Partial Backtracking in CDCL Solvers

Chuan Jiang and Ting Zhang

Iowa State University, Ames IA 50011, USA
{cjiang,tingz}@iastate.edu

Abstract. Backtracking is a basic technique of search-based satisfiability (SAT) solvers. In order to backtrack, a SAT solver uses conflict analysis to compute a backtracking level and discards all the variable assignments made between the conflicting level and the backtracking level. We observed that, due to the branching heuristics, the solver may repeat lots of previous decisions and propagations later. In this paper, we present a new backtracking strategy, which we refer to as partial backtracking. We implemented this strategy in our solver Nigma. Using this strategy, Nigma amends the variable assignments instead of discarding them completely so that it does not backtrack as many levels as the classic strategy. Our experiments show that Nigma solves 5% more instances than the version without partial backtracking.

Keywords: satisfiability, backtracking, conflict-driven conflict learning.

1 Introduction

Most modern SAT solvers are based on *conflict-driven clause learning* (CDCL). As a basic technique of CDCL solvers, *backtracking* helps the solver jump out of a local search space where no solution could ever be found [1]. In CDCL solvers, backtracking is non-chronological and guided by conflict analysis to determine how far the solver would jump back. The first non-chronological backtracking strategy was introduced in GRASP [1]. When GRASP meets a conflict, it keeps the current level and flips the value of the most recent decision variable. Backtracking only occurs if the flipping still leads to a conflict. Later, *random backtracking* was proposed to introduce randomness into selecting the backtracking level [2,3]. Essentially, the learnt clause is used for randomly deciding which variable is to be flipped. Nowadays, most solvers utilize a non-randomized backtracking strategy [4], which is referred to as *classic backtracking* in this paper. This strategy is more aggressive than that used in GRASP, since backtracking is always carried out after each conflict, making the resulting assignment trail always look like the one obtained when the learnt clause has already been included in the formula.

No matter what kind of backtracking a solver takes, it is observed that sometimes the solver backtracks quite far, which is almost equivalent to a restart. However, due to the wide adoption of VSIDS [4] and phase saving [5], the solver may make similar decisions as the ones before backtracking and hence repeat

K. McMillan, A. Middeldorp, and A. Voronkov (Eds.): LPAR-19, LNCS 8312, pp. 490–502, 2013.

some propagations. In this paper, we present a new backtracking strategy, referred to as *partial backtracking*. We implemented this strategy in our solver Nigma. Using this strategy, Nigma amends the variable assignments between the conflicting level and the assertion level instead of discarding them completely. Nigma still backtracks after each conflict, but it does not have to backtrack as many levels as those solvers using classic backtracking. Our experiments show that Nigma backtracks 10% $\sim$ 60% fewer levels than the version with classic backtracking.

This paper is organized as follows. Section 2 introduces the basic notions in SAT solving and CDCL solvers. Section 3 analyzes the classic backtracking strategy and the phenomenon of repeated propagation. Section 4 presents the implementation details of the partial backtracking strategy. Several optimizations on the implementation are discussed in Section 5. Section 6 presents the experiment results, showing the performance of our solver Nigma is improved after adopting the partial backtracking strategy. Section 7 concludes with some discussion on the future work.

2 Preliminaries

In this section, we introduce the basic notations and terminology on SAT solving and CDCL solvers.

A *literal* is either a Boolean variable x or its negation $\neg x$, and a *clause* is a disjunction of literals. A formula is said in *conjunction normal form* (CNF) if it is a conjunction of clauses. The *satisfiability* problem is to determine if there exists an assignment that evaluates a given Boolean formula to TRUE.

We say a variable or literal is *free* if it is unassigned and a clause is *unit* if it only contains one free literal and all other literals have been assigned FALSE. A unit clause essentially asserts that the sole free literal must be assigned TRUE. We call this assertion an *implication*, written as $l@dl$, indicating that the literal l is implied to be TRUE at the decision level dl (the definition of decision level is given below).

