\ell)$. A *decision* literal ℓ does not have a justification, hence $\rho(\ell) = \blacksquare$.

The *decision level* $\delta(\ell)$ of a literal ℓ is defined as follows: if ℓ is a decision, then its decision level is the number of decisions in π preceding ℓ, plus one; otherwise, its decision level is the maximum level of all literals in $\rho(\ell) \setminus \{\ell\}$. When relevant, we write $\ell@n$ to denote that a literal ℓ is on decision level n. The decision level of a set of literals S is $\delta(S) = \max_{\ell \in S} \delta(\ell)$. Furthermore, we define a *literal weight function* $\zeta : \mathcal{L} \to \mathbb{R}$ that maps a literal $\ell \in \mathcal{L}$ to a real number. We also extend ζ to sets of literals by defining $\zeta(S) = \sum_{\ell \in S} \zeta(\ell)$.

Given a partial assignment π, we partition $\pi = \tau \cdot \omega$ into the set of propagated literals τ and the queue of literals to propagate ω. We say that a literal ℓ is propagated if $\ell \in \tau$ or $\neg\ell \in \tau$. We note that some related work calls *propagated* what we call *implied*; this distinction becomes relevant in Section 4.3.

To visualize the implications of literals on a partial assignment π, we use *implication graphs*, as shown in Figure 1. An implication graph is a directed acyclic graph with a node for each literal in π and edges indicating the justification of each literal. A literal ℓ in the graph is the implied consequence of all literals with an edge to ℓ using clause $\rho(\ell)$. Decision literals do not have any incoming edges. A literal ℓ *depends on literal* ℓ' if there is a path from ℓ' to ℓ in the implication graph.

3 Related Work

Non-Chronological Backtracking (NCB) has been the standard backtracking approach in most state-of-the-art SAT solvers [6, 13] since watched literals became popular [18]. When a NCB algorithm finds a conflict, it performs conflict analysis to learn a new clause preventing the same conflict from happening again. The standard algorithm searches a unique implication point (UIP), i.e., a clause with one literal at the highest decision level. The UIP results from binary resolutions of the conflicting clause and the justifications leading to it. Usually, we stop at the first UIP encountered (1UIP).

NCB is characterized by backjumping. To repair the conflict, NCB will backtrack to the second-highest level in the learned clause. In the example Figure 1a, the clause $C_9 = \neg w@1 \vee \neg y@2 \vee \neg z@4$ has the highest decision level 4 and second-highest decision level 2. The algorithm then backjumps to decision level 2 (Figure 1c).

Chronological Backtracking (CB) was reintroduced more recently [20, 17, 19, 11]. It won the SAT2018 competition with MAPLE_LCM and is now part of CADICAL [4]. It was shown that CB also improves performance in applications such as model counting [24] and enumeration [27] due to a more systematic exploration of the search space, but also MaxSAT [19] due to its higher flexibility in incremental solving.

Less restrictive than NCB, CB only requires backtracking to the highest decision level in the learned clause, minus one. In the example Figure 1a, the clause C_9 has the highest decision level 4. The algorithm then backtracks to decision level 3 (Figure 1d).

More precisely, CB may backtrack anywhere between the second highest and the highest decision level minus one, and heuristics can be used to select the appropriate level [20, 17]. In the following, we will only consider the variant of CB that backtracks to the highest decision level minus one, as it is simpler to differentiate from NCB.

Beyond SAT Solving. Some attempts have been made to improve the backtracking processes in CSP, notably [15]. However, these approaches did not consider the inner workings of propagation algorithms and did not learn clauses. Additionally, CADICAL introduced a *facade* for better integration in SMT solver with CB [4]. CADICAL shows a partial assignment to the SMT solver that does not reflect the state of the SAT solver and pretends to use a stack. This facade is orthogonal to our approach, and could be used in conjunction with graph backtracking. Some work has been done to add implications graphs with redundancy in the SMT congruence closure algorithm [1]. GB might also have applications in local search [10] due to its minimalistic approach to backtracking.

4 Graph Backtracking

We now present our *graph backtracking (GB)* algorithm. We highlight the key features of CDCL-based SAT solving using our GB approach, in particular when compared to the existing strategies of chronological backtracking (CB) [20] and non-chronological backtracking (NCB) [23]. To ease readability, the differences between GB and (N)CB are highlighted in blue in the respective algorithms.

GB in a Nutshell. The main idea of GB is to use implication graphs to determine the set of literals that can be unassigned upon conflict detection. We cluster literals into so-called *chunks*: groups of literals that can be undone together. We can repair the assignment by undoing any chunk involved in the conflict (Section 4.1). With this strategy, we obtain a CDCL-based SAT solving approach using GB, summarized in Algorithm 4, where treatment of propositional variables in case of conflict analysis is more surgical and controlled.

In (N)CB, literals are clustered according to their decision levels. This clustering forms a coarse over-approximation of the actual dependencies between literals. The decision levels create the simplifying assumption that literals assigned at a level d depend on all literals below d. Hence, when backtracking to level d, all literals above d need to be unassigned. Our GB algorithm refines this clustering by using *chunks*. A chunk ck_ℓ is a set of literals depending on the decision literal ℓ. That is, the set of literals reachable from a decision ℓ in the implication graph. Naturally, an implied literal can depend on multiple decisions and belong to several chunks. We write $\gamma(\ell)$ as the set of chunks containing a literal ℓ. For simplicity, we assume that $\gamma(\ell) = \gamma(\neg\ell)$. Our notation extends to sets of literals (such as clauses) S as $\gamma(S) = \bigcup_{\ell \in S} \gamma(\ell)$. A decision literal d only belongs to its associated chunk $\gamma(d) = \{ck_d\}$. A literal ℓ justified by a clause $C = \rho(\ell)$ belongs to the chunks $\gamma(\ell) = \gamma(C \setminus \{\ell\})$.

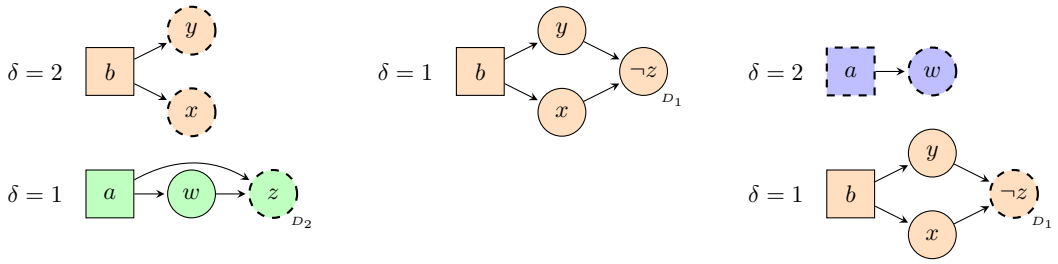
▶ **Example 1.** Consider Figure 1a. The implication graph contains four chunks, each highlighted in a distinct color, $ck_a = \{a, w, y\}, ck_b = \{b, x, y\}, ck_c = \{c, z, h_1, h_2\}$, and $ck_d = \{d, z, h_2\}$. The conflicting clause $C_8 = \neg w \vee \neg y \vee \neg z \vee \neg h_2$ belongs to the four chunks $\gamma(C_8) = \{ck_a, ck_b, ck_c, ck_d\}$. GB can therefore choose to backtrack ck_a since it is the lightest, as it does not contain any heavy literals h_i. (Section 4.1 explains why ck_b is not selected.)

When a conflict clause C is discovered, in GB we may undo any of the chunks in $\gamma(C)$ to resolve the conflict. Section 4.1 describes how this choice is made. The undone chunks influence the conflict analysis procedure of GB, as explained in Section 4.2. To maintain clean invariants, watched literals require special care in GB, as detailed in Section 4.3.

4.1 Repair Chunk Selection

Upon discovering a conflict C, a CDCL solver needs to undo parts of the assignment, such that a newly learned clause C', derived from the conflict, becomes unit and implies a flipped literal. To this end, GB may undo any set of chunks[3] Γ_i that contains exactly one[4] chunk from $\gamma(C)$, i.e., $|\Gamma_i \cap \gamma(C)| = 1$. GB then selects the lightest chunk set Γ^* among all valid candidates Γ_i. Conflict analysis then discovers C' containing exactly one literal ℓ belonging to Γ^*. After backtracking Γ^*, ℓ is implied by C'.

Among all backtrack candidates, the selection of Γ^* is guided by the user-provided cost function ζ. A careless choice of backtracked chunks might not lead to termination, as illustrated next.



(a) Conf. $D_1 = \neg x \vee \neg y \vee \neg z$ **(b)** Undo chunk ck_a. **(c)** Conf. $D_2 = \neg a \vee \neg w \vee z$.

Figure 2 Potential loop in conflict discovery with arbitrary backtracking.

[3] Here we talk about chunk sets and not individual chunks for generality, relevant in Sections 5 and 6.
[4] Undoing more chunks may make it impossible to learn a unit clause.

▶ **Example 2.** Consider Figure 2. The implication graph contains two decision variables a and b, with their respective chunks ck_a and ck_b. On Figure 2a, BCP (Section 4.3) discovers a conflict clause $D_1 = \neg x \vee \neg y \vee \neg z$, which has a single literal in the chunk ck_a. The chunk ck_a can then be undone as shown in Figure 2b to fix the conflict. The literal $\neg z$ is added to ck_b. CDCL might decide a again, and discovers another conflict $D_2 = \neg a \vee \neg w \vee z$ (Figure 2c). If now ck_b is undone, and the decision b is taken, we are back to the same state as Figure 2a.

As further elaborated in Section 4.2, the choice of chunk for backtracking influences the clause that is learned after conflict analysis. We write $1\text{UIP}_{\Gamma_i}(C)$ to denote the first UIP found when unassigning chunks Γ_i for the conflict clause C. The *chunk level* of a chunk ck_ℓ, written $\delta_\gamma(ck_\ell)$, is the decision level $\delta(\ell)$ of its decision literal ℓ. We extend the notation to sets of chunks Γ_i as $\delta_\gamma(\Gamma_i) = \max_{ck \in \Gamma_i} \delta_\gamma(ck)$.

Safeguard. The loop in Example 2 arises because neither conflict leads to learning a new clause. In Figure 2a, D_1 is already a UIP with respect to ck_a, i.e., $1\text{UIP}_{ck_a}(D_1) = D_1$. Later, in Figure 2c, D_2 is also a UIP with respect to ck_b.

Redundant clauses can also be learned in CB. However, progress is ensured by the ordering of decision levels. Indeed, in CB, a literal flipped due to a conflict will remain on the trail at least until another flipped literal is added at a lower level. It is then impossible to encounter infinitely many conflicts before reaching decision level 0, and progress is guaranteed.

