DPVIS – A Tool to Visualize the Structure of SAT Instances

Carsten Sinz and Edda-Maria Dieringer

Symbolic Computation Group, WSI for Computer Science
University of Tübingen, 72076 Tübingen, Germany
{sinz, dieringe}@informatik.uni-tuebingen.de

Abstract. We present DPVIS, a Java tool to visualize the structure of SAT instances and runs of the DPLL (Davis-Putnam-Logemann-Loveland) procedure. DPVIS uses advanced graph layout algorithms to display the problem's internal structure arising from its variable dependency (interaction) graph. DPVIS is also able to generate animations showing the dynamic change of a problem's structure during a typical DPLL run. Besides implementing a simple variant of the DPLL algorithm on its own, DPVIS also features an interface to MiniSAT, a state-of-the-art DPLL implementation. Using this interface, runs of MiniSAT can be visualized—including the generated search tree and the effects of clause learning. DPVIS is supposed to help in teaching the DPLL algorithm and in gaining new insights in the structure (and hardness) of SAT instances.

1 Introduction

Although SAT is an NP-complete problem, there are many real-world instances that can be solved surprisingly fast by modern (mainly DPLL-based) solvers. The typical explanation for this phenomenon that can be found in the literature is that those instances are equipped with some kind of internal (and sometimes hidden) "structure" that makes these problems tractable. The term "structure", due to its vagueness, leaves much room for interpretation, though, and it remains unclear how this structure manifests itself and how it could be exploited. We have therefore proposed a visualization approach [1], that is supposed to deliver some hints on why solving a particular instance is hard or easy.

Our approach is based on the problem's *(variable) interaction graph* [2], which is supposed to appropriately reflect (at least part of) the problem's structure. In a SAT instance given by a set S of clauses over a set X of propositional variables, the interaction graph's vertices are the instance's propositional variables, and variables occurring together in any clause of S are connected by an edge. Thus, if two variables are connected, this indicates that assigning a truth value to one of the connected variables has the potential to determine the truth value of the other (thus they *interact*).

DPVIS (see Fig. 1) reads problems in DIMACS CNF format and shows their variable interaction graph. The interaction graph is laid out using algorithms that are known to reflect graph clustering and symmetry especially well [3]. Moreover, DPVIS visualizes changes of the interaction graph during the run of a DPLL algorithm. This includes visualizing the effects of unit propagation (Boolean constraint propagation) on the interaction graph as well as showing the search tree generated by the DPLL algorithm.

We expect DPVIS to be a useful tool for examining the structure of SAT instances—especially in generating hypotheses on what makes an instance tractable. Moreover, we believe that DPVIS can be a valuable tool in teaching the DPLL algorithm. As DPVIS can also display learned clauses, non-chronological back-jumping, and search restarts, it encompasses most features that can be found in a modern implementation of the DPLL procedure.

