Abstract

1 Introduction

The vast majority of modern SAT solvers used in industrial applications are based on the conflict-driven clause learning (CDCL) algorithm. In a CDCL SAT solver, backtracking occurs after every conflict, where all literals from one or more decision levels become unassigned, before the solver resumes making decisions and performing unit propagations. It turns out that many of the literals that are unassigned during a backtrack will often be re-assigned again in roughly the same order when the solver continues. This results in many of the exact same propagation steps being repeated over and over again during the execution of the solver, and avoiding these repetitions could result in a significant speedup.

Chronological backtracking [4, 5, 7] addresses this inefficiency by heuristically backtracking fewer levels up the trail after learning a clause. However, chronological backtracking breaks some traditional invariants of the SAT solver, namely that literals are supposed to appear on the trail in monotonically increasing order by decision level, and can be complex to implement. Additionally, it does not allow the search to have as much flexibility in deciding the order of literals to assign, since leaving a large portion of the trail intact ”in a sense” forces decisions originally used to construct that portion. Some empirical evidence also suggests that it is only beneficial when used sparingly (e.g., when the number of levels to be backtracked over is greater than 100).

We propose an alternative method, ”trail saving”, where we cache the part of the trail that is unassigned during a backtrack and then attempt to restore implied parts of it as each new decision is made. Trail saving preserves the traditional invariants of the SAT solver and its basic version is very simple to implement. It also allows the search to choose the order of decisions, but helps make propagation faster. We explore the theoretical speedup that trail saving provides, develop some enhancements to make the idea more effective, and demonstrate experimentally that it speeds up state-of-the-art SAT solvers by a non-trivial factor.

2 Background

We will assume the reader is familiar with SAT solving and the basics of the CDCL algorithm [8]. We will use the notion of a trail to represent a sequence of literals corresponding to the current partial assignment. The literals appear on the trail in the order in which they were assigned. Associated with every literal on the trail is a decision level. Decision level 0 represents all literals which are implied by the formula itself and subsequent decisions start the beginning of a new decision level. When a literal is made true by unit propagation, the literal is given its decision level and a reason clause, where the reason clause is the clause that was made unit by propagation in order to imply this literal. Most modern solvers use the two-literal watch scheme to make unit propagation more efficient, where each clause is watched by two non-false literals. When a watcher for a clause becomes false, a replacement is searched for in the clause. If no such replacement exists

3 Trail Saving

It can be observed that a SAT solver often repeats the same propagations over and over again during execution. Although no two descents down the trail will ever be identical because of clause learning, the solver will often propagate many of the same literals with the same reasons in a similar order to what it had done previously, especially since the order of the decisions usually remains similar. We plan to cache the part of the trail that we are backtracking over and use it to restore some of those literals without propagating them from scratch.

Algorithm 1: propagate()

```
1:  **while** queue not empty **do**
2:    $l :=$ next literal on queue
3:  BEGIN CODE TO ADD FOR TRAIL SAVING
4:    **if** trail saving turned on **then**
5:      **if** val(top of trail cache) $=$ true **then**
6:        **while** trail cache not empty **do**
7:          move top of trail cache to next literal
8:          let tc $=$ top of trail cache
9:          **if** reason(tc) is null **then**
10:           break
11:         **else if** val(tc) $=$ false **then**
12:           return conflict with reason(tc)
13:         **else if** val(tc) $=$ unassigned **then**
14:           enqueue tc with reason reason(tc)
15: END CODE TO ADD FOR TRAIL SAVING
16:   **for** each clause $c \in$ watchlist($\neg l$) **do**
17:     **for** each literal $x \in c$ **do**
18:       **if** $val$(x) $\neq$ false and x is not already a watcher of c **then**
19:         let $x$ be a watcher of $c$ and add $c$ to watchlist($\neg x$)
20:         break
21:     **if** no new watcher was found **then**
22:       **if** other watcher $x_2$ of $c =$ false **then**
23:         return c as conflict
24:       **else if** other watcher $x_2$ of $c$ is unassigned **then**
25:         enqueue $x_2$
```

3.1 Storing the trail cache

The first step that is needed to use ”trail saving” is to store the literals which are being unassigned during a backtrack, along with their respective reason clauses, in a ”trail cache”. Decisions should be marked with a null reason. The literals should appear in the same order on the trail cache as they had appeared on the trail. We will adopt the notation that the ”top” of the trail cache is initially set to be the literal that appeared the earliest on the trail and will get updated as the trail cache gets used up. Note that initially the top of the trail cache is always a decison (i.e., the start of a new decision level). As the literals from the trail cache get restored (i.e., assigned), the top of the trail cache points to the next literal not yet restored.