In GB, this ordering is weakened as follows: candidate chunks either lead to learning a new clause, or contain the most recent chunk in the conflict.

▶ **Definition 3** (Terminating backtrack candidates). *Let C be a conflicting clause in the clause set F. Let $\gamma(\pi)$ be the set of all chunks in the current partial assignment π, and $2^{\gamma(\pi)}$ be the powerset of $\gamma(\pi)$. The set Γ of* terminating backtrack candidates *is defined as:*

$$\Gamma = \left\{ \Gamma_i \in 2^{\gamma(\pi)} : |\Gamma_i \cap \gamma(C)| = 1 \wedge \left(1\text{UIP}_{\Gamma_i}(C) \notin F \vee \delta_\gamma\big(\Gamma_i \cap \gamma(C)\big) = \delta(C) \right) \right\}$$

Selecting Γ^* from the terminating backtrack candidates guarantees the termination of GB while allowing the preservation of heavy literals. The first condition $1\text{UIP}_{\Gamma_i}(C) \notin F$ ensures that the learned clause is new. Alternatively, if the second condition $\delta_\gamma(\Gamma_i \cap \gamma(C)) = \delta(C)$ holds, the chunks are ordered and an identical argument to CB applies.

▶ Remark 4. It is not always trivial to check the redundancy of the learned clause. Let $C_1 \in F$ be the conflict detected by BCP and $C_2 \in F$ be another conflict. If $1\text{UIP}_{\Gamma_i}(C_1) = C_2$, then the first condition fails. A simple way to detect such a situation is to go through the watch lists of the literals in the learned clause and check whether it is subsumed in the formula. However, this case is very rare in practice and can be ignored in an efficient implementation.

In Example 2, backtracking ck_a would not learn a new clause and ck_b is more recent in the conflict $\delta_\gamma(ck_a) < \delta(C)$. Therefore ck_a is not a terminating backtrack candidate.

In Example 1, our safeguard explains why we did not select ck_b for backtracking. C_8 is already a UIP with respect to ck_b. Backtracking ck_b does not learn a new clause and ck_b is not the most recent chunk in the conflict.

Chunk Selection. Selecting the chunk set Γ^* becomes a minimization problem over the set of terminating backtrack candidates Γ. That is:

$$\Gamma^* = \textsc{SelectChunk}(C, \zeta) = \operatorname*{arg\,min}_{\Gamma_i \in \Gamma} \zeta \left(\bigcup_{ck \in \Gamma_i} ck \right) \tag{1}$$

▶ Remark 5. Note that, if the cost function ζ only returns positive values, SelectChunk only selects backtrack candidates with a single chunk. However, in some cases, it might be beneficial to backtrack multiple chunks together (see Sections 5 and 6.3).

Algorithm 1 Conflict Analysis.

```
1: procedure ANALYZE(C, Γ*)
2:     D ← C                                   ▷ Current learned clause.
3:     n ← |D ∩ ⋃_{ck∈Γ*} ck|                  ▷ Number of literals in D that will be undone
4:     for ℓ ∈ π do                            ▷ Iterate right to left
5:         if ℓ ∈ D ∧ Γ* ∩ γ(ℓ) ≠ ∅ then
6:             D ← D ⊗_ℓ ρ(ℓ)                  ▷ Binary Resolution
7:             n ← |D ∩ ⋃_{ck∈Γ*} ck|
8:         if n = 1 then                       ▷ Found a UIP
9:             return D
```

Algorithm 2 Backtracking.

```
1: procedure UNDO(Γ*)
2:     for ℓ ∈ π do                            ▷ Iterate left to right
3:         if γ(ℓ) ∩ Γ* ≠ ∅ then
4:             π ← π \ {ℓ}                     ▷ Unassign ℓ
5:             γ(ℓ) ← ∅, ρ(ℓ) ← ■, δ(ℓ) ← ∞
6:             continue
7:         δ(ℓ) ← UPDATELEVEL(ℓ)               ▷ Recalculate the decision level of ℓ
8:         if η(ℓ) ∩ Γ* ≠ ∅ then               ▷ see Section 4.3
9:             τ ← τ \ {ℓ}                     ▷ ℓ needs to be repropagated
10:    τ ← π ∩ τ                               ▷ Remove the unassigned literals
11:    ω ← π \ τ
```

4.2 Conflict Analysis & Backtracking

Our conflict analysis procedure within GB is summarized in Algorithm 1, showcasing a variant of the standard 1UIP scheme [23] adapted to chunks. Binary resolution between the clauses C and D over a literal ℓ is denoted by $C \otimes_\ell D$, which is defined as $(C \setminus \{\neg\ell\}) \cup (D \setminus \{\ell\})$.

▶ **Example 6.** In Figure 1a, the call ANALYZE$(C_8, \{ck_a\})$ in Algorithm 1 handles y first, since it is the last literal on the trail belonging to ck_a. It will perform resolution on $\rho(y) = C_3$, giving the clause $C_9' = C_8 \otimes_y C_3 = \neg w \vee \neg x \vee \neg z \vee \neg h_2$. The procedure ends with the learned clause C_9' since w is the last literal belonging to ck_a.

As shown in Example 6, chunk-based conflict analysis in GB adapts to any chunk and produces an implying learned clause. The backtracking procedure of GB is presented in Algorithm 2. We can now retain literals whose decision levels are higher than the backtracked chunk. In Figure 1b, we keep b, c, d, y, z, x, h_1 and h_2 in the trail, whose decision levels are higher than a. This is why at line 7 of Algorithm 2, we need to recalculate the decision level of each literal after the lowest undone decision. Line 9 of Algorithm 2 ensures the correctness of the two-watched-literal scheme in GB (Theorem 9), as explained next.

4.3 Unit Propagation & Watch Literals

One of the reasons for the success of NCB in CDCL-based SAT solvers is the use of fast unit propagation algorithms. The most common technique to achieve Boolean Constraint Propagation (BCP) is the two-watched-literal scheme [18]. Each clause is said to be watched by two of its literals – called *watched literals* or *watchers* – such that the solver only needs to

look at clauses watched by the negation of the currently being propagated literal to ensure the discovery of every conflict and unit clause. We write $\mathrm{WL}(\ell)$ to denote the watch list of a literal ℓ, that is the list of clauses watched by ℓ. The following invariant is the guiding principle for existing backtracking solutions:

▶ **Invariant 1** (Watched literals [11]). *Consider the trail $\pi = \tau \cdot \omega$. For each clause $C \in F$ watched by c_1, c_2, we have $\neg c_1 \in \tau \Rightarrow c_2 \in \pi$.*

It states that for each clause C, if one of its watchers c_1 is falsified and propagated, then the other watcher c_2 must be satisfied. During propagation, BCP checks whether a literal ℓ can be added to the propagated set, i.e., whether it is possible to enforce $\neg c_1 \in (\tau \cdot \ell) \Rightarrow c_2 \in \pi$. If this is not possible, then a conflict is raised. Note that conflicts do not violate the invariant, since they are detected while the watched literals are still in the queue ω. Invariant 1 is not trivially preserved in GB upon backtracking. We, thus, revise it into Invariant 2 in order to show soundness of GB in Theorem 9.

Limitations of Watched Literals in GB. Invariant 1 is naturally maintained by NCB, since the partial assignment behaves exactly like a stack. If during the propagation of a literal ℓ a clause is detected to be satisfied, it will remain so at least until ℓ is unassigned. That is, the right side $c_2 \in \pi$ holds at least as long as the left side $\neg c_1 \in \tau$ holds. Therefore, backtracking does not violate the invariant.

This property does not apply to GB by construction. In Figure 1, we propagated x before y, yet y was unassigned first. Therefore, backtracking does not preserve Invariant 1. The clause $C_3 = y \vee \neg x \vee \neg w$ would satisfy Invariant 1 if $c_1 = \neg x$ and $c_2 = y$, but no longer after backtracking since $y \notin \pi$. To restore it, x needs to be removed from the propagated set τ.

A surprisingly effective, yet simple, solution used in CB is to repropagate all literals in the trail located after the first literal of the backtracked level, called the restoration point [20, 11]. In CB, repropagation is cheap because watch lists are already cleaned up, and the restoration point is usually relatively close to the end of the trail. However, in GB, this point is not trivial to find, and may be located early, leading to a lot of unnecessary propagations.

Another interesting aspect of GB is that there is no total ordering between literals. In (N)CB, it is possible to choose literals with the highest decision level to ensure that the watchers of a clause will always be unassigned before the other literals in the clause. In GB, this is no longer the case. Consider again C_3 in the example of Figure 1a, we would like to select the watchers such that they will be unassigned after backtracking. It is clear that y should be one of the watched literals, since it belongs to both ck_a and ck_b. However, we can either choose $\neg w$ or $\neg x$ as the second watcher. Let us select $\neg w$. In this case, after backtracking ck_b, the clause violates Invariant 1, since $\neg w \in \tau$ but $y \notin \pi$. If we had selected $\neg x$ instead, backtracking ck_a would also violate the invariant. We cannot find a perfect watcher since $\gamma(w) \not\subseteq \gamma(x)$ and $\gamma(x) \not\subseteq \gamma(w)$.

Cross-Chunks of Watched Literals in GB. To solve the aforementioned issues on watch literals in GB, we introduce the concept of *cross-chunks*, written as $\eta(\ell)$ and used in Algorithm 2. The cross-chunks of a literal ℓ is the set of chunks that, when backtracked, require ℓ to be repropagated. In other words, if $ck \in \eta(\ell)$ and ck is undone, then ℓ needs to be repropagated; as ensured in line 9 of Algorithm 2: The literal ℓ is removed from the propagated set τ and put in the queue ω to be repropagated. Cross-chunks allow skipping most of the repropagations. The backtracking procedure of Algorithm 2 preserves the following stronger invariant:

Algorithm 3 Boolean Constraint Propagation.