CDCL solvers check the satisfiability of Boolean formulas through *Boolean constraint propagation* (BCP) and conflict analysis. BCP is an iterative process of searching for unit clauses and obtaining implications until reaching a fixed point or encountering a conflict, that is, a clause whose literals are all assigned FALSE. We call the clause with all literals being assigned FALSE a *conflicting clause*. Most solvers store implications in the *implication queue* and propagate them one by one in FIFO manner. Algorithm 1 shows the propagation of an implication with two watched literals [4] and Algorithm 2 shows the iterative process of propagation.

If BCP terminates with a conflict, then the solver extracts the reason as a clause and adds it into the Boolean formula to avoid recurrence of the same conflict in the future. This process is called *conflict analysis* or *learning* and the new added clause is called a *learnt clause*. It is always desirable for a learnt clause to become unit after backtracking to some level.

Algorithm 1. $Propagate(l@dl)$

```
1: wl_1 ← ¬l
2: for all clause c where wl_1 is watched do
3:    Search for a non-FALSE unwatched literal l' in c
4:    if Exists l' then
5:       Unwatch wl_1
6:       Watch l'
7:    else
8:       wl_2 ← the other watched literal in c
9:       if wl_2 is FALSE then
10:         ImplicationQueue.Clear()
11:         ConflictAnalysis()
12:         return
13:      else if wl_2 is TRUE then
14:         continue
15:      else
16:         ImplicationQueue.Push(wl_2@dl_curr) {dl_curr is the current level}
17:      end if
18:   end if
19: end for
```

Algorithm 2. $BCP()$

```
1: while ImplicationQueue is not empty do
2:    l@dl ← ImplicationQueue.Pop()
3:    Propagate(l@dl)
4: end while
```

If BCP terminates without conflicts, then the solver selects a free variable and gives it a value heuristically. This variable assignment is referred to as a *decision* and pushed into a stack. A *decision level* is associated with each decision to denote the its depth in that stack.

We refer the readers to [6] for more information on SAT solving and CDCL solvers.

3 Classic Backtracking

In this section, we present the classic backtracking and identify the phenomenon of *repeated propagation*.

According to the classic backtracking, the solver resolves conflicts by backtracking to the *assertion level* dl_{asrt}, which is the second highest level among the literals in the learnt clause (we say a level dl_1 is higher than dl_2 if $dl_1 > dl_2$), and hence erasing all the variable assignments between dl_{asrt} and the *conflicting level* dl_{conf}, which is the level where the conflict occurs. After backtracking, the learnt clause becomes unit and the solver invokes BCP. This kind of backtracking unavoidably discards all the propagations between dl_{asrt} and dl_{conf}.

$\neg x_1 \vee x_2$
$\neg x_3 \vee \neg x_4$
$\neg x_1 \vee x_4 \vee x_5 \vee x_6$
$x_5 \vee x_{13}$
$\neg x_7 \vee x_8$
$\neg x_7 \vee x_9$
$\neg x_2 \vee \neg x_8 \vee x_{10}$
$\boxed{\neg x_8 \vee \neg x_9 \vee \neg x_{10}}$
$x_4 \vee x_7 \vee \neg x_{11}$
$x_7 \vee x_{11} \vee x_{12}$
$x_6 \vee x_{11}$

(a) Clauses

Variable	Activity Score	Last Value
x_1	10	TRUE
x_3	8.1	TRUE
x_2	7.2	TRUE
x_5	6.4	FALSE
x_{12}	6	FALSE
x_7	5.5	TRUE
x_6	3.7	FALSE
x_{13}	2.5	TRUE
x_{10}	2.2	TRUE
x_8	1.5	TRUE
x_4	0.5	FALSE
x_9	0	FALSE
x_{11}	0	FALSE

(b) Variables

Level	Assignments
1	x_1, x_2
2	x_3, $\neg x_4$
3	$\neg x_5$, x_6, x_{13}
4	$\neg x_{12}$
5	x_7, x_8, x_9, x_{10}

(c) Assignments

Fig. 1. The status before backtracking

Peter van der Tak et al. observed that CDCL solvers may reassign the same variables to the same Boolean values after a restart, and proposed the *partial restart* strategy [7]. One important reason of reassignments is the wide adoption of VSIDS [4] and phase saving [5]. We observed that backtracking exhibits a similar phenomenon, which we refer to as *repeated propagation* (note that a restart is a special form of backtracking). We give an example to illustrate this phenomenon.