3.2 Restoring the trail cache

We will base trail saving on a convenient observation. If the top of the trail cache becomes true at any point, every literal after the top of the trail cache up until the next decision literal (i.e., the next literal with a null reason) is implied by the current trail. The top of the trail cache can become true either by decision or unit propagation, and this observation still holds.

Proof Let L_0 be the set of literals on the trail after backtracking, L_1 be the set of literals on the current trail, and $L_c[i]$ denote the set of the first i literals on the trail cache. The ith literal on the trail cache is implied by $L_0 \cup L_c[i-1]$, as long as the ith literal was not a decision. $L_0 \subseteq L_1$, since L_1 starts out as L_0 and adds non-contradictory literals (by decision or unit propagation). Therefore, the ith literal on the trail cache is implied by $L_1 \cup L_c[i-1]$. Assume the top of the trail cache is true. Then, from above, every literal up until the next decision would be implied. ■

Once the backtrack has been completed and the solver resumes with decisions and propagations, we will attempt to restore parts of the trail cache automatically during propagation to make it faster. When a literal is to be propagated, we first check the top of the trail cache to determine its truth value. If that literal's truth value is true, we assign all of the literals that come after it on the trail cache to be true up until (and excluding) the next the next literal with a null reason. If at any time the top of the trail cache is a literal with a non-null reason whose truth value is already false, we determine a conflict has occurred with the conflict clause being that literal's reason. At the beginning of a new backtrack, the trail cache is cleared and a new trail cache is generated.

A potentially quadratic speedup can be realized during the propagation of these literals since they are being propagated as a set, rather than one at a time. We will illustrate this with an example.

Example 1 Suppose the literals $x_1, ...x_n$ are on the trail cache, with none of them having a null reason, and x_1 becomes true (e.g., the solver makes a decision for x_1 to be true). Then, $x_2, ...x_n$ are implied by the current trail. Now we will examine the propagation of a clause c which consists of the literals $l, \neg x_1, \neg x_2, ..., \neg x_n$ with and without trail saving. Assume l and $\neg x_1$ are the current watched literals.

Without trail saving, $\neg x_1$ becomes false and a new watcher is searched for and found 1 literal later, $\neg x_2$, and then $\neg x_1$ and $\neg x_2$ are swapped. Next $\neg x_2$ becomes false, a new watcher is searched for and found 2 literals later, $\neg x_3$, and then $\neg x_2$ and $\neg x_3$ are swapped. This process will continue until all x_n literals are false, c becomes unit, and l becomes forced to be true. This propagation took $1 + 2 + ... + n$ searches through clause c, for a total of $O(n^2)$ searches.

With trail saving, $x2, ...x_n$ are all assigned to be true as soon as x_1 is made true, so immediately the values of literals $\neg x_1, ..., \neg x_n$ in c are false. During propagation, a new watcher is searched for to replace $\neg x_2$, n literals are saerched over without finding any, thus c is detected as unit and l becomes forced to be true. This propagation took $O(n)$ searches through clause c. ■

Another potential speed up occurs when trail saving detects conflicts early without performing any of the pending propagations. This happens when the top of the trail cache is a literal with a non-null reason and the truth value of this literal is already false, and we can use that literal's reason clause as the conflict clause.

3.3 Enhancements

The first enhancement to trail saving is to keep the trail cache intact instead of clearing it before each backtrack, and to add the literals that will be backtracked over to the already existing trail cache. In order to preserve soundness, we must add the literals from the current backtrack to the beginning of the trail cache, such that the trail cache has as its prefix the literals which are being backtracked over in the same order they appear on the current trail, and as its suffix the previous trail cache. In order for the suffix to be of any use, one must actually not store the last level (i.e. the conflicting level) to the trail cache at all, since the learned clause will prevent the conflicting level from being duplicated in the future. Because the trail cache can now grow indefinitely, occasionally it needs to be trimmed by removing duplicate literals. We decided to trim the trail cache whenever its size exceeded the total number of variables.

The next enhancement to trail saving is to scan the trail cache to check if there is a literal that is already assigned false within some limit. The limit we found worked best is to scan the trail until the third null reason is found (i.e. "looking ahead 2 decisions"). If a literal that is already assigned false is found, we know that we are guaranteed to have a conflict in the near future, and we force the SAT solver's subsequent decisions to come from the trail cache instead of using the regular variable selection heuristic (i.e. EVSIDS).