1: **procedure** BCP( )
2: **while** $\omega \neq \emptyset$ **do**
3: $\ell \leftarrow \omega[0]$, $c_1 \leftarrow \neg\ell$
4: $\eta(\ell) \leftarrow \gamma(\ell)$ ▷ Reset and enforce $\gamma(\ell) \subseteq \eta(\ell)$
5: **for** $C \in \mathrm{WL}(c_1)$ **do**
6: $c_2 \leftarrow$ the other watched literal in C
7: **if** $c_2 \in \pi \wedge \gamma(c_2) \subseteq \eta(c_1)$ **then**
8: **continue** ▷ C already satisfies Invariant 2
9: $r \leftarrow \textsc{SearchReplacement}(C, c_1, c_2)$
10: $\mathrm{WL}(c_1) \leftarrow \mathrm{WL}(c_1) \setminus \{C\}$ ▷ Evict C from the watch list of c_1
11: $\mathrm{WL}(r) \leftarrow \mathrm{WL}(r) \cup \{C\}$ ▷ Add C to the watch list of r
12: **if** $\neg r \notin \pi$ **then** ▷ Good replacement found
13: **continue**
14: **if** $\neg c_2 \in \pi$ **then** ▷ Conflict
15: **return** C
16: **if** $c_2 \in \pi$ **then** ▷ Missed implication
17: $\eta(r) \leftarrow \eta(r) \cup \gamma(c_2)$
18: **continue**
19: $\omega \leftarrow \omega \cdot c_2, \rho(c_2) \leftarrow C, \delta(c_2) \leftarrow \delta(C \setminus \{c_2\})$
20: $\gamma(c_2) \leftarrow \gamma(C \setminus \{c_2\})$, $\eta(r) \leftarrow \eta(r) \cup \gamma(c_2)$
21: $\omega \leftarrow \omega \setminus \{\ell\}, \tau \leftarrow \tau \cdot \ell$
22: **return** ■

▶ **Invariant 2** (GB watched literals). *Consider the trail $\pi = \tau \cdot \omega$. For each clause $C \in F$ watched by c_1, c_2, we have $\neg c_1 \in \tau \Rightarrow \big[c_2 \in \pi \wedge \gamma(c_2) \subseteq \big(\gamma(c_1) \cup \eta(c_1)\big)\big]$.*

The revised invariant states that if a watched literal c_1 is falsified and propagated, then the other watcher c_2 must be satisfied and c_2 will hold as long as $\neg c_1$ is in the propagated set. That is, if c_2 is unassigned during backtracking, then c_1 will either be unassigned as well, or repropagated. During BCP, we update the cross-chunks of the watched literals to keep a record of literals that need to be repropagated after backtracking.

Boolean Constraint Propagation. With the adjustments to cross-chunks of watched literals and Invariant 2, we adapt BCP to GB, as shown in Algorithm 3. Compared to standard two-watched-literal BCP algorithms, the main difference of Algorithm 3 comes with the handling of chunks, cross-chunks, and skip conditions. We note that, in practice, cross-chunks are always used in union with the chunks of a literal. Therefore, we improve efficiency by enforcing that $\gamma(\ell) \subseteq \eta(\ell)$ (Line 4). This way, when checking whether a clause can be skipped during BCP, only one set comparison is required. In the example of Figure 1a, let us set the watchers of the clause C_3 to be y and $\neg x$. When propagating x, we will imply y with $\gamma(y) \leftarrow \{ck_a, ck_b\}$ and update $\eta(x) \leftarrow \{ck_a, ck_b\}$ to satisfy Invariant 2. Upon undoing ck_a, x will be repropagated since $ck_a \in \eta(x)$.

4.4 CDCL-Based SAT Solving Using GB

Our CDCL-based SAT solving approach using GB is summarized in Algorithm 4, exploiting cost-based trail repair, conflict analysis, backtracking, and BCP, as presented in Sections 4.1 to 4.3. For conciseness, we only provide proof sketches here and refer to [12] to more details.

▶ **Lemma 7** (Graph backtracking invariant). *The graph backtracking CDCL-based approach of Algorithm 4 preserves Invariant 2.*

Proof. Invariant 2 is maintained by BCP, since BCP only adds a literal ℓ to the propagated set τ if all clauses satisfy $\neg c_1 \in (\tau \cdot \ell) \Rightarrow \big[c_2 \in \pi \wedge \gamma(c_2) \subseteq \big(\gamma(c_1) \cup \eta(c_1)\big)\big]$. Only the clauses watched by $\neg\ell$ are checked, as they are the only ones for which $\neg c_1 \in (\tau \cdot \ell)$ can change from false to true.

Invariant 2 is also maintained by backtracking, since it ensures $\neg c_1$ is unassigned or removed from τ before c_2 is unassigned.

The other operations of Algorithm 4 do not modify the watch lists nor the propagated set, and therefore do not violate Invariant 2. ◀

▶ **Lemma 8** (Graph backtracking trail). *In Algorithm 4, no clause is falsified by the propagated set τ, or unit under τ and not satisfied by π.*

Proof. We prove by contradiction. Let a clause C be falsified by τ or be unit under τ and not satisfied by π. Then one of its watched literals c_1 is falsified and propagated, and the other watched literal c_2 is not satisfied. This contradicts Invariant 2 and Lemma 7. ◀

▶ **Theorem 9** (Graph backtracking). *Algorithm 4 is sound, and terminating.*

Proof. Soundness follows directly from Lemma 8: a solution is only returned when the propagated set $\pi = \tau$ and all variables are assigned. From Lemma 8, no clause is falsified by π, and therefore the solution is correct. Learned clauses are entailed by the original formula with binary resolution. Therefore, if the formula finds a conflict without any decision, then the empty clause is entailed by the original formula, and the formula is unsatisfiable.

The terminating backtrack candidates of Section 4.1 guarantee termination. Upon conflict, either a new clause is learned, or the most recent chunk in the conflict is undone. In the former case, there are finitely many clauses entailed by the original formula, and therefore learning a new clause is progress. In the latter case, progress must be made because of the ordering of chunks similarly to CB (Section 4.1). ◀

5 Simulating (N)CB with GB

In this section, we show that graph backtracking is a generalization of both NCB and CB. Importantly, we present how to craft an appropriate cost function ζ to simulate NCB and CB with GB. As such, GB can be used as a replacement for (N)CB, while retaining the behavior and guarantees offered by (N)CB.

Let us define a non-deterministic transition system over the state $\langle \pi, \delta, \rho, \phi \rangle$, where ϕ is the set of clauses and π, δ, ρ are as defined in Section 2. The transitions T are the standard CDCL steps of *decide*, *BCP*, *analyze*, and *backtrack* with associated conditions, e.g., conflict analysis is only triggered when a conflict is detected by BCP.

We say that a SAT algorithm A_1 *simulates* another SAT algorithm A_2 iff for any formula F we have the following: any sequence of solver states $\langle \pi, \delta, \rho, \phi \rangle$ executing A_1 on F is possible to be encountered executing A_2 on F. Note that we do not require the same sequence of states on A_1, A_2, but that the same states are reachable in A_1, A_2. Some non-determinism in simulating SAT algorithms lies in the order of implications and conflicts due to the orders of clauses and watch lists.

Algorithm 4 CDCL with Graph Backtracking.

$\pi = \tau = \omega = \emptyset$
$\rho(\ell) = \blacksquare, \delta(\ell) = \infty$
$\forall \ell.\ \gamma(\ell) = \eta(\ell) = \emptyset$
procedure CDCL(F, ζ)
    Fill the watch lists WL for F
    while $\top$ **do**
        $C \leftarrow$ BCP() ▷ Algorithm 3
        if $C = \blacksquare$ **then**
            if $|\pi| = |\mathcal{V}|$ **then** ▷ All variables are assigned
                return SAT
            $\ell \leftarrow$ DECIDE() ▷ No change from (N)CB
            $\omega \leftarrow \omega \cdot \ell, \delta(\ell) \leftarrow \delta(\pi) + 1, \gamma(\ell) \leftarrow \{ck_\ell\}$
            continue
        if $\gamma(C) = \emptyset$ **then**
            return UNSAT
        $\Gamma^* \leftarrow$ SELECTCHUNK(C, ζ) ▷ Equation (1)
        $D \leftarrow$ ANALYZE(C, Γ^*) ▷ Algorithm 1
        UNDO(Γ^*) ▷ Algorithm 2
        $c_2 \leftarrow$ the unassigned literals in D
        $\omega \leftarrow \omega \cdot c_2, \rho(c_2) \leftarrow D, \delta(c_2) \leftarrow \delta(D \setminus \{c_2\}), \gamma(c_2) \leftarrow \gamma(D \setminus \{c_2\})$
        $F \leftarrow F \cup \{D\}$
        if $|D| \geq 2$ **then**
            $c_1 \leftarrow$ another literal in D
            $\text{WL}(c_1) \leftarrow \text{WL}(c_1) \cup \{D\}, \text{WL}(c_2) \leftarrow \text{WL}(c_2) \cup \{D\}$
            $\eta(c_1) \leftarrow \eta(c_1) \cup \gamma(c_2)$

Simulating CB. We write GB_ζ as GB parametrized with the weight function ζ and define the cost function ζ_{CB} to be:

$$\zeta_{\text{CB}}(\ell) = \begin{cases} -1 & \text{if } \delta(\ell) \geq \delta(C) \\ 1 & \text{otherwise} \end{cases}$$

where C is the conflicting clause.

▶ **Theorem 10** (GB simulates CB). *$GB_{\zeta_{\text{CB}}}$ simulates CB.*

Proof. The minimal solution to the optimization problem (1) is the set of chunks Γ^* whose decision levels are higher than or equal to the decision level of the conflict C. That is, $\bigcup_{ck \in \Gamma^*} ck = \{\ell \in \pi : \delta(\ell) \geq \delta(C)\}$ yields the set of literals that CB would backtrack.[5] We next prove that $\text{GB}_{\zeta_{\text{CB}}}$ simulates CB by induction on the sequence of solver states.
Base case. The base case $\langle \pi = \emptyset, \forall \ell.\ \delta(\ell) = \infty, \forall \ell.\ \rho(\ell) = \blacksquare, \phi = F \rangle$ is trivially the same for both $\text{GB}_{\zeta_{\text{CB}}}$ and CB.
Step case. For a transition step $T \in \{decide, BCP, analyze, backtrack\}$, we show that from state $S = \langle \pi, \delta, \rho, \phi \rangle$, if T_{CB} can reach state $S' = \langle \pi', \delta', \rho', \phi' \rangle$ in CB, then $T_{GB_{\zeta_{\text{CB}}}}$ can also reach S', and vice versa.

[5] We note that Γ^* always satisfies the safeguard mechanism of Section 4.1.

