Trail Saving on Backtrack

Randy Hickey and Fahiem Bacchus

Department of Computer Science, University of Toronto, Canada, rgh000@gmail.com, fbacchus@cs.toronto.edu

Abstract.

1 Introduction

Fix later—also [10] can't be first use of trail savings?

The vast majority of modern SAT solvers that are used to solve real-world problems 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. Traditionally, CDCL solvers would backtrack non-chronologically to the conflict level, which is the second highest decision level remaining in the conflict clause after conflict analysis has resolved away all but one literal from the current decision level [7]. Recently, however, it has been shown that backtracking chronologically (i.e. backtracking one level only after conflict analysis) [5,8] can be effective on many instances and was implemented in solvers that won the last two SAT competitions. Although chronological backtracking breaks some of the conventional invariants of CDCL solvers, it has been formalized and proven correct [6].

There are at least two possible side-effects of chronological backtracking which may explain its effectiveness. It has been observed that when a solver backtracks non-chronologically after a conflict, many of the literals that are unassigned during the backtrack will be re-assigned again in roughly the same order when the solver continues. By using chronological backtracking, the solver keeps a lot more of its partial assignment intact and saves from having to reconstruct a lot of the trail via propagation that it otherwise would have. Saving from having to reconstruct portions of the trail after backtracking was first introduced in the context of restarts [10]. The second side-effect of chronological backtracking is that it forces the solver to stay in the same local neighbourhood during search, since it unassigns less of the partial assignment and one could argue that this might lead to finding additional conflicts more quickly or reaching a satisfying assignment (if one exists) more quickly. However, this second side-effect also leads to less search flexibility, since the solver does not jump out of local neighbourhoods as often and does not obey the variable selection heuristic. We will argue that it is the first side-effect that leads to the effectiveness of chronological backtracking and not the second by developing a technique which also saves from having to repeat propagations, but allows the search to remain flexible and obey the variable selection heuristic.

Our alternative method is "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 performs similarly well as chronological backtracking without suffering from some of the aforementioned drawbacks.

2 Background

elaborate backtrack after learning a clause. Also describe properties of the trail (a) reason valid every reason clause is made unit by its prefix, (b) non-contradictory, (c) non-conflicting (no clause is falsified), and (d) propagation complete

We will assume the reader is familiar with SAT solving and the basics of the CDCL algorithm [9]. 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 and the other watcher is true or unassigned, then the clause is unit. If no such replacement exists and the other watcher is also false, then the clause is a conflict and needs to be analyzed to produce a learnt clause.

3 Trail Saving

1. Use simple backtrack and save to motivate and state the invariant.
2. Introduce notation for concatenating two trails.
3. Name and state the invariant in text (e.g. a defn)—concenating the cached trail with the current trail yields a trail that is always reason valid, but not necessarily non-contradictory, non-conflicting, nor propagation complete).
4. Proof (simple) that the invariant holds for caching on backtrack.
5. On backtrack at least the asserted literal and any new propagants at the backtrack level are added.
6. Prove that the invariant holds (???do we need an invariant or simply a property of the augmented trail) for trails with extras inbeween. $T_{old} + T_{newlyAdded} + T_{saved}$ also satisfies the invariant
7. re-acheving propagation completeness for the augmented trail (a) only add one level of the cached trail at a time, and do unit prop after (b) only add non-contradicted decision from the old trail. Also note that conflicts are valid and can be learnt from even if the augmented trail is not propagation complete.

8. Finally figure out a way to formalize the extra technique of storing on top after repeated backtracking.
9. Explain how freedom for the search is maintained by makeing decisions then consulting the trail.
10.

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.

3.1 Storing the trail cache

The first step that is needed to use "trail saving" is to store all of 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 keep a pointer, which we will refer to as the "top" of the trail cache, and initially set it to be the literal which had appeared the earliest on the trail. Note that initially the top of the trail cache always has a null reason (i.e., it was the start of a new decision level on the previous trail). As the literals from the trail cache get restored (i.e., assigned), the top of the trail cache will be updated to the next literal not yet restored.

3.2 Restoring the trail cache

We will base trail saving on the following critical invariant. If the top of the trail cache becomes true at any point, even after an arbitrary number of decisions and propagations have been made since the last backtrack, the set of literals on the trail cache after the top and up to but excluding the next literal with a null reason is still implied by the current trail. The top of the trail cache can become true either by decision or unit propagation, and this invariant still holds.

