SAT-Web: A Web-Based Educational SAT Visualisation Tool

James Madgwick[1,*], Martin Mariusz Lester[1]

[1]University of Reading, PO Box 217, Reading, Berkshire, RG6 6AH, United Kingdom

Abstract
Provision of tools for explaining SAT solvers and visualising SAT instances helps to facilitate easier entry to the field for newcomers, yet this remains an under-explored area. In this tool paper we present SAT-Web, a web-based educational tool for explaining SAT solving, and the first tool of its kind which can be used entirely from a web-browser. Our tool provides visualisations with search trees and variable interaction graphs, and includes a DPLL solver with tracing. We explain the design, techniques and implementation details behind the tool. We also review approaches and tools for instance visualisation and existing pedagogical tools designed to explain solvers. Finally, we discuss potential future improvements and opportunities for SAT visualisation and pedagogical tools.

Keywords
visualisation, pedagogical tools, explaining solving, web-based tools

1. Introduction

The volume and significance of SAT research has continued to increase throughout the past quarter century, yet there has been comparably little focus on educational materials and tools for explaining SAT concepts and the operation of key algorithms.

While some earlier visualisation tools have claimed a secondary educational use, the first dedicated tool for educational SAT solving [1] was only published in 2018. Although other tools have been developed since, this area remains underdeveloped when compared to the wider field of SAT tools and mature solvers.

Educational tools have been aimed at an audience who have limited prior knowledge of SAT fundamentals and solver algorithms, such as undergraduate university students.

These tools all generally implement the older and simpler DPLL algorithm [2], with most also supporting CDCL approaches, which have been adopted by all modern SAT solvers [3] since being introduced in 1999 [4]. They focus on explaining algorithms, by providing tracing and visualisations to demonstrate and explain the steps taken (e.g. variable assignments) when an instance is evaluated to determine satisfiability. By helping to develop an understanding of SAT, these tools hope to stimulate additional interest in the field more generally.

We introduce a new educational SAT tool (SAT-Web) which builds on the concepts introduced by prior tools, while being accessed entirely from within a web-browser. The source code is available on GitHub and also archived on Zenodo [5]. SAT-Web can be accessed online via GitHub Pages.

This use of web technologies to provide easier access is unique compared to previous tools, which all require a downloaded application to be run locally. The tool also provides an instance structure graph visualisation and a dynamic representation of problem formulae using mathematical notation, features not found in prior educational tools.

We examine techniques for visualising instances and discuss previous work on the development of educational tools in Section 2. In Section 3 we present the SAT-Web tool itself, detailing the tool's features, design decisions, and implementation details. After comparing the features of SAT-Web against existing tools in Section 4, we conclude with some ideas for future work in Section 5.

16th Pragmatics of SAT International Workshop, August 11, 2025, Glasgow, Scotland
*Corresponding author.
james@madgwick.xyz (J. Madgwick); m.lester@reading.ac.uk (M. M. Lester)
https://www.reading.ac.uk/computer-science/staff/dr-martin-lester (M. M. Lester)
0009-0009-4900-0515 (J. Madgwick); 0000-0002-2323-1771 (M. M. Lester)

2. Visualisation techniques and Educational tools

There are generally two motivations for visualising instances and the solving process. The primary reason has been to improve understanding of the internal structure of problems; to identify why some instances are easier to solve than others in order to develop optimisations for solvers. A secondary purpose is for explaining SAT and helping those unfamiliar with its concepts through use of visualisations. These techniques include textual representations of the solving process, diagrams, and graphs.

Graphs have always been a popular technique for visualisation of instances, with one of the first examples of visualisation (due to Slater in 2004) using connected graphs drawn using the popular `dot` tool [6]. Shortly afterwards, Sinz and Dieringer developed a dedicated tool called DPvis [7] to gain insight into instance structure and provide "hints on why solving a particular instance is hard or easy". At this time, Brien and Malik described understanding of solver run-time behaviour as "lacking", as it was mostly limited to examining metrics [8]. More recently, Newsham and others have identified the concept of *communities* inside instance structure (partly through visualisations) and directly linked it to solver run time [9].

Over the last 20 years a variety of SAT visualisation tools have been developed; Table 1 shows a summary of visualisations and related features supported by a selection of tools.

Table 1
Visualisations and related features across a selection of tools.

	DPvis [7]	3DVIS [10]	iSAT [11]	SATGraf [12]	LearnSAT [1]	SATViz [13]
Search Tree	X				X	
Variable Interaction Graph	X	X	X	X		X
Implication Graph				X	X	
Animations	X			X		X
Interactive	X	X		X		X

Varieties of graphs include *search tree graphs*, *variable interaction graphs*[1] and *implication graphs*. Others, such as the *common variable graph*, *hypergraph*, and *bipartite variable clause graph*[2] can also be used. However, these are not often used in tools because they lack useful information, represent similar information to other graphs, or for *hypergraphs*, because they are difficult to render visually [10]. It should be noted that with the exception of *hypergraphs*, the conversion from an original problem to the graph is lossy, preventing reconstruction of an original SAT instance [10]. See Figure 1 for examples.

Variable interaction graphs show relationships between variables across all instance clauses. Each variable is represented as a node, with edges drawn between variables which exist in the same clause [10]. These graphs can be drawn without making any assignments or needing any information from a solver; this ease of generation may explain why they are featured in many visualisation tools. They are also helpful for understanding an instance's structure, with some instances exhibiting specific shapes, clusters, and other visual patterns. The graph layout is often computed using the force-directed placement algorithm to generate node positions [7], with 3D graphs using a multi-level extension of this method [10]. The layout can also be arranged to highlight communities of related nodes [12].

