J Autom Reasoning (2007) 39:219–243
DOI 10.1007/s10817-007-9074-1

Visualizing SAT Instances and Runs of the DPLL Algorithm

Carsten Sinz

Received: 1 February 2006 / Accepted: 1 December 2006 / Published online: 26 July 2007
© Springer Science + Business Media B.V. 2007

Abstract SAT-solvers have turned into essential tools in many areas of applied logic like, for example, hardware verification or satisfiability checking modulo theories. However, although recent implementations are able to solve problems with hundreds of thousands of variables and millions of clauses, much smaller instances remain unsolved. What makes a particular instance hard or easy is at most partially understood – and is often attributed to the instance's *internal structure*. By converting SAT instances into graphs and applying established graph layout techniques, this internal structure can be visualized and thus serve as the basis of subsequent analysis. Moreover, by providing tools that animate the structure during the run of a SAT algorithm, dynamic changes of the problem instance become observable. Thus, we expect both to gain new insights into the hardness of the SAT problem and to help in teaching SAT algorithms.

Keywords SAT instance · DPLL procedure

1 Motivation

Progress in SAT-solving has been tremendous over the past years. Problems that were completely out of reach 10 years ago can now be handled with success [8]. Especially in hardware verification, SAT solvers (and the associated method of bounded model checking [15]) have made a glittering success story [44, 65], and in many areas replaced the previously predominant BDD-based model checkers [13]. But in other combinatorial problem domains such planning [40], configuration [41], satisfiability modulo theory [5], or software verification [64, 70] SAT-solvers have played out their strengths.

C. Sinz (✉)
Institute for Formal Models and Verification, Johannes Kepler University Linz,
Altenbergerstr. 69, 4040 Linz, Austria
e-mail: carsten.sinz@jku.at

Although the advent of new methods and optimized implementations made SAT-solvers very successful on these "real-world" instances, large classes of SAT instances remain (e.g., unsatisfiable random 3-SAT problems with a clause-variable-ratio near the phase transition point [14, 32, 42, 45]) on which these solvers fail – even on instances that are considerably smaller (by four to five orders of magnitude). This is not surprising, as SAT is an NP-complete problem [55]. The reasons for this phenomenon, however, are at best partially understood, and besides results that are often not directly applicable to large industrial instances (e.g., treewidth), no a priori criteria are available to decide whether an instance is tractable. In fact, determining the hardness of a particular instance may be even more complex than solving the SAT instance itself.

The standard argument found in the literature to explain this dichotomy is that real-world instances are equipped with some kind of internal (and sometimes hidden) structure that makes these problems tractable [66]. The term "structure," because of its vagueness, leaves much room for interpretation, though, and it remains unclear how this structure manifests itself and could be exploited. Among the methods proposed are randomization [35], clause learning [43], symmetry and component detection [1, 10, 16, 56], and tree decomposition [52, 63]. The first two techniques (randomization and clause learning) can be found in most implementations of modern SAT-solvers for real-world instances today [26, 34, 47]. Concepts explaining the boundary between tractability and intractability (besides graph-based properties such as treewidth) include backbone variables [46] and backdoor sets [69]. Although highly valuable from both an epistemological and a practical point of view, these concepts do not deliver an a priori criterion to directly "read off" the computational hardness of a given instance.

We propose a novel, mainly empirical approach in order to shed some light on an instance's internal structure [60]. A major ingredient of our method is the visualization of internal variable dependencies and clause dependencies as they emerge from a graph transformation of the SAT problem. From the graph we hope to be able to derive new criteria for problem hardness.

Different graph representations of SAT instances have been proposed in the literature, including variable interaction graphs [52], resolution graphs (Van Gelder, personal communication, 2005), implication graphs [3], factor graphs [12], and hypergraph variants of these [31, 50].[1] We have decided to use variable interaction graphs and resolution graphs for the visualization approach presented in this paper, as these graphs are both amenable to layout algorithms (as opposed to hypergraphs) and general enough (as opposed to implication graphs). To render graphs, we use existing graph layout algorithms or, more specifically, force-directed placement procedures [24, 29]. These procedures are known to reflect clustering and symmetry – properties that we are interested in – especially well [29].

Besides computing static layouts, we have developed a tool called DPvis that is able to generate animations showing the dynamic change of a problem's structure during a run of the DPLL (Davis–Putnam–Logemann–Loveland) procedure. The

[1] Unfortunately, there is no established nomenclature for these graphs. Szeider [63], for example, uses the name "primal graph" for our variable interaction graphs and the notion "incidence graph" for what we call factor graph; Galesi and Kullmann [30] call resolution graphs "conflict graphs."

DPLL procedure is used (with variations and extensions) by almost all SAT-solver implementations for real-world problems today.

The rest of this paper is organized as follows. First we introduce basic notions of SAT-solving and graph layout. Then we present our implementations DPvis and 3Dvis, which compute two- and three-dimensional graph layouts of SAT instances. We then show results obtained by our visualization and summarize observations we have made. Finally, we report on related work and conclude with a brief discussion of future research.

2 Theoretical Background

In this section we define basic notions of SAT-solving, present different transformations from SAT instances to graphs, and give a short introduction into graph layout algorithms, as far as it is needed here. We also state known results on treewidth and tree decompositions.

2.1 SAT and the DPLL Algorithm

The propositional logic satisfiability problem (SAT) asks whether a formula in conjunctive normal form (CNF) – a SAT instance – is satisfiable, that is, whether there exists an assignment to the variables such that the whole formula evaluates to true. More formally, a SAT instance is defined as follows.

Definition 2.1 Given a finite set X of propositional variables, a SAT instance $S = \{C_1, \ldots, C_m\}$ (over X) is a finite set of clauses, where a clause $C = \{l_1, \ldots, l_k\}$ is a finite set of literals, and a literal l is either a variable $x \in X$ or its negation $\neg x$ (with $x \in X$). We denote the set of literals (over X) by $L := X \cup \neg X = X \cup \{\neg x \mid x \in X\}$.

Logically, a clause $C = \{l_1, \ldots, l_k\}$ is interpreted as the disjunction of its literals, namely, as $(l_1 \vee \cdots \vee l_k)$, and a SAT instance $S = \{C_1, \ldots, C_m\}$ as the conjunction of its clauses, namely, as $C_1 \wedge \cdots \wedge C_m$.

The problem of determining whether a formula in CNF is satisfiable is the classical NP-complete problem [55]. Different algorithms have been proposed to solve it [17, 18, 54], with the Davis–Putnam–Logemann–Loveland (DPLL) algorithm being the one mainly used in practical applications today. We have depicted the basic DPLL algorithm in Fig. 1. It has been extended in various ways in modern imple-

Fig. 1 Pseudo-code of the basic DPLL algorithm (without clause learning)

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

mentations, the most important ones being clause learning and nonchronological backtracking [43], fast Boolean constraint propagation [47], and random restarts [35].

2.2 Graph Representations of SAT Instances

In order to obtain graph representations from SAT instances, a mapping from the latter to the former is needed. As an intermediate step in generating such a mapping, it is useful to consider hypergraphs [31] first, because they allow for a natural, lossless transformation from the propositional logic domain to a graphlike setting (see also [50]).

Definition 2.2 A hypergraph $H = (V, E)$ is a pair of a vertex set V and a set of hyperedges E, where E is a subset of the powerset of V, that is, $e \subseteq V$ for all $e \in E$.

This suggests a natural translation of a SAT instance S to a hypergraph H, by setting the vertex set V of H to the set of literals of S and letting the clauses of S generate the hyperedges of H. We thus obtain $H = (L, S)$ as a natural representation of S as a hypergraph.

Unfortunately, it is hard to render hypergraphs graphically, except for the case where all hyperedges are of size two, in which the hypergraphs coincide with ordinary graphs. Thus, different ways have been proposed in the literature [46, 52, 62] to obtain ordinary graphs by modifying the hypergraph representation in some way. In what follows, we assume a SAT instance consisting of a clause set S over a set of variables X. Figure 2 illustrates these graph representations on an exemplary formula.

Factor graph Factor graphs have been used in the context of SAT solving by Braunstein and Zecchina [12] and Szeider [63] (among others). A factor graph $G_F = (V, E)$ is a bipartite graph, in which the vertex set consists of the union of the instance's clauses and variables; thus $V = X \cup S$. An (undirected) edge is drawn between variable x and clause c if and only if $x \in c$ or $\neg x \in c$. It is obtained from the hypergraph representation $H = (S, L)$ by adding a new vertex for each hyperedge (i.e., for each clause), connecting the new vertex with all elements of the hyperedge, ignoring signs of literals. In a slight variant of the factor graph (called *directed factor graph*), a directed graph is used instead of an undirected one, where the direction of the arc indicates whether the variable occurs positively or negatively in the clause.

Variable interaction graph The variable interaction graph $G_I = (X, E)$ is obtained from the hypergraph $H = (L, S)$ by replacing each hyperedge e^* with a set of ordinary edges, one edge for each pair of elements of e^*, and merging nodes of positive and negative literals for each variable [52]. Thus, an edge $\{x, y\}$ is drawn between two variables x and y if they occur together in at least one clause.

