An A-Maze-ing SAT Solving Visualization

Norbert Manthey

Knowledge Representation and Reasoning Group
Technische Universität Dresden, 01062 Dresden, Germany
norbert.manthey@tu-dresden.de

Abstract. Modern SAT solvers are extremely powerful and solve many problems in industry and academia. Yet, these systems are highly complex and incorporate many research results from recent years, so that details can only be taught by experts. We present an abstract and simple visualization of solving the SAT problem – finding an exit in a maze – that models many techniques of modern solvers, ranging from simple search over verifying unsatisfiability proofs and formula simplification to parallel search techniques that include sharing of knowledge.

1 Introduction

The satisfiability testing (SAT) problem is one of the most relevant problems of computer science, as SAT is the representative problem for the complexity class $\mathcal{NP}$ [14]. Due to the numerous improvements to SAT solvers, many industrial problems are successfully reduced to SAT [10,12,20,22,24,43,61], because using highly optimized and specialized SAT solvers makes all these improvements accessible. However, keeping track with all developments in SAT solving is quite hard, as the complexity of award winning SAT solvers increases continuously over the years [1,2]. Still, many ideas from SAT solving are very valuable and might also be interesting to grasp, even without giving all the formal details.

Related work on abstract descriptions of SAT technology focused on formal aspects. DPLL(T) [50] and similar systems [3,46] focusses on the search of a SAT solver without formula simplification, which is included in the system Generic CDCL [31]. Heule et al. look more on the soundness of formula modifications and omit the search process [39]. From all these abstract descriptions it is hard to intuitively understand how a SAT solver behaves in the search space, and how other techniques like formula simplifications modify the search space.

With this work, we provide an abstract visualization of SAT solving as an intuitive and easy entry that covers most modern techniques without requiring too much details. Instead of constructing a model for a given formula, we show how an exit can be found in a maze. With this visualization we can show the difference between the DPLL [17] and the CDCL algorithm [47], and furthermore can motivate restarts [7,11,23,33,54], clause removal [5,6] and additional reasoning during search. Parallel solving approaches [30,48,57] like parallel portfolio solving can be modelled as well as search space partitioning. Finally, also

emitting unsatisfiability proofs based on reverse unit propagation [21,26,60] and verifying these proofs can be done in the maze visualization.

Obviously, such a visualization is not meant to cover all modern solving techniques, but we believe that supporting known results with a visualization is very useful. Especially for introducing SAT solving to unexperienced persons, or when presenting search algorithm extensions, this work is considered very valuable. By looking at specific situations when finding an exit in the maze, the motivation for certain developed SAT techniques, for example clause learning or clause removal, can be recognized. Furthermore, other problems of the search of modern SAT solvers might become visible when more methods are translated to the maze visualization. Besides the mediated intuition, this work also provides a large list of references that facilitates entering the the research field of satisfiability testing.

In the following section the preliminaries of SAT solving are given. Next, in Section 3 the visualization is introduced by using the plain search algorithms. Additionally, the visualization of exemplary techniques is presented. Section 4 parallel SAT solving approaches are modelled in the maze visualization. Finally, in Section 5 a conclusion is given.

2 Preliminaries

We consider a fixed infinite set $\mathcal{V}$ of Boolean *variables*. A *literal* is a variable v (*positive literal*) or a negated variable $\overline{v}$ (*negative literal*). The *complement* $\overline{x}$ of a positive (negative, resp.) literal x is the negative (positive, resp.) literal with the same variable as x. The complement of a set of literals S, denoted by $\overline{S}$ is defined as $\overline{S} := \{\overline{x} \mid x \in S\}$. A clause C is a finite set of literals, and is understood as disjunction of literals. *Formulas* are finite multisets of clauses, and are understood as conjunction of clauses. A sequence of literals M is *consistent*, if whenever $x \in M$, then $\overline{x} \notin M$. A consistent sequence M is also treated as a set whenever convenient. The set of variables that occur in a formula F is denoted with $\mathsf{vars}(F)$. Likewise, we apply vars to sets and sequences of formulas.

An *interpretation* I is a consistent sequence of literals, and is called *total*, if $\mathsf{vars}(I) = \mathcal{V}$. The *reduct* $F|_I$ of a formula F with respect to an interpretation I represents the resulting formula after the reducing F with respect to I. The reduct removes all satisfied clauses and all falsified literals, and is defined as

$$F|_I := \{C \setminus \overline{I} \mid C \in F, C \cap I = \emptyset\}.$$

An interpretation I *satisfies* a formula F, if all clauses of a formula are satisfied by the interpretation, i.e. $F|_I = \emptyset$.

