To Link or not to Link? That is a Watch List!

Robin Coutelier
TU Wien, Vienna, Austria

Abstract

The two-watched literal scheme is a powerful method used in state-of-the-art SAT solvers to reduce the number of clauses checked during Boolean constraint propagation. In this paper, we explore the representation of watch lists using a linked list data structure. We explain why this representation intuitively has advantages over the traditional array-based representation. We then empirically evaluate the performance of this representation and explain why it is not used in practice when combined with the blocker optimization. Both paradigms were implemented in the NapSAT SAT solver. Based on the implementation process, we detail difficulties raised by a linked list representation. Experimenting with this alternative representation shows insights into cache behaviors. However, we conclude that it should not be used for general-purpose SAT solvers.

2012 ACM Subject Classification Automated Reasoning, Constraint and logic programming

Keywords and phrases Watcher Lists, SAT, CDCL

Digital Object Identifier 10.4230/LIPIcs.POS.2024.1

Supplementary Material *NapSAT (Source Code GitHub)*: `https://github.com/RobCoutel/NapSAT`

Acknowledgements We thank the reviewers for their valuable feedback. As a reviewer pointed out, a similar approach was presented in [1] and [7]. This work is therefore not novel, but rather an independent re-discovery of an existing method. The author acknowledges support from the ERC Consolidator Grant ARTIST 101002685; the TU Wien Doctoral College TrustACPS; the FWF SpyCoDe SFB projects F8504; the WWTF Grant ForSmart 10.47379/ICT22007. NapSAT is a project started at the University of Liège under the supervision of Prof. Pascal Fontaine and continued at the TU Wien under the supervision of Prof. Laura Kovács. We thank the latter for proofreading the paper.

1 Introduction

Modern SAT solvers are based on Conflict Driven Clause Learning (CDCL) [9]. During a typical execution of the CDCL algorithm, most of the computation time is spent on Boolean constraint propagation (BCP). BCP searches for literals that can be implied by the current assignment and a clause via unit propagation. There may be a large number of clauses to inspect during the propagation of a literal ℓ. To reduce this number, the *two watched literals scheme* was introduced [7]. As the name suggests, the idea is to watch each clause with two literals ℓ_1, ℓ_2. Each literal ℓ is associated with a watch list $\mathrm{WL}(\ell)$, which contains all the clauses watched by ℓ. Provided that certain invariants are maintained on the watched literals, this allows one to only check the clauses in the list $\mathrm{WL}(\neg\ell)$ when propagating ℓ, greatly reducing the number of clause visits during CDCL.

This research was conducted without the knowledge of the literature on linked watch lists. Reviewers pointed out that this idea is not novel and has been documented in [1] but was experimented on in [7] before. In this paper, we explore for a third time the representation of watch lists using a linked list data structure. We evaluate our work with a practical implementation in the NapSAT SAT solver. We detail the implementation difficulties raised by the linked list representation and empirically evaluate the performance of using watch lists via arrays and linked lists. We conclude that the linked list representation is not suitable for general applications.

Pragmatics of SAT.
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

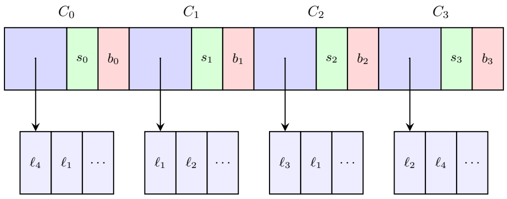


Figure 1 Representation of clauses in NapSAT. Clauses are stored as a fixed size structure containing a pointer to the literals, the size s of the clause, and a blocker literal b.

2 NapSAT Background

We assume that the reader is familiar with the CDCL algorithm and the basic data structures used in SAT solvers [4,7]. In this section, we introduce the data structures used in NapSAT relevant to understanding the rest of the paper.

NapSAT is a CDCL-based SAT solver written in C++. The code is available on GitHub[1] and GitLab[2], and consists of approximately $\sim 5,800$ loc, among which the core corresponds to $\sim 2,100$ loc[3]. NapSAT supports different variations of chronological backtracking [3]. In this paper, however, we will only consider the classical non-chronological variant, as it is the most common in practice.

NapSAT is an experimental solver that is not intended to compete with state-of-the-art solvers yet. The author therefore acknowledges that some techniques and representations may not be standard and comparable to more sophisticated solvers. However, most arguments presented in this paper are general enough to be applied to more advanced solvers. In the future, we plan to integrate the methods suggested by the reviewers to improve the NapSAT. We shall mark discussion on other solvers in a "remark" environment.