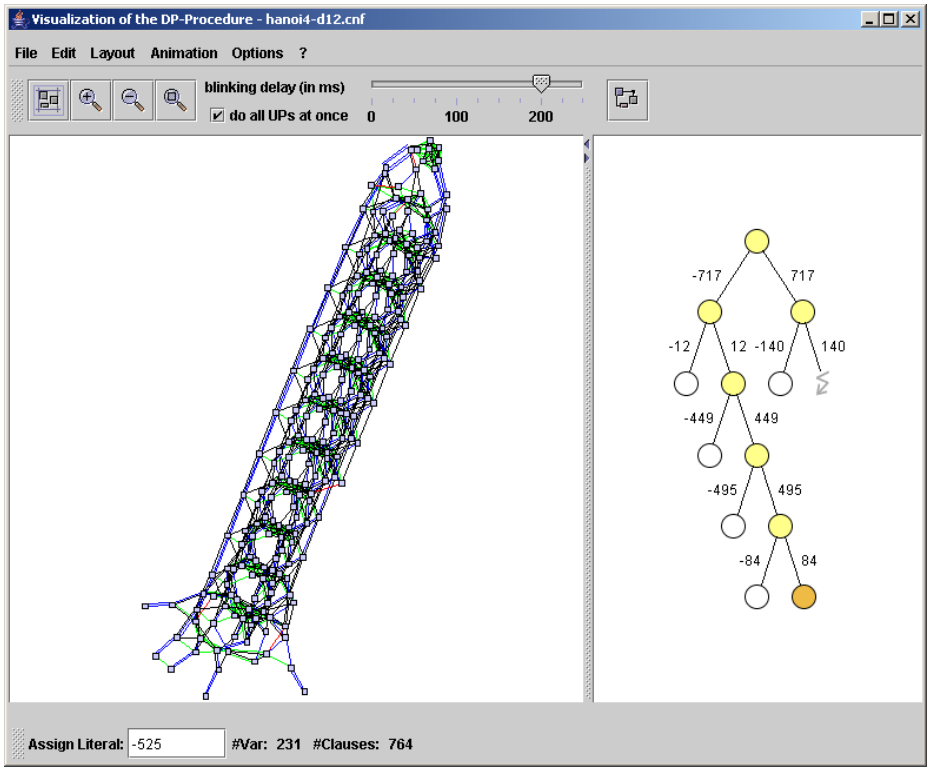

Fig. 1. DPVIS tool showing a visualized SAT instance (**hanoi4** of the DIMACS Benchmark Collection): The variable interaction graph is shown on the left and a manually generated, partial search tree on the right.

2 Theoretical Background

2.1 SAT Instances as Graphs

To transform a SAT instance represented as a set S of clauses into a (directed or undirected) graph $G = (V, E)$ with vertex set V and edge set E a number of methods have

been suggested [2, 4–6]. We have decided to use a variant of Rish and Dechter's *interaction graphs*[1] for our visualization, as they put an emphasis on variable dependencies.

The interaction graph is an undirected graph, where the vertex set V is the set of variables of S and $\{x, y\} \in E$ if and only if there is a clause $c \in S$ that contains both variables x and y. Note, that this is a lossy representation, as the signs of literals in clauses are ignored. For the purpose of structural analysis and visualization we consider this not a disadvantage, however.

For our visualization experiments we have decided to use a refined variant of interaction graphs, in which 2-clauses (i.e. clauses containing exactly two literals) are represented as visually emphasized directed or undirected arcs in the graph. We make the following distinction:

1. A clause with two positive literals is shown as a red, double-ended arrow.
2. A clause with one positive and one negative literal, say $(\neg x \vee y)$, is written as a blue arrow from variable x to variable y, indicating the implication direction.
3. A clause with two negative literals is written as a green, double-ended arrow.

All other clauses are displayed without highlighting, i.e. as black edges between the involved variables. Moreover, we allow duplicate edges in our graphs; thus, e.g., two clauses both involving variables x and y result in a double edge between the corresponding vertices.

Special treatment of 2-clauses is motivated by their importance for tractability: A problem consisting only of 2-clauses is solvable in linear time [4]. Moreover, experiments with random instances from the $(2 + p)$-model (problems with a fraction of p 3-clauses and $(1 - p)$ 2-clauses) indicate that random instances with up to 40 percent of 3-clauses (i.e. $p \leq 0.4$) might be computationally tractable [7].

2.2 Graph Layout

The *graph layout* or *graph drawing* problem consists of generating a geometric representation of a graph in two or three dimensions. Nodes have to be positioned in the Euclidean space while optimizing certain layout properties like minimal number of edge crossings, uniform edge length, reflection of inherent symmetry, etc [3, 8].

Different algorithms are available for graph layout, a prominent one being the *spring embedder model* of Eades [9] or its close relative, the *force-directed placement* algorithm of Fruchterman and Reingold [3]. Both are known to produce layouts that reflect symmetry convincingly.[2] The physical model used by the spring-embedder assumes metal springs of a certain length attached between each pair of connected nodes. The springs impose attractive and repellent forces on the nodes depending on their current distance in the layout. The layouter attempts to minimize the sum of all affecting forces by iteratively repositioning the nodes.

[1] Interaction graphs are also known as *primal graphs*. Slater [6] calls them *co-occurrence-of-variables-graphs*.

[2] We have considered other existing layouts as well, like hierarchical or orthogonal layout, but found them not being equally suitable for our purpose.