- *decide* is identical in $\mathrm{GB}_{\zeta_{\mathrm{CB}}}$ and CB.
- *BCP* in $\mathrm{GB}_{\zeta_{\mathrm{CB}}}$ and CB differ only on the handling of watched literals, but they both find the same implications and conflicts by virtue of Invariant 1. Chunks γ and cross-chunks η do not affect the states S.
- When Γ^* is chosen for *analyze* in $\mathrm{GB}_{\zeta_{\mathrm{CB}}}$, it intersects with the conflict in the highest chunk in the conflict. Therefore, $1\mathrm{UIP}_{\Gamma^*}(C) = 1\mathrm{UIP}_{\{ck:\delta_\gamma(ck)=\delta(C)\}}(C)$. Thus, $\phi' = \phi \cup \{1\mathrm{UIP}_{\Gamma^*}(C)\}$ is the same in $\mathrm{GB}_{\zeta_{\mathrm{CB}}}$ and CB.
- As previously noted, the set of literals *backtrack*ed by $\mathrm{GB}_{\zeta_{\mathrm{CB}}}$ is the same as CB. Therefore, the resulting state after backtracking is the same in $\mathrm{GB}_{\zeta_{\mathrm{CB}}}$ and CB. The solvers may differ in the propagation sets τ and ω, but not in their union.

The implication justifications ρ are used identically in $\mathrm{GB}_{\zeta_{\mathrm{CB}}}$ and CB: given the same partial assignment π, both methods can find the same justifications for the same literals. Since decision levels δ are defined based on decisions and justifications ρ, they also coincide. Therefore, for every transition T, if T can reach state S' from S in $\mathrm{GB}_{\zeta_{\mathrm{CB}}}$, then it can also be reached in CB; and vice versa. ◀

Simulating NCB. NCB is similar to CB, except that in NCB, the backjump level is not known until conflict analysis has been performed. Let $C' = 1\mathrm{UIP}_{\{ck:\delta_\gamma(ck)=\delta(C)\}}(C)$ be the 1UIP of C with respect to the highest chunk in the conflict, and let ℓ' be the unique highest decision level literal in C'. The cost function ζ_{NCB} is defined as:

$$\zeta_{NCB}(\ell) = \begin{cases} -1 & \text{if } \delta(\ell) > \delta\left(C' \setminus \{\ell'\}\right) \\ 1 & \text{otherwise} \end{cases}$$

▶ **Theorem 11** (GB simulates NCB). *$GB_{\zeta_{NCB}}$ simulates NCB.*

Proof. The set of backtracked chunks Γ^* is the set whose chunk levels are higher than the second-highest decision level in the learned clause, which is exactly the set of literals that NCB backtracks. The proof then proceeds as for Theorem 10. ◀

6 Advanced Techniques with Graph Backtracking

We present further improvements and details for implementing a performant GB in Algorithm 4.

6.1 Chunks and Backtrack Candidates

We represent chunks and cross-chunks as sparse indexed bitsets. They allow for efficient set operations by exploiting bit-level parallelism, while also being memory efficient.

While it is theoretically interesting to consider negative weights (see Section 5), ζ often has its image over the positive reals. Therefore, computing the set of backtrack candidates Γ for a conflict C is not done using the powerset of chunks $2^{\gamma(\pi)}$, but rather by iterating over the chunks in the conflict only $\gamma(C)$. Indeed, if $\forall \ell.\ \zeta(\ell) > 0$, then the optimal solution always has a single chunk, and computing the cost of chunk combinations is wasted. We also add a limit on the number of backtrack candidates to consider, to avoid combinatorial explosions (see Section 6.3). In the remainder of the paper, we will only consider the case of positive weights.

6.2 Blockers

The two-watched-literal scheme can be improved with the use of blockers [26]. A blocker is a literal b in a clause C stored in the watch list of a watcher ℓ such that if $b \in \pi$, then C does not need to be inspected when propagating $\neg\ell$. In NCB, Invariant 1 can be relaxed to include blockers as follows.

▶ **Invariant 3** (Watched literals with blockers [11]). *Consider the trail $\pi = \tau \cdot \omega$. For each clause $C \in F$ watched by c_1, c_2 and blocked by b in $\mathrm{WL}(c_1)$, we have $\neg c_1 \in \tau \Rightarrow [c_2 \in \pi \vee b \in \pi]$.*

However, the same issues as discussed in Section 4.3 arise. The order of propagation is not necessarily compatible with the backtracking order. Similarly to Invariant 2, we therefore strengthen Invariant 3 with cross-chunks.

▶ **Invariant 4** (GB blocked watched literals). *Consider the trail $\pi = \tau \cdot \omega$. For each clause $C \in F$ watched by c_1, c_2 and blocked by b in $\mathrm{WL}(c_1)$, we have*

$$\neg c_1 \in \tau \Rightarrow \big[[c_2 \in \pi \wedge \gamma(c_2) \subseteq \eta(c_1)] \vee [b \in \pi \wedge \gamma(b) \subseteq \eta(c_1)]\big].$$

6.3 Chunk Merging

During BCP, we may discover that a clause C is actually a missed implication for a decision $\ell \in ck_\ell$ such that $ck_\ell \notin \gamma(C \setminus \{\ell\})$. In case of conflict, backtracking the chunk ck_ℓ alone would not be fruitful since it would be reimplied by C right after and the conflict would be rediscovered once more. Therefore, when backtracking ck_ℓ, we also should backtrack one of the chunks in $\gamma(C \setminus \{\ell\})$. We can merge the chunk ck_ℓ with $\gamma(C \setminus \{\ell\})$ either during propagation (eagerly), or during conflict repair (lazily).

Eager chunk merging (ECM) is simple and similar to the handling of missed lower implications in CB [19]. When a clause C, implying a decision literal ℓ, is discovered, we update the implication graph by reimplying ℓ with the clause C. The effect of this implication is then rippled through the implication graph, de facto eliminating the chunk ck_ℓ and merging it with the chunks in $\gamma(C \setminus \{\ell\})$. The trail is stably reordered to restore the topological sort. ECM is not always optimal. For instance, let the conflict clause C' spanning over both ck_ℓ and some chunk $ck \in \gamma(C \setminus \{\ell\})$. If ck_ℓ is merged eagerly, then we would backtrack ck_ℓ as well as ck. However, if the chunks were not merged, we could backtrack ck alone, which is necessarily lighter than $ck_\ell \cup ck$.

On the other hand, *lazy chunk merging* (LCM) virtually merges chunks during conflict repair. To do so, during propagation, we remember potential reimplications similarly to [11], and ensure that no cycle is created. When creating backtrack candidates Γ (Section 6.1), we enhance Γ by expanding chunks that would be merged. This way, the algorithm is more informed for chunk selection, and never needs to materialize the merge. It simply undoes the union of the chunks. However, if a lot of virtual merges are introduced, the number of options to consider during conflict repair can explode and deteriorate performance.

We note $\textsc{Analyze}(C, \Gamma^*)$ is adapted such that when the decision literal of a chunk is analyzed, the missed implication is used for resolution, similarly to Algorithm 4 in [11].

▶ **Example 12.** In the example of Figure 1, the clause $C_7 = a \vee \neg c \vee \neg d$, can merge ck_a with ck_c and ck_d. Using LCM, the backtrack options are $\Gamma = \{\{ck_a, ck_c\}, \{ck_a, ck_d\}, \{ck_c\}, \{ck_d\}\}$. To avoid unnecessary cost estimation, we perform subsumption simplifications on the chunks to be merged. In our example, we do not calculate the weights of $\{ck_a, ck_c\}$ and $\{ck_a, ck_d\}$, as they will always be more expensive than $\{ck_c\}$ and $\{ck_d\}$ respectively.

6.4 Lightest vs. Learning Chunks

Section 4.1 introduced some safeguards to ensure progress. Namely, if a chunk does not lead to learning a new clause, and is not at the highest level in the conflict C, then we did not consider it in the terminating backtrack candidates Γ. However, this constraint can be relaxed. If for some chunks $\Gamma_i \in \Gamma$, we manage to learn a new clause, we can safely backtrack any chunk in $\{\Gamma_j \in 2^{\gamma(\pi)} : |\Gamma_j \cap \gamma(C)| = 1\}$, even if it did not lead to learning the new clause. In particular, we can backtrack the lightest chunk.

▶ **Example 13.** Consider Figure 1a. Undoing ck_b is not allowed because of the safeguard mechanism. However, in Example 12, ck_b would be the cheapest choice because of LCM. We learn a new clause $1\text{UIP}_{ck_d}(C_8) = C_9 = \neg w \vee \neg y \vee \neg z$ and backtrack ck_b.

7 Empirical Evaluation

We implemented our graph backtracking approach in the experimental solver NapSAT [22]. Our solver is based on the CDCL algorithm [23] using the two-watched-literal scheme [18], VSIDS branching heuristic [18] with phase caching [13] and clause shrinking [25]. The core implementation consists in roughly 2.7k effective lines of C++ code, covering all options of GB presented in Sections 4 and 6, as well as standard (N)CB. For a fair comparison, we attempted to maximize code sharing between the different backtracking options; as such, only 2% of the code of NCB is not shared with GB, whereas GB represents around 27% more code than NCB[6]. Since our focus is on the comparison of backtracking strategies, we did not implement auxiliary techniques, such as pre- and in-processing, clause minimization, and non-redundant clause deletion (we only purge clauses satisfied at root level). To eliminate engineering differences, we compare the different backtracking strategies in the same solver. All our computations and results were executed on an AMD EPYC 7502 2×32-core processor running at 2.5 GHz and equipped with 1 TB of RAM.

Weight heuristic. We used a simple weight heuristic to minimize propagations. If a literal ℓ is propagated, then $\zeta(\ell) = 8$; otherwise, $\zeta(\ell) = 1$. The coefficient 8 was chosen without any fine-tuning. A power of two is however beneficial for stability of floating point arithmetic. We also add a small penalty to chunks whose decisions are located earlier in the trail. This encourages the solver to behave more like CB when the weights only differ slightly.

Configurations. Below, we present the configurations that were evaluated. NapSAT always uses blockers (Section 6.2).

- NCB: Non-chronological backtracking as described in Section 3 and [13].
- CB: Chronological backtracking as described in Section 3 and [20].
- LSCB: Lazy Strong Chronological Backtracking as described in [11].
- GB: Vanilla graph backtracking as described in Section 4.
- LCM: Lazy chunk merging as described in Section 6.3.
- ECM: Eager chunk merging as described in Section 6.3.
- BB: Backtracking the best chunk as described in Section 6.4.