In NapSAT, literals are represented as 32-bits unsigned integers whose least significant bit indicates the polarity. Clauses are stored as a fixed-size structure containing a pointer to the literals, the size s of the clause, and a blocker literal b. The blocker must be in the clause and is used to quickly check if the clause is satisfied. Each clause takes 16 bytes of memory (8 for the pointers to literals, 4 for the size, and 4 for the blocker.). All clauses are stored in a contiguous vector, allowing us to identify them with a 32-bits unsigned integer instead of a 64-bits pointer. Figure 1 illustrates the representation of NapSAT clauses. We assume that the watched literals are the first two literals in the clause as in [1,4,7].

▶ Remark 1. A main difference between NapSAT and state-of-the-art solvers is the location of the blockers. In NapSAT, the blocker is stored in the clause data structure, while in state-of-the-art solvers, the blocker is stored in the watch list [2,4]. The advantage of storing the blocker in the clause is that it allows one to share it between the two watch lists. However, since most state-of-the-art solvers seem to agree on storing the blocker in the watch list, we will also implement and evaluate this representation in NapSAT in the future. In the following, we consider the data structures used in NapSAT but discuss how our arguments can be applied to different techniques.

---

[1] `https://github.com/RobCoutel/NapSAT`
[2] `https://gitlab.uliege.be/smt-modules/sat-library`
[3] as of commit df5a9ca4

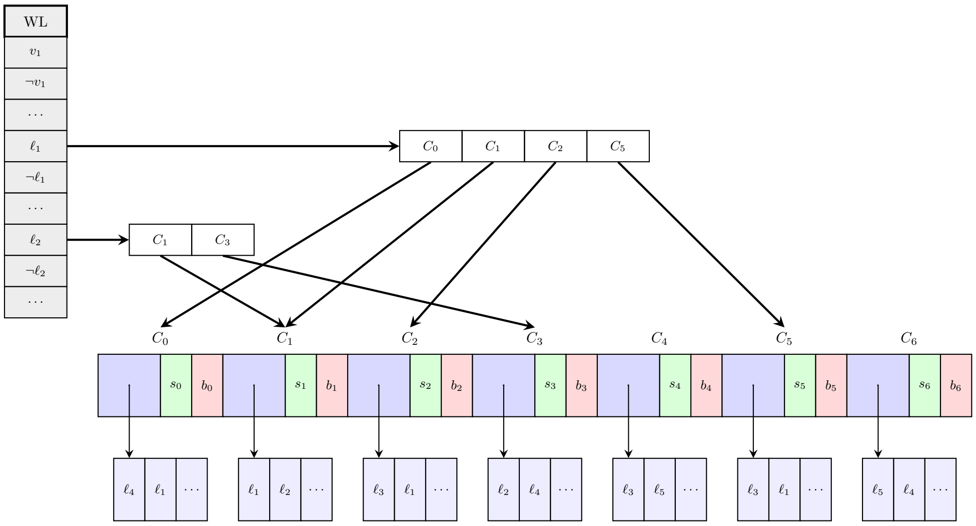


Figure 2 Array-based representation of watch lists. The watch list of ℓ_1 is $\{C_0, C_1, C_2, C_5\}$ and the watch list of ℓ_2 is $\{C_1, C_3\}$.

3 Watch Lists Representations

3.1 Array-Based Representation

The traditional representation of watch lists is an array of pointers to an array of references to clauses (potentially with blockers). For example, Figure 2 shows the watch lists of ℓ_1 and ℓ_2. The watch list of ℓ_1 is $[C_0, C_2, C_1, C_5]$ and is simply represented as an array of indices to the clauses. There exist variations of this representation, such as the use of pointers to the clauses instead of indices, as in CaDiCaL [2].

For this paper, we will consider indices as on Figure 2. This representation has the advantage of being very simple and flexible. It is easy to implement and does not complicate the maintenance of the code base.

The complexity of removal of an arbitrary element in the list is $O(n)$, where n is the number of clauses in the watch list. However, the solver seldom removes arbitrary clauses from the list. More often, a clause is removed as it is inspected during propagation. In this case, the complexity is $O(1)$ since we know the index of the clause in the watch list. There are two main approaches to removing an element from a list. Either (i) we swap the clause with the last clause in the list and decrement the size of the list, or (ii) we shift all the elements after the clause to the left, as in MiniSat [4]. Both approaches are sensible, since most of the time, the propagation will continue until the end, and the shifting naturally happens during the propagation. In the array-based implementation of NapSAT, we use the first approach (i). Watching a new clause is done in $O(1)$ time if the watch list is not full, and $O(n)$ otherwise, by simply pushing the clause at the end of the list (maybe after reallocation if the list is full).