Implication graphs are used to show how the value of one variable implies the value of another, either between variables in a single clause or across a subset of clauses. They are primarily used for demonstrating the operation of unit propagation. Implying values for solving directly is possible only for 2SAT problems [14], but they can still be useful to determine an implication within a clause where all but one variable has been assigned.

In addition to a visualisation of the initial problem, some tools provide an animation of the solving process to demonstrate the reduction in complexity caused by variable elimination. This is implemented

[1]Sometimes called *variable instance graphs* [9].
[2]Also known as a *bipartite graph* [13] or *factor graph* [10].

using either a built-in solver or by communicating with an external solver [12]. Other tools use a textual output to show a log of events, such as assignment and backtracking [15]. The most advanced tools include playback features, such as a pause or rewind [7]. Occasionally tools feature plots of metrics gathered during the solving process, such as decision depth, number of implications and details of learned conflict clauses [8].

Search trees are a form of graph representing the sequence of variable assignments, beginning from a root node and branching downwards as variables are assigned. They are effective for intuitively visualising backtracking as moving upwards through branches in the tree. Assignments implied by unit propagation can be shown as additional nodes within the tree [1]. Learnt clauses in CDCL can also be shown [16], although this creates a busy diagram.

While a few early visualisation tools (such as DPvis) were presented as having an educational aspect, most tools were targeted at only SAT researchers. Only more recently have tools with a purely educational purpose been developed.

LearnSAT is arguably the most prominent educational tool, being built specifically for the purpose of explaining and teaching SAT solving [1]. It uses a command line interface and can export search trees and implication graphs. The implementation uses Prolog with built in solvers for DPLL and CDCL, with a tutorial and example problems provided. The recent pedagogical tool EduSAT [17] is supplied as a Python library with a DPLL solver and offers an *understanding through coding* approach using Jupyter notebooks. The similarly recent Interactive SAT Tracer (SAT-IT) [15] is a Java app with a graphical user interface providing a trace of the solving process and details of conflict analysis. Users can step through every part of the process using buttons and load problems using the standard DIMACS CNF format, which is used for the provided example problems.

The educational tools described above all run locally, requiring downloads and additional runtime dependencies to be installed on the user's system before they can be used. With the exception of SAT-IT, they also require prior knowledge of specific programming languages and lack a simple user interface. The authors of EduSAT [17] and SAT-IT [15] specifically highlight these downsides of LearnSAT.

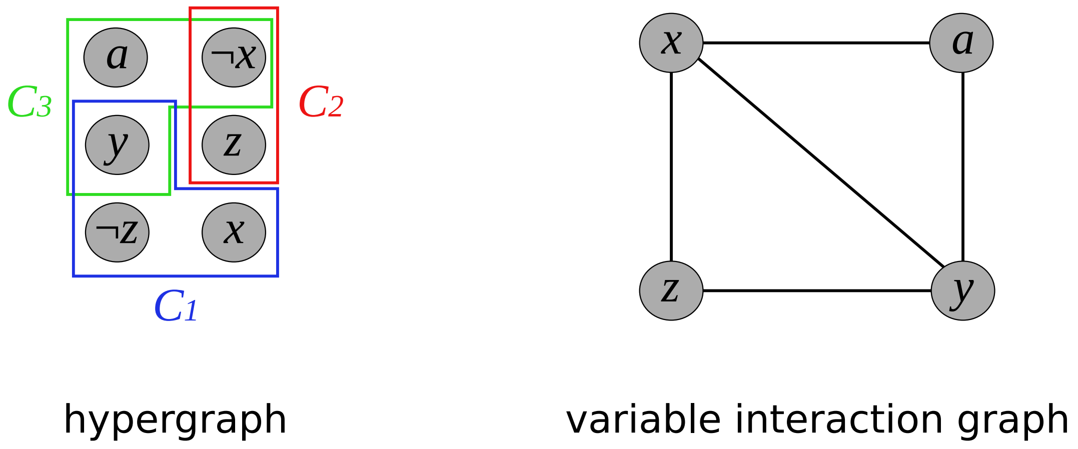


Figure 1: Hypergraph and variable interaction graph for the SAT instance $S = (C1, C2, C3) = (\{y, \neg z, x\}, \{a, \neg x, y\}, \{\neg x, z\})$. Based on examples by Sinz [10].

3. Design and implementation

As described above, the relatively new field of educational SAT tools lacks an application which can easily be accessed and used without requiring additional setup. We therefore decided to implement a new tool to address these limitations. This section covers the feature selection, design, and implementation of SAT-Web, our browser based educational SAT tool.