Benchmarks. To illustrate the performance of graph backtracking, we generated 1000 satisfiable instances of 3-coloring problems using the `cnfgen` tool [16] with arguments `kcolor 3 gnm 650 1469`. We aimed at finding satisfiable instances sufficiently difficult for our solver,

[6] Calculated using `gcc -coverage` with `lcov`

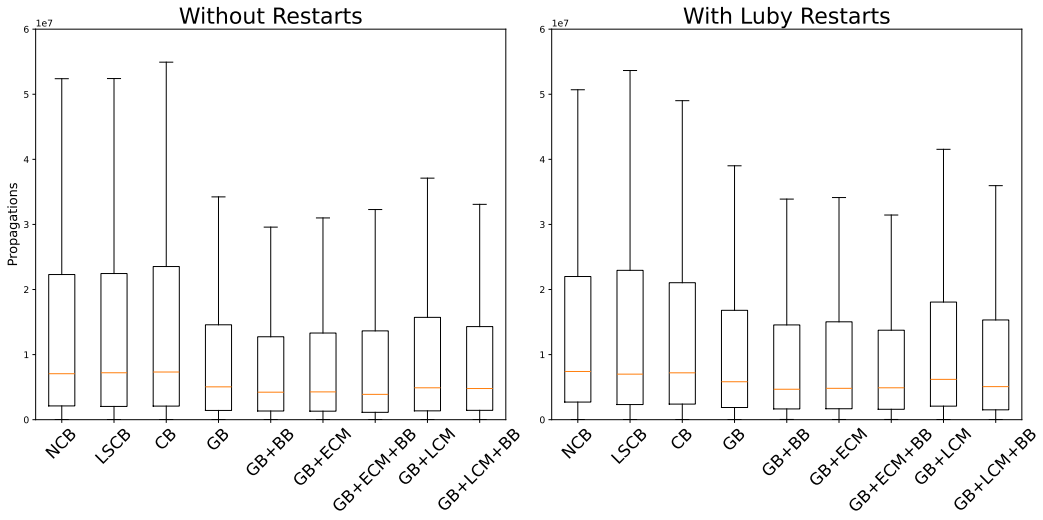


Figure 3 Comparison of the number of propagations for NCB, CB and GB variants.

while not too difficult to solve within a reasonable time. Each problem is run with a timeout of 2 hours. Since the SAT competition problems are tuned to existing SAT techniques, they are not the best fit for our experiments. Graph coloring problems are well-suited for our application because they do not involve too many variables. Further, we could tune the hardness of problems to the capabilities of our solver, which is important to get a good picture of the performance of the different backtracking strategies.

Result analysis. Our experiments are summarized in Table 1 and Figure 3. Our approach consistently executes fewer propagations than (N)CB. Our best configuration is GB+ECM as it executes 47% fewer propagations than NCB and decreases the runtime by around 30%.

In our experiments, ECM makes the solver more stable, reaching fewer timeouts and reducing the variance of the runtime. This is because, without chunk merging, chains of conflicts can occur and lead to undoing the same number of literals in more backtracking steps, which is very costly. On the other hand, LCM introduces a lot of variance in the cost of computing backtrack candidates and their weights. This is why LCM has the lowest propagations per second, but also the highest variance in runtime. While the results of our experiments suggest that LCM is not very effective, it is still worth investigating it further. Problems with many variables and binary clauses may benefit LCM because the cost of eager merging is proportional to the number of literals that need to be reimplied.

Backtracking the best chunk (BB) has mixed results, improving the performance of GB alone, and with LCM, but not with ECM. However, the penalty witnessed with ECM is not very large. It seems that BB is a good option because it balances the safeguard mechanism described in Section 4.1. This mechanism is meant to prevent a very unlikely scenario where the solver would loop forever. Sacrificing the best chunk for learning a clause goes against the philosophy of GB. BB gets us back to the core idea of GB, while keeping termination.

Unsurprisingly, restarts penalize GB because they clash with the intention of GB to minimize the number of unassignments. The comparison can be found on Figure 3. Indeed, restarts work well when propagations are cheap, which is less the case in GB. Reordering the decisions is less important in GB since the solver can backtrack any of the decisions involved in the conflict, and not only the most recent one.

The results also highlight the trade-off between time and number of propagations. While

Option	Timeout	PAR2		Time (s)		Propagations $\times 10^6$	
NCB	16	380.7	-	106 ± 428	-	23.7 ± 46.4	-
LSCB	12	335.0	88.00%	100 ± 388	94.27%	22.1 ± 41.0	93.07%
CB	11	333.1	87.50%	103 ± 414	97.33%	22.9 ± 42.8	96.30%
GB	17	378.1	99.31%	103 ± 402	96.91%	14.2 ± 25.3	60.05%
GB+BB	12	304.2	79.71%	91.6 ± 380	85.98%	12.6 ± 23.0	53.27%
GB+ECM	11	299.7	78.72%	72.4 ± 310	67.89%	12.6 ± 23.8	52.96%
GB+ECM+BB	12	303.5	79.72%	84.9 ± 366	79.66%	13.0 ± 25.0	54.64%
GB+LCM	14	376.8	98.98%	124 ± 485	117.1%	15.8 ± 29.9	66.39%
GB+LCM+BB	16	387.4	101.8%	97.0 ± 332	91.05%	13.9 ± 24.2	58.71%

Table 1 Statistics of the experiments. The timeout column indicates the number of instances that timed out for each configuration. PAR2 indicates the penalized average of runtimes, where each timeout is counted as 2 times the timeout value (2 hours). The average times and propagations are computed excluding the 50/1000 instances for which at least one of the configurations timed out. Standard deviation is shown after the ± symbol. Relative values (in percentages) are computed with respect to NCB and displayed on the right of the respective columns.

GB executes significantly fewer propagations, each of them is slower than in (N)CB. This is because of the chunk and cross-chunk maintenance, and the more complex backtracking procedure, which requires going through the entire trail to find the literals that need to be propagated again. GB shines particularly when the weight of chunks varies significantly.

8 Conclusion and Future Work

We introduce a graph backtracking (GB), yielding a more surgical backtracking approach for CDCL solvers. GB generalizes the backtracking process of (N)CB and adapts to user preferences over literal unassignments. Our GB approach is sound and terminating. We implemented GB in the experimental solver NapSAT and showed that it performs better on graph coloring benchmarks. Particularly, GB backtracks fewer literals in single-shot SMT conflicts, compared to (N)CB.

The AVATAR framework [28] is a promising application for graph backtracking as it does not require a given order of literals in the trail, only sets of assigned literals. The integration of NapSAT in the first-order prover Vampire [3] is ongoing. Scaling the algorithms to large number of decisions, and therefore chunks is still an open problem. Integrating GB in SMT solving is another future work direction. SMT solvers typically follow a more rigid assignment structure for their theory solvers, which does not fit well with graph backtracking. Relaxing this stack assumption to benefit from chronological and graph backtracking is future work.

References

1 Bruno Andreotti and Haniel Barbosa. Producing shorter congruence closure proofs in a state-of-the-art SMT solver. In *VMCAI*, volume 16417 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2026.

2 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

3 Filip Bártek, Ahmed Bhayat, Robin Coutelier, Márton Hajdú, Matthias Hetzenberger, Petra Hozzová, Laura Kovács, Jakob Rath, Michael Rawson, Giles Reger, Martin Suda, Johannes Schoisswohl, and Andrei Voronkov. The vampire diary. In *CAV (3)*, volume 15933 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2025.

4 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. Cadical 2.0. In *CAV (1)*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024.

5 Armin Biere, Mathias Fleury, and Florian Pollitt. CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT entering the SAT competition 2023. In *SAT Competition 2023 – Solver and Benchmark Descriptions*, volume B-2023-1 of *Department of Computer Science Report Series B*, pages 14–15. University of Helsinki, 2023.

6 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

7 Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. Satisfiability modulo custom theories in Z3. In *VMCAI*, volume 13881 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 2023.

8 Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

9 Yasmine Briefs and Mathias Fleury. From building blocks to real sat solvers. First-Order Reasoning, Below and Beyond: Workshop in Honor of Christoph Weidenbach's 60th Birthday, Colocated with CADE'30, 30th International Conference on Automated Deduction, Stuttgart, Germany, July 28-31, 2025, 2025. Presentation only.

10 Shaowei Cai and Kaile Su. Configuration checking with aspiration in local search for SAT. In *AAAI*, pages 434–440. AAAI Press, 2012.

11 Robin Coutelier, Mathias Fleury, and Laura Kovács. Lazy reimplication in chronological backtracking. In *SAT*, volume 305 of *LIPIcs*, pages 9:1–9:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

12 Robin Coutelier, Thomas Hader, and Laura Kovács. Generalizing CDCL with Graph Backtracking, 2026. URL: `ADD`, `arXiv:ADD`.

13 Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

14 Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. Satisfiability modulo user propagators. *J. Artif. Intell. Res.*, 81:989–1017, 2024.

15 Matthew L. Ginsberg. Dynamic backtracking. *J. Artif. Intell. Res.*, 1:25–46, 1993.

16 Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. Cnfgen: A generator of crafted benchmarks. In *SAT*, volume 10491 of *Lecture Notes in Computer Science*, pages 464–473. Springer, 2017.

17 Sibylle Möhle and Armin Biere. Backing backtracking. In *SAT*, volume 11628 of *Lecture Notes in Computer Science*, pages 250–266. Springer, 2019.

18 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001.

19 Alexander Nadel. Introducing Intel(R) SAT solver. In *SAT*, volume 236 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

20 Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *SAT*, volume 10929 of *Lecture Notes in Computer Science*, pages 111–121. Springer, 2018.

21 Mathias Preiner, Hans-Jörg Schurr, Clark Barrett, Pascal Fontaine, Aina Niemetz, and Cesare Tinelli. Smt-lib release 2024 (non-incremental benchmarks), April 2024. `doi:10.5281/zenodo.11061097`.

22 Coutelier Robin. NapSAT solver. Accessed March 2026. URL: `https://github.com/RobCoutel/NapSAT`.

23 João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

24 Mate Soos and Kuldeep S. Meel. Engineering an efficient probabilistic exact model counter. In *CAV (3)*, volume 15933 of *Lecture Notes in Computer Science*, pages 72–91. Springer, 2025.

25 Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

26 Niklas Sörensson and Niklas Eén. Minisat 2.1 and minisat++ 1.0-sat race 2008 editions. *SAT*, page 31, 2009.

27 Giuseppe Spallitta, Roberto Sebastiani, and Armin Biere. Disjoint projected enumeration for SAT and SMT without blocking clauses. *Artif. Intell.*, 345:104346, 2025.

28 Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 696–710. Springer, 2014.

A Soundness, Completeness and Termination

We prove that the graph backtracking approach of Algorithm 4 is sound, complete, and terminating.

▶ **Lemma 7.** *The graph backtracking CDCL-based approach of Algorithm 4 preserves Invariant 2.*

Proof. The proof is similar to the proof of soundness of lazy strong chronological backtracking [11]. We show that Invariant 2 is preserved by each of the procedures of GB. The detailed steps for BCP are included in the source code of NapSAT.

