Review Article

Exploiting Structure: A Survey and Analysis of Structures and Hardness Measures for Propositional Formulas

Rick Adamy[1], Elias Kuiter[1], Gunter Saake[1]

1. Otto-von-Guericke Universität Magdeburg, Germany

The Boolean satisfiability problem (SAT) and its many variations lie at the core of many algorithmic problems in both academia and industry. Due to being NP-complete, general instances of SAT cannot be solved efficiently. However, exploiting certain structures or properties of a formula can greatly accelerate the computation of solutions or serve as a measure for the hardness of a SAT instance. In this paper, we describe and discuss such exploitable properties and structures. First, we describe known exploitable structures found in propositional formulas like blocked clauses, unit clauses, pure literals, backbones, and autark assignments. Second, we describe hardness indicators for propositional formulas such as the variable-to-clause ratio, as well as advanced structural measures like centrality, modularity, and self-similarity. In particular, we give an overview on the selected structures and measures and discuss their applications. We also identify relationships between them to clarify their complex interactions and potential for use in solvers.

Corresponding authors: Rick Adamy, rick.adamy@ovgu.de; Elias Kuiter, kuiter@ovgu.de; Gunter Saake, saake@ovgu.de

1. Introduction

The Boolean satisfiability problem (SAT) is a core problem of propositional logic with close ties to problems important to industry. Among them are selection problems (e.g., finding cliques or vertex covers in graphs), separation problems (e.g., finding maximum matchings or maximum cuts in graphs) and arrangement problems (e.g bin packing and scheduling), which can all be expressed or modeled as a propositional formula. Thus, solving such problems can be reduced to finding a satisfying assignment for the corresponding formula [1]. In fact, any NP problem (i.e., computationally hard problems with no efficient solving algorithm for all instances), can be reduced to the question of whether a formula is satisfiable or not [2]; that is, to the Boolean satisfiability problem (SAT). In addition to the wide applicability of SAT, it also offers an easy to handle standard format, the conjunctive normal form (CNF), which allows solving methods to be developed independently from the concrete problems and their individual format.

Due to this universality of the SAT problem, finding efficient ways to solve it are of importance to many areas of academia and applications in industry such as hardware and circuit verification [3], software verification like static driver verification [4] or the analysis of feature models [5], which model dependencies of features in software product lines [6]. However, due to SAT being in NP itself it is provably not possible (if $P \neq NP$) to construct a general solution algorithm that is efficient for all instances.

While efficiently solving arbitrary instances is therefore not possible, it is still possible to identify helpful properties and structures of specific formula types, which then allow the creation of specialized solvers, like $\mathrm{SATZILLA}$ [7], $\mathrm{HORDESAT}$ [8] or the SPEAR theorem prover [9], which are efficient for certain types of formulas. In addition, there are classes of formulas with more restrictions to their structure, such as Horn formulas (which can be solved in linear time [10]) or feature-model formulas [11] (which are mostly efficient to solve in practice). In this survey, we give an overview over a selection of such structures found in SAT formulas and their application in specialized solving algorithms, as well as a selection of difficulty indicating measures on them. Furthermore, we identify connections between the selected structures and measures and highlight their potential use in solvers.

2. Background

In the following section we briefly introduce the most important concepts that we use to study structures and measures of propositional formulas, as well as methods to solve them.

2.1. SAT and Propositional Logic

First we introduce the basics of propositional logic, the SAT problem and their representations. For this let $\mathbf{S} = \{x_1, x_2, ..\}$ be a set of *variables* and let $\mathbf{L} = \mathbf{S} \cup \{\neg x | x \in \mathbf{S}\}$ be the corresponding set of *literals* over a set of variables. At the core of SAT are propositional formulas in conjunctive normal form (CNF):

Definition 2.1. A formula $C = l_1 \vee .. \vee l_m$ is called a *clause* of size m if it is a disjunction of m literals. A formula $\phi = C_1 \wedge .. \wedge C_n$ is called a *CNF formula* of length n if it is a conjunction of n clauses.