A set of formulas M *entails* a formula G, in symbols $M \models G$, if all total interpretations I that satisfy all formulas $F \in M$ also satisfy G. Let I be an interpretation, F be a formula, and M be the set of all literals in the sequence I. Then I *models* F, in symbols $I \models F$, if the set M entails F. Then, I is called *model* of F. A formula is called *satisfiable*, if there exists a model of F. Otherwise, F is called *unsatisfiable*. The *satisfiability testing* (SAT) problem is to answer the question whether a given propositional formula F is satisfiable. More details and background on satisfiability testing can be found in [14].

3 Visualizing Modern SAT Solving

Modern SAT solvers are extremely powerful due to the improvements of the last 20 years, starting with the *conflict driven clause learning* (CDCL) algorithm [47]. Only due to novel reasoning and simplifications [15, 18, 58], heuristics [23, 33, 49, 62] and data structures [32, 49, 55], these systems are used extensively in both academia and industry to solve NP problems [14]. However, in most universities only the old DPLL algorithm from the 1960s is taught [17], which, besides the termination rules to determine satisfiability, basically uses *search*, *unit propagation* and *backtracking*. Nowadays, SAT solvers are parallel software systems [30, 48, 57] which furthermore allow to print traces such that the unsatisfiability answer can be verified [21, 26, 60]. This work is not meant to present an abstraction that can be linked directly to all modern developments[1]. Instead, the presented abstraction is believed to be useful to built a first intuition, which might built a good starting point for diving into the all the formal necessities and little details.

3.1 Visualization Based on the DPLL Algorithm

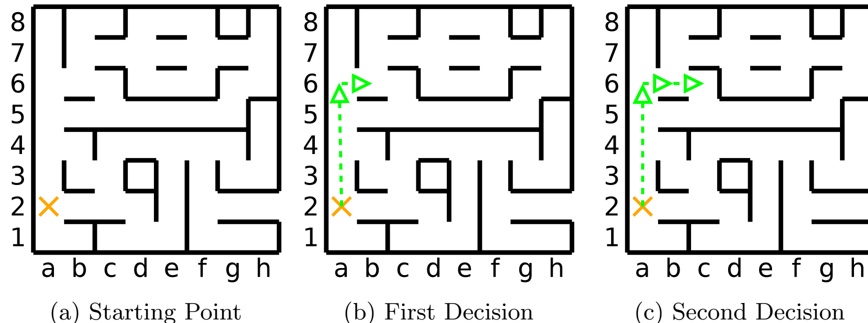
Teaching all the ideas behind the novel algorithms is difficult, as most of these algorithms are quite complex. Hence, we propose an abstract visualization of solving the SAT problem which can help to get a good intuition of modern SAT solving before starting to look into the details. The visualization compares solving the SAT problem with finding an exit in a maze by an agent. While modern SAT solvers construct an interpretation iteratively, the maze constructs the path to the exit. Similarly to the DPLL algorithm [17] that uses rules to modify the interpretation, there are a few rules to follow in the maze:

1. when moving left, backtracking has to be applied
2. when moving right, search decisions can be made

Essentially, the agent is not allowed to make search decisions when moving right. A move of the agent in the maze is based on columns, and not in fields. Hence, all fields in a column that can be reached from the current location of the agent without leaving the column and without jumping over bars are considered to be reached without making a step. This way, in the maze in Fig. 1a the agent can access field (B6) from field (A2) in a single step. There is another major difference between the maze visualization and solving propositional formulas: while in the maze multiple search decisions to continue the search path are possible, variables of propositional formulas can be assigned only two truth values.

With these constraints, a satisfiable formula corresponds to a maze whose exit can be found with a path that crosses each column only in one block, i.e. by never moving left.

[1] We would be happy, if such a direct connection, i.e. a translation from a maze to CNF would exist such that all steps on the CNF can be represented in the maze and vice versa. However, the maze example would correspond to solving an MDD.



(a) Starting Point (b) First Decision (c) Second Decision

Fig. 1: Stage of sequential search

A maze is given in Fig. 1a, where the starting point is marked with a cross. For now, let the agent who searches the exit in the maze be called SAM. In the illustration, the agent will always be located at the end of the current path. Assume SAMs search heuristic chooses to start the search as high as possible, so that SAM visits the field (B6) next, as shown in Fig. 1b. Afterwards, he visits the field (C6), as presented in Fig. 1c.