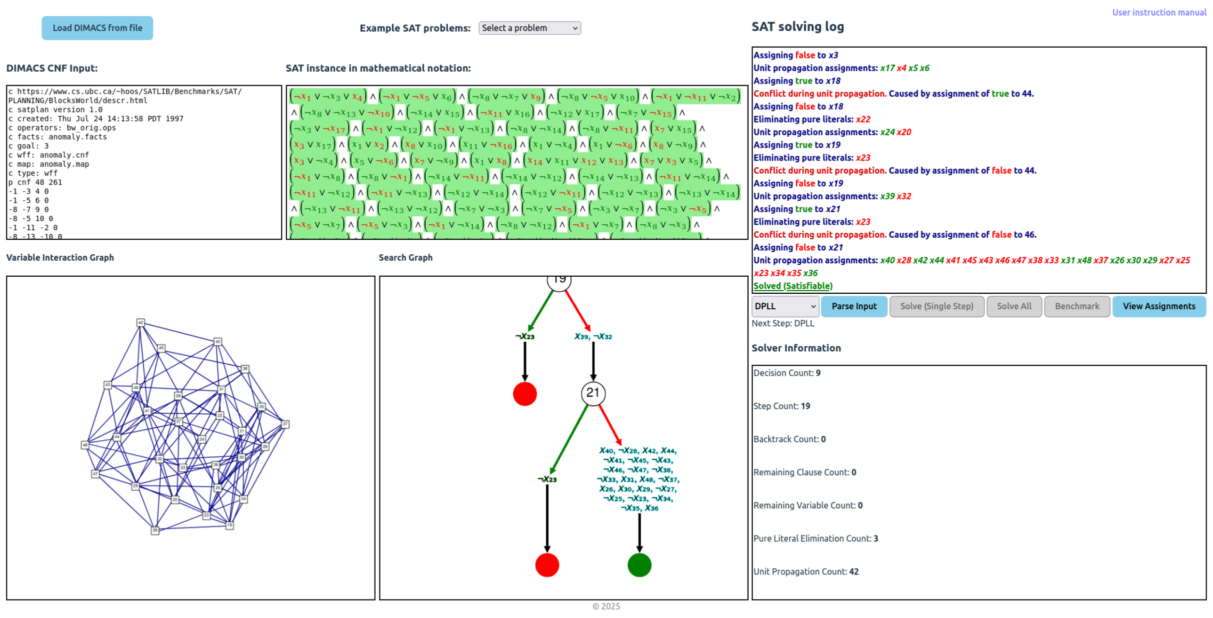
The tool is built as a single page application (SPA) style interactive web app, using a layout similar to SAT-IT with all components visible on the screen at the same time. Users can access the tool simply by navigating to a URL where the site is hosted, without any downloads or runtimes being required. Similar to existing tools, a panel based layout is used with separate spaces set aside for each feature.

Figure 2 shows the web app interface. It has the following components: CNF input area (using DIMACS format), example SAT problem selection, solver control panel, log for solver tracing, instance representation using mathematical notation, solver information panel, search tree graph visualisation, and variable interaction graph visualisation.

The *CNF input* area allows manual entry of problems and loading problems directly from a local file on the user's computer. The *example SAT problem selector* provides problems ranging from very simple (solvable by hand), to some well known more complex problems which cannot be solved by backtracking solvers in a reasonable time. This selection allows users to get started without needing to find any problems themselves. The *mathematical notation panel* (Figure 3) uses colour to indicate the currently assigned boolean value of literals and clauses within the current problem. The *variable interaction graph visualisation* (Figure 5) can be zoomed, panned, and nodes in the graph can be moved around. These features are also present in the *search graph visualisation* (Figures 6 and 7). The *solver control panel* controls all aspects of the app, such as parsing CNF input, selecting algorithms, and viewing final variable assignments. The *solver log* (Figure 4) displays traces explaining the internal operations of the solver as the user steps through a problem, with colour used to emphasise assignments and conflicts. The *solver information panel* displays details such as how many times backtracking has occurred and the number of assignments made by unit propagation or pure literal elimination. Most components have an associated explanatory tooltip box, shown when hovering over the component or its heading.

SAT-Web operates entirely as a client-side web app without any remote server components. This decision was made to simplify deployment and use, allowing the app to be hosted using a basic web-server. This also accelerated the software development process by removing the requirement to write an API for client-server interaction. A downside to this approach is all code needs to run in the user's browser, which significantly reduces the selection of programming languages and libraries compared to a client-server app.

The *Svelte* JavaScript web framework (used with the *Vite* frontend build tool) provides the backbone of the app, controlling all user input and UI events. All the features listed above form individual Svelte components. The *Cytoscape.js* [18] library is used for providing the visualisations. This library is aimed at an academic audience and has a specific focus on graphs, with good support for force-directed layout algorithms and trees.



```
c https://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/
PLANNING/BlocksWorld/descr.html
c satplan version 1.0
c created: Thu Jul 24 14:13:58 PDT 1997
c operators: bw_orig.ops
c facts: anomaly.facts
c goal: 3
c wff: anomaly.cnf
c map: anomaly.map
c type: wff
p cnf 48 261
-1 -3 4 0
-1 -5 6 0
-8 -7 9 0
-8 -5 10 0
-1 -11 -2 0
-8 -13 -10 0
```