3.2 Linked List Representation

When using linked watch lists, we remove one level of indirection. The watch list of a literal ℓ is now a simple reference to one of the clauses watched by ℓ. The clause data structure is now enhanced with two clause references, one for each watched literal. Exploring the watch list

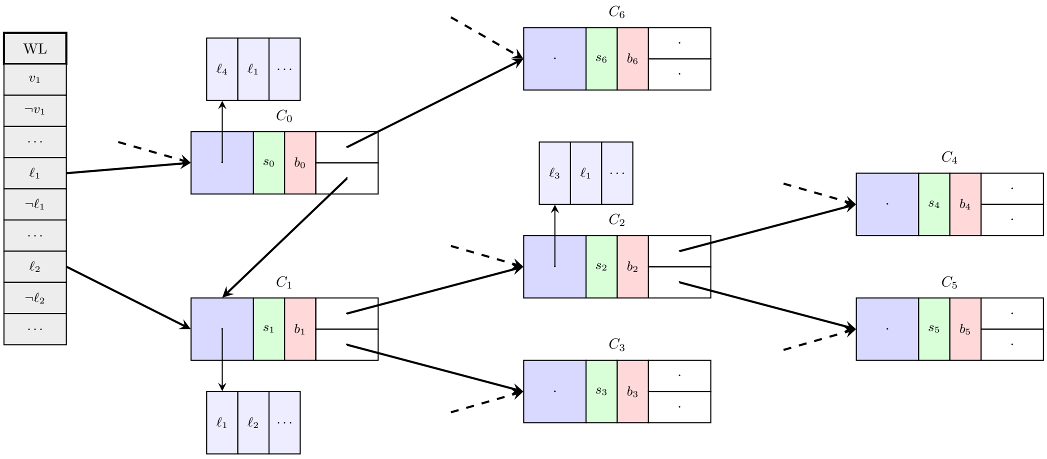


Figure 3 Representation of watch lists using the linked list data structure. The watch list of ℓ_1 is $\{C_0, C_1, C_2, C_5\}$ and the watch list of ℓ_2 is $\{C_1, C_3\}$.

of a literal ℓ now only requires following the first pointer of the clause if ℓ is the first watched literal, and the second pointer otherwise. This representation is shown in Figure 3. Clauses are now 24 bytes long, however, this space is saved from the list of references, compared to the array-based representation.

Similarly to the array representation, removal of an arbitrary clause from the list is $O(n)$, but removal during propagation is $O(1)$. It simply requires connecting the previous clause to the next clause in the list. Doing this efficiently and elegantly is however not trivial since the connection might be either with the first or the second pointer of the previous clause. For example, in Figure 3, if the solver removes C_1 from the watch list of ℓ_1, we connect the second pointer of C_0 to C_2 since ℓ_1 is the second watched literal of C_0 and the first watched literal of C_1. Watching a new clause C by ℓ is always done in $O(1)$ time, by connecting C to WL(ℓ) and updating WL(ℓ) to C, thereby pushing C at the front of the list.

Why linked lists? Using linked lists has several advantages over the array-based representation. First, it allows us to reduce the dereference level by one. Indeed, there is no longer a need for a pointer to the array of clause references. A reference to the first clause in the list is sufficient. Furthermore, in NapSAT, when exploring the list, the clause must be dereferenced anyway, there does not seem to be an extra cost of using this technique.

▶ Remark 2. The previous statement is however not true when using the blocker technique as in MiniSAT [4] or CaDiCaL [6] where the blockers are stored in the watch lists. A blocked clause does not need to be dereferenced. In that regard, comparing linked lists and arrays in NapSAT should yield better results than comparing linked lists and arrays in other solvers. Since linked lists perform worse than arrays in NapSAT (Section 4), we can expect that the difference would be even more significant in other solvers.

Second, it is more memory efficient. When elements are moved from one watch list to another, the array-based representation might require to reallocate memory in the destination list, and some memory might be wasted in the origin list. With linked lists, the memory is allocated during the creation of the clause, and the only lost memory is the pointers at the end of the list (which are not larger than the field remembering the size of the array). A simple and not very rigorous evaluation with Valgrind [8] on 10 problems with 150 variables from the uniform random 3-SAT satisfiable problems of SATLIB [5] (`uf150-01.cnf` to `uf150-010.cnf`) showed us that the linked list scheme uses about 13% less memory than the

Table 1 Intuitive comparison of the array and linked list representation of watch lists.