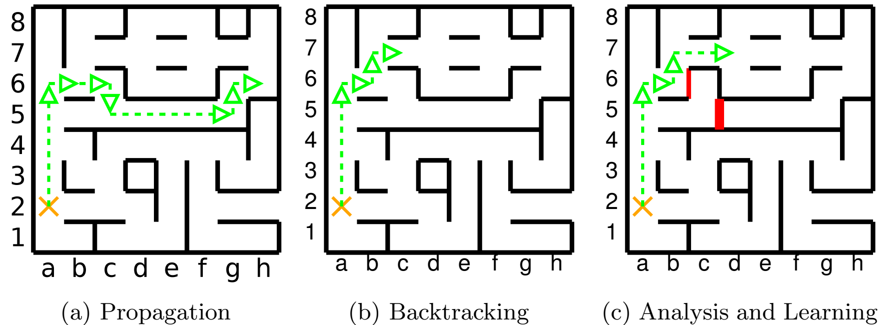
Once SAM is located in field (C6), there is no choice, so that *propagation* is performed, as depicted in Fig. 2a. However, after propagation SAM notices that there is no exit and no possibility to move forward. This situation is called a *conflict*. In the SAT solving world, the corresponding partial interpretation would remove all literals from a clause, resulting in the empty clause that cannot be satisfied by extending the interpretation further. Once a conflict is found, the DPLL algorithm performs backtracking by undoing the last decision. Likewise, SAM returns to (B6), as shown in Fig. 2a, and does not choose (C6) this time again.

Note that the order how decisions are taken is ordered based on the path that is constructed by the agent. In a SAT solver, the order of the variables that are chosen to construct the partial interpretation is not fixed [13, 19, 47, 49].

3.2 The CDCL Algorithm

The major difference between the CDCL algorithm and the former DPLL algorithm is the following: After finding a conflict, we do not only perform backtracking, but furthermore create a learned clause, and add this learned clause (temporarily [5, 6, 56]) to the formula [47, 62]. This learned clause is illustrated with the bar between fields (C6) and (D5) in Fig. 2c – where we could argue that the bar might block any other part of the path from the last decision to the found dead end, but creating a link between different locations of the bar and different kinds of learned clauses is beyond the scope of the work[2]. Once the bar is added to the maze, this bar avoids entering the same unsolvable subspace again. Likewise, in the SAT world, the learned clause prevents the search to enter the same

[2] For an analysis of different SAT algorithms see [62].



(a) Propagation (b) Backtracking (c) Analysis and Learning

Fig. 2: Stages of sequential search in presence of a conflict

search space. To correctly handle the backtracking information, the choice that failed in the maze is also closed with a small auxiliary bar.[3] In the illustration, this auxiliary bar is thinner than the bar that represents the learned clause. In the extreme case, adding these learned clauses can result in an exponentially superior behavior of the CDCL algorithm over the DPLL algorithm [53].

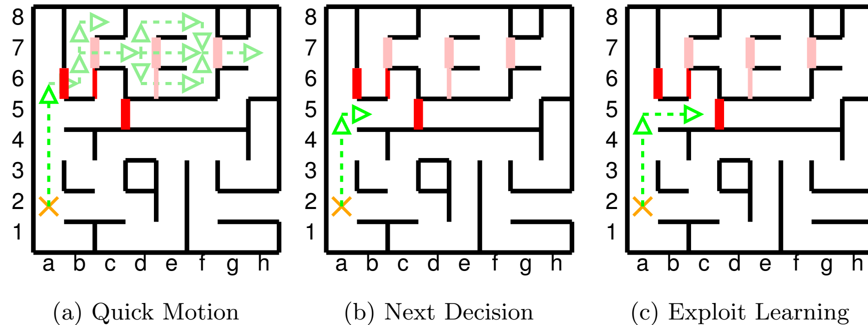
To illustrate the effect, let Sam continue his search in quick motion. From (b6) he visits field (d7), tests all possibilities and adds a few bars to the maze. After concluding that he cannot find the exit from there, he visits (8c) without success and finally returns to the very first row. All these steps are added to the maze in Fig. 3a in a light fashion. Next, Sam enters the field (b5), as shown in Fig. 3b. By propagation, he quickly finds his first learned bar, as in Fig. 3c, and he can continue with backtracking instead of further search.

In this small example the maze behind the first learned bar is rather simple, so that not too many search steps are saved. However, assume behind that bar multiple other learned clauses are necessary to prove that there is no exit in this part of the maze. Without learning, Sam would have to repeat the exact steps again. Likewise, the DPLL procedure would perform these redundant search steps. Learning the clause has hence the potential to save a lot of work. Similarly, the CDCL algorithm is stronger than the DPLL algorithm in solving formulas.

3.3 Visualizing Modern Solving Techniques

Modern SAT solvers include a set of algorithm extensions that make them effective on large industrial problems. While the maze cannot show the importance of good data structures – e.g. the *Two Watched Literal* data structure [49] – heuristics like *restarts*, *learned clause removal* and *formula simplification* can be illustrated. This section gives an example for each of these techniques.

[3] While in SAT there are only two possible truth assignments, the maze actually represents multiple choices, so that backtracking has to be made clear explicitly, while in the binary case it is clear where to continue. Hence, this additional clause becomes necessary, essentially to be able to verify unsatisfiability proofs (see Section 3.4).