DPVIS builds upon the commercially available graph layout package yFiles of yWorks (http://www.yworks.com), from which it uses the *Organic Layouter* and *Smart Organic Layouter*, both of which implement force-directed placement algorithms.

3 DPVIS Functionality

Upon start-up, DPVIS reads a SAT instance in DIMACS CNF format from a file, performs unit propagation, and displays the instance's variable ineraction graph with variables (the graph's nodes) placed randomly (on the left part of the screen) and an empty DPLL search tree (on the right). The user can then choose a graph layouter and DPVIS computes a visually more attractive layout. Thereafter, variables can be set to *true* or *false* (by clicking on the graph's nodes with either the left or right mouse button), or an automatic DPLL run can be started. In the first case, after setting a variable, unit propagation is automatically initiated.

A detailed description of all available display and animation options is given in the following paragraphs.

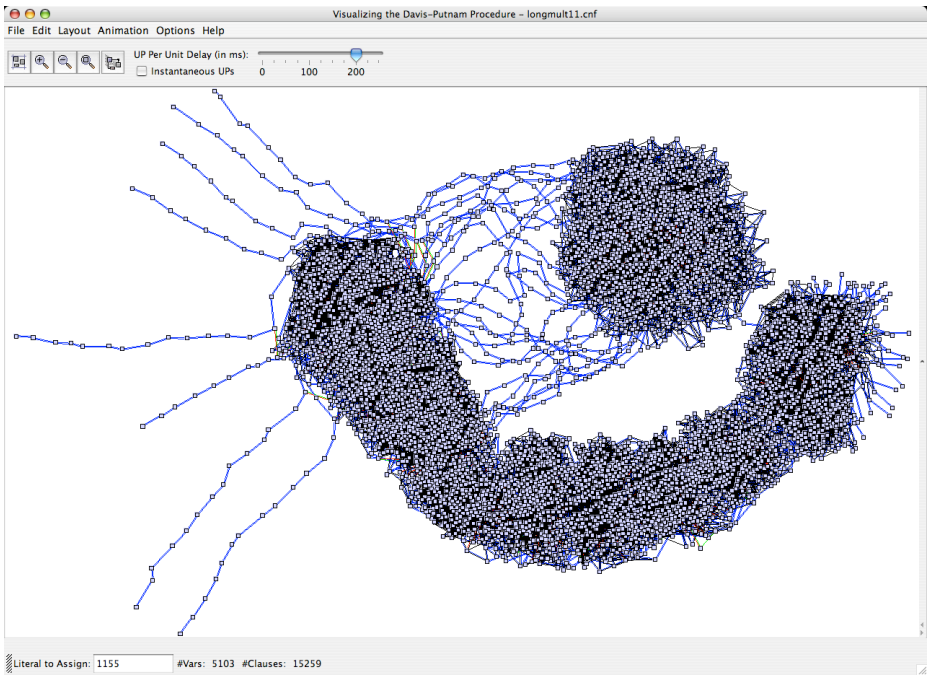

Fig. 2. Larger layout example: Bounded model-checking instance **longmult11** (see Sec. 4 for a reference) with 5103 variables and 15259 clauses laid out using DPVIS' Smart Organic Layouter.

3.1 Visualizing Internal Structure

To display the static problem structure and show the effects of setting individual variables and subsequent unit propagation, DPVIS offers these features:

- **Two Different Layout Algorithms:** Graph layouts can be computed using two different force-directed layout algorithms, each equipped with a set of additional parameters, e.g. for controlling preferred edge length or graph compactness.
- **Zooming the Interaction Graph**: After the layout is computed, the user can zoom into the interaction graph to focus on an area of special interest. The user may also search for variables and center the view on them, or optimally fit the graph into the available display area.
- **Setting Variables to *true* resp. *false***: By clicking on a variable, the user can set a variable to *true* (by clicking the left mouse button) or *false* (right mouse button). Each setting of a variable also extends the search tree by a new leaf.
- **Performing Unit Propagation**: Unit propagation is automatically initiated after setting of a variable: first, all variables affected by unit propagation are highlighted; then they are removed one by one (or at once, if this option is selected) and the interaction graph is updated accordingly, resulting in an animated view of unit propagation.

An example of a typical layout obtained with DPVIS is displayed in Fig. 2.