Proof 1 We will refer to the set of literals that become unassigned during the backtrack after conflict analysis has taken place as L_u, and will refer to the rest of the trail that remains assigned as L_0. We will refer to all of the literals that get placed on the trail after L_0 during subsequent propagations and decisions as L_1. We will refer to the current trail $L_0 \cup L_1$ as L_T. The trail cache will contain the literals in L_u ordered by their appearance on the old trail. Let $L_c[i]$ denote the set of the first i literals on the trail cache. Let $L_c[top]$ be the top of the old trail, which gets updated as parts of the trail cache get restored.

The ith literal on the trail cache is implied by $L_0 \cup L_c[i-1]$, as long as the ith literal has a non-null reason, since $L_0 \cup L_c[i-1]$ had forced $L_c[i]$ on the previous trail.

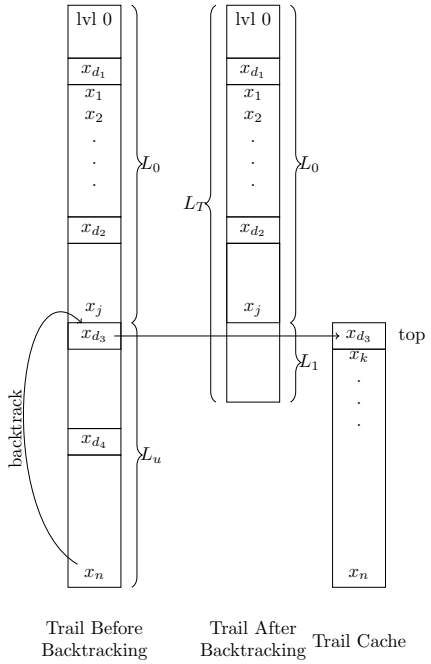

Fig. 1. Storing into a trail cache after backtracking.

$L_0 \subseteq L_T$, since $L_T = L_0 \cup L_1$. Therefore, the ith literal on the trail cache is also implied by $L_T \cup L_c[i-1]$.

Assume $L_c[top]$ is true, which means that $L_c[0], ..., L_C[top] \subseteq L_1 \subseteq L_T$. Then, from above, every literal on the trail cache up to but excluding the next literal with a nun-null reason 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 by using the invariant stated above. Each time the propagation routine is about to propagate a new literal, we first check the top of the trail cache to determine its truth value. If that literal's truth value is already true, we assign all of the literals that come after it on the trail cache to be true up until (and excluding) the next literal with a null reason.

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, as argued in [3,1]. Restoring literals from the trail cache automatically not only leads to quicker propagation of future literals, but also means that the propagation routine has less work to do as it searches for new watchers in clauses on watchlists. All of the reasons attached to these literals that are automatically restored will now be satisfied, thus the propagation routine can skip over them rather than having to access each of them potentially multiple times to find they are unit.

If at any time the top of the trail cache is a literal l with a non-null reason whose truth value is already false, we determine a conflict has occurred with the conflict clause being l's reason from the trail cache. This is because we know from Proof 1 that l is implied to be true at this point by the reason on the trail cache (i.e., the reason on the trail cache is unit), but $\neg l$ was also previously assigned to be true, therefore we have a conflict. This leads to a third potential speed up, since a conflict has been detected and none of the pending propagations have to be completed. Since trail saving can sometimes save hundreds or thousands of literals at a time, the number of propagations required to detect a conflict could be very costly.

1 Algorithm: propagate()

```
1:  while queue not empty do
2:
3:    restoreTrailCache()
4:
5:    l := next literal on queue
6:    for each clause c ∈ watchlist(¬l) do
7:      for each literal x ∈ c do
8:        if val(x) ≠ false and x is not already a watcher of c then
9:          let x be a watcher of c and add c to watchlist(¬x)
10:         break
11:     if no new watcher was found then
12:       if other watcher x_2 of c = false then
13:         return c as conflict
14:       else if other watcher x_2 of c is unassigned then
15:         enqueue x_2
```

1 Algorithm: restoreTrailCache()

```
1:  if trail saving turned on then
2:    let tc = top of trail cache
3:    if val(top of trail cache) = true then
4:      while trail cache not empty do
5:        move top of trail cache (tc) to next literal
6:        if reason(tc) is null then
7:          break
8:        else if val(tc) = false then
9:          return conflict with reason(tc)
10:       else if val(tc) = unassigned then
11:         enqueue tc with reason reason(tc)
```

3.3 Enhancements

In the standard version of trail saving outlined above, the trail cache needs to be deleted after each backtrack in order to make room for the new backtracked literals to be cached. 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 beginning of 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, and as its suffix the previous trail cache.