(a) Quick Motion (b) Next Decision (c) Exploit Learning

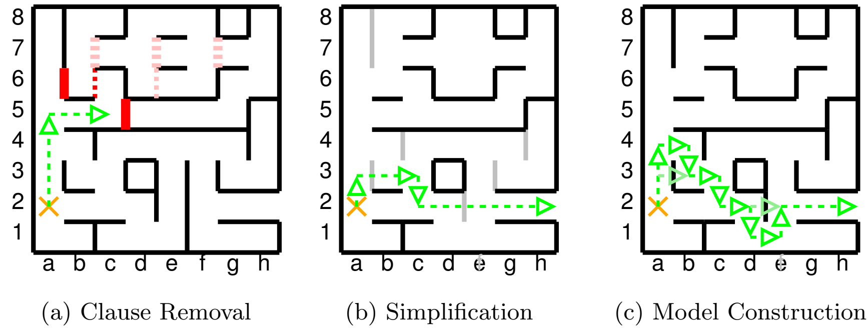
Fig. 3: Exploiting the learned clauses in future search

Restarts By adding *restarts* to SAT solvers, their behavior became much more stable [23]. A restart means to clear the interpretation while keeping all the other information that has been gathered during search. In the maze, this means that Sam simply needs to go back from his current position to the starting point and start search over from there. Since the values within the search heuristic might have changed, it is likely that Sam enters another part of the maze next. In SAT solvers, the assigned truth values for the next search decisions is tried to be the same as before the restart – however, the order of the search decisions will be different [51].

Clause Removal From a completeness point of view, keeping some of the learned clauses until the exit is found is necessary. Otherwise Sam might enter the same part of the maze again and again without making progress. On the other hand, there are many learned clauses that are redundant. This effect is shown in the maze in Fig. 4a. The solid learned clauses are necessary, whereas the dashed learned clauses are not reachable any more and can be removed. Learned clause removal simply removes some learned clauses. As this removal is driven by a heuristic, sometimes clauses are removed that are not redundant. On the other hand, the illustration shows nicely that most of the learned clauses are actually not useful for the search any more after a few more clauses are learned [56]. In modern SAT solvers heuristics have been added to identify the clauses that are actually useful and to remove other clauses [5, 6].

Formula Simplification The maze in Fig. 4b shows how a maze might be modified by simplification. In general, modifications are feasible if they keep at least one valid path to the exit, if there exists one, and otherwise to not create a path to the exit if reaching the exit was not possible before[4]. One of the most effective formula simplification techniques [9], *variables elimination* [18, 59] can be visualized by merging neighboring columns. In the example, this simplification is possible for the columns e and f. In Fig. 4c these two columns

[4] Again, a valid path to the exit cannot move left.



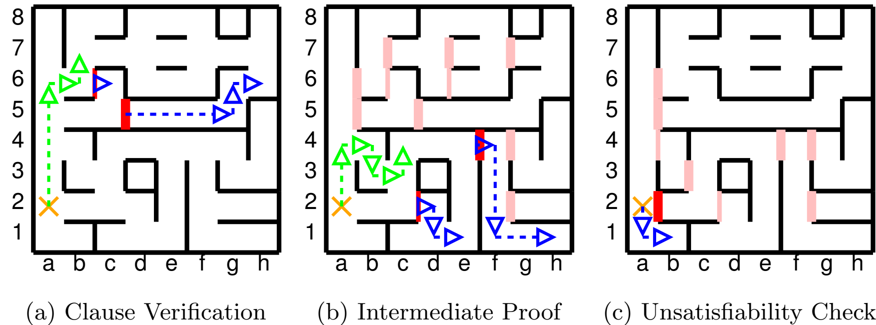
(a) Clause Removal (b) Simplification (c) Model Construction

Fig. 4: Advanced solving techniques

are merged. Furthermore, some of the walls of the maze became light grey. This modification is feasible, because removing vertical walls that have a free end is sound. This technique could be related with *blocked clause elimination* [38], or redundant clauses in general [27, 39]. When removing blocked clauses from a formula additional models are possible for the modified formula, which might not be valid for the original formula any more. Likewise, other techniques like forcing paths to fields that are located on all possible solution paths would be possible. Such a modification would correspond to looking for backbones via *probing* [42] or *look-ahead* techniques [29].

Now, Sam can find a path to the exit in the modified maze, as added to the maze in 4b. For simplification techniques that keep equivalence in SAT, the solution path does not have to be modified to become a valid solution path for the original maze. However, once the walls become solid again, the path has to be corrected, if the solution path crossed these walls. This correction is illustrated in Fig. 4c. When the columns (e) and (f) are separated again, the path might be modified once more to result in a valid solution. In SAT solving, fixing the path corresponds to changing the solution for the modified formula to obtain a solution for the original formula [37, 39].