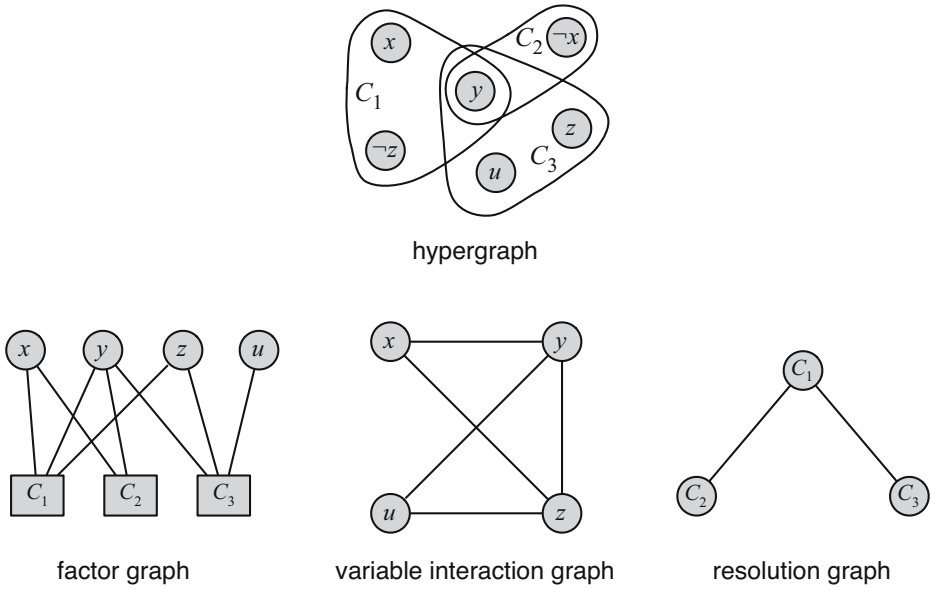

Fig. 2 Different hypergraph and graph representations of the SAT instance $S = (C_1, C_2, C_3) = (\{x, y, \neg z\}, \{\neg x, y\}, \{u, y, z\})$

Resolution graph In the resolution graph $G_R = (S, E)$ an (undirected) edge is drawn between two clauses C_1 and C_2 if and only if there is a variable $x \in X$ such that $x \in C_1$ and $\neg x \in C_2$. Clauses C_1 and C_2 that are adjacent in G_R can thus be resolved. (Note that we also allow tautological resolvents here.)

Whereas directed factor graphs are lossless representations of SAT instances, all of (undirected) factor graphs, variable interaction graphs, and resolution graphs are lossy representations of SAT instances – in the sense that the mapping from SAT instances to graphs is not injective. In the case of factor graphs and variable interaction graphs, signs of literals are ignored (and thus cannot be "reconstructed"), and clauses are not represented uniquely, as the following example shows: the 3-clause $\{x, y, \neg z\}$ and the set of three two-clauses $\{\{\neg x, y\}, \{x, \neg z\}, \{y, z\}\}$ share the same interaction graph and thus cannot be differentiated. For resolution graphs, as parallel edges are merged, the transformation is also not invertible. Lossy representations, however, allow one to abstract away certain properties that are not currently in focus.

In the variable interaction graph, a path from a variable x to a variable y means that assigning a value to x may influence possible assignments to y (e.g., exclude that y is set to true). Conversely, if there is no path from x to y, these variables are independent, which also means that the SAT problem for the components containing x and y, respectively, can be solved independently. In the resolution graph, a path from a clause C to a clause D indicates that literals of C and D may be merged by a series of resolution steps (cf. del Val's no-merge resolution for knowledge compilation [20] or Plaisted and Zhu's ordered semantic hyper-linking [51]).

2.3 Treewidth and Tree Decompositions

Treewidth is a notion that was introduced by Robertson and Seymour in their work on graph minors [53]. The treewidth $\mathrm{tw}(G)$ is a structural parameter of a graph G. It plays an important role in algorithmic graph theory, as many graph algorithms become tractable when restricted to graphs of bounded treewidth.

Treewidth is defined via *tree decompositions*, where a tree decomposition of a graph $G = (V, E)$ is defined as a pair (T, χ), where T is a tree and χ a labeling of the vertices of T by sets of vertices of G. (T, χ) is a tree decompostion of G provided the following conditions hold:

- For every vertex $v \in V$ there is a vetex t in T with $v \in \chi(t)$.
- For every edge $(v, w) \in E$ there is a vertex t in T such that $\{v, w\} \subseteq \chi(t)$.
- For any vertices t_1, t_2, t_3 in T, if t_2 lies on a path from t_1 to t_3, then $\chi(t_1) \cap \chi(t_3) \subseteq \chi(t_2)$.

The *width* of a tree decomposition (T, χ) is defined as the maximum of $|\chi(t)| - 1$ over all vertices t in T. The *treewidth* $\mathrm{tw}(G)$ of G is the minimum width over all its tree decompositions.

For a fixed k, deciding whether a graph has treewidth at most k can be decided in linear time [11]. However, computing the treewidth of a given graph is an NP-hard problem [2].

Tree decompositions can be used to obtain fast algorithms for subclasses of otherwise intractable problems. For SAT, Gottlob et al. have shown, for instance, that clause sets with bounded treewidth of the variable interaction graph are fixed-parameter tractable [36]. Szeider proved that clause sets with bounded treewidth of the factor graph are fixed-parameter tractable [63].

A problem is called *fixed-parameter tractable (FPT)* [23, 49] if its run-time depends exponentially only on the parameter k but is otherwise polynomial, that is, it is $\mathcal{O}(f(k) \cdot l^{\alpha})$, where f is any function, l is the size of the problem instance (for SAT the number of occurring literals), and α is a constant independent of k.

Treewidth is also closely related to bucket elimination algorithms for SAT. Dechter [19, 52], for example, has presented such an algorithm. Zabiyaka and Darwiche have defined the notion of *functional treewidth* [71] and give results of an algorithm based on this notion for circuit benchmarks from the LGSynth93 suite. Narodytska and Walsh [48] have applied an algorithm that exploits structural properties related to treewidth to automotive configuration problems. Array Logic [27], working with decompositions and partial solution tables, is also closely related to tree decompositions and has been applied to configuration problems. Especially in the application area of product configuration, treewidth seems to be a promising concept.

2.4 Graph Layout Algorithms

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, and reflection of inherent symmetry [22, 29].

Different algorithms are available for graph layout, a prominent one being the *spring-embedder model* of Eades [24] or its close relative, the *force-directed placement* algorithm of Fruchterman and Reingold [29]. Both are known to produce layouts that reflect symmetry very well.[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 (i.e., the energy of the whole system) by iteratively repositioning the nodes.

Unfortunately, using only local spring forces is not sufficient to obtain a globally untangled graph [67]. As an example, Walshaw [67] considers a chain of three vertices a, b, and c that are connected by two edges $\{a, b\}$ and $\{b, c\}$. Now, having only local spring forces, a placement where a and c are put at the same location and b one spring length away would be a state of minimal energy and thus considered optimal. However, having a and c put at the same place is typically not regarded as an acceptable layout. On a larger scale, repulsion is necessary to push whole regions that are not immediately connected away from each other. The standard technique [24, 29, 67] to avoid such situations is to add global repellent forces between all vertices. These are typically simulated by a physical model that considers nodes as equally charged particles that push away from each other. A drawback of this approach is that a simplistic algorithm to compute global forces requires quadratic time in the number of vertices. However, there are more advanced algorithms like the Fast Multipole Method (FMM) [37] or the Barnes–Hut-algorithm [4] that can avoid the quadratic behavior – at the cost of a more complicated implementation, though.

3 Implementations

We have developed two tools for visualizing SAT instances: DPvis, which generates two-dimensional layouts of SAT instances and which can also animate runs of the DPLL algorithm, and 3Dvis, which produces three-dimensional layouts.

3.1 DPvis: Visualizing the DPLL Algorithm

DPvis is a Java tool to visualize the structure of SAT instances and runs of the DPLL procedure. It builds on 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. DPvis is also able to generate animations showing the dynamic change of a problem's structure during a DPLL run. Besides implementing a simple variant of the DPLL algorithm on its own, DPvis features an interface to MiniSAT [26], a state-of-the-art DPLL implementation, by which runs of MiniSAT can be visualized – including the generated search tree and the effects of clause learning. A screenshot of DPvis is shown in Fig. 3.

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

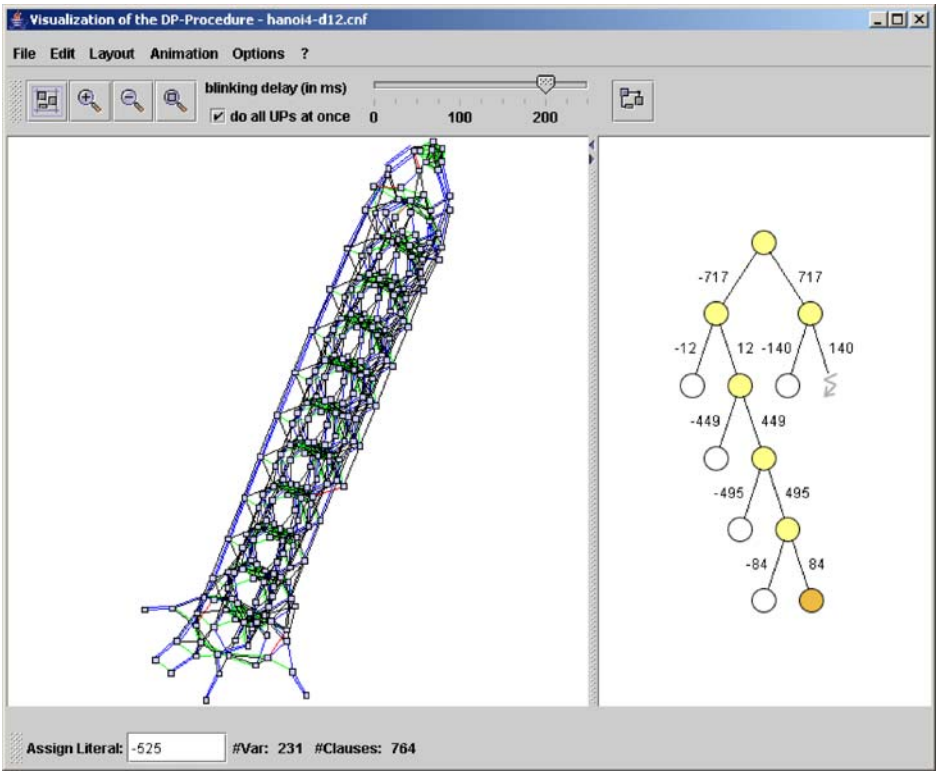

Fig. 3 DPvis tool showing a visualized SAT instance (`hanoi4` of the DIMACS benchmark collection[3]): The variable interaction graph is shown on the left and a manually generated, partial search tree on the right

DPvis uses a refined variant of the variable interaction graph, in which 2-clauses (aka binary clauses, i.e., clauses containing exactly two literals) are represented as visually emphasized directed or undirected arcs in the graph. Different colors are used depending on the signs of the literals occurring in the 2-clause. Special treatment of 2-clauses is motivated by their importance for tractability: a problem consisting only of 2-clauses is solvable in linear time [3]. 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% of 3-clauses (i.e., $p \leq 0.4$) might be computationally tractable [46]. Binary clauses occur to a considerable extent in real-world SAT instances (for example, in hardware verification) as well.