Consider the clauses and variable assignments in Figure 1a and Figure 1b. Since the solver tends to select the most active free variables and their last values as decisions, we have the resulting assignment trail shown in Figure 1c. Then the solver encounters a conflict while propagating x_8 at the level 5 (the conflicting clause is framed in Figure 1a). The clause $\neg x_7 \vee \neg x_2$ is learnt by 1-UIP [8] and thus $dl_{asrt} = 1$. According to VSIDS, the solver will only increase the activity scores (assuming the increment is 1) of the variables involving in the conflict, namely, $\{x_2, x_7, x_8, x_9, x_{10}\}$. Therefore, the activity scores of the variables assigned between dl_{conf} and dl_{asrt}, $\{x_3, x_4, x_5, x_6, x_{12}, x_{13}\}$, remain the same. As shown in Figure 2c, in the decision immediately after backtracking to dl_{asrt}, x_3 will be chosen and assigned TRUE again at the level 2. Note that the resulting set of variable assignments at the level 2 is a superset of that before backtracking. The set of variable assignments at the level 3 is also similar to that before backtracking, except that x_6 has been "lifted" to the level 2.

$\neg x_1 \vee x_2$
$\neg x_3 \vee \neg x_4$
$\neg x_1 \vee x_4 \vee x_5 \vee x_6$
$x_5 \vee x_{13}$
$\neg x_7 \vee x_8$
$\neg x_7 \vee x_9$
$\neg x_2 \vee \neg x_8 \vee x_{10}$
$\neg x_8 \vee \neg x_9 \vee \neg x_{10}$
$x_4 \vee x_7 \vee \neg x_{11}$
$x_7 \vee x_{11} \vee x_{12}$
$x_6 \vee x_{11}$
$\neg \mathbf{x_7} \vee \neg \mathbf{x_2}$

(a) Clauses

Variable	Activity Score	Last Values
x_1	10	TRUE
x_2	**8.2**	TRUE
x_3	8.1	TRUE
x_7	**6.5**	TRUE
x_5	6.4	FALSE
x_{12}	6	FALSE
x_6	3.7	**TRUE**
x_{10}	**3.2**	TRUE
x_8	**2.5**	TRUE
x_{13}	2.5	TRUE
x_9	**1**	**TRUE**
x_4	0.5	FALSE
x_{11}	0	FALSE

(b) Variables

Level	Assignments
1	x_1, x_2, $\neg x_7$
2	x_3, $\neg x_4$, $\neg x_{11}$, x_{12}, x_6
3	$\neg x_5$, x_{13}
4	x_{10}
5	x_8, $\neg x_9$

(c) Assignments

Fig. 2. The status after backtracking

By comparing the variable assignments before and after each backtracking, we have Figure 3 that shows the percentage of discarded variable assignments that are chosen as decisions or propagated again before the next backtracking. It is interesting to see that the solver tends to either enter a totally different search space or stubbornly stick to its previous choices. But for a majority of backtrackings, a large proportion of discarded variable assignments are repeated. Note that we only consider those backtrackings that go back more than 10 levels and do not take account of restarts. Also, the variable assignments on the conflicting level are not counted in computing this percentage.

4 Partial Backtracking

In this section, we present the partial backtracking strategy that allows the solver to backtrack to some level dl_{back} such that $dl_{conf} > dl_{back} \geq dl_{asrt}$, therefore saving the propagations between dl_{back} and dl_{asrt}.

There are two reasons that classic backtracking prefers to use the assertion level as the backtracking level. First, after each backtracking, the learnt clause becomes unit and hence BCP can be invoked. Second, the succeeding BCP will not cause any consistency issue. To adopt the partial backtracking strategy, we need to update BCP procedure so that the two conditions are still met.