Aspect	Array		Linked list	
Dereference level	2 levels	(−)	1 level	(+)
Memory usage	Extensible	(−)	Fixed	(+)
Insertion	$O(1)$ or $O(n)$	(−)	$O(1)$	(+)
Arbitrary removal	$O(n)$	(=)	$O(n)$	(=)
Removal during propagation	$O(1)$	(=)	$O(1)$	(=)
Bookkeeping overhead	Low	(+)	High	(−)
Code complexity	Low	(+)	Medium	(−)

array-based scheme. We do not desire to make any strong claim about this, nor spend more time on this evaluation, but it is an interesting observation. Furthermore, the additional cost of adding two 32 bits integers to the clause data structure is negligible. Trying to artificially double the size of the clauses (32 bytes) using the `alignas` keyword in C++, without any benefit, had only a limited impact on the runtime of the solver.

▶ Remark 3. The PicoSAT paper [1] observes a similar save in memory for the linked list representation.

Finally, singly linked lists can easily be extended to doubly linked lists that allow $O(1)$ removal of any clause in the list. This could be interesting in a context where clauses are often removed by the user.

Implementation complications. The main difficulty of implementing a linked list scheme, aside from the necessary bookkeeping, is simplifying the clause set. Indeed, the array-based approach allows having a clause in more than two watch lists at a time, and then cleaning it up in post-processing. This is not possible with the linked list representation. In particular, NapSAT supports different backtracking strategies that maintain different invariants on the watched literals [3]. This makes the implementation of the linked list representation more complex.

Table 1 summarizes the comparison of the observations made in this section. We denote by (+) (resp. (−)) when a feature benefits (resp. harms) the respective representation, and (=) when the feature is equivalent in both representations.

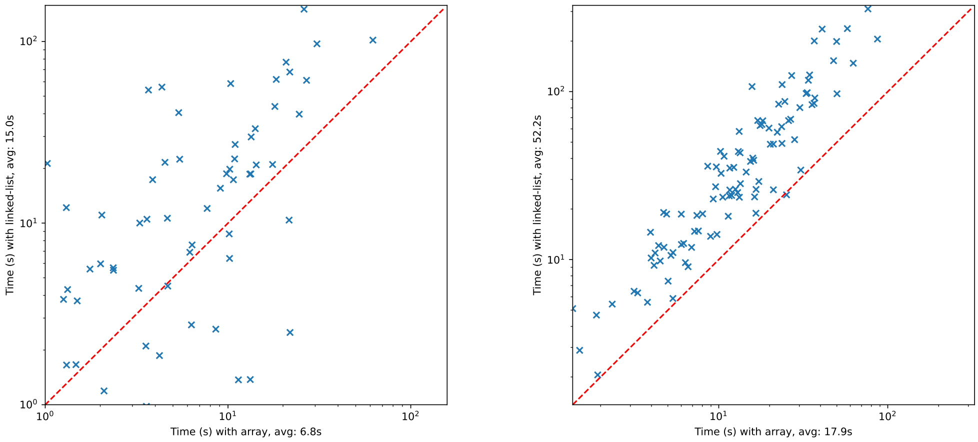
4 Empirical Evaluation

Figure 4 shows the difference of runtime between both implementations[4] of watch lists in NapSAT. Our experiments were conducted on the random 3-SAT instances of the SATLIB [5] with 250 variables. While the measurements were conducted on a single core of an Intel Core i7-10750H of a laptop with 16GB of DDR4 RAM, the results are significant enough to assert that they are not due to noise and that the linked list representation is significantly slower than the array-based representation: measured at 2.21 for SAT instances and 2.91 for UNSAT instances.

Note that the entropy in the results is also because changing the representation modifies the behavior of the solver. In particular, the watch lists change order differently in both implementations. When using linked lists, the removal of an element in the list is stable, that is, the order of the list does not change. However, when removing an element in the array

---

[4] commits 0f300b57 and df5a9ca4 for linked lists and array respectively



(a) SAT instances. The linked list representation runs on average 2.21 times longer than the array implementation.

(b) UNSAT instances. The linked list representation runs on average 2.91 times longer than the array implementation.

Figure 4 Comparison of the runtime of the array and linked list representation of watch lists on the random 3-SAT instances of the SATLIB with 250 variables. All runs were performed with default options of NapSAT (non-chronological backtracking with clause deletion).

implementation, the last element of the list is moved to the position of the removed element. Furthermore, when inserting a clause in the watch list, the array implementation pushes it to the end of the list, while the linked list implementation pushes it to the beginning of the list. This has an impact on which conflicts are found by the solver. However, this effect is quasi-random, and we can observe similar trends on all sizes of the SATLIB.