3.2 Animating DPLL Runs

DPVIS also allows generating and visualizing complete runs of the DPLL procedure. There are three possibilities to generate such a run. They differ in the way in which the case-splitting literal L is selected in the DPLL algorithm (see Fig. 3).

```
boolean DPLL(ClauseSet S)
{
  while (S contains a unit clause {L}) {
    delete clauses containing L from S      // unit-subsumption
    delete ¬L from all clauses in S         // unit-resolution
  }
  if (∅ ∈ S) return false                   // empty clause?
  if (S = ∅) return true                    // no clauses?
  choose a literal L occurring in S         // case-splitting on L
  if (DPLL(S ∪ {{L}})) return true          // first branch
  else if (DPLL(S ∪ {{¬L}})) return true    // second branch
  else return false
}
```

Fig. 3. Pseudo-code of the basic DPLL algorithm.

1. **Manual Selection of Case-Distinction Variable**: The user selects the case-splitting variable by clicking on a node in the variable interaction graph or by entering the variable index into a field at the bottom of the screen. This option delivers enough flexibility for the user to experiment with his own (manually generated) variable selection heuristics, e.g. one that tries to generate independent subproblems.

2. **Simple Variable Selection Heuristics**: DPVIS also implements two simple variable selection heuristics: one that randomly selects the case-splitting variable, and one that is based on counting literal occurrences: the variable with the maximal number of occurrences (making no distinction between positive and negative occurrences) is selected.
3. **Interface to Modern DPLL Implementation**: An interface to external SAT-solvers is also contained in DPVIS. This allows DPVIS to read traces of DPLL runs from solvers like zChaff or MiniSAT[3]. Currently, the only interface available is that to MiniSAT (see Fig. 4). Using this interface, DPLL traces containing information about case-splitting, unit propagation, learned clauses and back-jumps can be animated and analyzed with DPVIS. The user can choose between playback mode, in which—starting with an initial interaction graph layout—the program trace is presented to the user (resembling a movie). Alternatively, the user can employ single-step mode, in which after each step he or she may compute a new graph layout.

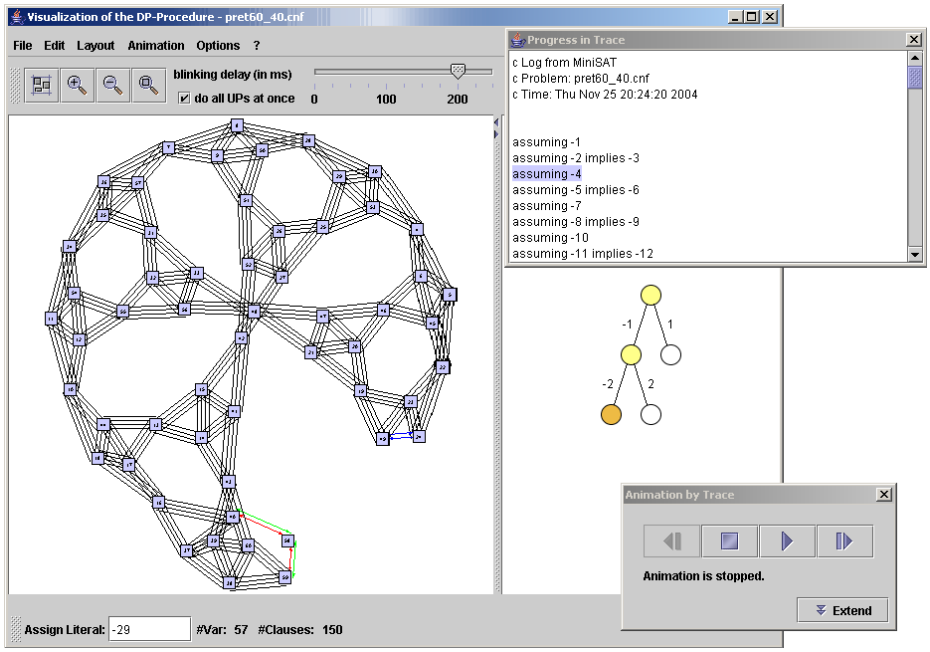

Fig. 4. DPVIS-interface to MiniSAT. Program traces (top right window) produced by MiniSAT are read in by DPVIS, and can be played back. Play-back is controlled by an animation console (bottom right window).