The clause of size 0 is called empty clause and is denoted by $\square$.

The set of all formulas in CNF is denoted by $\mathbf{CNF}$. A clause on its own is also a formula in $\mathbf{CNF}$ as it is a CNF formula of length 1. It is sufficient to only look at formulas in CNF, as any arbitrary propositional formula can be transformed into an equivalent formula in CNF by repeatedly applying the axioms of propositional logic [12]. This in turn can cause a translated formula to grow exponentially in length. For such cases, alternative transformations into equisatisfiable CNFs like the Tseitin transformation [13] [14] can be used, which grow only linear in length. In order to determine whether a formula is satisfiable, we need to define truth value assignments.

Definition 2.2. A function α: $S \rightarrow \{0, 1\}$ is called truth value assignment (henceforth just assignment). Let var(ϕ) be the set of symbols used in ϕ. An assignment α is called fitting for ϕ if it only assigns symbols from var(ϕ). α is called partial w.r.t. ϕ if there is a $s \in$ var(ϕ) not assigned by it. α is called complete w.r.t. ϕ if it is not partial. We only examine fitting assignments and use $(x_1 = i_1, .., x_n = i_n)$ as shorthand notation for the assignment α such that $\alpha(x_j) = i_j, j \in \{1, .., n\}$.

Based on the assigning function the notion of satisfiability of a formula can be defined as follows:

Definition 2.3. Let α be an assignment, x be a variable, C a clause and ϕ a formula in CNF.

- α satisfies x if $\alpha(x)$ = 1
- α satisfies $\neg x$ if $\alpha(x)$ = 0
- α satisfies C if it satisfies at least one of the literals in C
- α satisfies ϕ if it satisfies all clauses of ϕ.

An assignment satisfying a formula is called a model for the formula. A formula containing an empty clause is unsatisfiable, as the empty clause is not satisfiable by definition.

Definition 2.4. A formula is called *tautology* if every complete assignment over its variables is a model.

Conversely a formula is called *contradiction* if it has no models. Based on the notions of CNF and the satisfiability of CNF formulas, the SAT problem is now defined as follows:

Definition 2.5. The set of all satisfiable formulas in CNF is denoted by SAT.

Adding an additional structural constraint to the shape of the formula yields the class of k-SAT problems defined as follows:

Definition 2.6. k-SAT is the set of all satisfiable formulas ϕ in CNF whose clauses are of size at most k.

The special case of k = 1 is trivial as it consists solely of clauses of size 1. Such a formula is unsatisfiable if and only if it contains a clause x and its negation $\neg x$, as a variable cant be assigned both 0 and 1 simultaneously. Otherwise a satisfying assignment can be constructed by assigning 0 to a variable if it occurs negated and 1 otherwise. In addition, the case k = 2 is similarly a problem in P, meaning there is at least one polynomial-time algorithm that can decide the satisfiability of any 2-SAT instance. In fact, a wide variety of such algorithms have been proposed over the years, the fastest of them even reaching linear time complexity, like those proposed by Krom [15] and Aspvall et al. [16]. For all cases of k > 2 however, k-SAT is generally an NP problem. Since SAT is reducible onto k-SAT (for k > 2), it makes any such k-SAT equally difficult and equally expressive, or NP-hard. Thus it can be sufficient to look at a specific case with more restriction on the shape of the formulas, like 3-SAT. In addition to the CNF, any formula ϕ can be encoded in an undirected graph, called variable incidence graph (VIG). The representation of a formula as a graph puts more emphasis on the relations between the variables which are captured by the graphs structure, giving an additional avenue to identify structure within a formula. Furthermore this allows to apply graph based measures and graph theory to SAT instances. VIGs can be defined as follows:

Definition 2.7. A VIG on a formula ϕ consists of the variables contained in ϕ as vertices, connecting two vertices with an edge if two variables appear together in a clause.