Learning UIP Clauses SAT solvers learn empowering clauses [52], which are clauses that improve the strength of unit propagation compared to the current state of the formula. After backtracking, these clauses lead to unit propagation, because they are created such that there is exactly one literal in the learned clause that becomes unassigned alone during backtracking. During conflict analysis, these literals refer to *unique implication points* (UIPs). It is believed that learning a certain type of these UIP clauses – namely 1st-UIP-clauses – is a good strategy [62]. The visualization supports this believe that learning 1st-UIP [49] clauses is stronger than learning clauses related to other UIPs: the clause learned in Fig. 2c corresponds to the 1st-UIP clause, because this bar is the closest branch on the path when backtracking from the conflict. Another possible learned clause would be located between the fields (b6) and (c6), which in the given step is



(a) Clause Verification (b) Intermediate Proof (c) Unsatisfiability Check

Fig. 5: Verifying the unsatisfiability answer of a run

the bar that is added to memorize backtracking. If only the latter clause (the 2nd-UIP) would be learned, then Sam would enter the same search space again.

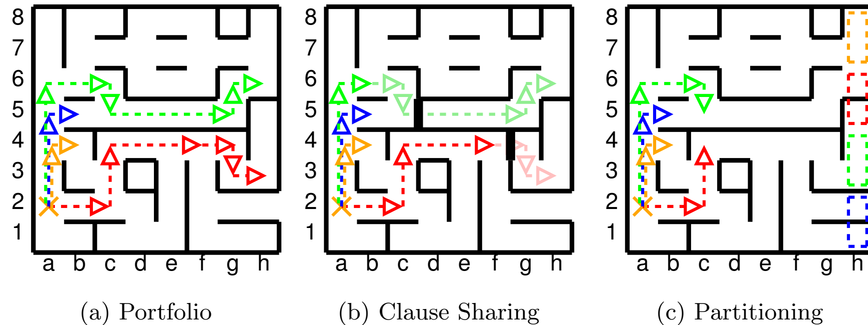
3.4 Verifying Unsatisfiability Proofs

Modern SAT solvers are used in many applications, including safety critical applications as hardware verification or software verification [14]. In verification, if a formula is satisfiable, then the model corresponds to a counter-example of the described statement. On the other hand, if a SAT solver returns the answer "unsatisfiable", then the property is assumed to hold. While testing whether a given interpretation satisfies a formula is simple, verifying unsatisfiability is rather difficult. Recent developments made it possible to generate proofs during a run of a CDCL solver with low overhead, while checking the proof remains efficient [21, 26, 60].

In the maze abstraction, both the (clausal) proof and the proof verification process can be illustrated. As we have to trust the verification, the verification task should be as simple as possible: given a starting field (x), then the verification has to check whether a dead-end is reached from this field by using only propagation. This operation is very close to the *reverse unit propagation*, which is used in modern proof verification tools [21]. Consider the maze in Fig. 5a[5]: from (c5) we find a dead-end by propagation (as in Section 3.2). On the other hand, starting from (c7) propagation does not reach a dead-end, because there is choice involved, namely visiting field (d6), (d7) or (d8).

Now for the example given in Fig. 5b, the clauses that have been learned can be verified in exactly the order they have been added to the maze. In the final maze 5c we can see that by the simple verification check it can be verified that there is no path to the exit of the maze, even though intermediate learned clauses have been removed already. This removal corresponds to clauses proofs with deletion information [26].

[5] Note, the exit has been closed to turn the maze into an unsatisfiable maze.



(a) Portfolio (b) Clause Sharing (c) Partitioning

Fig. 6: Finding an exit in parallel

4 Parallel Solving Algorithms

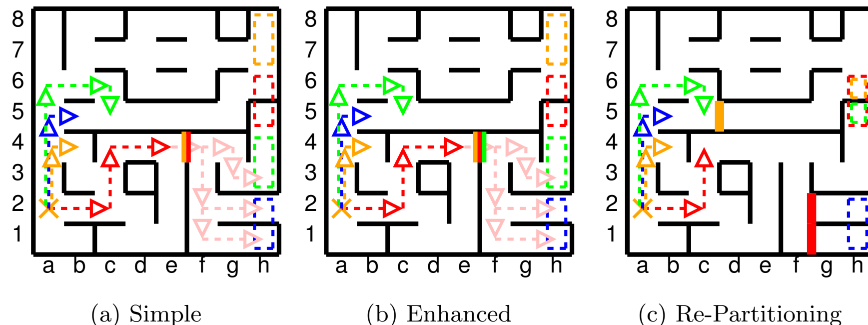
Due to parallel computing resources, parallel SAT solving techniques have been presented [30, 48, 57]. The maze shows why unit propagation cannot be parallelized easily [40]: Sam does not know in advance where he will be going, and parallel tasks might be redundant due to the bars that are added to the maze over time.