Besides providing two different layout algorithms, DPvis allows zooming into the interaction graph, performing unit propagation, and manually setting variables to true or false. During all these operations, the variable interaction graph is updated to reflect the current partial assignment. Moreover, DPvis allows one to navigate freely

[3]The `hanoi4` instance represents the well-known Towers of Hanoi game with four disks, encoded as a planning problem. Individual actions of the plan as well as their dependencies are clearly visible as linearly arranged clusters in the graph layout.

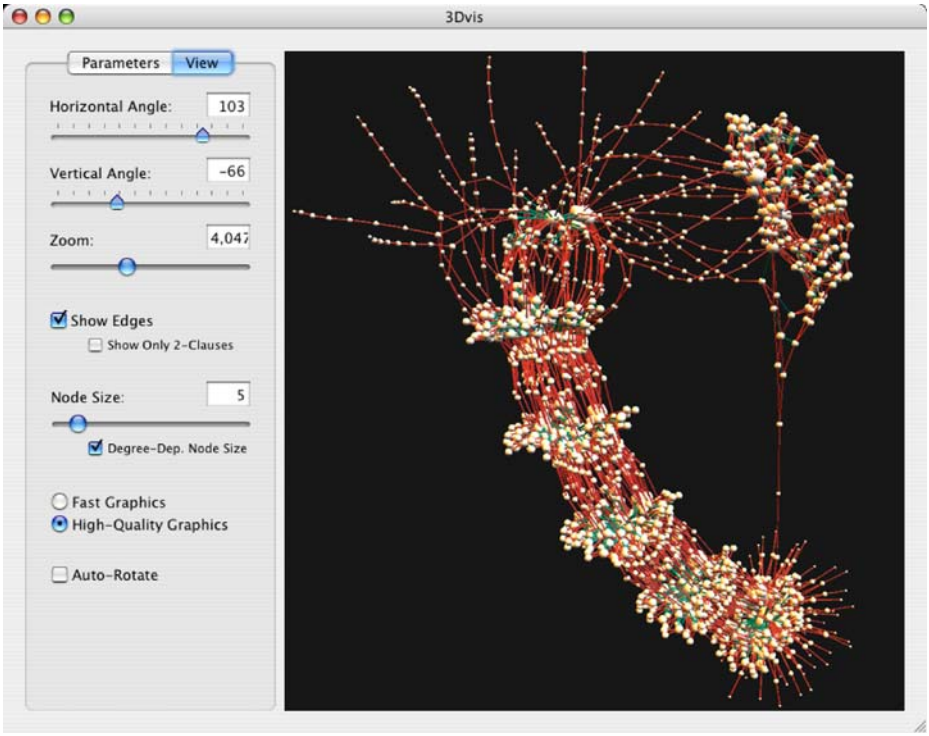

Fig. 4 3Dvis showing an instance on equivalence checking of two hardware multipliers (`longmult5`, available at http://www.cs.cmu.edu/~modelcheck/bmc.html)

in the search tree generated by the DPLL algorithm and thus to compare reduced instances (due to partial variable assignments) at different points of the search tree.

The interface to MiniSAT allows playback of DPLL traces generated by this SAT-solver. Such traces contain information about case splitting, unit propagation, learned clauses, and back-jumps, all of which can be animated and analyzed with the help of DPvis.

DPvis was developed in collaboration with E.-M. Dieringer and was implemented mainly by her. DPvis is available as a Java applet and can be downloaded from http://www-sr.informatik.uni-tuebingen.de/~sinz/DPvis. Details on DPvis are described in a different paper [60].

3.2 3Dvis: Three-Dimensional Graph Layout

Our second implementation, 3Dvis, generates three-dimensional layouts. It implements the multilevel force-directed graph-drawing algorithm by Walshaw [67], is written in C++, and makes use of OpenGL. There is a Linux version of 3Dvis with a simple user interface based on the OpenGL Utility Toolkit (GLUT), and a Mac OS X version with a Cocoa/Objective-C user interface (see Fig. 4).[4]

[4]The Linux version of 3Dvis can be downloaded from http://www-sr.informatik.uni-tuebingen.de/~sinz/3Dvis. The Mac OS X version is available from the author of this paper on request.

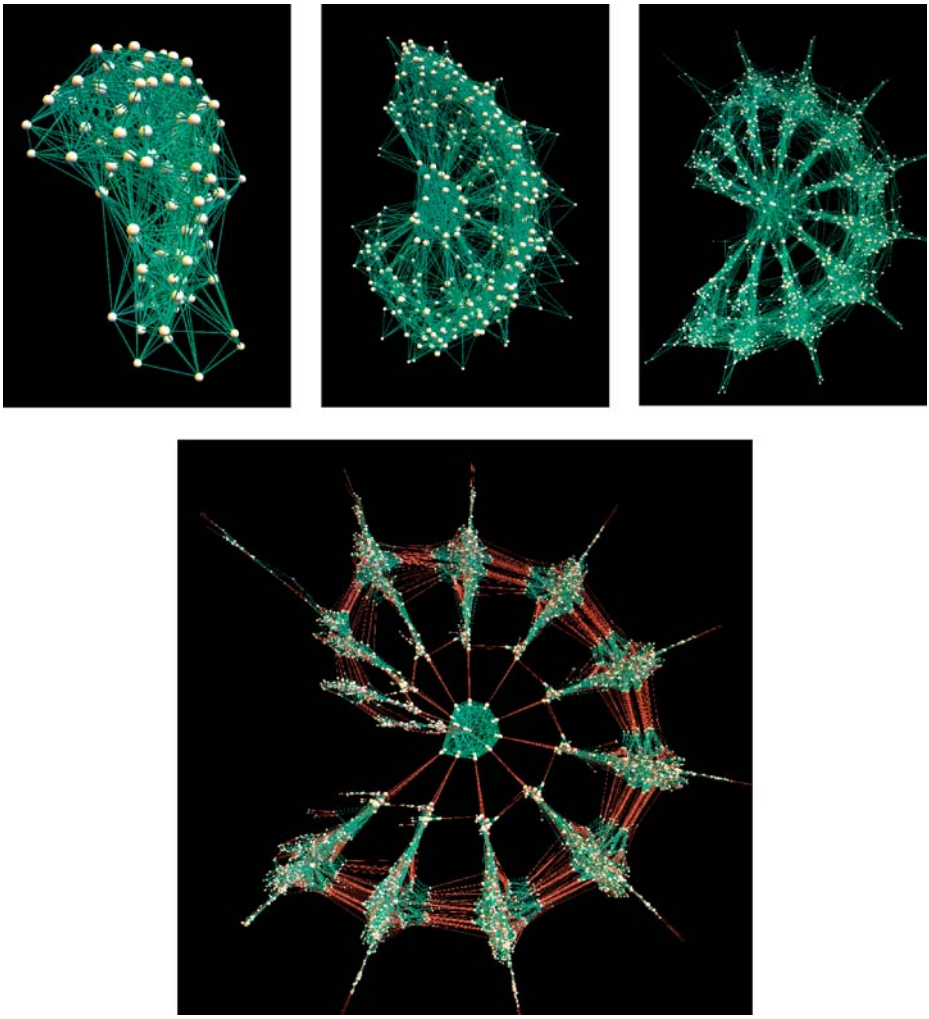

Fig. 5 Layouts computed by the multilevel algorithm of 3Dvis. On the *top row*, layouts of coarsened graphs for SAT instance `bmc-ibm-5` on levels $l = 7, 5, 3$ are shown, the final layout ($l = 0$) is shown on the *bottom*. `bmc-ibm-5` is an instance from bounded model checking [9], and the 12 time steps of the unrolled transition relation are clearly visible as clusters on an outer circle. Adjacent time steps are connected by 2-clauses that interrelate current and next state variables. The last three time steps (*top left*) are smaller because of a bounded cone of influence optimization. The checker automaton, which is connected to all time steps, is clearly visible in the center

In a first step, Walshaw's multilevel algorithm computes a sequence $G_0, \ldots, G_l$ of increasingly coarsened graphs, starting with the initial variable interaction graph G_0 and repeating until the size of the coarsest graph consists of only two nodes. Each coarsening step works by computing a matching (not necessarily perfect), where each vertex is matched with one of its neighbors (such that no vertex has more than one neighbor in the matching, but solitary vertices may remain). Computing such matchings reduces the size of the graph (i.e., the number of vertices) at most by half

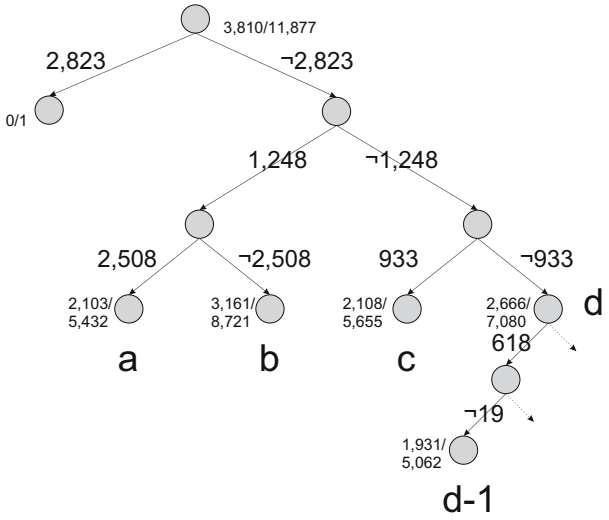

Fig. 6 Search tree of a depth-bounded run of the DPLL algorithm on the `longmult8` benchmark problem

in each coarsening step. So, after approximately $\mathcal{O}(\log_2 |V|)$ steps, the coarsest graph is reached.