4 Related Work

4.1 Trail Saving for Assumption-Based SAT

Trail saving was briefly introduced in the context of assumption-based SAT [2] in order to speed up the propagation of the assumptions each time the solver backtracked past the assumption levels. Note that the process is the same, but it only works to restore the propagants and implicants of the assumptions and only after a learnt unit clause. Becuase the number of propagants and implicants of the assumptions tends to become dominated by the number of assumptions in the applications it was tested on, the technique did not yield a significant improvement. It would be interesting to combine the extended version of trail saving outlined in this paper with trail saving over the assumptions to see if that could produce better results.

4.2 Comparison to Chronological Backtracking

As mentioned earlier, chronological backtracking also achieves some of the theoretical speedups that trail saving achieves. One advantage of chronological backtracking is that it does not require any caching of the trail or accessing such a cache during propagation; it simply leaves a vast portion of the trail intact rather than backtracking all the way to the conflict level. Chronological backtracking will also speed up propagations slightly more than trail saving, because it leaves more literals assigned at once which leads to clauses being detected as unit more quickly.

One disadvantage that chronological backtracking has compared to trail saving is that it doesn't allow the search as much flexibility to jump out of local neighbourhoods, as the common variable selection heuristics tend to depend on [6, 1, 3]. Because chronological backtracking leaves a large portion of the trail assigned, it will behave similarly to how the solver would behave if it was to force the decisions that do occur within this large portion of literals, whereas trail saving does not force any decisions. As variable selection heuristics become more refined, it is likely that adhering to them will become more crucial to performance.

Another disadvantage is that, because propagation is not performed at every decision level, literals that remain assigned during chronological backtracking (but otherwise would have been unassigned) do not have a chance to move their decision levels up. An example of this is when the solver was supposed to backtrack over a literal that would end up being unit propagated at level 0, but stays assigned at its current decision level when chronological backtracking is used. This could potentially effect the lbd score or length of the learnt clause that gets generated at the next conflict. Additionally, chronological backtracking breaks the invarant that all literals appear on the trail in monotonically increasing order by decision level, which makes its implementation slightly more complicated.

5 Experiments and Results

We implemented our techniques in two different SAT solvers, MapleSAT and Cadical, both of which have finished at or near the top of SAT competitions for the past several years. Each of these solvers was run as is without any trail saving or chronological backtracking in one version, with chronological backtracking in another version, and with the various types of trail saving we described in the remaining versions.

We then ran each solver on the 400 benchmarks used in the 2018 SAT Competition and the 400 benchmarks used in the SAT 2019 Competition. The experiments were run

	cadical	cadical_chrono	cadical_trail	cadical_trail_enhanced
Instances Solved	527	515	520	524
Par-2 (x 10^6s)	3.220	3.269	3.248	3.216
	maple	maple_chrono	maple_trail	maple_trail_enhanced
Instance Solved	458	470	472	473
Par-2 (x 10^6s)	3.797	3.690	3.694	3.667

Figure 1: Table of results

on 2.7 GHz Intel cores with each process given 7 GB of memory and 5000 seconds of CPU time. We chose not to output or verify the proofs generated by any of the solvers. The results are reported in Figure 1.

When comparing regular cadical with non-chronological backtracking against cadical with trail saving, the average time taken to solve those instances that were solved by both solvers decreased and the number of propagations per second increased. Although the total number of solved instances decreased for both versions of trail saving, the Par-2 score increased for both of them. The trail saving versions also solved more instances and had better Par-2 scores than cadical with chronological backtracking (default parameters).

When comparing regular MapleSAT with non-chronological backtracking against either of the MapleSAT trail saving versions, the total number of instances solved increased significantly, the average time taken to solve those instances which were solved by both solvers decreased, and the number of propagations per second increased. When comparing the trail saving versions to MapleSAT with chronological backtracking (default parameters), the trail saving versions perform slightly better in both total instances solved and PAR-2 scores.