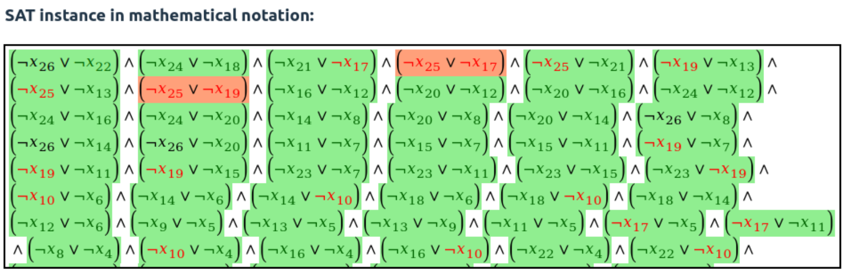
$(\neg x_1 \lor \neg x_3 \lor x_4) \land (\neg x_1 \lor \neg x_5 \lor x_6) \land (\neg x_8 \lor \neg x_7 \lor x_9) \land (\neg x_8 \lor \neg x_5 \lor x_{10}) \land (\neg x_1 \lor \neg x_{11} \lor \neg x_2)$
$\land (\neg x_8 \lor \neg x_{13} \lor \neg x_{10}) \land (\neg x_{14} \lor x_{15}) \land (\neg x_{11} \lor x_{16}) \land (\neg x_{12} \lor x_{17}) \land (\neg x_7 \lor \neg x_{15}) \land$
$(\neg x_3 \lor \neg x_{17}) \land (\neg x_1 \lor \neg x_{12}) \land (\neg x_1 \lor \neg x_{13}) \land (\neg x_8 \lor \neg x_{14}) \land (\neg x_8 \lor \neg x_{11}) \land (x_7 \lor x_{15}) \land$
$(x_3 \lor x_{17}) \land (x_1 \lor x_2) \land (x_8 \lor x_{10}) \land (x_{11} \lor \neg x_{16}) \land (x_1 \lor \neg x_4) \land (x_1 \lor \neg x_6) \land (x_8 \lor \neg x_9) \land$
$(x_3 \lor \neg x_4) \land (x_5 \lor \neg x_6) \land (x_7 \lor \neg x_9) \land (x_1 \lor x_8) \land (x_{14} \lor x_{11} \lor x_{12} \lor x_{13}) \land (x_7 \lor x_3 \lor x_5) \land$
$(\neg x_1 \lor \neg x_8) \land (\neg x_8 \lor \neg x_1) \land (\neg x_{14} \lor \neg x_{11}) \land (\neg x_{14} \lor \neg x_{12}) \land (\neg x_{14} \lor \neg x_{13}) \land (\neg x_{11} \lor \neg x_{14}) \land$
$(\neg x_{11} \lor \neg x_{12}) \land (\neg x_{11} \lor \neg x_{13}) \land (\neg x_{12} \lor \neg x_{14}) \land (\neg x_{12} \lor \neg x_{11}) \land (\neg x_{12} \lor \neg x_{13}) \land (\neg x_{13} \lor \neg x_{14})$
$\land (\neg x_{13} \lor \neg x_{11}) \land (\neg x_{13} \lor \neg x_{12}) \land (\neg x_7 \lor \neg x_3) \land (\neg x_7 \lor \neg x_5) \land (\neg x_3 \lor \neg x_7) \land (\neg x_3 \lor \neg x_5) \land$
$(\neg x_5 \lor \neg x_7) \land (\neg x_5 \lor \neg x_3) \land (\neg x_1 \lor \neg x_{14}) \land (\neg x_8 \lor \neg x_{12}) \land (\neg x_1 \lor \neg x_7) \land (\neg x_8 \lor \neg x_3) \land$

```
Assigning false to x3
Unit propagation assignments: x17 x4 x5 x6
Assigning true to x18
Conflict during unit propagation. Caused by assignment of true to 44.
Assigning false to x18
Eliminating pure literals: x22
Unit propagation assignments: x24 x20
Assigning true to x19
Eliminating pure literals: x23
Conflict during unit propagation. Caused by assignment of false to 44.
Assigning false to x19
Unit propagation assignments: x39 x32
Assigning true to x21
Eliminating pure literals: x23
Conflict during unit propagation. Caused by assignment of false to 46.
Assigning false to x21
Unit propagation assignments: x40 x28 x42 x44 x41 x45 x43 x46 x47 x38 x33 x31 x48 x37 x26 x30 x29 x27 x25
x23 x34 x35 x36
Solved (Satisfiable)
```

Next Step: DPLL

Decision Count: **9**
Step Count: **19**
Backtrack Count: **0**
Remaining Clause Count: **0**
Remaining Variable Count: **0**
Pure Literal Elimination Count: **3**
Unit Propagation Count: **42**

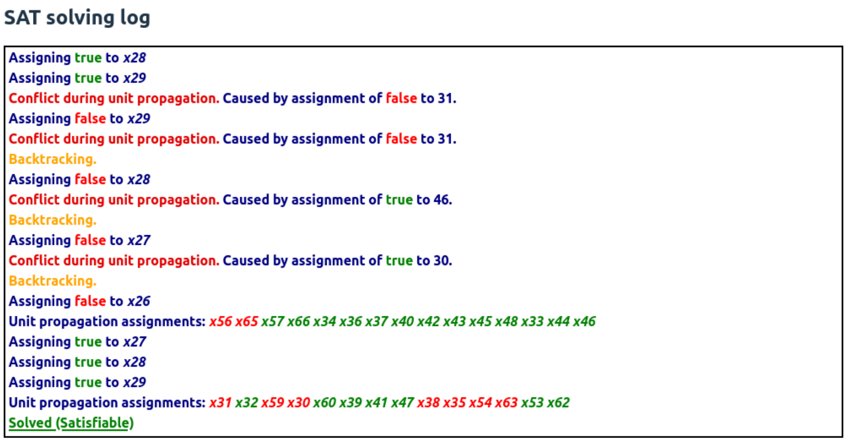
Figure 2: The SAT-Web interface after an example problem has been loaded and solved successfully.