BCP. The BCP algorithm transfers, one by one, literals from the waiting queue ω to the propagated set τ. BCP() preserves Invariant 2 by induction. At the start, $\pi = \tau = \omega = \emptyset$ and the invariant is trivially satisfied. Let us assume the current partial assignment $\pi = \tau \cdot \omega$ satisfies Invariant 2 before executing BCP(). A literal $\ell \in \omega$ can be added to the propagated set τ if we first ensure that for each clause $C \in F$ watched by c_1, c_2,

$$\neg c_1 \in (\tau \cdot \ell) \Rightarrow [c_2 \in \pi \wedge \gamma(c_2) \subseteq \eta(c_1)] \tag{2}$$

All clauses that are not watched by $\neg\ell$ satisfy Equation (2) by the inductive hypothesis, and do not need to be inspected. We therefore only consider clauses watched by $\neg\ell$. When we attempt to add ℓ to τ, for each clause C, one of following can occur:

- If $c_2 \in \pi \wedge \gamma(c_2) \subseteq \eta(c_1)$, then nothing needs to be done.
- If there exists a replacement literal $r \in C \setminus \{c_2\}$ such that $\neg r \notin \pi$, we watch r instead of c_1 and have $\neg r \notin (\tau \cdot \ell)$ since $\ell \in \omega \subseteq \pi$.
- If $\neg c_2 \in \pi$ and no good replacement exists, then BCP() returns the conflict clause C and ℓ is not added to τ.
- If c_2 is unassigned and no good replacement exists, then c_2 is added to ω such that $\gamma(c_2) = \gamma(C)$. We then update $\eta(c_1)$ to include $\gamma(c_2)$.
- If $c_2 \in \pi$ but $\gamma(c_2) \not\subseteq \eta(c_1)$, then the algorithm forces $\gamma(c_2) \subseteq \eta(c_1)$ by updating $\eta(c_1)$.

The cases discussed above cover all possible scenarios. For each, either ℓ is not added to τ because of a conflict, or Equation (2) is satisfied.

Chunk Selection & Conflict Analysis. Chunk Selection and Conflict analysis do not change the state of the partial assignment, nor the clauses. Therefore, they trivially preserve the invariant.

Backtracking. Backtracking may change the partial assignment by removing literals. However, if a clause watched by c_1, c_2 was such that $\neg c_1 \in \tau \wedge c_2 \in \pi$, then either c_2 is left untouched, or $\neg c_1$ is removed from τ (either removed from π or placed in ω). Therefore, after backtracking, it is not possible that $\neg c_1 \in \tau \Rightarrow c_2 \in \pi$ is violated. The chunk sets of literals remain unchanged, and therefore the invariant is preserved.

CDCL. Only two aspects of CDCL have not been examined so far. First, the decisions made during the search. However, since this part of the algorithm only adds literals to the waiting queue ω, the invariant is trivially preserved. Second, when the learned clause is added to the clause set F, the invariant is restored because we know that the new clause has exactly one unassigned literal (by design of the UIP), and therefore, pushing the new implication and updating the cross-chunks of the other literal is sufficient to maintain the invariant. ◀

Strong with Lemma 7, we can now show that the graph backtracking algorithm find all implications and cannot miss a conflict.

▶ **Lemma 8.** *In Algorithm 4, no clause is conflicting or unit with the propagated set τ and not satisfied by π.*

Proof. By contradiction, if a clause C was falsified or unit by the propagated set τ, then at least one of its watchers would be falsified in τ, and the other cannot be satisfied. This contradicts Lemma 7. ◀

▶ **Theorem 9.1.** *CDCL-based SAT solving using GB in Algorithm 4 is sound.*

Proof. With Lemma 8, a satisfiying assignment cannot be conflicting with any clause of the formula. Therefore, if GB CDCL returns SAT, the assignment is indeed satisfying. If GB CDCL returns UNSAT, then the empty clause was derived by resolution from clauses of the formula. Therefore, the formula is indeed unsatisfiable. ◀

▶ **Theorem 9.2.** *CDCL-based SAT solving using GB in Algorithm 4 is complete.*

Proof. If the formula is satisfiable, then there exists a satisfying assignment π. Since the algorithm only learns clauses that are entailed by the formula, π remains a satisfying assignment of the formula throughout the execution of the algorithm. Therefore, the algorithm cannot return UNSAT. If the formula is unsatisfiable, then by completeness of resolution, the empty clause can be derived by resolution from clauses of the formula. Therefore, the algorithm eventually returns UNSAT. ◀

▶ **Theorem 9.3.** *CDCL-based SAT solving using GB in Algorithm 4 terminates.*

Proof. The termination of each subalgorithm of GB CDCL follows from the finite set of variables and clauses. It does not need further explanation as it is similar to other CDCL variants.

Progress is made by two means: (i) Assigning literals that do not belong to a chunk (root level in NCB). When a literal ℓ is assigned and does not belong to any chunk, it means it is a direct implication of a subset of the formula F. Therefore, it cannot be backtracked. Since there are finitely many variables, assigning a literal ℓ such that $\gamma(\ell) = \emptyset$ is progress towards termination. (ii) Learning a new entailed clause makes progress since there are finitely many distinct clauses entailed by the formula F.

We show that always eventually, if the algorithm is not finished, it will make progress. When a conflict C is discovered, we perform conflict analysis and obtain the entailed clause C' one of two things happens:

1. $C' \notin F$ and we make progress by virtue of (ii)
2. $C' \in F$ and the learned clause is redundant. Let $\gamma(C)$ be the set of chunks on which C' is conflicting.
   a. If $|\gamma(C')| = 1$, then, ignoring root level literals, the learned clause must be of length one as well because of the UIP algorithm. After backtracking, we imply a literal ℓ such that $\gamma(\ell) = \emptyset$ and make progress by virtue of (i).
   b. If $|\gamma(C')| > 1$, then the algorithm undoes the chunk $ck \in \gamma(C')$ whose decision d is at the highest decision level among $\gamma(C')$ (the one that was decided last). This is enforced by the safeguard of Section 4.1. Let the literal $\ell \in ck$ be the UIP. We call *hot potatoes* the literals in a chunk that were flipped because of a conflict. As long as a potato lives, the conflict clause that created it cannot happen again (since the clause is satisfied). Hot potatoes might get undone, but only if a lower one is added. Hot potatoes can only sink in terms of decision level. Therefore, it cannot be that the same conflicts are infinitely rediscovered without making progress.

◀

B Conflict Search Strategy

When a conflict is detected, a few options are possible. The simplest, and most commonly used, is to interrupt BCP and trigger conflict repair immediately. However, this is not necessarily always the best option. Indeed, we sometimes want to gather all conflicts that can be discovered at once, especially when the trail does not have a monotonic structure like in CB or GB [19, 9].

We differentiate between three strategies: (i) immediate, (ii) partial, and (iii) exhaustive conflict repair. (i) Immediate conflict repair is the simplest strategy. As soon as a conflict is detected, BCP is interrupted and conflict analysis is triggered. (ii) Partial conflict repair propagates all literals in the first conflict clause before triggering conflict analysis. Finally, (iii) exhaustive conflict repair continues BCP until the queue ω is empty.

State-of-the-art solvers like Kissat [5] and CaDiCaL [4] do immediate conflict repair, with Intel SAT [19] doing exhaustive.

Immediate Conflict Repair. In light of the termination criterion presented in Section 4.1, it is important to learn at least one new clause for each conflict repair that does not occur at the highest chunk. However, a hidden problem occurring in both GB and CB is the possibility to learn an existing, or subsumed clause. This can happen if multiple conflicts occur simultaneously. Using immediate conflict repair, we analyze the first conflict C to obtain D. If $\exists C' \in F$ such that $C' \subseteq D$, we would not make progress. A brutish way to verify whether the clause D is new, is to traverse the watch lists of each literal of D. If the clause is subsumed, another chunk must be selected.

Partial and Exhaustive Conflict Repair. While no known attempt was successful in finding value in exhaustive conflict repair in NCB, it has been experimented on in the context of CB [19, 9]. In applications where the procedure calling the SAT solver is orders of magnitude more expensive than the SAT solver itself, it might be worth gathering more conflicts to select the lightest set of literals with more accuracy. Collecting more conflicts is transparent to the user and allows better choices of chunk to backtrack.

By virtue of the topological order of propagations, partial conflict repair ensures that if the first conflict's learned clause D is subsumed by another existing clause C', then C' is conflicting and will also be discovered. Exhaustive conflict repair ensures that if any learned clause is subsumed, the subsuming clause will part of the conflicting clause set.

Partial and exhaustive conflict repair are non-trivial to implement. Gathering all the conflicts is not difficult, but handling them simultaneously is. We only present the exhaustive conflict repair procedure in broad strokes, with details in the code base. Partial conflict repair procedure is similar, but with weaker guarantees on the novelty of learned clauses.

Considering a set κ of conflicts discovered during BCP, the first step is to find the set $\Gamma = \{\Gamma_i | \forall C \in \kappa . \Gamma_i \cap \gamma(C) \neq \emptyset\}$ of all set of chunks Γ_i that can be backtracked to repair all conflicts at once. We use a greedy algorithm enhanced with subsumption to find a small set of solutions. Then, we expand each Γ_i with the lazy merges (Section 6.3) and perform another subsumption simplification.

Note that this definition of backtrack candidates after subsumption ensures that at least one of the conflicts can either learn a UIP, or is already a UIP itself. Therefore, we are guaranteed to be able to imply a flipped literal after backtracking, which is necessary to ensure progress.

We filter all sets of chunks Γ_i that cannot lead to new clauses and do not contain the highest chunk level out of Γ. We then select Γ^*, the lightest chunks in Γ with respect to ζ.

Finally, we attempt to learn new clauses for each conflict $C \in \kappa$ using the selected Γ^*. If

all learned clauses are subsumed by existing clauses, we select another Γ_i and try again. We stop when we learn a new clause, or when we analyze a top level chunk.

C Additional Empirical Analysis

In this section, we provide additional representation and comments of our experimental results. The option acronyms and experimental setup were defined and described in Section 7.

Additionally, we evaluated the different conflict research strategies as descived in Appendix B and the impact of restarts on GB. These results are available in Appendix C.

However, due to the large number of configurations and cost of exhaustive conflict search, the following results were obtained on smaller problems than the ones presented in Section 7. We generated 1000 satisfiable instances of 3-coloring problems using the `cnfgen` tool [16] with arguments `kcolor 3 gnm 400 920`.