As this low level parallelization is not effective, high-level parallelization is used with two flavors: parallel *portfolio* solvers [4,25] solve the same formula with differently configured sequential solvers, and *search space partitioning* solvers [28, 34, 36] partition the search space and afterwards solve the partitions and the whole formula. Additionally, the performance is improved by knowledge sharing.

Parallel Portfolio Solvers Assume we have four computing resources. This corresponds to searching with four agents in the maze, where each agent adds *private* bars to the maze as in the CDCL algorithm. Such a situation is illustrated in Fig. 6a. As in portfolio SAT solvers, these private bars can be shared with all agents as long as only simple simplification of the maze – simplifications that exactly preserve all path to the exit – are used. Likewise, clause sharing is sound in portfolio SAT solvers as long as all solvers only apply equivalence preserving simplification techniques [45].

By making the additional bars public, solvers usually benefit. Furthermore, it can be seen that there also exist redundant bars which should not be shared as they do not help the search process of other agents, for example all the bars that have been removed in the sequential example in Fig. 5a. Hence, modern solvers do not share all clauses, and also filter shared clauses before these clauses are received [8].

Search Space Partitioning Solvers Search space partitioning of a formula F is done by adding *partitioning constraints* K, which are formulas as well [34,36]. For creating n disjoint partitions F_i, $1 \leq i \leq n$, n constraints K_i are created



(a) Simple (b) Enhanced (c) Re-Partitioning

Fig. 7: Clause sharing in search space partitioning solvers, and re-partitioning

such that the following conditions hold: $K_i \wedge K_j \equiv \perp$ for $1 \leq i < j \leq n$, and $\bigvee_{i=1}^{n} K_i \equiv \top$. The formula of partition i is created as $F_i := F \wedge K_i$.

This partitioning can be illustrated as follows: One column of the maze is partitioned. Then, the path of an agent has to be located in a position of the partition that is assigned to the agent. For simplicity, this column is chosen to be on the most right column. Then, each agent is allowed to find a solution only in his partition – an agent can only leave the maze if the exit is located behind a field of the partition of the agent. Fig. 6c illustrates this partitioning for four agents, where the exit can be reached only for the blue agent, i.e. the fourth partition from the top of the maze.

Clause sharing in partitioning solvers is more restrictive than for the portfolio approach, because now the shared clauses also depend on the partitioning column [35, 36]. Fig. 7a shows how the first agent (i.e. the red agent, with the lowest start in the maze) added a bar in the lower half of the maze. Without much analysis it is clear that this bar is also valid for the agent whose partition contains the rows (8) and (7). With a little more effort, it is also clear that sharing this bar with the agent whose partition contains rows (6) and (5) is sound (see Fig. 7b). The sharing of learned clauses in partitioning based SAT solvers has been improved similarly [41].

Further parallelism can be added by allowing another agent to solve the whole maze, so that this agent is allowed to cross the partitioning column anywhere. Adding such a fifth agent illustrates the difference between *plain partitioning* and *iterative partitioning* [34]. Likewise, once an agent found that his partition does not lead to the exit, he becomes idle. This idle agent can be re-used by re-partitioning an existing partition. Either, the partition is split, such that the two agents work on disjoint parts of the search space, or the new partition becomes a sub-partition of the original partition. Splitting corresponds to *plain partitioning* [28, 34], and the latter approach is related to *iterative partitioning* [34, 36]. Re-using idle agents is illustrated in Fig. 7c: the partition with rows (5) and (6) is now solved by three agents. Before an agent starts to find a valid path for the new partition, his privately added bars have to be removed. Otherwise, these agents might not find an exit any more and hence might not find the global

exit at all. Naturally, if the learned bars have been shared correctly, some of the previously private bars might remain valid (e.g. the bar in the upper half of the maze). On the other hand, bars that are valid for the new partition and its parent partition can be used safely, for example the bars in the lower part of the maze.

5 Conclusion

We present a very abstract, yet adequate, visualization of solving the SAT problem that is capable of modelling many modern techniques. This visualization is based on finding the exit in a maze. The covered techniques range from simple heuristic search with clause learning over verifying unsatisfiability proofs and formula simplification to parallel search techniques that include sharing of knowledge. We believe that this form of representation helps to understand the way SAT solvers work, and makes reasons for design decisions clearer. Additionally, while presenting modern SAT solving algorithms, this work also provides pointers to literature that might be helpful for learning more about the formal aspects of SAT solving.