[3] SAT-solvers that are intended to interface with DPVIS have to be slightly modified in order to output the appropriate program traces.

Independent of which mode was used to generate the DPLL search tree, the following additional options are available:

- **Free navigation in the search tree**: At each point in the DPLL program trace, the user can stop the trace and navigate to any point in the thitherto existing search tree.
- **Re-compute layout at any time and each node of the search tree**: When the play-back of the animation trace is stopped (or interrupted), different layout algorithms and parameter settings may be applied either to the current state (see Fig. 5) or (by combining it with the search tree navigation feature) to already visited search nodes. Different states during search can thus easily be compared.

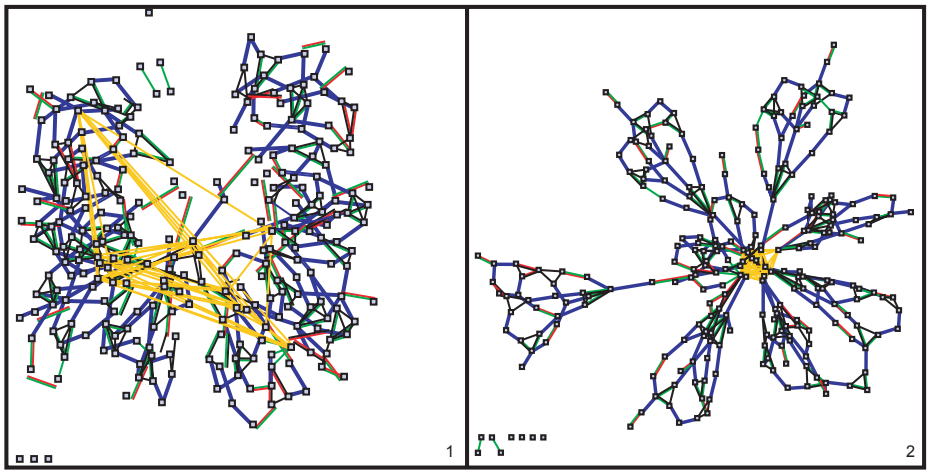

Fig. 5. At each point during play-back of a DPLL trace, the layout of the interaction graph may be re-computed: after having learned some clauses (left diagram, yellow arcs; instance **ssa0432-003** of the DIMACS Benchmark Collection), re-computing the graph layout shows the locality of the learned lemmas (right diagram).

4 Two Sample Applications

We now briefly present two examples, how DPVIS can effectively be employed. The first is concerned with the analysis of DPLL search spaces, the second deals with so-called *top-level assignments*.

4.1 Comparing Search Spaces

The intention of this experiment is to compare search spaces resulting from random problem instances with search spaces generated by “real-world” problems. We use the following SAT instances for our experiments:

longmult1: encoding of a bounded model checking (BMC) problem (equivalence of output bit 0 of two hardware multiplier designs)

uuf50-0188: unsatisfiable uniform random 3-SAT instance with 50 variables and a clause-variable-ratio of $\alpha = 4.36$ (i.e. near the satisfiability threshold)

ssa2670-141-d7: instance from test-pattern generation (checks for *single-stuck-at* fault), where seven randomly selected variables have been fixed (to random Boolean values in order to reduce the search space to a size comparable with the other instances)

bw_large.b-d8: encoding of an AI planning problem, with eight randomly selected variables fixed to random Boolean values (as with ssa2670-141-d7)

The **longmult1**-instance can be obtained from www.cs.cmu.edu/~modelcheck/bmc.html, all other instances can be downloaded from www.satlib.org. All instances are unsatisfiable, additional statistical information about these problem instances can be found in Table 1.

Table 1. Characteristic numbers for some SAT instances (all instances unsatisfiable), including MiniSAT results (number of conflicts and decisions) and balancedness-coefficients.