Then, in reverse order, layouts are computed, for the coarsest graph first, continuing until a layout for the original graph is obtained. In each layout step, nodes for G_i are initially positioned according to the positions computed for G_{i+1}. Then the force-directed layouter is started on G_i with this approximate placement as initial layout. By computing approximate layouts for coarser graphs first, the whole layout process can be accelerated. Details on Walshaw's algorithm can be found in one of his papers [67]. Figure 5 shows an example of the iterative multilevel layout algorithm in action. On the top, layouts for the coarsened graphs G_7, G_5, and G_3 are shown; on the bottom the final result, a layout for G_0, is depicted.

4 Experiments

In this section, we report on experimental results obtained with our implementations DPvis and 3Dvis. We start with layout examples and then present run-times of our implementation of Walshaw's three-dimensional layout algorithm.

4.1 Graph Layout Examples

We started our experiments on visualization with an example from hardware verification that stems from equivalence checking of a 16-bit sequential shift-and-add multiplier with a combinatorial multiplier (see [9] for further information). We used the file `longmult8`, representing equivalence of bit 8 of the two circuit designs.[5]

Whereas one single variable interaction graph already reveals a lot of information about the instance's structure, we observed that by visualizing depth-bounded runs of a DPLL-style algorithm and comparing different interaction graphs of the same

[5] http://www.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html

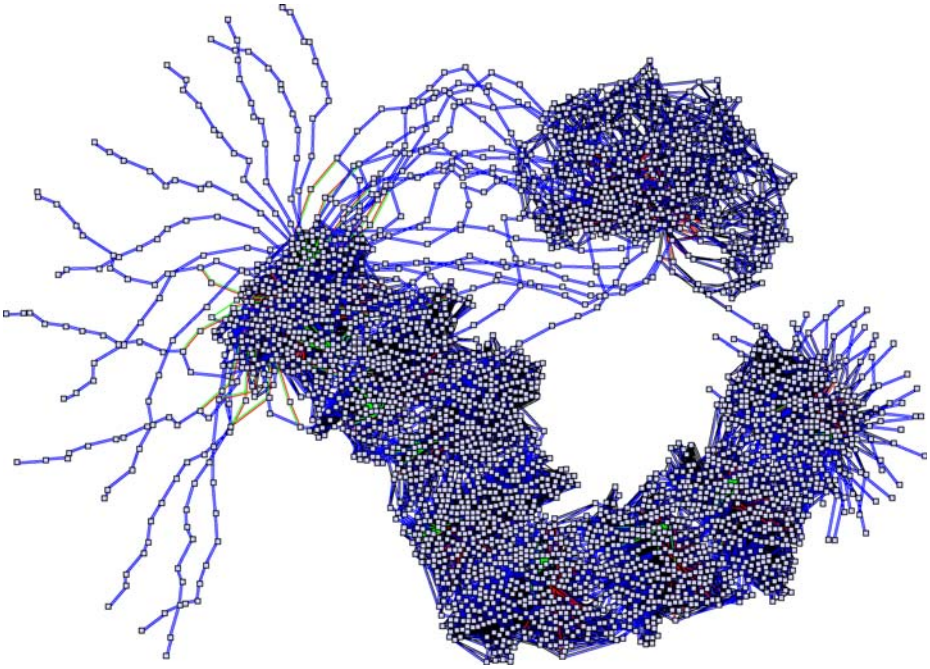

Fig. 7 Variable interaction graph of the instance `longmult8` (top level, after unit propagation)

instance at different states during search, we obtained considerably more information. We therefore simulated such depth-bounded runs of a typical DPLL algorithm in our experiments. The depth-bounded search tree obtained that way for the `longmult8` instance is shown in Fig. 6.

The picture shows the search tree up to depth 3 (one branch extended down to depth 5), with the initial instance as the root node of the tree. Each node is labeled with two numbers (n/k), where n indicates the number of propositional variables of the instance and k the number of clauses. Child nodes are generated by case distinction (on the indicated variable) and are simplified by subsequent unit propagation. On search depth 3, we have four nontrivial clause sets (a, b, c, d) with between 2,103 and 3,161 variables. To illuminate the dynamics of the interaction graphs, we have added a further node at an increased search depth of 5, named d-1. Variable interaction graphs for these five snapshots are shown in Figs. 8, 9, and 10; the original (complete) instance is shown in Fig. 7. Diagram a-1 on top of Fig. 10 additionally shows a magnified view of the top left part of Diagram a.

We performed additional experiments with another instance of hardware verification, the bounded model checking instance `bmc-ibm-2`. This instance encodes a verification problem (IBM CPU Part #2) from IBM's research laboratory in Haifa. The instance contains 3,628 variables and 14,468 clauses. It can – together with a short description about how it was generated – be downloaded from the Canadian SATLIB site.[6] The interaction graph of this instance is shown in Fig. 11.

[6]See respective link on http://www.satlib.org.

Table 1 Run-times of 3Dvis on different SAT instances

SAT Instance	#vars	#clauses	#edges	Run-time L3/L0	Memory
vmpc-21	441	45,339	32,193	0.5 / 7.3	6.8
queueinvar-10	886	5,622	17,578	1.3 / 6.2	6.3
barrel-5	1,407	5,383	12,201	1.3 / 15.2	6.4
longmult-4-up	1,764	5,081	5,124	0.9 / 8.1	5.5
longmult-5-up	2,181	6,355	6,429	1.1 / 9.8	5.7
barrel-7	3,523	13,765	70,512	9.8 / 75.1	16.3
1dlx-c-liveness	6,874	65,479	134,672	18.0 / 468.4	21.3
x1mul.miter	8,760	55,571	51,610	18.0 / 218.6	13.2
bmc-ibm-5	9,395	41,207	55,130	6.3 / 75.0	12.4
c7552mul.miter	11,282	69,529	70,553	10.3 / 105.6	14.4
9dlx-iq1	24,604	261,473	687,398	141.4 / 3,822.6	85.6
bmc-ibm-6	51,639	368,352	726,085	64.8 / 38,972.1	83.2
bmc-galileo-8	58,074	294,821	379,050	63.4 / 45,568.2	62.2
bmc-galileo-9	63,624	326,999	427,698	74.8 / –	68.5
clauses-2	75,528	272,784	392,910	291.7 / 3,247.1	79.6
9dlx-iq3	69,789	968,295	2,599,017	890.0 / –	319.8
9dlx-iq6-bug7	173,811	5,609,632	9,921,587	4,365.0 / 39,404.3	1,223.8
9dlx-iq6-bug9	235,184	8,063,818	14,453,844	8,390.4 / –	1,757.1

The instances were selected from the industrial section of previous SAT Competitions [7]. The size of each instance (number of variables and clauses) is shown, as well as the size of the graph, the run-time (in seconds) to compute a layout on levels $l = 3$ and $l = 0$ (final layout), and the memory consumption (in MB). A dash indicates a time-out.

For comparison, we initiated experiments with further SAT instances outside the realm of hardware verification. We used an instance from automotive product configuration,[7] one instance of the well-known pigeonhole problems (`hole10`), and a random 3-SAT formula with 100 variables and 425 clauses. The last two instances are known to be hard for resolution-based SAT-solvers, whereas the configuration instance is known to be very easy. The variable interaction graphs shown on the left of Fig. 12 correspond to a state during the run of a DPLL algorithm where a few literals already have been fixed. After setting of three further variables and subsequent unit propagation, the interaction graphs shown on the right resulted.

Besides generating variable interaction graphs, we also built resolution graphs. Figure 13 shows two resolution graphs for restriction a of instances `longmult8` and for instance `bmc-ibm-2`. Clustering and symmetry of the resolution graphs are similar to the respective variable interaction graphs. This feature suggests that properties of an instance such as symmetries carry over from one representation to the other (for the component structure of a SAT instance this is obvious).

4.2 Run-time of 3Dvis

In this section we report on the performance of our implementation 3Dvis of Walshaw's algorithm for three-dimensional graph layout. All performance measurements were done with the GLUT/C++ implementation under Linux on a machine

[7] File `C202_FW`, downloadable at http://www-sr.informatik.uni-tuebingen.de/~sinz/DC.