The **Linked list** and **Array** lines of Table 2 also show that the linked list scheme does not scale well. The runtime ratio increases with the size of the problems.

5 Practical Difficulties of Using Linked Lists

A seemingly insignificant drawback of the linked list scheme is that it is not possible to explore the watch list without dereferencing the pointer to the literals of the clause. Since the solver needs to know which branch to choose, it needs to know the order of the two watched literals. To do so, either a copy must be kept inside of the clause data structure, further increasing the bookkeeping overhead, or the literals must be dereferenced. This largely negates the advantage of the blocker. Indeed, when the blocker is satisfied, we wish to avoid checking the literals. However, in our linked list representation, this is not possible.

▶ Remark 4. In the PicoSAT paper [1], it was suggested to use a bit in the link to store this information. For example, a link to the clause C in a watch list of literal ℓ would be marked with a bit set to 0 if ℓ is the first watched literal of C, and 1 otherwise. This would allow us to avoid dereferencing the literals. However, swapping literals in the clause is now a problem since we would need to update the other link as well.

To test this claim, we have ensured that the literals were artificially dereferenced in NapSAT before checking the blocker. The results are shown in Table 2. We can see that the cost of dereferencing the literals is significant and is responsible for a nonnegligible part

Table 2 Comparison of the average runtime of the different watch list representations on the uniform random 3-SAT instances of the SATLIB.

	`uf200`	`uuf200`	`uf225`	`uuf225`	`uf250`	`uuf250`
Linked list	0.28 s	0.75 s	1.78 s	5.10 s	15.00 s	52.20 s
Array	0.20 s	0.44 s	1.10 s	2.63 s	6.80 s	17.92 s
Array with dereference	0.17 s	0.46 s	1.16 s	2.92 s	8.52 s	24.52 s

Table 3 Final comparison of the array and linked list representation of watch lists. Empirically, the array representation is faster and easier to implement.

Aspect	**Array**		**Linked list**	
Dereference level	2 levels	(−)	1 level	(+)
Memory usage	Extensible	(−)	Fixed	(+)
Bookkeeping overhead	Low	(+)	High	(−)
Code complexity	Low	(+)	Medium	(−)
Usefulness of blockers	High	(+)	Low	(−)

of the slowdown. However, there seems to be more to it. The rest of the performance drop is probably due to the increased bookkeeping overhead. For example, swapping the watched literals is a common practice in SAT solvers, and can be done in propagation with 4 assembly instructions using xor operations without branching. However, in the linked list representation, we cannot simply swap the literals, we also need to conditionally update the clause references. The branching and the additional memory accesses slow down the propagation. This leads us to believe that copying the literals to avoid dereferencing them might not be a good idea. For further investigation, we would check the impact of the order of clauses in the lists on the runtime of the solver.

▶ Remark 5. As opposed to this work, PicoSAT [1] results are more promising. It might be because PicoSAT does not use blockers, or because the linked list representation is more efficient in PicoSAT than in NapSAT. It would be interesting to investigate this further.

6 Conclusion

This paper studied the benefits and challenges of implementing watch lists using arrays and linked lists. Table 3 summarizes the comparison between both representations of watch lists. While linked lists have some merits, our experiments show that they do not pay off in practice. The blocker is such a powerful tool that saving a bit of memory is not worth negating its benefits. We therefore do not recommend using linked lists for general-purpose SAT solvers.

References

1 Armin Biere. Picosat essentials. *J. Satisf. Boolean Model. Comput.*, 4(2-4):75–97, 2008. URL: `https://doi.org/10.3233/sat190039`, `doi:10.3233/SAT190039`.

2 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

3 Robin Coutelier, Mathias Fleury, and Laura Kovács. Lazy reimplication in chronological backtracking. In *To Appear in the proceeding of SAT*, 2024.

4 Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. `doi:10.1007/978-3-540-24605-3\_37`.

5 Holger H Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. *Sat*, 2000:283–292, 2000.

6 Norbert Manthey. Cadical modification–watch sat. *SAT COMPETITION 2021*, page 28, 2021.

7 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. `doi:10.1145/378239.379017`.

8 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100. ACM, 2007. `doi:10.1145/1250734.1250746`.

9 João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227. IEEE Computer Society / ACM, 1996. `doi:10.1109/ICCAD.1996.569607`.