Instance	#Vars	#Clauses	#Confl.	#Dec	B-Coeff.
longmult1	631	1611	5	20	3.7021
uuf50-0188	50	218	145	186	1.0611
ssa2670-141-d7	753	1619	41	253	7.3231
bw_large.b-d8	889	9949	4	19	3.0603

The table's columns show—from left to right—the instance name, the number of variables and the number of clauses occurring in the problem (after having performed unit propagation), the number of conflicts and decisions produced during a MiniSAT run, and the B-coefficient, which is an indicator of the balancedness of the search tree of the instance (as generated by MiniSAT). More exactly, for a search tree T the B-coefficient B is defined by $B := h/\lceil \mathrm{ld}(c) \rceil$, where h is the height of the search tree (the length of the longest path from the root), c is the number of conflicts produced by MiniSAT (i.e. the number of leaves in the search tree), and $\lceil \mathrm{ld}(c) \rceil$ is the dual logarithm of c rounded up to the next larger integer, i.e. the height of a balanced tree with c nodes. Thus, a B-coefficient of 1 indicates a balanced tree, whereas a large coefficient stands for an unbalanced, degenerate tree.

The search spaces for all four problem instances are shown in Fig. 6. The random 3-SAT instance (Fig. 6-2) reveals an almost balanced search tree, whereas the other, "real-world" instances (1, 3, 4) possess more or less degenerate trees. This observation is also reflected by the B-coefficient of the real-world instances being much higher than that of the random 3-SAT instance.

We suppose that hard SAT instances have low B-coefficients whereas easy instances possess high ones. Further assuming that the search space is uniform (in the sense of having similar B-coefficients in different parts), the B-coefficient may be used to estimate the run-time of a SAT solver after having processed only a small fraction of the search space.

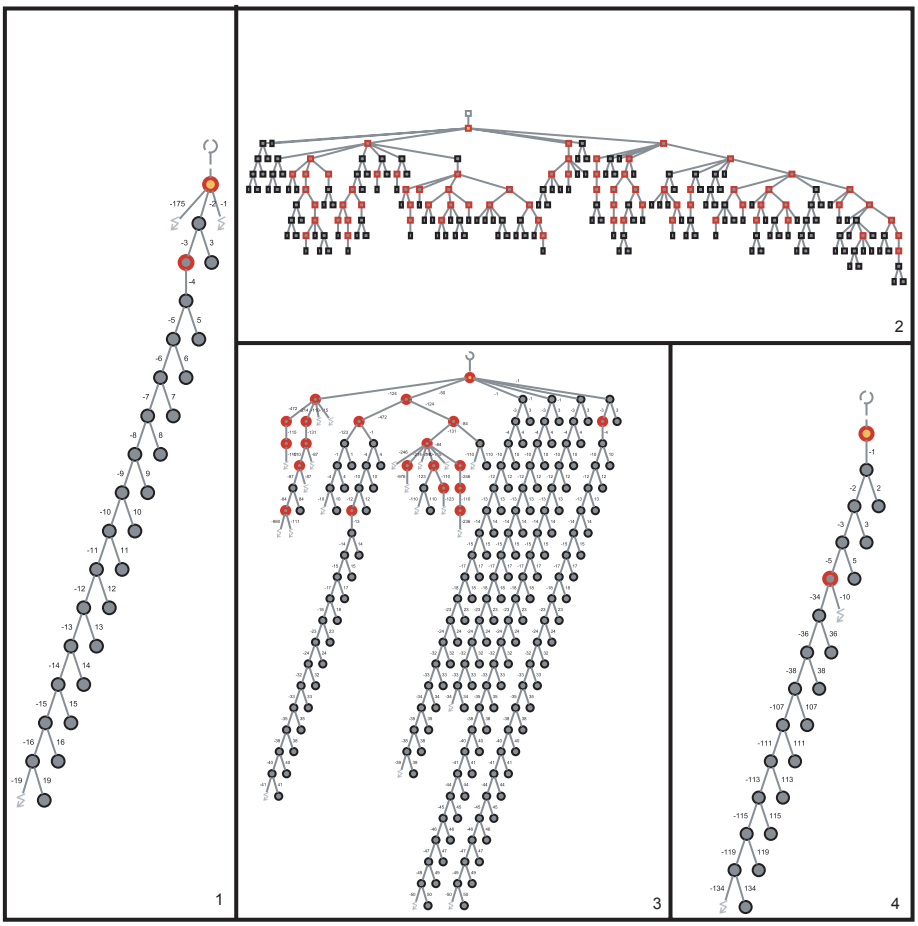

Fig. 6. Search trees for different problem instances: 1: bounded model checking (longmult1); 2: random 3-SAT (uuf50-0188); 3: test-pattern generation (ssa2670-141_d7); 4: planning problem (bw_large.b-d8).