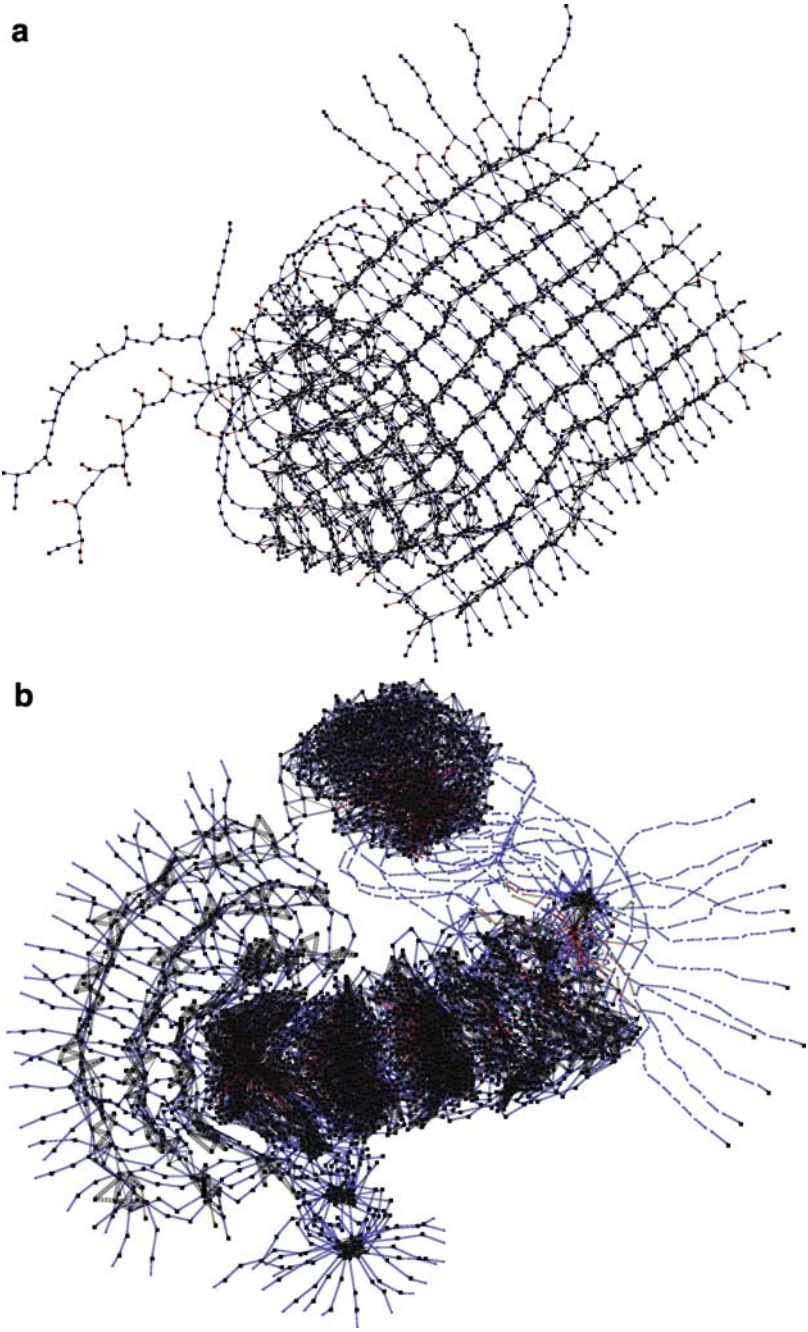

Fig. 8 Variable interaction graphs for two subinstances of `longmult8`, obtained by setting variable 2,823 to false, variable 1,248 to true, variable 2,508 to true (**a)** or false (**b)**, and subsequent unit propagation. In **a**, the grid structure of the combinatorial multiplier is clearly visible, whereas the shift-and-add multiplier could be considerably simplified by fixing the value of the three variables. In **b**, large parts of the shift-and-add multiplier have remained mainly unchanged, and individual time steps of the sequential circuit are clearly visible as large, connected clusters

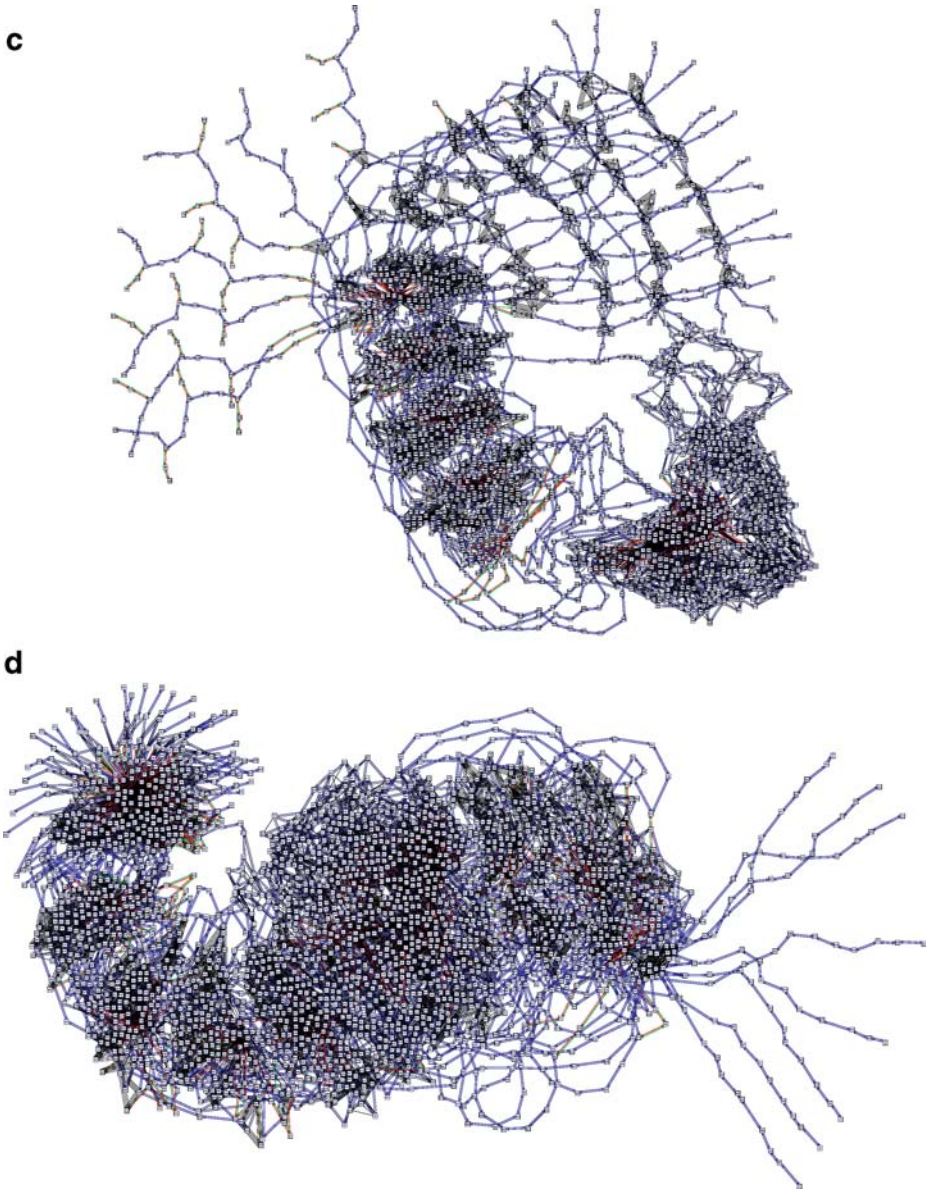

Fig. 9 Variable interaction graphs for two subinstances of `longmult8`, obtained by setting variables 2,823 and 1,248 to false, variable 933 to true (**c)** or false (**d)**, and subsequent unit propagation. It is noteworthy that almost no simplification by unit propagation occurred in **d**, which clearly contrasts with Fig. 8a, resulting from setting only two variables to opposite values

with a Pentium 4 processor running at 3 GHz (hyperthreading switched off) and 2 GB of main memory. The results of our measurements can be seen in Table 1.

Layouts for instances with up to 10,000 variables are computed in a few minutes; layouts for larger graphs are still feasible but may require several hours to be computed. However, as can also be seen from Fig. 5, layouts of coarsened graphs