C.1 Experimental Setup

Target metric. The motivation of GB is to give some control to choose which literals should be unassigned upon conflict discovery. To illustrate the performance of GB, we use an approximation of the work performed by a user solving incremental SAT instances. We measure the number of *synchronizations* (syncs) between the solver and the user. That is, before each decision of the SAT solver, we assume a user synchronizing its internal state with the SAT solver (similar to user propagators [14] and how `veriT` [8] works). If a variable has changed polarity in the assignment compared to the last synchronization, we count one `sync`. To minimize the number of synchronizations, the decision heuristic assigns literals to the same polarity as the last synchronization.

Decision polarity. In standard mode, NapSAT uses phase cashing to assign decision literals to the same polarity as its last assignment. To minimize the changes between synchronizations, we replace phase caching with a more aggressive approach that assigns decision literals to the same polarity as the last synchronization. This way, the solver prefers to keep literals synchronized. In addition, we also add a small penalty to chunks whose decisions are located earlier in the trail. The encourages the solver to behave more like CB when it does not matter. Further, experiments show that analyzing conflicts on earlier chunks generally leads to larger learned clauses.

Additional options. We also evaluated the impact of the conflict strategy (immediate, partial and exhaustive conflict repair) and the impact of restarts on GB. In addition to the options described in Section 7, we also evaluated the following configurations:

PCR Partial conflict repair.
ECR Exhaustive conflict repair.

In terms of synchronizations, LSCB and CB are very similar, we therefore did not run LSCB for these experiments. No timeout was used for these experiments.

C.2 Results and Analysis

Raw statistics. Table 2 shows the average time, number of synchronizations for each configuration. Graph backtracking (GB), no matter its variant, consistently executes fewer synchronizations than NCB and CB, on average. Our best configuration, GB+BB, executes around 37% fewer synchronizations than NCB, with a reasonable overhead in execution time.

Option	Time (s)	Time relative	Sync $\times 10^3$	Sync relative
NCB	1.43 ± 4.66	-	85.1 ± 118	-
CB	1.26 ± 3.26	88.09% (88.09%)	74.0 ± 98.1	86.87% (86.87%)
GB	2.24 ± 6.35	156.2% (156.2%)	49.4 ± 75.6	57.99% (57.99%)
GB+BB	3.16 ± 20.3	220.9% (220.9%)	53.4 ± 106	62.77% (62.77%)
GB+ECM	2.27 ± 10.8	158.6% (158.6%)	55.7 ± 96.4	65.38% (65.38%)
GB+ECM+BB	1.82 ± 5.20	126.8% (126.8%)	52.0 ± 79.8	61.08% (61.08%)
GB+LCM	2.54 ± 6.61	177.1% (177.1%)	54.4 ± 82.6	63.96% (63.96%)
GB+LCM+BB	3.09 ± 12.6	215.7% (215.7%)	56.2 ± 99.9	66.00% (66.00%)
NCB+PCR	1.71 ± 4.75	- (119.1%)	83.1 ± 110	- (97.59%)
CB+PCR	1.65 ± 4.77	96.67% (115.2%)	73.0 ± 97.0	87.88% (85.76%)
GB+PCR	3.17 ± 11.0	185.8% (221.3%)	49.7 ± 79.4	59.82% (58.38%)
GB+BB+PCR	3.41 ± 11.6	199.6% (237.8%)	51.6 ± 83.0	62.12% (60.62%)
GB+ECM+PCR	2.80 ± 13.3	164.3% (195.7%)	52.4 ± 88.4	63.10% (61.58%)
GB+ECM+BB+PCR	2.49 ± 8.50	146.0% (174.0%)	53.1 ± 82.8	63.95% (62.41%)
GB+LCM+PCR	3.40 ± 12.3	199.2% (237.3%)	52.7 ± 82.1	63.38% (61.86%)
GB+LCM+BB+PCR	5.73 ± 41.2	335.8% (400.0%)	58.3 ± 113	70.16% (68.47%)
NCB+ECR	27.3 ± 89.3	- (1909%)	78.6 ± 106	- (92.37%)
CB+ECR	27.3 ± 102	99.73% (1903%)	66.0 ± 89.4	83.95% (77.54%)
GB+ECR	74.8 ± 589	273.5% (5221%)	44.3 ± 77.6	56.38% (52.07%)
GB+BB+ECR	72.3 ± 853	264.3% (5046%)	43.2 ± 74.8	54.97% (50.77%)
GB+ECM+ECR	51.5 ± 655	188.2% (3593%)	48.4 ± 87.5	61.52% (56.82%)
GB+ECM+BB+ECR	47.5 ± 263	173.7% (3316%)	49.1 ± 82.6	62.47% (57.70%)
GB+LCM+ECR	68.8 ± 385	251.7% (4804%)	50.1 ± 84.3	63.66% (58.80%)
GB+LCM+BB+ECR	50.7 ± 172	185.4% (3539%)	47.2 ± 70.6	59.97% (55.39%)

Table 2 Average time and number of synchronizations for each configuration with **restarts disabled**. Standard deviation is shown after the ± symbol. The relative values are computed with respect to the NCB configuration with the same conflict research method and restart policy. In parentheses, we also show the relative values with respect to the NCB with immediate conflict repair and no restart.

ECR can bring this number to 44%, but at a significant cost in terms of execution time and number of propagations.

Cactus plots. Figures 4 and 5 show cactus plots of the number of problems solved within a given time and number of synchronizations and conflicts. Figure 4 displays the results for the experiments presented in Section 7, while Figure 5 shows the results for the experiments presented in this section.

We filtered out the redundant options for readability

Conflict Search Strategies. The experiments show that ECR increases the computation cost of the solver by orders of magnitude, while only reducing the number of synchronizations by a few percents. ECR does benefit GB slightly more than CB and NCB, but overall, the cost of ECR seems quite prohibitive for the gain it provides.

PCR does not show any significant improvement in terms of number of synchronizations. The implementation effort of PCR being very similar to ECR, we believe that PCR is not worth the cost either.

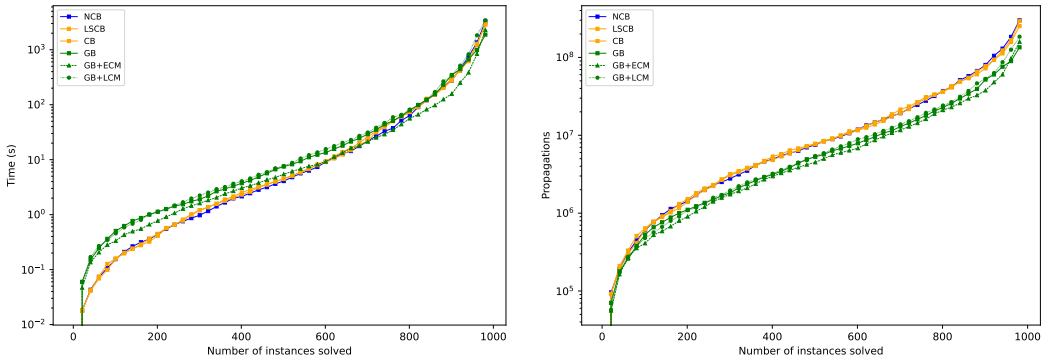
Different conflict search strategies could be useful in applications with very high cost of

Option	Time (s)	Time relative	Sync $\times 10^3$	Sync relative
NCB	1.82 ± 7.11	- (127.2%)	85.1 ± 135	- (99.98%)
CB	1.68 ± 5.48	92.28% (117.4%)	83.0 ± 122	97.56% (97.55%)
GB	3.53 ± 12.2	193.7% (246.4%)	63.0 ± 99.9	73.97% (73.96%)
GB+BB	3.63 ± 20.3	199.5% (253.8%)	63.7 ± 105	74.81% (74.81%)
GB+ECM	2.88 ± 9.80	158.0% (201.1%)	66.4 ± 110	78.00% (77.99%)
GB+ECM+BB	3.41 ± 20.2	187.2% (238.2%)	66.7 ± 125	78.39% (78.39%)
GB+LCM	4.66 ± 26.2	256.0% (325.7%)	66.0 ± 126	77.58% (77.57%)
GB+LCM+BB	4.23 ± 16.9	231.9% (295.0%)	66.6 ± 116	78.28% (78.26%)
NCB+PCR	2.56 ± 16.5	- (179.0%)	82.3 ± 130	- (96.70%)
CB+PCR	1.96 ± 6.86	76.55% (137.0%)	75.4 ± 107	91.54% (88.52%)
GB+PCR	8.15 ± 109	317.8% (568.9%)	64.3 ± 117	78.15% (75.57%)
GB+BB+PCR	5.61 ± 33.3	218.7% (391.4%)	62.0 ± 103	75.28% (72.79%)
GB+ECM+PCR	4.17 ± 21.4	162.6% (291.1%)	65.6 ± 110	79.64% (77.02%)
GB+ECM+BB+PCR	4.85 ± 35.3	189.3% (338.8%)	65.5 ± 116	79.54% (76.92%)
GB+LCM+PCR	8.18 ± 86.5	319.0% (570.9%)	67.4 ± 123	81.86% (79.16%)
GB+LCM+BB+PCR	5.96 ± 31.5	232.6% (416.3%)	66.0 ± 110	80.14% (77.50%)
NCB+ECR	38.7 ± 162	- (2702%)	78.4 ± 118	- (92.02%)
CB+ECR	30.6 ± 101	78.93% (2132%)	68.9 ± 93.1	87.99% (80.97%)
GB+ECR	90.8 ± 346	234.5% (6336%)	51.6 ± 75.9	65.87% (60.62%)
GB+BB+ECR	102 ± 463	263.7% (7126%)	51.3 ± 81.3	65.50% (60.28%)
GB+ECM+ECR	94.4 ± 1329	243.8% (6588%)	58.5 ± 118	74.71% (68.75%)
GB+ECM+BB+ECR	74.4 ± 651	192.3% (5196%)	59.5 ± 104	75.98% (69.92%)
GB+LCM+ECR	181 ± 1619	468.8% (12669%)	58.2 ± 111	74.24% (68.32%)
GB+LCM+BB+ECR	129 ± 684	333.7% (9017%)	54.6 ± 89.8	69.62% (64.07%)

Table 3 Average time and number of synchronizations for each configuration with **restarts enabled**. Standard deviation is shown after the ± symbol. The relative values are computed with respect to the NCB configuration with the same conflict research method and restart policy. In parentheses, we also show the relative values with respect to the NCB with immediate conflict repair and no restart.