SAT instance in mathematical notation:

$(\neg x_{26} \vee \neg x_{22}) \wedge (\neg x_{24} \vee \neg x_{18}) \wedge (\neg x_{21} \vee \neg x_{17}) \wedge (\neg x_{25} \vee \neg x_{17}) \wedge (\neg x_{25} \vee \neg x_{21}) \wedge (\neg x_{19} \vee \neg x_{13}) \wedge$
$(\neg x_{25} \vee \neg x_{13}) \wedge (\neg x_{25} \vee \neg x_{19}) \wedge (\neg x_{16} \vee \neg x_{12}) \wedge (\neg x_{20} \vee \neg x_{12}) \wedge (\neg x_{20} \vee \neg x_{16}) \wedge (\neg x_{24} \vee \neg x_{12}) \wedge$
$(\neg x_{24} \vee \neg x_{16}) \wedge (\neg x_{24} \vee \neg x_{20}) \wedge (\neg x_{14} \vee \neg x_{8}) \wedge (\neg x_{20} \vee \neg x_{8}) \wedge (\neg x_{20} \vee \neg x_{14}) \wedge (\neg x_{26} \vee \neg x_{8}) \wedge$
$(\neg x_{26} \vee \neg x_{14}) \wedge (\neg x_{26} \vee \neg x_{20}) \wedge (\neg x_{11} \vee \neg x_{7}) \wedge (\neg x_{15} \vee \neg x_{7}) \wedge (\neg x_{15} \vee \neg x_{11}) \wedge (\neg x_{19} \vee \neg x_{7}) \wedge$
$(\neg x_{19} \vee \neg x_{11}) \wedge (\neg x_{19} \vee \neg x_{15}) \wedge (\neg x_{23} \vee \neg x_{7}) \wedge (\neg x_{23} \vee \neg x_{11}) \wedge (\neg x_{23} \vee \neg x_{15}) \wedge (\neg x_{23} \vee \neg x_{19}) \wedge$
$(\neg x_{10} \vee \neg x_{6}) \wedge (\neg x_{14} \vee \neg x_{6}) \wedge (\neg x_{14} \vee \neg x_{10}) \wedge (\neg x_{18} \vee \neg x_{6}) \wedge (\neg x_{18} \vee \neg x_{10}) \wedge (\neg x_{18} \vee \neg x_{14}) \wedge$
$(\neg x_{12} \vee \neg x_{6}) \wedge (\neg x_{9} \vee \neg x_{5}) \wedge (\neg x_{13} \vee \neg x_{5}) \wedge (\neg x_{13} \vee \neg x_{9}) \wedge (\neg x_{11} \vee \neg x_{5}) \wedge (\neg x_{17} \vee \neg x_{5}) \wedge (\neg x_{17} \vee \neg x_{11})$
$\wedge (\neg x_{8} \vee \neg x_{4}) \wedge (\neg x_{10} \vee \neg x_{4}) \wedge (\neg x_{16} \vee \neg x_{4}) \wedge (\neg x_{16} \vee \neg x_{10}) \wedge (\neg x_{22} \vee \neg x_{4}) \wedge (\neg x_{22} \vee \neg x_{10}) \wedge$

Figure 3: Mathematical notation component. Literals are coloured based on their resultant boolean value, with clauses highlighted based on if they are satisfied or not.



SAT solving log

```
Assigning true to x28
Assigning true to x29
Conflict during unit propagation. Caused by assignment of false to 31.
Assigning false to x29
Conflict during unit propagation. Caused by assignment of false to 31.
Backtracking.
Assigning false to x28
Conflict during unit propagation. Caused by assignment of true to 46.
Backtracking.
Assigning false to x27
Conflict during unit propagation. Caused by assignment of true to 30.
Backtracking.
Assigning false to x26
Unit propagation assignments: x56 x65 x57 x66 x34 x36 x37 x40 x42 x43 x45 x48 x33 x44 x46
Assigning true to x27
Assigning true to x28
Assigning true to x29
Unit propagation assignments: x31 x32 x59 x30 x60 x39 x41 x47 x38 x35 x54 x63 x53 x62
Solved (Satisfiable)
```

Figure 4: SAT tracing log component showing entries after solving the `ii8a1.cnf` problem [19].