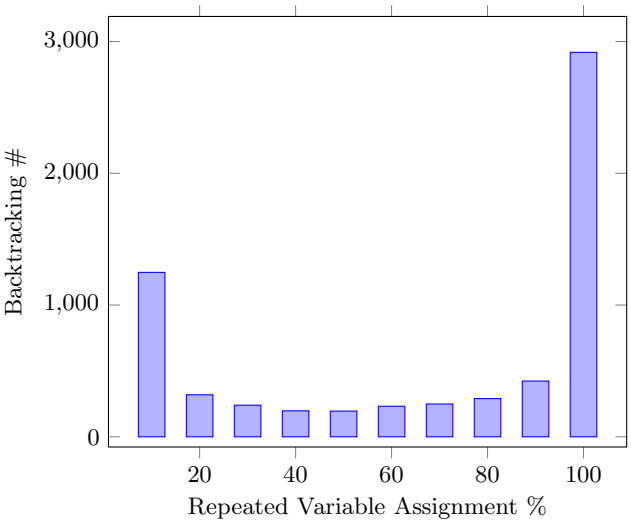


Fig. 3. Repeated variable assignment percentage while solving ACG-15-5p1.cnf from SAT Challenge 2012

The first condition can be easily satisfied by backtracking to any level lower than dl_{conf} but higher than or equal to dl_{asrt}. We note that the assertion level is the lowest level that the solver can backtrack to while keeping the learnt clause unit. The main complications come from maintaining the second condition. There are four kinds of issues BCP may encounter after backtracking to a level higher than dl_{asrt}. In Section 4.1, we will discuss these issues and give the corresponding solutions at clause level. The complete solution will be given in Section 4.2.

4.1 Complications and Solutions for Partial Backtracking

Unusual Implication. Classic backtracking guarantees that the solver always obtains implications at the current level dl_{curr}, that is, for any implication $l@dl$ in the implication queue, $dl = dl_{curr}$ (see Algorithm 1). However, this is not true for partial backtracking. A simple counterexample is the implication obtained from the learnt clause. This implication is at dl_{asrt}, which is lower than or equal to dl_{curr} after backtracking partially ($dl_{curr} = dl_{back} \geq dl_{asrt}$). Moreover, this implication may result in more implications, which can be scattered at any level between dl_{asrt} and dl_{curr}.

To the best of our knowledge, no existing solver exploits this guarantee in any essential way. In the implementation of Nigma, we simply relax this restriction.

Inappropriate Watched Literal. Generally, if a clause becomes unit and its sole free literal gets assigned according to this implication, its watched literals are

certainly assigned at the highest decision level among all its literals. This condition may be violated after backtracking partially.

Consider a clause $x_1 \vee \neg x_2 \vee x_3$. Suppose x_3 is assigned FALSE at the level 10, and x_1 and x_2 are free. So x_1 and $\neg x_2$ are watched for this clause. During BCP after backtracking partially, x_1 may be assigned FALSE at the level 6. In this case, it is inappropriate to still watch x_1. Since the level of x_3 is higher than the level of x_1, x_3 should be watched instead.

In order to solve this issue, we use the following procedure, where $\delta(l)$ is a function that returns the decision level where the literal l gets assigned.

- $AdjustWatchedLiteral(wl, c)$
  Pre-condition: The literal wl is watched in the clause c; All the unwatched literals in c are FALSE.
  Description: Search for an unwatched literal l in c such that $\delta(l) > \delta(wl)$ and for any unwatched literal l' in c, $\delta(l) \geq \delta(l')$. If successful, unwatch wl, watch l and return l. Otherwise, return wl.

Spurious Conflict. As we noted before, BCP may lead to conflicts. A standard conflict has the following implicit feature: the two FALSE literals with the highest levels in the conflicting clause are assigned at the same level. However, during BCP after backtracking partially, the solver might encounter a *spurious conflict* where these two literals are assigned at different levels.