In this paper, the basic search process with clause learning, restarts and simple formula simplifications are covered. Furthermore, verifying unsatisfiability proofs is shown with the maze visualization. Finally, we present how parallel solving methods work and how clause sharing is performed. There are many more techniques that can be modelled, but that have not been presented here, for example finding all solutions, incremental SAT solving or simplifying the formula during search. On the other hand, there are also techniques, that cannot be easily explained based on the maze, for example formula simplification techniques that are based on *extended resolution* [44] or cutting planes reasoning [15, 16]. Nevertheless, we believe that the presented abstraction – finding an exit in a maze – provides an easy entry to SAT technology, before starting with all the formal details.

References

1. *Proceedings of SAT Challenge 2013*, volume B-2013-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, Helsinki, Finland, 2013.
2. *Proceedings of SAT Competition 2014*, volume B-2014-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, Helsinki, Finland, 2014.
3. H. Arnold. A linearized DPLL calculus with clause learning. `http://opus.kobv.de/ubp/volltexte/2009/2908/`, 2010.
4. G. Audemard, B. Hoessen, S. Jabbour, J.-M. Lagniez, and C. Piette. Revisiting clause exchange in parallel SAT solving. In *SAT 2012*, volume 7317 of *LNCS*, pages 200–213, 2012.
5. G. Audemard, J.-M. Lagniez, B. Mazure, and L. Saïs. On freezing and reactivating learnt clauses. In *SAT 2011*, volume 6695 of *LNCS*, pages 188–200, 2011.

6. G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, pages 399–404, 2009.
7. G. Audemard and L. Simon. Refining restarts strategies for SAT and UNSAT. In *Principles and Practice of Constraint Programming*, volume 7514 of *LNCS*, pages 118–126, 2012.
8. G. Audemard and L. Simon. Lazy clause exchange policy for parallel SAT solvers. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 197–205. Springer, 2014.
9. A. Balint and N. Manthey. Boosting the performance of SLS and CDCL solvers by preprocessor tuning. In *POS-13*, volume 29 of *EPiC Series*, pages 1–14. EasyChair, 2014.
10. R. Béjar and F. Manyà. Solving the round robin problem using propositional logic. pages 262–266. AAAI Press / The MIT Press, 2000.
11. A. Biere. Adaptive restart strategies for conflict driven SAT solvers. In *SAT 2008*, volume 4996 of *LNCS*, pages 28–33, 2008.
12. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th Conference on Design Automation*, pages 317–320. ACM Press, 1999.
13. A. Biere and A. Fröhlich. *Evaluating CDCL Variable Scoring Schemes*, pages 405–422. Springer International Publishing, Cham, 2015.
14. A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
15. A. Biere, D. Le Berre, E. Lonca, and N. Manthey. Detecting cardinality constraints in CNF. In *SAT 2014*, volume 8561 of *LNCS*, pages 285–301, 2014.
16. W. J. Cook, C. R. Coullard, and G. Turán. On the complexity of cutting-plane proofs. *Discrete Applied Mathematics*, 18(1):25–38, 1987.
17. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
18. N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT 2005*, volume 3569 of *LNCS*, pages 61–75, 2005.
19. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT 2004*, volume 2919 of *LNCS*, pages 502–518, 2004.
20. C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *SAT 2007*, volume 4501 of *LNCS*, pages 340–354, 2007.
21. E. I. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE 2003*, pages 10886–10891. IEEE Computer Society, 2003.
22. E. I. Goldberg, M. R. Prasad, and R. K. Brayton. Using SAT for combinational equivalence checking. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 114–121. ACM, 2001.
23. C. P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1-2):67–100, 2000.
24. P. Großmann, S. Hölldobler, N. Manthey, K. Nachtigall, J. Opitz, and P. Steinke. Solving periodic event scheduling problems with SAT. In *Advanced Research in Applied Artificial Intelligence*, volume 7345 of *LNCS*, pages 166–175, 2012.
25. Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: A parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2009.