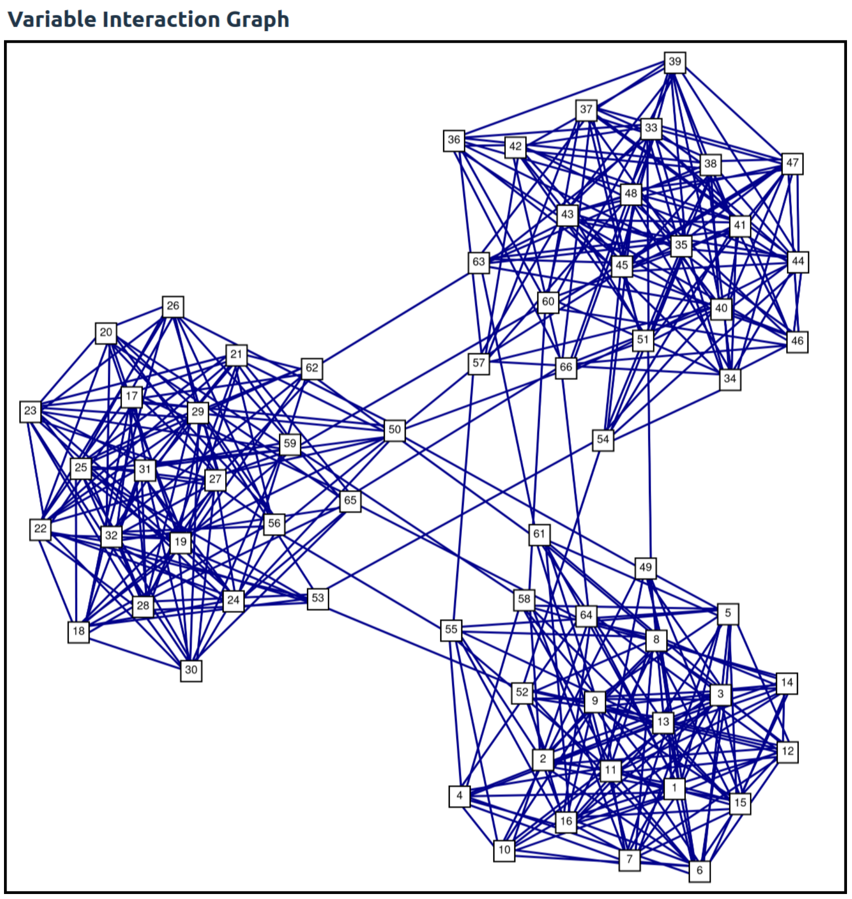


Figure 5: Variable interaction graph generated before solving has started for the `ii8a1.cnf` problem [19].

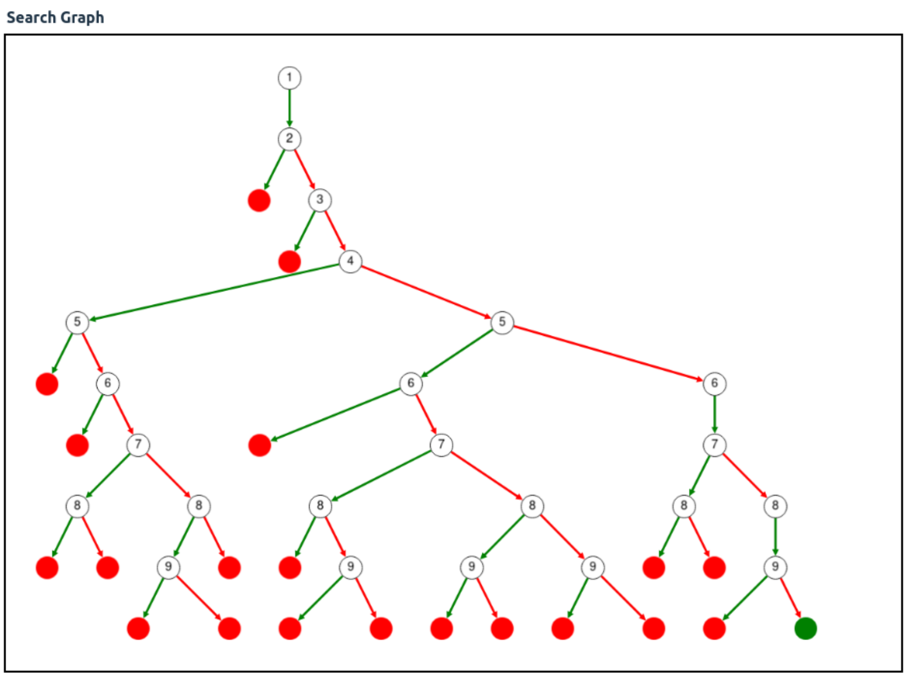


Figure 6: Search graph component showing a small problem solved using backtracking. The green node indicates that satisfiability was achieved, red nodes represent conflicts.

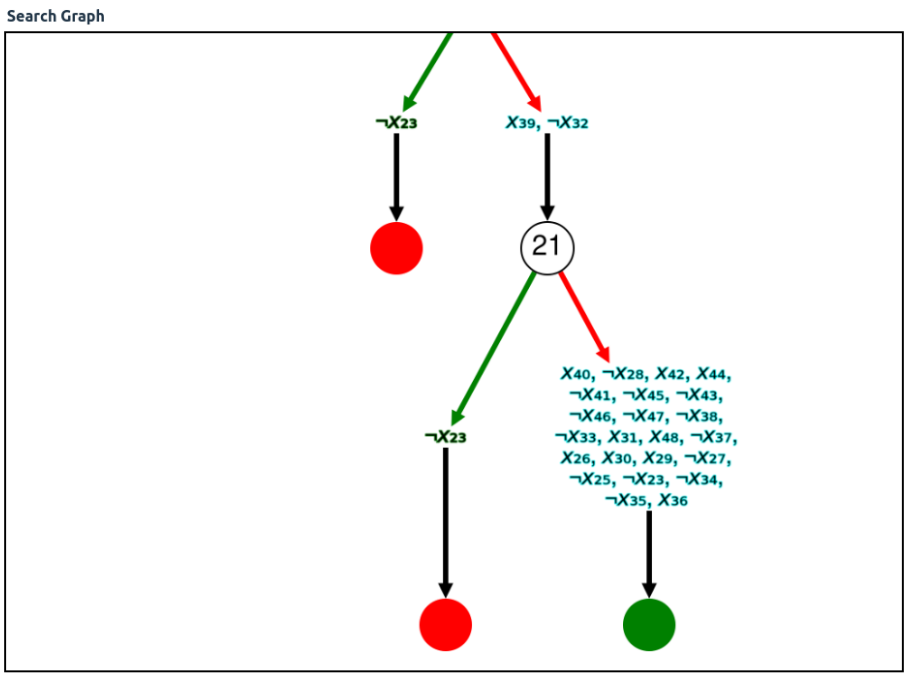

Search Graph
$\neg x_{23}$
$x_{39}, \neg x_{32}$
21
$\neg x_{23}$
$x_{40}, \neg x_{28}, x_{42}, x_{44}, \neg x_{41}, \neg x_{45}, \neg x_{43}, \neg x_{46}, \neg x_{47}, \neg x_{38}, \neg x_{33}, x_{31}, x_{48}, \neg x_{37}, x_{26}, x_{30}, x_{29}, \neg x_{27}, \neg x_{25}, \neg x_{23}, \neg x_{34}, \neg x_{35}, x_{36}$