We give a simple example to illustrate the spurious conflict. Consider a clause $x_1 \vee \neg x_2$. After backtracking partially, we may have two implications $\neg x_1@10$ and $x_2@15$ at the same time. This is a conflict (as all the literals are FALSE), but it is different from the standard one.

The spurious conflict cannot be resolved by the standard learning procedure. From another perspective, the spurious conflict essentially implies that the FALSE literal with the highest level should have been implied at the second highest level among the literals in the conflicting clause. In other words, without learning, we can immediately obtain an implication by simply backtracking to a level between the highest level and the second highest level in the conflicting clause. That level can also be but not necessary the second highest level because we are able to handle the unusual implication now. We have the following procedure to resolve spurious conflicts.

- $ResolveSpuriousConflict(c)$
  Pre-condition: All the literals in the clause c are FALSE; The literals wl_1 and wl_2 are watched in c; $\delta(wl_1) \neq \delta(wl_2)$.
  Description: If $\delta(wl_1) > \delta(wl_2)$, backtrack to the level $\delta(wl_1) - 1$ and push the implication $wl_1@\delta(wl_2)$ into the implication queue. If $\delta(wl_1) < \delta(wl_2)$, backtrack to the level $\delta(wl_2) - 1$ and push the implication $wl_2@\delta(wl_1)$ into the implication queue.

Wrong Decision Level. After backtracking partially, some assigned variables need to update their decision levels. For example, consider a clause $x_1 \vee x_2$.

Initially, x_1 is assigned TRUE at the level 18 and x_2 is free. Suppose at the level 20, a conflict is identified and the solver backtracks to the level 19 while $dl_{asrt} = 5$. Further suppose that the succeeding BCP induces the implication $\neg x_2@15$. As a result, the decision level of x_1 should be modified to 15. The issue can be solved by backtracking to the level 17 and get the implication $x_1@15$. The following procedure is used for this purpose.

– *ResolveWrongDecisionLevel*(c)
  Pre-condition: All the unwatched literals in the clause c are FALSE; c has a TRUE watched literal wl_{true} and a FALSE watched literal wl_{false}; $\delta(wl_{true}) > \delta(wl_{false})$.
  Description: Backtrack to the level $\delta(wl_{true}) - 1$ and push the implication $wl_{true}@\delta(wl_{false})$ into the implication queue.

Both processes of resolving spurious conflict and wrong decision level might trigger further backtracking. A helper procedure, *ClearInvalidImplications*, is defined to adjust the implication queue accordingly.

– *ClearInvalidImplications*()
  Description: Remove invalid implications from the implication queue. An implication $l@dl$ is *invalid* if $dl > dl_{curr}$.

In spite of the possible chained backtracking, whenever BCP terminates, the current decision level is always higher than or equal to the assertion level.

4.2 BCP after Partial Backtracking

As mentioned before, the standard BCP needs an adjustment if the solver takes a partial backtracking. Algorithm 3 shows the procedure *PropagateAmending* that is a special propagating procedure to be used after backtracking partially. Algorithm 4 shows the procedure *BCPAmending* that replaces the standard BCP procedure.

Let us revisit the example in Section 3. At this time, when the conflict occurs at the level 5, the solver takes a partial backtracking to the level 4 (see Figure 4a). While propagating the implication $\neg x_7@1$, the solver obtains $\neg x_{11}@2$ (unusual implication) (see Figure 4b) due to $x_4 \vee x_7 \vee \neg x_{11}$. In the next iteration of propagation, the solver identifies a spurious conflict ($x_7 \vee x_{11} \vee x_{12}$) and has to go back one level to resolve it (see Figure 4c). Due to the existence of $x_6 \vee x_{11}$, x_6 should have been implied at the level 2 (wrong decision level), so the solver goes back one level again (see Figure 4d). Then BCP terminates because no more implication or conflict can be found. It is clearly seen that the solver amends the existing assignment trail conservatively, not simply discarding a significant portion of it. We note that under this strategy, it is possible that the solver enters a search space which is quite different from the one resulting from the classic backtracking.