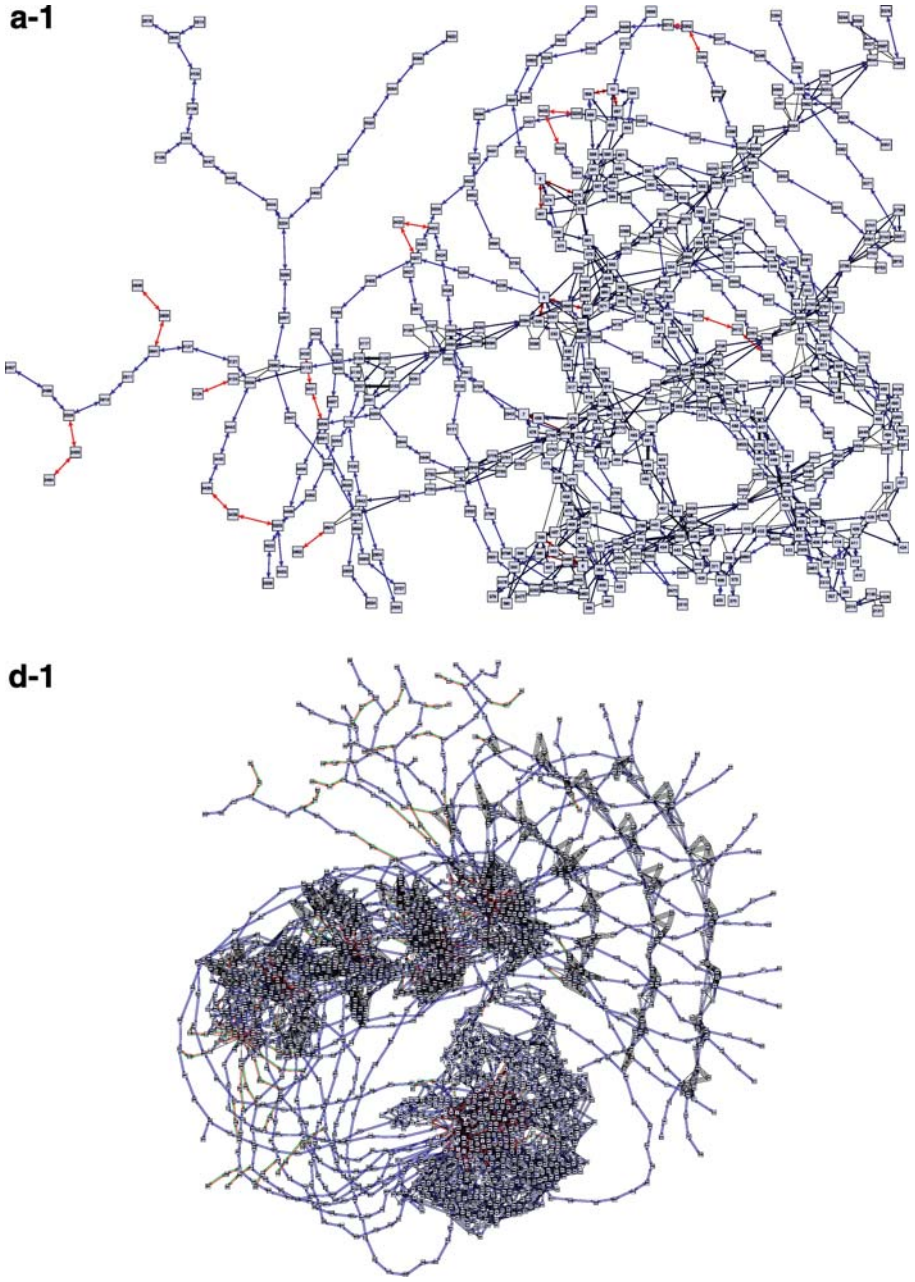

Fig. 10 Two further variable interaction graphs: **a-1** displays a magnified view of the top left part of Fig. 8a, and **d-1** shows the instance obtained from Fig. 9d by additionally setting variable 618 to true, variable 19 to false, and subsequent unit propagation. **a-1** reveals long chains of variables connected by bi-implications. Each of these chains could be contracted to only one variable (by an equivalence detection algorithm) without changing the semantics of the partially processed SAT instance. These chains, however, emerge only after fixing the values of the two variables (and subsequent unit propagation) but do not occur in the original instance. **d-1** shows that subinstance (**d**) is considerably simplified by fixing the values of only two further variables

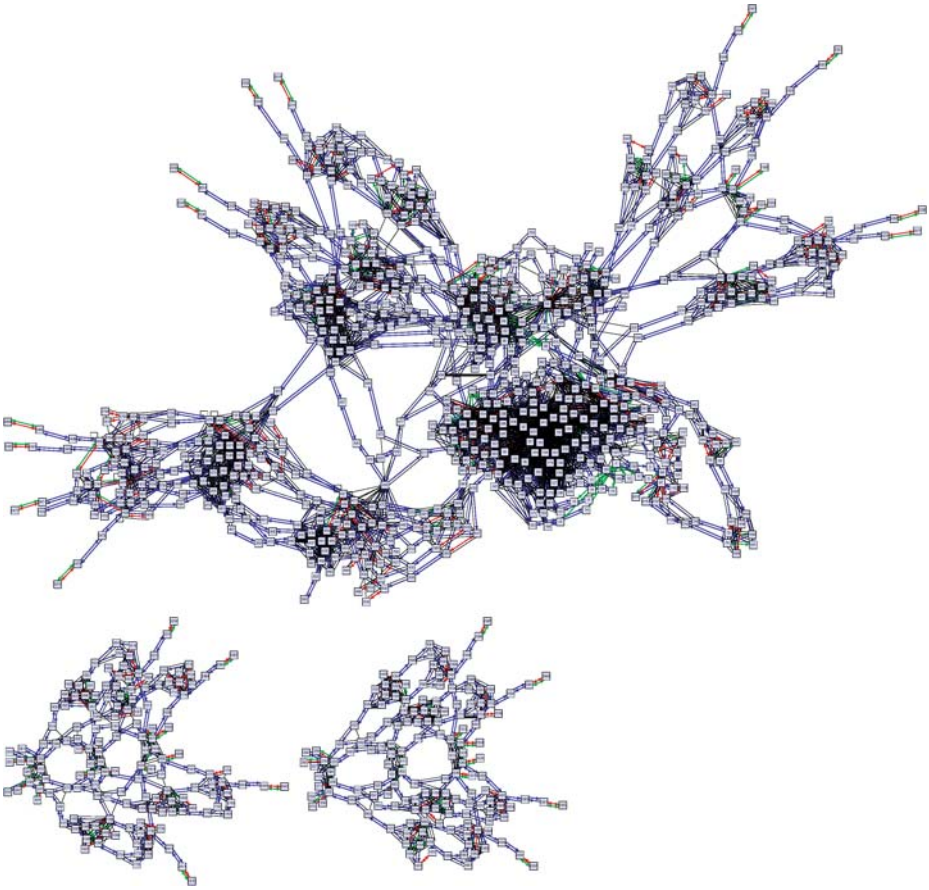

Fig. 11 Variable interaction graph for the SAT instance `bmc-ibm-2`. It is noticeable that the instance – without further restrictions just by unit propagation – already decays into three independent components

(e.g., on level $l = 3$) can already give a rough estimate how the final layout will look and are computed much faster.

5 Observations

We divide our observations into those made on the static graph presentations and those that take the change of the graph structure into account (cf. Figs. 8, 9, and especially Fig. 12).

5.1 Static Aspects

When considering graphs obtained from hardware verification instances, we made the following observations:

1. There are noticeably many implication chains of considerable length (this is best seen in Diagram a-1 of Fig. 10). In many cases these chains are even made up of

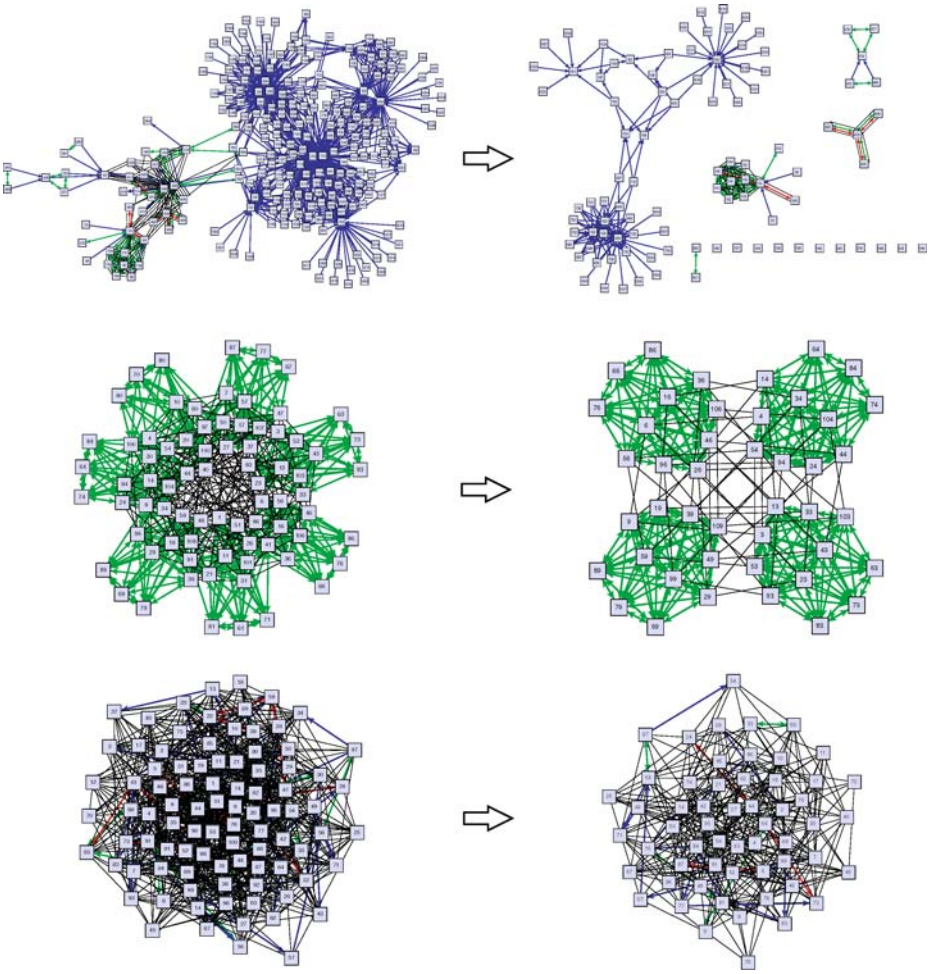

Fig. 12 Variable interaction graphs for different SAT instances before (*left*) and after (*right*) three steps of a DPLL algorithm run (and subsequent simplification by unit propagation). On the *top*, an instance from automotive product configuration is shown, in the *middle* a pigeon hole formula, and on the *bottom* a random 3-SAT formula with a clause-variable-ratio near the phase-transition point. Whereas the product configuration instance (which is easy for current DPLL-based solvers) reduces considerably after fixing the value of three variables (and even decomposes into many independent subcomponents), this effect does not show up on the other instances that are hard for DPLL. These hard instances reveal a "self-similar" behavior between the whole instance and the subproblems that occur during runs of the DPLL algorithm, which might partially explain their hardness

equivalences. Many of them do not occur directly in the original instance but only after setting of a few variables. This fact indicates that incorporation of equivalence handling (or, more generally, special treatment of 2-clauses) may be fruitful for such instances.

2. Decomposition into independent subproblems occurs frequently, at least on higher levels of the search tree. Some problem instances already decay into independent components after unit propagation (as, e.g., `bmc-ibm-2`; cf. Fig. 11).