Proof 2 We will use all of the same set names as Proof 1 and also refer to the set of literals on our previous trail cache as L_{pc}. We concatenate the literals in L_{pc} to the end of L_c to get L_{sc}. We know that L_{pc} is a valid trail cache for $L_0 \cup L_u$. If we reach the end of L_c without any conflicts, we know that $L_u \subseteq L_T$. Since $L_0 \subseteq L_T$, then $L_u \cup L_0 \subseteq L_T$ and therefore L_{pc} is a valid trail cache for L_T. ■

In order for the previous trail cache to be of any use, one must actually never store the last level of the current trail to the trail cache at all, since a learnt clause prevents the last level from being duplicated in the future. If any part of the previous trail cache was used to construct this last level, it must also be discarded from the previous trail cache before the previous trail cache is concatenated to the end.

When concatenating trail caches together as above, it is important to note that the running trail cache can grow indefinitely. The trail cache can be trimmed periodically in the following way. Traverse down the trail cache and whenever a literal appears more than once, delete its entry in the trail cache. Whenever a literal x appears whose complement $\neg x$ has already appeared on the trail cache, x should be deleted and everything remaining until the end of the trail cache should be deleted (in order to keep the trail cache free of conflicts). We decided to trim the trail cache whenever its size exceeded twice the total number of variables in the SAT instance, as that is what worked best in preliminary experiments.

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.

Sometimes using trail saving to automatically restore literals with their previous reasons will lead to a literal getting a reason that is "worse" (in terms of number of literals, but one could use a different metric) than the reason it would have ended up with under normal propagation (i.e. without trail saving). The last enhancement we made to trail saving was to stop restoring literals from the old trail as soon as we hit a reason that we suspected might be "bad". For this, we kept a running mean and standard deviation of the length of all reasons used in the solver so far, and decided to stop restoring from the trail whenever we hit a reason that was greater in length than two standard

deviations above the mean length. This way, when normal propagation resumes, we give it a chance to find a better reason than the one that trail saving had stored from the last descent down the trail.

Another potential enhancement one could use is to use the trail cache to try to learn a second clause automatically when the solver reaches a conflict. However, we did not find a way to implement this to make it beneficial and the number of levels we have to scan for a second conflict would have to be more than the number of levels we already look ahead on each decision (when the solver is not under conflict).

4 Related Work

4.1 Trail Saving for Assumption-Based SAT

Trail saving was briefly introduced in the context of assumption-based SAT [3] 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 [7,2,4]. 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 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.

	Total Solved	SAT	UNSAT	Par-2 (x 10^6 s)
cadical	527	315	212	3.220
cadical_chrono	515	305	210	3.269
cadical_trail	520	307	213	3.248
cadical_trail_enhanced	524	315	209	3.216

Fig. 2. Table of results for cadical or "chrono", version published in [6].

	Total Solved	SAT	UNSAT	Par-2 (x 10^6 s)
cadical	492	289	203	3.5020
cadical_chrono	498	292	206	3.4400
cadical_trail	493	290	203	3.5228
cadical_trail_enhanced	496	292	204	3.4847

Fig. 3. Table of results for cadical, version pulled from github as of July 30, 2020.

	Total Solved	SAT	UNSAT	Par-2 (x 10^6 s)
maple	458	259	199	3.797
maple_chrono	470	271	199	3.690
maple_trail	472	271	201	3.694
maple_enhanced	473	277	195	3.667

Fig. 4. Table of results for MapleLCMDist.

6 Future Work

With more clever data structures the efficiency of restoring the trail could be improved. There are a number of parameters which can be tuned, including how often to refresh the trail, how large the cutoff point for rejecting "long" reasons should be, how long to wait to prune the trail cache if one is appending multiple trail caches together, how far to look ahead on the trail cache for conflicts (in terms of number of decisions or total number of literals). We only experimented with a few different settings of these parameters that made sense, but more careful fine-tuning of these parameters could further enhance the effectiveness of the technique. There are potentially other techniques one could use a trail cache for, including inprocessing steps that involve probing. A look down the trail cache is an incomplete but very fast way to do a probing step, as opposed to propagation. It is also possible that trail saving and chronological backtracking could be combined in some way to produce even better results.

References

1. Gent, I.P.: Optimal implementation of watched literals and more general techniques. J. Artif. Intell. Res. **48**, 231–251 (2013), https://doi.org/10.1613/jair.4016
2. 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
3. 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
4. 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
5. 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
6. 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
7. 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
8. 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
9. 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
10. van der Tak, P., Ramos, A., Heule, M.: Reusing the assignment trail in CDCL solvers. JSAT **7**(4), 133–138 (2011), https://satassociation.org/jsat/index.php/jsat/article/view/89