Figure 7: Search graph component showing part of the tree produced when solving a problem using DPLL. Assignments made by unit propagation have a cyan outline, with pure literal elimination assignments indicated by a green outline.

While using an existing SAT solver within the app was considered, this was ruled out because all well known solvers are written using compiled languages (e.g. C, C++) which can only be used in a browser with *WebAssembly* [20]. This would have greatly increased complexity, coupled with the modifications required to existing solvers for extracting traces [7]. Instead of reusing existing solver code, backtracking and DPLL solver implementations were written using *TypeScript*. The solver code for DPLL uses the repeating loop approach [15], rather than a recursive function [10] because this is more suited to running a single iteration of the algorithm before pausing solving to render visualisations.

Tracing represents the tracking of internal activities within a solver algorithm. To store and handle traces an event architecture was developed, where an event is a single item of solver activity. For example, a variable assignment and the detection of a conflict are both types of event in this architecture. Events are added to an event log in order of occurrence as the user steps through the solving process. All possible solver activity has an event representation; event objects can also contain complex object types. Information is extracted from events and used by the UI and visualisations, e.g. entries in the *solver log* correspond with an event or multiple related events.

The *variable interaction graph* component is rendered by processing instance clauses to generate an array of Cytoscape elements, with all duplicate links removed as only one edge is drawn between each pair of variables in the graph. When first drawn, the graph is formatted using the *cose* Cytoscape layout (a force-directed physics layout algorithm). The nodes and edges shown are then updated with every step of the solver algorithm. The visual styling applied to nodes and edges was based on the variable interaction graphs produced by DPvis [7]. Figure 5 shows the panel after an example problem has been loaded.

The *search graph* component uses the *dagre* Cytoscape layout, which provides a hierarchical directed acyclic graph. It shows all decisions and variable assignments made while solving an instance. Cytoscape allows for configurable styling based on the class of an element or on the contents of the element object itself. This was used to apply dynamic styles mimicking the concise appearance of the search trees presented by Fichte and others [16]; compared to the more verbose appearance of the trees used in LearnSAT [1]. Cytoscape elements in the graph are generated using events, with each event type associated with an edge or node, except for backtracking events which are used to track the current position in the tree.

4. Comparison

Compared with existing educational SAT tools, SAT-Web provides a unique feature set. A high-level comparison of the features of existing tools and SAT-Web is shown in Table 2. The existing tools described previously mostly relied on command line input and manually writing scripts. LearnSAT was

Table 2
Comparison of educational SAT tool features.

	SAT-IT [15]	LearnSAT [1]	EduSAT [17]	SAT-Web
Built in examples	X	X	X	X
Textual output	X	X	X	X
Mathematical notation				X
Search tree/graph		X		X
Implication graph		X		
Variable interaction graph				X
Backtracking solver	X	X		X
DPLL solver	X	X	X	X
CDCL solver	X	X		
Graphical User Interface	X			X
Usable without download				X

the only tool to provide visualisation outputs, but only in the form of image files placed into a separate folder, as it uses a terminal interface. All existing tools also require additional runtime dependencies to be installed before being downloaded and run locally. Representing a problem using mathematical notation is a unique feature of SAT-Web, not found in any previous tool. While solver performance is a less important aspect of educational tools, testing showed the DPLL implementation in SAT-Web to be least 10 times quicker than SAT-IT when solving problems from the AIM benchmark set [19].

5. Future Work and Conclusion

While SAT-Web contains the primary educational components of previous tools, there are a variety of additional concepts which could be explored to enhance the tool's usefulness. In particular, making the practical applications of SAT more apparent by demonstrating the link between a real world problem and the SAT instance encoding it. Visualising problems in their original format is demonstrated in Clingraph [21], where a Sudoku grid is filled up interactively as variable assignments are made to an Answer Set Programming problem representing the puzzle. A more innovative concept involves transforming instances into a visual maze [22], where the path through the maze represents a set of variable assignments and a path to the exit shows satisfiability.

We plan to improve SAT-Web by adding further features and functionality. The most useful of these would be support for the CDCL algorithm, bringing SAT-Web in line with most other tools. As the existing components are not sufficient to explain the operation of CDCL, this would require the addition of at least an annotated implication graph to demonstrate visually how learnt clauses are formed. The tabular representation used in [23] for explaining CDCL *trails* could also be considered. The tool's basic step by step animation could be enhanced by adding an option to run a set number of steps together with a specified delay between each, creating a smooth animation – similar to the functionality in DPvis [7]. Another feature could be the ability to export video files, as used by SATViz [13]. It would also be useful to make existing visualisations more interactive, such as collapsible nodes in search trees and clicking on graph nodes to show further information. 3D rendering could be added as an option for the variable interaction graph, as shown in [10], although this would require a suitable JavaScript library supporting real time 3D.