We shall point out that, when the implication to be propagated happens to be at the current level, the effect of *PropagateAmending* is exactly the same as

Algorithm 3. *PropagateAmending*($l@dl$)

```
1: wl_1 ← ¬l
2: for all clause c where wl_1 is watched do
3:    Search for a non-FALSE unwatched literal l' in c
4:    if Exists l' then
5:       Unwatch wl_1
6:       Watch l'
7:    else
8:       wl_1 ← AdjustWatchedLiteral(wl_1, c)
9:       wl_2 ← the other watched literal in c
10:      if wl_2 is FALSE then
11:         if δ(wl_1) > δ(wl_2) then
12:            wl_2 ← AdjustWatchedLiteral(wl_2, c)
13:         end if
14:         if δ(wl_1) == δ(wl_2) then
15:            Backtrack to δ(wl_1)
16:            ConflictAnalysis() {Standard conflict}
17:            ClearInvalidImplications()
18:            return
19:         else
20:            ResolveSpuriousConflict(c) {Spurious conflict}
21:            ClearInvalidImplications()
22:         end if
23:      else if wl_2 is TRUE then
24:         if δ(wl_2) > δ(wl_1) then
25:            ResolveWrongDecisionLevel(c) {Wrong decision level}
26:            ClearInvalidImplications()
27:         end if
28:      else
29:         ImplicationQueue.Push(wl_2@δ(wl_1))
30:      end if
31:   end if
32: end for
```

Algorithm 4. *BCPAmending*()

```
1: while ImplicationQueue is not empty do
2:    l@dl ← ImplicationQueue.pop()
3:    PropagateAmending(l@dl)
4: end while
```

Level	Assignments
1	x_1, x_2, $\neg x_7$
2	x_3, $\neg x_4$
3	$\neg x_5$, x_6, x_{13}
4	$\neg x_{12}$

(a)

Level	Assignments
1	x_1, x_2, $\neg x_7$
2	x_3, $\neg x_4$, $\neg x_{11}$
3	$\neg x_5$, x_6, x_{13}
4	$\neg x_{12}$

(b)

Level	Assignments
1	x_1, x_2, $\neg x_7$
2	x_3, $\neg x_4$, $\neg x_{11}$, x_{12}
3	$\neg x_5$, x_6, x_{13}
4	

(c)

Level	Assignments
1	x_1, x_2, $\neg x_7$
2	x_3, $\neg x_4$, $\neg x_{11}$, x_{12}, x_6
3	
4	

(d)

Fig. 4. The status after backtracking partially

Propagate. This indicates that *PropagateAmending* is essentially a generalization of *Propagate*.

5 Optimization

In this section, we discuss optimizations applicable to *PropagateAmending* and *BCPAmending*.

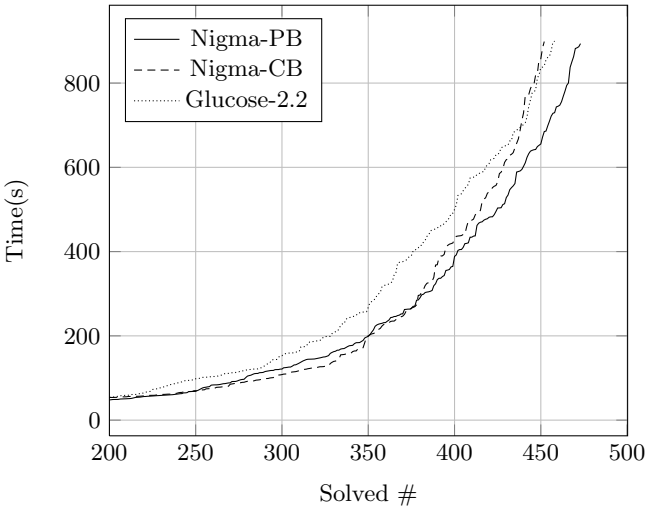
First, the implication queue can be constructed as a priority queue. As we described before, most CDCL solvers organize implications in a queue and propagates them in FIFO manner. However, since the implications in the queue can be scattered on different levels, unnecessary propagations can be avoided by giving higher priority to the implication at the lowest level in the queue. The intuition is that propagation may induce backtracking due to spurious conflict and wrong decision level, making some implications invalid and removed from the queue. For example, suppose that we have the implications x_1@10 and $\neg x_2$@20 in the implication queue. If propagating x_1@10 incurs a backtracking to some level lower than 20, $\neg x_2$@20 becomes invalid and the solver needs not propagate it.