26. M. J. Heule, W. A. Hunt Jr, and N. Wetzler. Trimming while checking clausal proofs. In *FMCAD 2013*, pages 181–188. IEEE, 2013.
27. M. J. Heule, M. Järvisalo, and A. Biere. Clause elimination procedures for CNF formulas. In *LPAR*, volume 6397 of *LNCS*, pages 357–371, 2010.
28. M. J. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing*, volume 7261 of *LNCS*, pages 50–65, 2012.
29. M. J. H. Heule and H. van Maaren. *Look-Ahead Based SAT Solvers*, chapter 5, pages 155–184. Volume 185 of Biere et al. [14], 2009.
30. S. Hölldobler, N. Manthey, V. H. Nguyen, J. Stecklina, and P. Steinke. A short overview on modern parallel SAT-solvers. In *Proceedings of the International Conference on Advanced Computer Science and Information Systems*, pages 201–206, 2011. ISBN 978-979-1421-11-9.
31. S. Hölldobler, N. Manthey, T. Philipp, and P. Steinke. Generic CDCL – A formalization of modern propositional satisfiability solvers. In *POS-14*, volume 27 of *EPiC Series*, pages 89–102. EasyChair, 2014.
32. S. Hölldobler, N. Manthey, and A. Saptawijaya. Improving resource-unaware SAT solvers. In *LPAR*, volume 6397 of *LNCS*, pages 519–534, 2010.
33. J. Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI*, pages 2318–2323, 2007.
34. A. E. J. Hyvärinen, T. A. Junttila, and I. Niemelä. Partitioning SAT instances for distributed solving. In *LPAR*, volume 6397 of *LNCS*, pages 372–386, 2010.
35. A. E. Hyvärinen, T. A. Junttila, and I. Niemelä. Grid-based SAT solving with iterative partitioning and clause learning. In *Principles and Practice of Constraint Programming – CP 2011*, volume 6876 of *LNCS*, pages 385–399, 2011.
36. A. E. Hyvärinen and N. Manthey. Designing scalable parallel SAT solvers. In *SAT 2012*, volume 7317 of *LNCS*, pages 214–227, 2012.
37. M. Järvisalo and A. Biere. Reconstructing solutions after blocked clause elimination. In *SAT 2010*, volume 6175 of *LNCS*, pages 340–345. 2010.
38. M. Järvisalo, A. Biere, and M. J. Heule. Blocked clause elimination. In *TACAS*, volume 6015 of *LNCS*, pages 129–144, 2010.
39. M. Järvisalo, M. J. Heule, and A. Biere. Inprocessing rules. In *Automated Reasoning*, volume 7364 of *LNCS*, pages 355–370, 2012.
40. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, 1990.
41. D. Lanti and N. Manthey. Sharing information in parallel search with search space partitioning. In *Learning and Intelligent Optimization*, LNCS, pages 52–58, 2013.
42. I. Lynce and J. P. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *ICTAI*, pages 105–110, 2003.
43. I. Lynce and J. P. Marques-Silva. Efficient haplotype inference with Boolean satisfiability. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, pages 104–109. AAAI Press, 2006.
44. N. Manthey, M. J. Heule, and A. Biere. Automated reencoding of Boolean formulas. In *Hardware and Software: Verification and Testing*, volume 7857 of *LNCS*, pages 102–117, 2013.
45. N. Manthey, T. Philipp, and C. Wernhard. Soundness of inprocessing in clause sharing SAT solvers. In *SAT 2013*, volume 7962 of *LNCS*, pages 22–39, 2013.
46. F. Marić. Formalization and implementation of modern SAT solvers. *Journal of Automated Reasoning*, 43(1):81–119, 2009.

47. J. P. Marques-Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. ICCAD '96, pages 220–227. IEEE Computer Society, 1996.
48. R. Martins, V. Manquinho, and I. Lynce. An overview of parallel SAT solving. *Constraints*, 17(3):304–347, 2012.
49. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM, 2001.
50. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, Nov. 2006.
51. K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT 2007*, volume 4501 of *LNCS*, pages 294–299, 2007.
52. K. Pipatsrisawat and A. Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 3*, AAAI'08, pages 1481–1484. AAAI Press, 2008.
53. K. Pipatsrisawat and A. Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175(2):512–525, 2011.
54. A. Ramos, P. van der Tak, and M. J. Heule. Between restarts and backjumps. In *SAT 2011*, volume 6695 of *LNCS*, pages 216–229, 2011.
55. L. O. Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, Canada, 2004.
56. L. Simon. Post mortem analysis of sat solver proofs. In D. L. Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop*, volume 27 of *EasyChair Proceedings in Computing*, pages 26–40. EasyChair, 2014.
57. D. Singer. Parallel Resolution of the Satisfiability Problem: A Survey. In *Parallel Combinatorial Optimization*. Wiley Interscience, Oct. 2006.
58. M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In *SAT 2009*, volume 5584 of *LNCS*, pages 244–257, 2009.
59. S. Subbarayan and D. K. Pradhan. NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. In *SAT 2005*, volume 3542 of *LNCS*, pages 276–291, 2005.
60. N. Wetzler, M. J. Heule, and W. A. Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In C. Sinz and U. Egly, editors, *SAT 2014*, volume 8561 of *LNCS*, pages 422–429. Springer, 2014.
61. D. Williamson, L. Hall, H. Hoogeveen, C. Hurkens, J. K. Lenstra, S. Sevastjanov, and D. Shmoys. Short shop schedules. *Operations Research*, 45:288–294, 1997.
62. L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 279–285. IEEE Press, 2001.