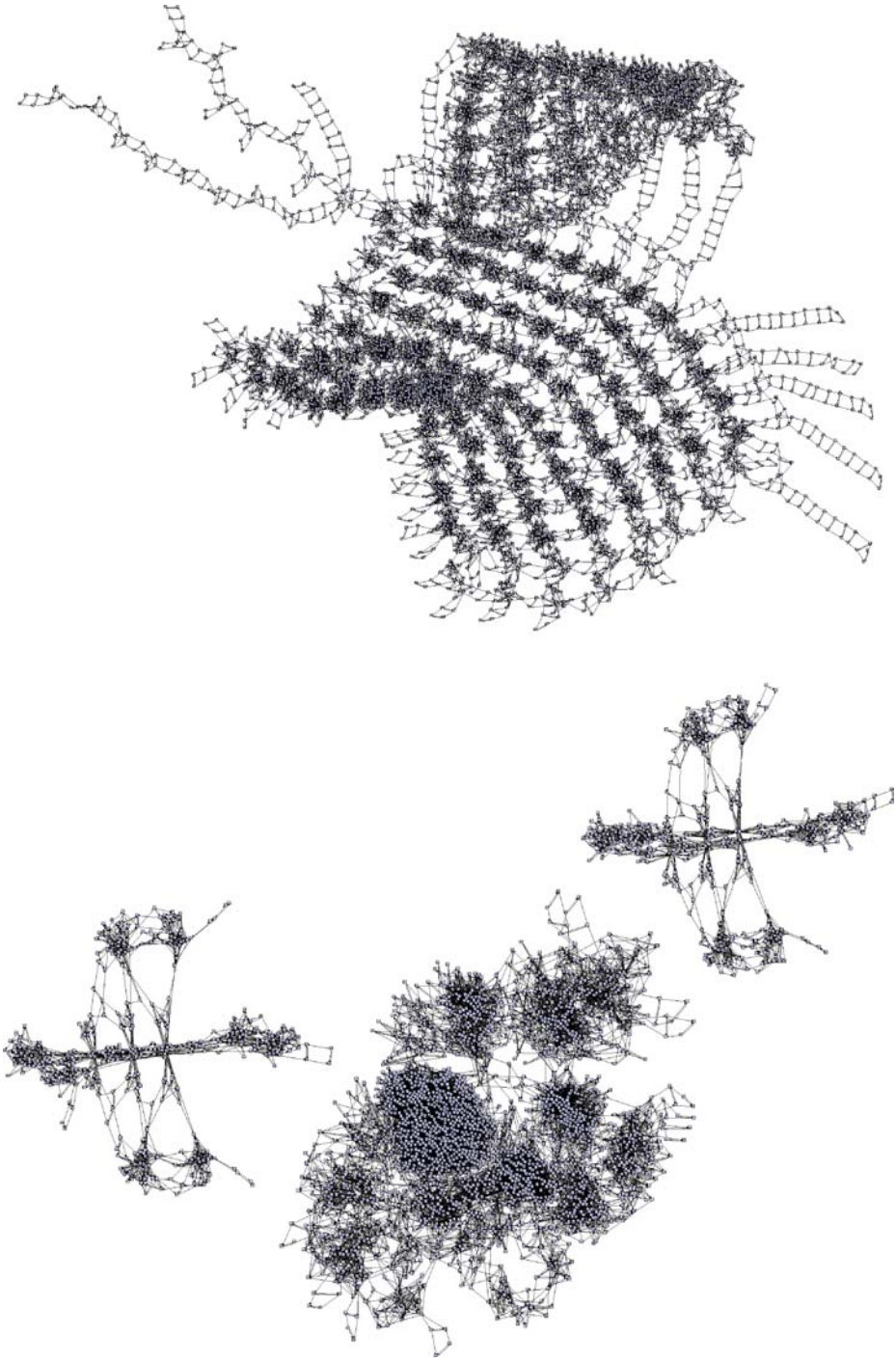

Fig. 13 Resolution graphs for two SAT instances: `longmult8` with the same restriction as in Fig. 8a on the *top*, and `bmc-ibm-2` on the *bottom*. Note the similarity with regard to symmetry and clustering compared to the respective variable interaction graphs in Figs. 8 and 11

This indicates that detection and special handling of independent components may be a useful feature to integrate into DPLL solvers.

Both aspects – implication chains and decomposition – have been considered in the SAT literature [10, 33, 50, 58] but are only partially implemented in today's solvers. Thus, it may be worthwhile trying to exploit these advantageous properties more directly in the future.

5.2 Dynamic Aspects

When considering the dynamic change of the graph structure of a SAT instance we observed the following:

1. Hard problem instances (pigeonhole, random 3-SAT) not only possess a lower rate of variable reduction by unit propagation but also lack the decomposition feature that appeared, for example, in the configuration and hardware verification instances (see Fig. 12). The hard instances seem to possess a "fractal" or "self-similar" behavior on problem reduction by unit propagation.
2. A huge amount of problem reduction is caused by so-called top-level assignments (TLAs) [26] (cf. also Fig. 7 in [60]). TLAs are unit clauses that are learned during the search process. Each top-level assignment 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. (The notion of top-level assignment is closely related to that of a *backbone variable* [59].)

In other experiments [60] we perceived that although TLAs occur in many instances (also in random 3-SAT instances), the number of TLAs is much higher for (comparatively easy) real-world problems than for (comparatively hard) random 3-SAT instances. Comparing the variable interaction graphs of 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 [60].

5.3 Hypotheses

From these observations we derive the following (preliminary) hypotheses:

1. Both long implication chains (cf. Fig. 10, Diagram a-1) and decomposition into independent subproblems may explain the benign behavior of real-world instances. These are properties that we have not yet observed in hard instances. It is important to consider decomposition not only at the top level (i.e., on the original instance) but also at deeper levels of the DPLL algorithm (cf. Fig. 12, top) when several variables are set and thus a partial assignment has been applied to the instance. The decomposition property at the top level is closely related to tree decompositions.
2. Hard instances seem to exhibit a "fractal," "self-similar," or "scale-free" [68] behavior on reductions by case splitting and unit propagation (cf. Fig. 12, middle and bottom). For such instances the conformation of the interaction graph does not change considerably during the search process.

Making use of these hypotheses in future algorithms seems to be more conceivable for the first one (see Paragraph 5.1), whereas it is not so evident how to make use of the second one. However, we dare to hope that observations based on graph structure similar to ours may stimulate generation of new ideas in both theory and practice.

6 Related Work

Connecting graph-based concepts with SAT-solving is a relatively frequent topic in the SAT literature. Rish and Dechter [52] consider a directional resolution algorithm on instances with a special variable interaction graph structure and gain tractability results for instances with bounded induced width (which is equivalent to bounded treewidth) and limited induced diversity ($\text{div}^* \leq 1$). Del Val [21] takes this approach a step further by examining acyclic component networks and by using literal interaction graphs instead of variable interaction graphs.

Galesi and Kullmann [30] study properties of the resolution graph resulting in the establishment of different tractable subclasses (e.g., for clause sets F with Hermitian rank $h(F) \leq 1$).

Previous work on connected components for the SAT problem was conducted mainly in the context of model counting [6, 39, 56], although there are also approaches for constraint satisfaction problems in general [28, 38]. Bayardo and Pehousek extended the SAT solver Relsat 2.0 developed by Bayardo and Schrag in order to count models using connected components [39]. Their algorithm tries to detect components recursively at each node of a DPLL search tree. However, they do not take clause learning into account. Sang et al. carry on the recursive decomposition approach of Bayardo and Pehousek, and combine it with clause learning and component caching [56]. Slater [61] compares different SAT solvers on clustered problem instances, without experimenting with specialized algorithms that exploit the structure, however.

Park and van Gelder [50] apply hypergraph partitioning algorithms to decompose a SAT instance into independent subproblems. They claim that using the dual hypergraph (in which the vertices are clauses and hyperedges encompass clauses with common variables) induces a more natural decomposition, as a (minimal) cut in the dual hypergraph coincides with a set of variables that has to be assigned in order to break the instance into independent components. However, their algorithm seems to be not competitive in practice. Biere and Sinz [10] modify a zChaff-like SAT-solver to detect independent components and present techniques to handle these components.

Another graph-based approach is taken by Walsh [66], who carries over the influential work of Watts and Strogatz on "small world" networks [68] to combinatorial search problems. Walsh examines different classes of search problems, for example, CSP translations of quasigroup problems, with respect to their "small world" properties (characteristic path length and clustering coefficient of the interaction graph).

Work on visualization of SAT instances was done by Slater [62] and, to some extent, by Selman [57]. Slater uses different graph translation techniques (variable interaction graph, literal interaction graph, 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 are force-directed placement algorithms. Selman [57] 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. Again, this layout cannot reveal symmetries and clustering as well as force-directed placement does. Eén and Biere [25] have implemented a visual demo of their SAT preprocessor SatELite that graphically shows how clauses are processed by their algorithm.

Considering graph visualization in general, there are many graph layout packages available, for example, Tulip or Graphviz.[8] Most of these packages focus on customizable layouts and do not allow for a tight integration with specialized SAT-related algorithms, however (cf. Fig. 13).

7 Conclusion

We have presented an approach to visualize SAT instance in two and three dimensions by converting them to graph structures. We have generated visually attractive graph layouts using two different graph translations (variable interaction and resolution) by applying force-directed placement algorithms. Moreover, we have developed a tool, DPvis, that can generate animations of runs of the DPLL algorithm showing both the generated search tree and the temporal change of the SAT instance's structure.

We assume that our techniques can help in obtaining a better understanding of what makes a particular instance hard or easy. Moreover, we believe that our tool DPvis can be of great help in teaching and understanding the DPLL algorithm and its modern variants that incorporate clause learning.

Possible directions for future work include the design and implementation of faster graph layout algorithms (by using, e.g., fast multipole methods [37]), visualizing clause activity and variable activity in zChaff-like solvers to obtain a better understanding of clause learning, and trying to establish a theoretical link between graph properties and hardness of SAT instances.

Acknowledgements I thank Edda-Maria Dieringer for implementing the visualization tool DPvis. Allen van Gelder suggested to use resolution graphs for visualization. Ofer Strichman provided me with an interpretation for the graph structure of Fig. 5. Moreover, I thank the anonymous reviewers for helpful comments and suggestions. And finally, I am indebted to Armin Biere for many fruitful discussions on SAT visualization.

[8] See, e.g., http://directory.google.com/Top/Science/Math/Combinatorics/Software/Graph_Drawing for a list of such packages.