Second, even if encountering a standard conflict in *PropagateAmending*, it is possible to postpone the conflict analysis. Suppose, while propagating x_1@10, the solver meets a standard conflict at the level 20. If the solver does not analyse the conflict immediately but continues propagating, it may backtrack to some level lower than 20 later due to spurious conflict or wrong decision level, making that conflict disappear automatically.

Third, it is unnecessary to call *PropagateAmending* in each iteration of *BCPAmending*. As mentioned before, *PropagateAmending* is a generalization of *Propagate* and it is more expensive than *Propagate*. If the implication to be propagated happens to be at the current level, calling *Propagate* directly instead of *PropagateAmending* will not cause any issue.

Solver	SAT	UNSAT	Solved #
Nigma-PB	222	251	473
Nigma-CB	212	240	452
Glucose-2.2	212	246	458

(a) The Number of Solved Instances



(b) Runtime Cactus Plot

Fig. 5. Experiment results of Nigma-PB, Nigma-CB and Glucose 2.2 on the benchmark suite from the application track of SAT Challenge 2012

Fourth, it is also unnecessary to backtrack partially every time a conflict occurs. The motivation of partial backtracking is to save propagations. Thus this strategy should be more efficient if a large number of propagations are going to be discarded or repeated. In Nigma, we measure the saving by the number of levels the solver would go back by classic backtracking, namely, $dl_{conf} - dl_{asrt}$. According to our experiments, when we set the triggering condition to $dl_{conf} - dl_{asrt} > 10$, around $5\% \sim 30\%$ of conflicts will trigger partial backtracking.

6 Experiment Results

In this section, we present experiment results using our solver Nigma, which is a CDCL solver based on MiniSat 2.2 [9]. The benchmark suite consists of the 600 instances from the application track of SAT Challenge 2012 [10]. We conducted experiments on a 3.40GHz $\times$ 8 Intel Core i7-2600K processor with 900 second timeout and 7GB memory limit per instance.

The versions of Nigma with partial backtracking and with classic backtracking are denoted by Nigma-PB and Nigma-CB, respectively. Nigma-PB is configured as follows: if $dl_{conf} - dl_{asrt} \leq 10$, the solver simply follows the classic backtracking

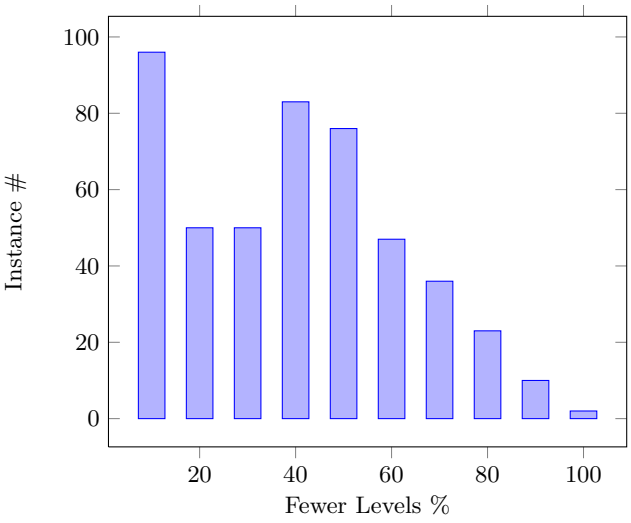


Fig. 6. Nigma backtracks fewer levels with partial backtracking

strategy; otherwise, the solver backtracks only one level, that is, it backtracks to the level $dl_{conf} - 1$. We use Glucose 2.2 [11] as an additional reference.