We expect that by making experiments with DPVIS other notions similar to that of the B-coefficient might come up in the future, hopefully allowing for a better distinction between hard and easy problem instances.

4.2 Analyzing the Effect of Top-Level Assignments

Eén and Sörensson introduced the notion of *top-level assignments* for unit-clauses that are learned during search [10]. Each top-level assignment (TLA) fixes the value of a variable for the whole instance and thus indicates that in each solution of the instance the TLA-variable must have that fixed value.[4]

In a further experiment with DPVIS, we compared the number of TLAs produced by different SAT instances during a MiniSAT run. Some results of these experiments are presented in Table 2.

Table 2. Number and percentage of top-level assignments (TLAs) for some SAT instances.

Instance	#Vars	#Clauses	#TLAs	#TLA/#Vars
longmult1	631	1611	151	23.93%
uuf50-0125	50	218	13	26.00%
ssa2670-141-d7	753	1619	336	44.62%
bw_large.b-d8	889	9949	233	26.21%

Although the number of top-level assignments is considerable in all cases, a more elaborate study with more problem instances from each problem class showed that the number of TLAs is typically much higher for real-world problems than for random 3-SAT problems (29.5% vs. 11.0% in our experiments).

Comparing (using DPVIS) the interaction graphs of the original instances with those where the detected TLAs were added, revealed a considerable simplification for the real-world instances, whereas the structure of random 3-SAT instances remained mainly unchanged (see Fig. 7).

5 Conclusion and Related Work

We presented DPVIS, a tool to visualize the structure of SAT instances and to display DPLL search trees. We see a twofold purpose for our visualization tool: First, it may help in building hypotheses on why problem instances are hard or easy. We have indicated in Sec. 4 how experiments in this direction might look like. Second, we suggest using DPVIS in teaching the basic DPLL algorithm and recent extensions of it like clause learning and restarts. By having an integrated view on both the problem structure (via its variable interaction graph) and the search tree, students can obtain a good intuition how the DPLL method works.

[4] The notion of a top-level assignment is closely related to that of a *backbone variable*[11].