References

1. Aloul, F.A., Sakallah, K.A., Markov, I.L.: Efficient symmetry breaking for boolean satisfiability. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), pp. 271–276 (2003)
2. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Algorithms and Discrete Methods **8**, 277–284 (1987)
3. Aspvall, M., Plass, M.F., Tarjan, R.E.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Inf. Process. Lett. **8**(3), 121–123 (March 1979)
4. Barnes, J., Hut, P.: A hierarchical $O(n \log n)$ force-calculation algorithm. Nature **324**, 446–449 (1986)
5. Barrett, C., de Moura, L., Stump, A.: SMT-COMP: Satisfiability modulo theories competition. In: Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005), pp. 20–23. Springer, Berlin Heidelberg New York (2005)
6. Beame, P., Impagliazzo, R., Pitassi, T., Segerlind, N.: Memoization and DPLL: Formula caching proof systems. In: Proceedings of the 18th Annual IEEE Conference on Computer Complexity (Complexity 2003), pp. 248–264 (2003)
7. Le Berre, D, Simon, L.: The essentials of the SAT 2003 competition. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), pp. 452–467 (2003)
8. Le Berre, D., Simon, L. (eds.): Special Volume on the SAT 2005 competitions and evaluations. Journal on Satisfiability, Boolean Modeling and Computation, vol. 2, (March 2006)
9. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS, pp. 193–207. Springer, Berlin Heidelberg New York (1999)
10. Biere, A.,Sinz, C.: Decomposing SAT problems into connected components. Journal on Satisfiability, Boolean Modeling and Computation **2**, 191–198 (2006)
11. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. **25**(6), 1305–1317 (1996)
12. Braunstein, A., Zecchina, R.: Survey and belief propagation on random k-SAT. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), pp. 519–528 (2003)
13. Bryant, R.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
14. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the really hard problems are. In: Proceedings of the 12th International Joint Conference on Artificial Intelligence, IJCAI-91, pp. 331–337. Sidney, Australia (1991)
15. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
16. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for CNF. In: Proceedings 41th Design Automation Conference (DAC 2004), pp. 530–534 (2004)
17. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7), 394–397 (July 1962)
18. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM **7**(3), 201–215 (1960)
19. Dechter, R.: Bucket elimination: a unifying framework for reasoning. Artif. Intell. **113**(1,2), 41–85 (1999)
20. del Val, A.: Tractable databases: How to make propositional unit resolution complete through compilation. In: Proceedings of 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94), pp. 551–561 (1994)
21. del Val, A.: Tractable classes for directional resolution. In: Proceedings 17th National Conference on AI, pp. 343–348. Austin, TX (2000)
22. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.: Algorithms for automatic graph drawing: an annotated bibliography. Comput. Geom. **4**, 235–282 (1994)
23. Downey, R.G. and Fellows, M.R.: Parameterized Complexity. Springer, Berlin Heidelberg New York (1999)
24. Eades, P.: A heuristic for graph drawing. Congressus Numerantium **42**, 149–160 (1984)

25. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005), pp. 61–75 (2005)
26. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), pp. 502–518. Springer, Berlin Heidelberg New York (May 2003)
27. Fleisher, H., Maissel, L.I.: An introduction to array logic. IBM J. Res. Dev. **19**(2), 98–109 (1975)
28. Freuder, E.C., Quinn, M.J.: Taking advantage of stable sets of variables in constraint satisfaction problems. In: Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI 1985), pp. 1076–1078 (1985)
29. Fruchterman, T., Reingold, E.: Graph drawing by force-directed placement. Software – Practice and Experience **21**(11), 1129–1164 (1991)
30. Galesi, N., Kullmann, O.: Polynomial time SAT decision, hypergraph transversals and the hermitian rank. In: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), pp. 89–104. Vancouver, Canada (May 2004)
31. Gallo, G., Longo, G., Pallottino, S.: Directed hypergraphs and applications. Discrete Appl. Math. **42**(2), 177–201 (1993)
32. Gent, I.P., Walsh, T.: The SAT phase transition. In: Proceedings of the 11th European Conference on Artificial Intelligence, pp. 105–109 (1994)
33. Gershman, R., Strichman, O.: Cost-effective hyper-resolution for preprocessing CNF formulas. In: Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005), pp. 423–429 (2005)
34. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. In: Proceedings of the Design, Automation and Test in Europe Conference and Exposition (DATE 2002), pp. 131–149. IEEE Computer Society, Paris, France (2002)
35. Gomes, C.P., Selman, B., Crato, N., Kautz, H.A.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reason. **24**(1/2), 67–100 (2000)
36. Gottlob, G., Scarcello, F., Sideri, M.: Fixed-parameter complexity in AI and nonmonotonic reasoning. Artif. Intell. **138**(1,2), 55–86 (2002)
37. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. J. Comp. Phys. **73**, 325–348 (1987)
38. Bayardo Jr., R.J., Miranker, D.P.: On the space-time trade-off in solving constraint satisfaction problems. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 1995), pp. 558–562 (1995)
39. Bayardo Jr., R.J., Pehoushek, J.D.: Counting models using connected components. In: Proceedings of the 17th Nat. Conference on Artificial Intelligence (AAAI 2000), pp. 157–162 (2000)
40. Kautz, H., Selman, B.: Planning as satisfiability. In: Proceedings of the 10th Europe Conference on Artificial Intelligence (ECAI'92), pp. 359–363. Wiley (1992)
41. Küchlin, W., Sinz, C.: Proving consistency assertions for automotive product data management. J. Autom. Reason. **24**(1,2), 145–163 (February 2000)
42. Kullmann, O.: The SAT 2005 solver competition on random instances. Journal on Satisfiability, Boolean Modeling and Computation **2**, 61–102 (2006)
43. Marques-Silva, J.P., Sakallah, K.A.: Conflict analysis in search algorithms for propositional satisfiability. In: Proceedings of the IEEE International Conference on Tools with Artificial Intelligence, pp. 467–469 (Nov. 1996)
44. McMillan, K.: Interpolation and SAT-based model checking. In: Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003), vol. 2725 of LNCS, pp. 1–13 (2003)
45. Mitchell, D.G., Selman, B., Levesque, H.J.: Hard and easy distributions of SAT problems. In: Proceedings of the 10th Nat. Conference on Artificial Intelligence (AAAI-92), pp. 459–465 (1992)
46. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: Determining computational complexity from characteristic "phase transitions". Nature **400**(6740), 133–137 (1999)
47. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC 2001), pp. 530–535. ACM (2001)
48. Narodytska, N., Walsh, T.: Constraint and variable ordering heuristics for compiling configuration problems. In: Configuration Workshop Proceedings, 17th European Conference on Artificial Intelligence (ECAI-06), pp. 2–7. Riva del Garda, Italy (August 2006)

49. Niedermeier, R.: Fixed-parameter Algorithms. Oxford Lecture Series in Mathematics and Its Applications. Oxford University Press (February 2006)
50. Park, T.J., Van Gelder, A.: Partitioning methods for satisfiability testing on large formulas. Inf. Comput. **162**, 179–184 (2000)
51. Plaisted, D.A., Zhu, Y.: Ordered semantic hyper-linking. J. Autom. Reason. **25**(3), 167–217 (2000)
52. Rish, I., Dechter, R.: Resolution versus search: Two strategies for SAT. J. Automated Reason. **24**(1,2), 225–275 (February 2000)
53. Robertson, N., Seymour, P.D.: Graph minors, II: Algorithmic aspects of tree-width. J. Algorithms **7**(3), 309–322 (1986)
54. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (1965)
55. Cook, S.A.: The complexity of theorem proving procedures. In: 3rd Symposium on Theory of Computing, pp. 151–158. ACM Press (1971)
56. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004). Vancouver, Canada (May 2004)
57. Selman, B.: Algorithmic adventures at the interface of computer science, statistical physics, combinatorics. In: Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004), pp. 9–12. Springer, Berlin Heidelberg New York (2004)
58. Sheeran, M., Stålmarck, G.: A tutorial on Stålmarck's proof procedure for propositional logic. Form. Methods Syst. Des. **16**(1), 23–58 (2000)
59. Singer, J., Gent, I.P., Smaill, A.: Backbone fragility and the local search cost peak. J. Artif. Intell. Res. **12**, 235–270 (2000)
60. Sinz, C., Dieringer, E.-M.: DPvis – a tool to visualize structured SAT instances. In: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), St. Andrews, Scotland, pp. 257–268. Springer, Berlin Heidelberg New York (June 2005)
61. Slater, A.: Investigations into satisfiability search. Ph.D. thesis, NICTA, Australian National University, Acton, Australia (2003)
62. Slater, A.: Visualisation of satisfiability problems using connected graphs, March 2004. http://rsise.anu.edu.au/~andrews/problem2graph
63. Szeider, S.: On fixed-parameter tractable parameterizations of SAT. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), pp. 188–202. Springer, Berlin Heidelberg New York (May 2003)
64. Vaziri, M., Jackson, D.: Checking properties of heap-manipulating procedures with a constraint solver. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03), number 2619 in LNCS, Warsaw, Poland, pp. 505–520. Springer, Berlin Heidelberg New York (2003)
65. Velev, M.N., Bryant, R.E.: Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. J. Symb. Comput. **35**(2), 73–106 (2003)
66. Walsh, T.: Search in a small world. In: Proceedings of the 16th International Conference on AI, pp. 1172–1177 (1999)
67. Walshaw, C.: A multilevel algorithm for force-directed graph-drawing. J. Graph Algorithms Appl. **7**(3), 253–285 (2003)
68. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**, 440–442 (1998.)
69. Williams, R., Gomes, C., Selman, B.: Backdoors to typical case complexity. In: International Joint Conference on Artificial Intelligence (IJCAI'03), pp. 1173–1178. Acapulco, Mexico (2003)
70. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: Proceedings of the 32nd Symposium on Principles of Programming Languages (POPL 2005), pp. 351–363. Long Beach, CA (January 2005)
71. Zabiyaka, Y., Darwiche, A.: Functional treewidth: bounding complexity in the presence of functional dependencies. In: Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006), pp. 116–129 (2006)