Figure 5a shows the number of instances solved by the three solvers and Figure 5b is the cactus plot of the results. It is clearly seen that when applying partial backtracking, Nigma-PB solved 21 more instances than Nigma-CB, and it also performs better than Glucose 2.2.

An in-depth view of the effect of partial backtracking is given in Figure 6, showing the percentage of fewer levels the solver backtracks for each solved instance. We note that, for a majority of instances, when the solver takes a partial backtracking, it backtracks 10% $\sim$ 60% fewer levels finally, compared with classic backtracking.

We also compare two additional metrics in the experiment, in order to explain the performance improvement by partial backtracking from a different perspective. The first metric is the number of decisions to solve an instance. Generally speaking, fewer decisions indicate the solver explores the search space in a better way [8]. According to the experiment, among the 439 instances solved by both Nigma-PB and Nigma-CB, 317 instances are solved by Nigma-PB with fewer decisions than by Nigma-CB.

The second metric is the number of decisions per conflict for a solved instance. We are interested in this metric because the power of CDCL solvers stems from identifying and learning from conflicts. The number of decisions per conflict reflects how frequently the solver identifies a conflict. The smaller this number is, the more often the solver detects and corrects its fault in making decisions. Partial backtracking has the potential to reduce this number as the solver might detect a standard conflict at a level higher than dl_{asrt} (see Line 14-18 in Algorithm 3) while retaining the ability to detect a standard conflict at

dl_{asrt}. The experiment result confirms our conjecture: 387 instances are solved by Nigma-PB with fewer decisions per conflict than by Nigma-CB.

7 Conclusions

In this paper, we presented the partial backtracking strategy which is essentially an extension of classic backtracking. This strategy amends the assignment trail instead of simply discarding a portion of it. As a result, some propagations need not to be repeated and the solver can go deeper in certain search space. Our experiments show that this new kind of backtracking improves the performance of CDCL solvers. Besides the optimizations mentioned in Section 5, we are investigating the following two aspects to further improve its efficiency.

First, in our current implementation, the solver backtracks to $dl_{asrt} - 1$ first. In fact, any level higher than dl_{asrt} can be used for the initial backtracking, as going back to that level still keeps the learnt clause unit. We are interested in designing a better heuristic to select the initial backtracking level.

Second, we would explore other criteria to trigger a partial backtracking. A promising candidate is the number of variable assignments the solver would discard by taking a classic backtracking.

References

1. Marques-Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
2. Lynce, I., Baptista, L., Marques-Silva, J.P.: Stochastic systematic search algorithms for satisfiability. Electronic Notes in Discrete Mathematics 9, 190–204 (2001)
3. Lynce, I., Marques-Silva, J.P.: Random backtracking in backtrack search algorithms for satisfiability. Discrete Applied Mathematics 155(12), 1604–1612 (2007)
4. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th Conference on Design Automation, New York, USA, pp. 530–535 (2001)
5. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
6. Marques-Silva, J.P., Lynce, I., Malik, S.: Conflict-driven clause learning sat solvers. In: Handbook of Satisfiability, pp. 131–154 (2009)
7. van der Tak, P., Ramos, A., Heule, M.: Reusing the assignment trail in cdcl solvers. Journal on Satisfiability, Boolean Modeling and Computation 7, 133–138 (2011)
8. Zhang, L., Madigan, C.F., Moskewicz, M.H., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, pp. 279–285. IEEE Press (2001)
9. Eén, N., Sörensson, N.: Minisat 2.2, `http://minisat.se/`
10. Balint, A., Belov, A., Järvisalo, M., Sinz, C.: Sat challenge (2012), `http://baldur.iti.kit.edu/SAT-Challenge-2012/index.html`
11. Audemard, G., Simon, L.: Glucose 2.2, `https://www.lri.fr/~simon/?page=glucose`