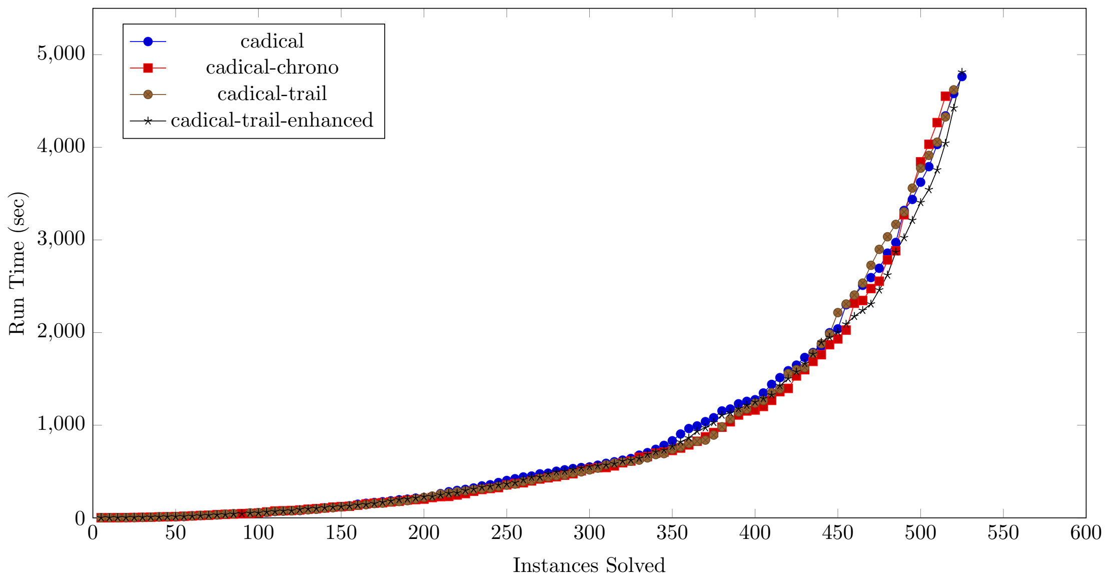

Figure 2: Comparison of run times for versions of cadical. cadical is without chronological backtracking, cadical-chrono is with chronological backtracking, cadical-trail is with the first version of trail saving, cadical-trail-2levels-ahead is the trail saving with all enhancements added.

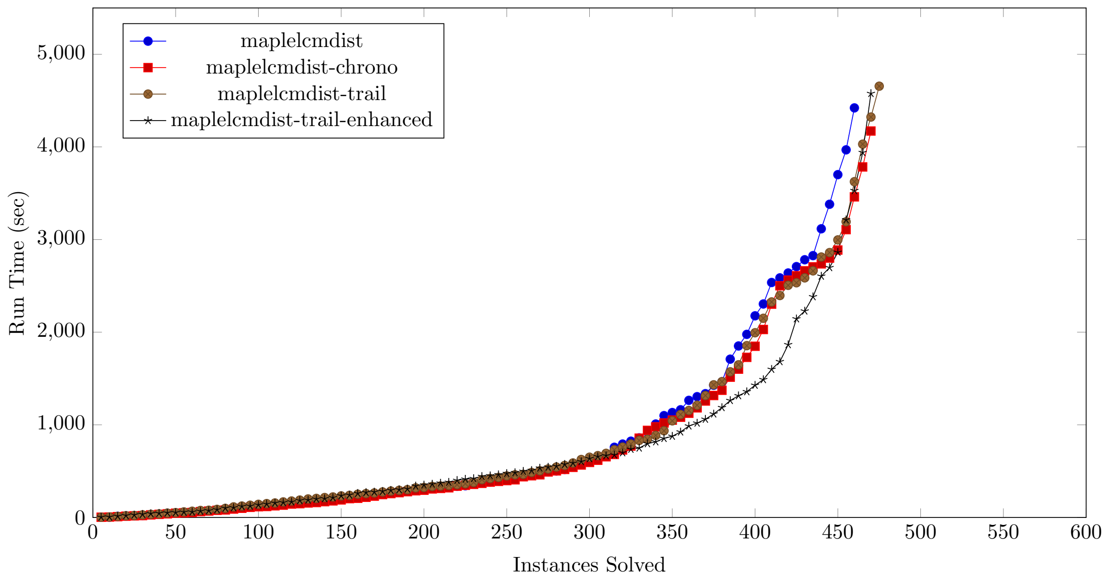

Figure 3: Comparison of run times for versions of MapleSAT. maplelcmdist is from the 2017 SAT Competition, maplelecmdist-chrono is with chronological backtracking (2018 SAT Competition), maplelcmdist-trail is with the first version of trail saving, maplelcmdist-trail-enhanced is with trail saving and all enhancements added.