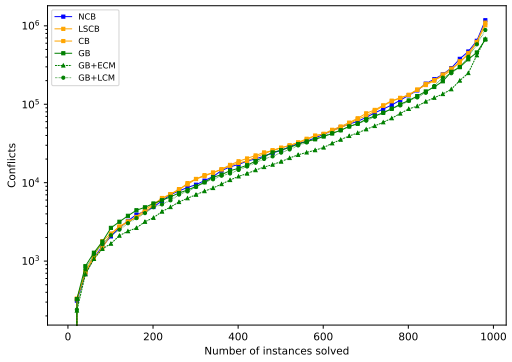
Option	Timeout	PAR2		Time (s)		Propagations $\times 10^6$	
NCB	6	219	-	66.6 ± 262	100.0%	20.5 ± 36.1	-
LSCB	10	244	111.5%	57.5 ± 215	86.43%	19.5 ± 32.6	95.06%
CB	4	182	83.45%	66.3 ± 299	99.66%	20.6 ± 37.7	100.4%
GB	13	331	151.2%	96.7 ± 347	145.3%	15.0 ± 25.0	72.96%
GB+BB	11	303	138.6%	83.6 ± 349	125.5%	12.8 ± 22.4	62.55%
GB+ECM	17	390	178.2%	91.3 ± 420	137.2%	14.6 ± 27.7	71.13%
GB+ECM+BB	15	350	159.7%	91.2 ± 419	137.0%	14.0 ± 27.4	68.11%
GB+LCM	16	423	193.2%	137 ± 541	206.8%	17.5 ± 31.3	85.05%
GB+LCM+BB	14	349	159.3%	98.5 ± 430	147.9%	14.2 ± 26.0	69.09%

Table 4 Statistics of the benchmarks described in Section 7 with **restarts enabled**. The timeout column indicates the number of instances that timed out for each configuration. PAR2 indicates the penalized average of runtimes, where each timeout is counted as 2 times the timeout value (2 hours). The average times and propagations are computed excluding the 50/1000 instances for which at least one of the configurations timed out. Standard deviation is shown after the ± symbol. Relative values (in percentages) are computed with respect to NCB and displayed on the right of the respective columns.



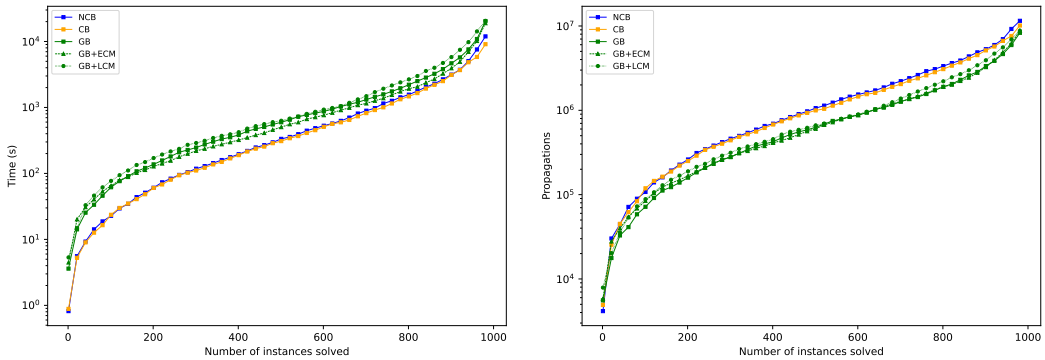
(a) Number solved over time.

(b) Number solved over number of propagations.



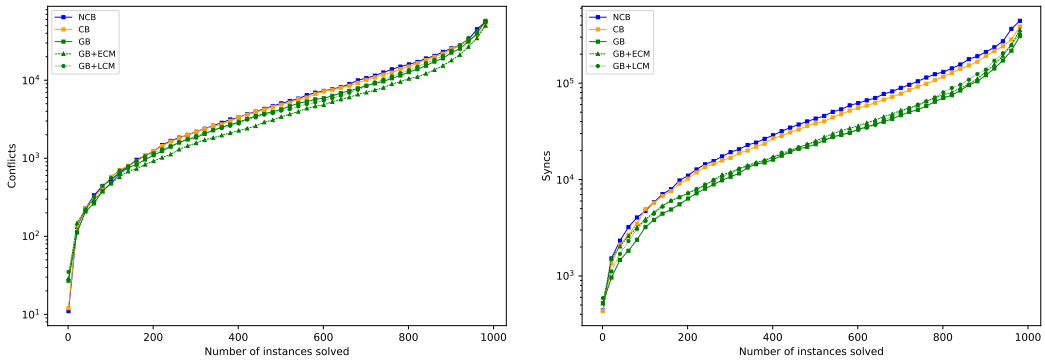
(c) Number solved over number of conflicts.

Figure 4 Cactus plot of the number of problems solved for NCB, CB, and GB for the experiments of Section 7. The lower the better.



(a) Number solved over time.

(b) Number solved over number of propagations.



(c) Number solved over number of conflicts.

(d) Number solved over number of syncs.

Figure 5 Cactus plot of the number of problems solved for NCB, CB, and GB for the experiments of Section 7. The lower the better.

synchronizations, where the overhead of ECR and PCR could be amortized. However, there probably are more efficient ways to improve performance of such applications.

We conclude that PCR/ECR are not worth the cost.

Chunk Merging. Interestingly, on pure SAT problems, ECM seems to be the most effective chunk merging strategy for computation time, while no merging seems slighlty better for the number of synchronizations. As indicated in Section 6.3, ECM makes backtracking more aggressive, which can lead to additional unassigned literals and therefore more synchronizations. On the other hand, LCM is more conservative but seems to have a detrimental effect on both the solving time and the number of synchronization. The solving time penalty probably comes from the fact that LCM makes chunk cost calculation more expensive. The reason for the increase in synchronizations is less clear. It could be due to the quality of learned clauses using the missed implications.

Restarts Unsurprisingly, restarts do not synergize well with GB. They increase both the solving time and the number of synchronizations. This is because restarts cancel a lot of the expensive bookkeeping of GB, without exploiting the benefits of GB. In particular, there is no point in trying to preserve the trail as much as possible, if we are going to throw it away with a restart. This is also illustrated by the fact that the relative performance of GB compared to NCB is much worse with restarts than without.

Learning vs. cheapest chunk. The option BB can sometimes have a negative effect on the performance of the solver. This might be due to the fact that it introduces clauses that are not really relevant to the conflict, which can lead to worse behavior after conflict repair. Interestingly, BB does seem to behave better with the larger problems ran in Section 7, which suggests that it might be more effective when the cost of synchronizations is higher.

Cactus plot analysis. The cactus plots in Figure 4 show that easy problems are solved faster with (N)CB, while GB seems to scale better on harder problems. This is probably because GB has a higher overhead than (N)CB, but saves more propagations on harder problem instances.

The cactus plots in Figure 5 show that GB consistently executes fewer synchronizations than (N)CB.

In both experiments, the number of conflicts remained relatively similar between the different configurations, with a slight decrease for GB. This might be particular to the category of problems we are looking at, where a more local search is effective.

D Analysis on SMT conflicts

In addition to analysis on graph coloring benchmarks, we generated conflicts from SMT solver to evaluate how different backtracking mechanisms are performing in this domain. The option acronyms were defined in Section 7.

Experiment setup. We patched the SMT solver CVC5 [2] to export every conflict within its default propositional solver (MiniSAT [13]). For each conflict we logged all current clauses as well as the trail leading to the conflict. We collected over 2.9 million conflicts by executing CVC5 on all benchmarks in the QF_UF and QF_NIA suite of the SMT-LIB release 2024 [21] with a timeout of 15 seconds per instance.

To reconstruct the conflicts in NapSAT, we converted each logged conflict to an input in the DIMACS CNF format, added unit clauses for assumptions, and instructed the solver to take the same decisions as within CVC5.

	tiny	small	medium	big	huge
#Instances	67,274	411,097	339,571	1,847,692	243,486
Ratios #BT					
NCB > GB	46.81%	49.53%	52.20%	48.95%	58.38%
NCB = GB	44.14%	33.94%	24.39%	20.72%	16.83%
NCB < GB	7.70%	16.52%	23.41%	30.33%	24.79%
Avg. #BT					
NCB	4.53 ± 1.42	16.16 ± 7.85	42.1 ± 12.7	139.0 ± 57.8	406 ± 400
GB	3.35 ± 1.33	12.18 ± 6.45	32.6 ± 12.1	127.6 ± 56.6	300 ± 206
Avg. #Conflicts					
NCB	1.01 ± 0.09	1.07 ± 0.32	1.20 ± 0.51	1.56 ± 0.83	1.83 ± 1.06
GB	1.10 ± 0.34	1.35 ± 0.82	1.68 ± 1.21	2.18 ± 1.56	2.62 ± 2.96

Table 5 Statistics on backtracked literals (BT) and conflicts. Standard deviation is shown after the ± symbol.

Results. Compared to the previous experiments we need to adapt our measurement metric slightly and count the number of backtracked literals instead of synchronizations. This is because we cannot track subsequent literal synchronizations after a conflict as any measurement beyond the immediate conflict handling requires the SMT solver in the loop. Furthermore, we utilize a constant cost function in order to optimize for the number of backtracked literals.

We start each measurement at the logged conflict and measure until the next SAT solver decision is to be made. Note that a conflict analysis propagates literals which may lead to subsequent conflicts. Those are included in our measurements as this closer reflects the behavior in an SMT solver. To make average values more meaningful, we have clustered the conflicts by the sum s of total backtracked literals for both methods:

tiny	$(s \leq 10)$
small	$(10 < s \leq 50)$
medium	$(50 < s \leq 100)$
big	$(100 < s \leq 500)$
huge	$(500 < s)$

Table 5 compares GB with NCB on the average number of backtracked literals (literals of the reconstructed trail that were unassigned during a conflict analysis), the average number of conflicts (the logged conflict plus subsequent conflicts), as well as the percentage of instances where one approach surpasses the other.

Analysis. Performing GB on conflicts on SMT solver generated conflicts performs fewer literal unassignments than NCB in most cases. Subsequent conflicts are, however, more likely to appear with GB. Analyzing a single conflict, GB performs fewer or equal backtracking operations than NCB by design. However, as GB and NCB backtrack different literals, they might encounter different secondary conflicts. In particular, GB encounters more of them. Intuitively, this is quite natural. Since GB preserves more of the trail, it is more likely to trigger additional implications that can lead to further conflicts. An extreme example is when NCB backtracks all the way to the root level. It could be that the trail now only contains a single literal, making further conflicts without decisions very unlikely. In this case, GB learns more clauses.