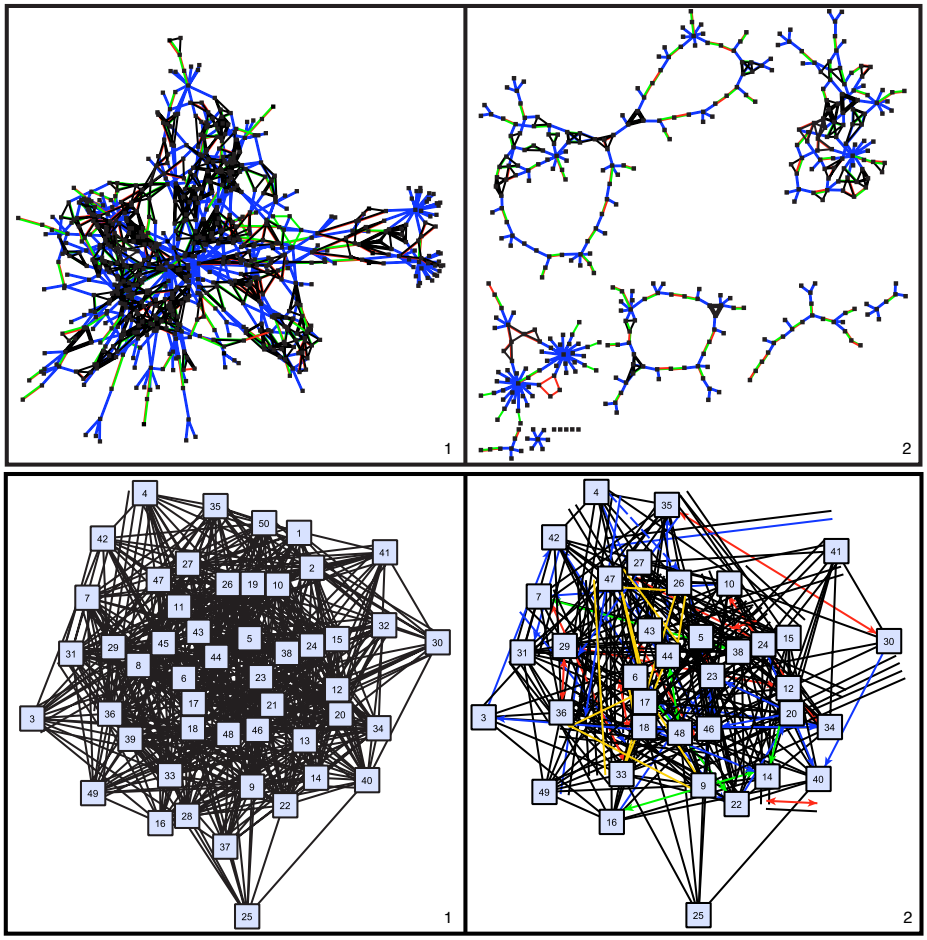

Fig. 7. Comparing interaction graphs before (on the left) and after (on the right) having added TLAs (top-level assignments): Real-world instance **ssa2670-141-d7** (shown on top) reveals a considerable simplification and decomposition into independent components after having added TLAs. The random 3-SAT instance **uuf50-0125** (shown below) also possesses a considerable amount of TLAs, but exhibits no such simplification.

Future experiments should include the analysis of an instance's component structure (resp. its decay into independent components) and the examination of long implication chains, as suggested in [1]. Moreover, in order to handle large graph layouts more conveniently, it would be desirable to have additional tools, e.g. for grouping sets of variables or merging them into a single node.

The only work on visualization of SAT instances that we are aware of is that of Slater [6] and preliminary work by Selman [12]. Slater uses different graph translation techniques (interaction graph, co-occurrence of literals and further ones) and visualizes the resulting graphs with the GraphVis software package from AT&T. However, the hierarchical layouter he uses is not as efficient in revealing symmetries as force-directed placement algorithms are. Selman uses a specialized three-dimensional layouter to display variable interaction graphs. In his approach nodes with different degree are placed on different "height levels", and nodes with the same degree are equally distributed on circles growing with the number of nodes that have to be placed on them.

Availability: DPVIS is available both as on-line version (Java Applet) and stand-alone application (Java Application including MiniSAT) from http://www-sr.informatik.uni-tuebingen.de/~sinz/DPvis.

References

1. Sinz, C.: Visualizing the internal structure of SAT instances (preliminary report). In: Proc. of the 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT 2004), Vancouver, Canada (2004)
2. Rish, I., Dechter, R.: Resolution versus search: Two strategies for SAT. J. Automated Reasoning **24** (2000) 225–275
3. Fruchterman, T., Reingold, E.: Graph drawing by force-directed placement. Software – Practice and Experience **21** (1991) 1129–1164
4. Aspvall, M., Plass, M., Tarjan, R.: A linear-time algorithm for testing the trurh of certain quantified boolean formulas. Information Processing Letters **8** (1979)
5. Park, T., Van Gelder, A.: Partitioning methods for satisfiability testing on large formulas. Information and Computation **162** (2000) 179–184
6. Slater, A.: Visualisation of satisfiability problems using connected graphs (2004) `http://rsise.anu.edu.au/~andrews/problem2graph`.
7. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: Determining computational complexity from characteristic 'phase transitions'. Nature **400** (1999) 133–137
8. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.: Algorithms for automatic graph drawing: An annotated bibliography. Computational Geometry **4** (1994) 235–282
9. Eades, P.: A heuristic for graph drawing. Congressus Numerantium **42** (1984) 149–160
10. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. of the 6th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT 2003), Springer (2003) 502–518
11. Singer, J., Gent, I., Smaill, A.: Backbone fragility and the local search cost peak. Journal of Artificial Intelligence Research **12** (2000) 235–270
12. Selman, B.: Algorithmic adventures at the interface of computer science, statistical physics, and combinatorics. In: Proc. of the 10th Intl. Conf. on Principles and Practice of Constraint Programming (CP 2004), Springer (2004) 9–12