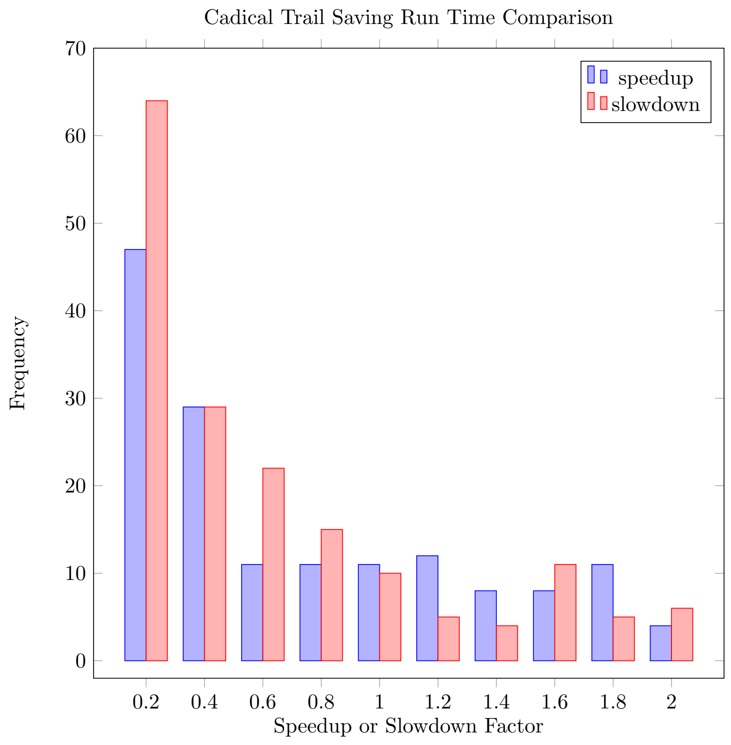
Cadical Trail Saving Run Time Comparison
Frequency
Speedup or Slowdown Factor
speedup
slowdown
0
10
20
30
40
50
60
70
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2

Figure 4:

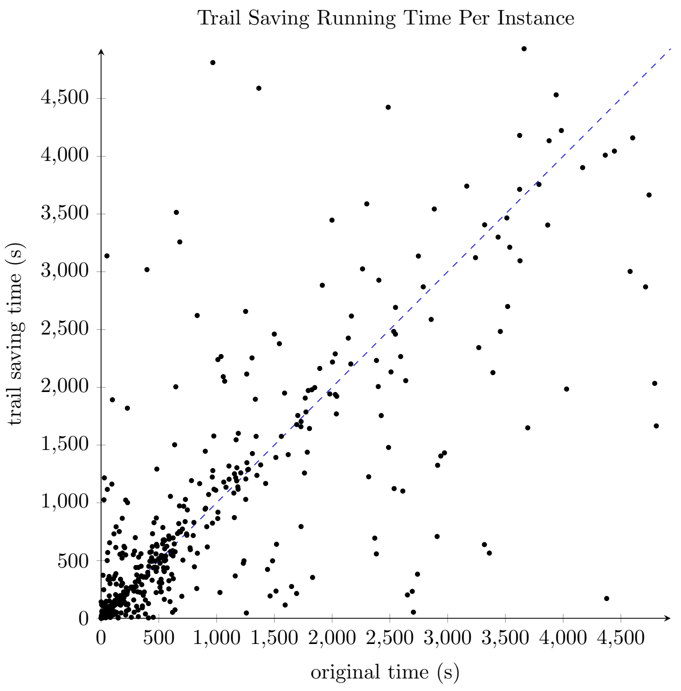
Trail Saving Running Time Per Instance
trail saving time (s)
original time (s)
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500

Figure 5:

6 Conclusion

References

[1] Heule, M., Weaver, S.A. (eds.): Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings, Lecture Notes in Computer Science, vol. 9340. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4, `https://doi.org/10.1007/978-3-319-24318-4`

[2] Hickey, R., Bacchus, F.: Speeding up assumption-based SAT. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 164–182. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_11, `https://doi.org/10.1007/978-3-030-24258-9\_11`

[3] Liang, J.H., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 123–140. Springer (2016). https://doi.org/10.1007/978-3-319-40970-2_9, `https://doi.org/10.1007/978-3-319-40970-2\_9`

[4] McMillan, K.L., Middeldorp, A., Voronkov, A. (eds.): Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings, Lecture Notes in Computer Science, vol. 8312. Springer (2013). https://doi.org/10.1007/978-3-642-45221-5, `https://doi.org/10.1007/978-3-642-45221-5`

[5] Möhle, S., Biere, A.: Backing backtracking. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 250–266. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_18, `https://doi.org/10.1007/978-3-030-24258-9\_18`

[6] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001. pp. 530–535. ACM (2001). https://doi.org/10.1145/378239.379017, `https://doi.org/10.1145/378239.379017`

[7] Nadel, A., Ryvchin, V.: Chronological backtracking. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10929, pp. 111–121.

Springer (2018). https://doi.org/10.1007/978-3-319-94144-8_7, `https://doi.org/10.1007/978-3-319-94144-8\_7`

[8] Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153. IOS Press (2009), `https://doi.org/10.3233/978-1-58603-929-5-131`