Considering that many of the early visualisation tools can no longer be downloaded, or are reliant on legacy runtime dependencies, we hope to inspire SAT researchers to consider using the web as a platform for building future SAT visualisation tools.

To conclude, we have presented a new pedagogical SAT tool which is the first of its kind to use web technologies. Our tool compares favourably with previous tools, supporting a unique set of features. Easy access to the tool online will hopefully allow students to understand SAT more efficiently and generate further interest in the field of educational SAT tools.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] M. Ben-Ari, LearnSAT: A SAT solver for education, J. Open Source Softw. 3 (2018). doi:10.21105/joss.00639.
[2] M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving, Commun. ACM 5 (1962) 394–397. doi:10.1145/368273.368557.
[3] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, IOS Press, 2009.
[4] J. Marques-Silva, K. Sakallah, GRASP: a search algorithm for propositional satisfiability, IEEE Trans. Comput. 48 (1999) 506–521. doi:10.1109/12.769433.

[5] J. Madgwick, M. Lester, SAT-Web: A web based educational SAT visualisation tool (software source code), 2025. doi:10.5281/zenodo.15801988.

[6] A. Slater, Visualisation of satisfiability problems using connected graphs, Technical Report, Australian National University, 2004. URL: https://users.cecs.anu.edu.au/~andrews/problem2graph/problem2graph.html.

[7] C. Sinz, E.-M. Dieringer, DPvis: a tool to visualize the structure of SAT instances, in: Theory and Applications of Satisfiability Testing: 8th Int. Conf., 2005, p. 257–268. doi:10.1007/11499107_19.

[8] C. Brien, S. Malik, Understanding the dynamic behavior of modern DPLL SAT solvers through visual analysis, in: Formal Methods in Computer Aided Design (FMCAD), San Jose, CA, USA, 2006, pp. 49–50. doi:10.1109/FMCAD.2006.35.

[9] Z. Newsham, V. Ganesh, S. Fischmeister, G. Audemard, L. Simon, Impact of community structure on SAT solver performance, in: Theory and Applications of Satisfiability Testing: 17th Int. Conf., 2014, pp. 252–268. doi:10.1007/978-3-319-09284-3_20.

[10] C. Sinz, Visualizing SAT instances and runs of the DPLL algorithm, J. Autom. Reason. 39 (2007) 219–243. doi:10.1007/s10817-007-9074-1.

[11] E. Orbe, C. Areces, G. Infante-López, iSat: Structure visualization for SAT problems, in: Logic for Programming, Artificial Intelligence, and Reasoning: 18th Int. Conf., 2012, p. 335–342. doi:10.1007/978-3-642-28717-6_26.

[12] Z. Newsham, W. Lindsay, V. Ganesh, J. H. Liang, S. Fischmeister, K. Czarnecki, SATGraf: Visualizing the evolution of SAT formula structure in solvers, in: Theory and Applications of Satisfiability Testing: 18th Int. Conf., 2015, pp. 62–70. doi:10.1007/978-3-319-24318-4_6.

[13] T. Holzenkamp, K. Kuryshev, T. Oltmann, L. Wäldele, J. Zuber, T. Heuer, M. Iser, SATViz: Real-time visualization of clausal proofs, in: Pragmatics of SAT: 13th Int. Workshop, 2022. doi:10.48550/arXiv.2209.05838.

[14] G. Kusper, C. Biró, B. Nagy, Resolvable networks — a graphical tool for representing and solving SAT, Mathematics 9 (2021). doi:10.3390/math9202597.

[15] M. Cané, J. Coll, M. Rojo, M. Villaret, SAT-IT: The interactive SAT tracer, in: Catalan Association for Artificial Intelligence: 25th Int. Conf., volume 375 of *Frontiers in Artificial Intelligence and Applications*, 2023, pp. 337–346. doi:10.3233/FAIA230704.

[16] J. K. Fichte, D. L. Berre, M. Hecher, S. Szeider, The silent (r)evolution of SAT, Commun. ACM 66 (2023) 64–72. doi:10.1145/3560469.

[17] Y. Zhao, Z. An, M. Ma, T. Johnson, EduSAT: A pedagogical tool for theory and applications of boolean satisfiability, arXiv e-prints (2023). doi:10.48550/arXiv.2308.07890.

[18] M. Franz, C. T. Lopes, D. Fong, M. Kucera, M. Cheung, M. C. Siper, G. Huck, Y. Dong, O. Sumer, G. D. Bader, Cytoscape.js 2023 update: a graph theory library for visualization and analysis, Bioinformatics 39 (2023). doi:10.1093/bioinformatics/btad031.

[19] DIMACS SAT benchmarks, in: Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993. URL: http://archive.dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf/.

[20] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, J. Bastien, Bringing the web up to speed with webassembly, SIGPLAN Not. 52 (2017) 185–200. doi:10.1145/3140587.3062363.

[21] S. Hahn, O. Sabuncu, T. Schaub, T. Stolzmann, Clingraph: A system for ASP-based visualization, Theory Pract. Log. Program. 24 (2024) 533–559. doi:10.1017/S147106842400005X.

[22] N. Manthey, An A-Maze-ing SAT solving visualization, Technical Report, Dresden University of Technology, 2015. URL: https://iccl.inf.tu-dresden.de/web/Techreport3031/en.

[23] D. E. Knuth, The Art of Computer Programming: Satisfiability, Volume 4, Fascicle 6, Pearson Ed. Inc., 2015.