VIGs may weight the edges by incorporating how often two variables appear together and the size of their shared clauses. Such a weight can be calculated by $w(x,y) = \sum_{C \in \phi \wedge x,y \in C} \frac{1}{\binom{|C|}{2}}$, which scores each shared clause by its size, giving higher scores to smaller clauses, before summing over the scores of all shared clauses. This weighting method does not consider positive and negative literals to be different. Based on the VIG, another useful concept are boxes over a VIG, which are important for certain measures.

Definition 2.8. A box B of size k in a VIG is a subset of k vertices, such that $\forall_{x,y \in B}(d_{x,y} < k)$. $d_{x,y}$ here is the weighted distance of the shortest path between x and y.

If $\mathbf{S}$ is a box of size s, then there are no two vertices $x, y \in \mathbf{S}$ such that their shortest path distance is longer than s. In essence this means that any two vertices of a box are always connected by a shortest path that does not exceed s. As a more geometrical interpretation, we can think of a box of size s as a circle with diameter s. For a set of s vertices $\mathbf{S}$, the weighted distances $d_{x,y}$ of the shortest paths between them can be thought of as line segments of length $d_{x,y}$. $\mathbf{S}$ now is a box of size s, if all line segments can be placed inside the circle without any of them intersecting the circle itself. In unweighted cases, instead of using weighted distances one can simply use the length of the shortest path, which is the number of edges, for a similar notion of a box.

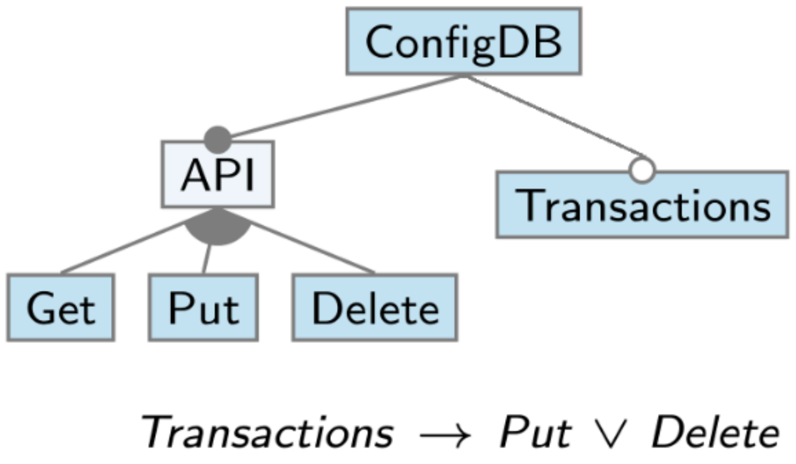


Figure 1. Graphical depiction of a feature model M, with root feature "ConfigDB", mandatory feature "API", optional feature "Transaction", the feature group "Get, Put, Delete" and the cross tree constraint "Transaction$\rightarrow$Put $\vee$ Delete"

2.2. Applications of SAT

Propositional formulas are, for instance, applied in software engineering for representing feature models, which are used to model dependencies of features in software product lines. Figure 1 depicts an example feature model for a simple database. Such a model can be transformed into a propositional formula that describes its valid configurations~[6]. For the example model M, an equivalent formula would be $\phi(M) =$ ConfigDB $\wedge$ (ConfigDB$\leftrightarrow$ API) $\wedge$ (Transaction $\rightarrow$ ConfigDB) $\wedge$ (API$\leftrightarrow$ Get $\vee$ Put $\vee$ Delete) $\wedge$ (Transaction $\rightarrow$ Put $\vee$ Delete). This formula can further be transformed into a CNF formula which can then be used to analyze the feature model, including determining core-, dead features of the model or checking for invalid feature selections. However, in order to do such analysis on a feature model, the underlying equivalent formula needs to be solved.

2.3. Solving SAT

Determining whether a given formula ϕ is a member of SAT (or k-SAT, respectively) lies at the core of the SAT problem, which can be answered by either constructing a satisfying assignment (model finding) or by finding a refutation proof, effectively showing the formula to be a contradiction.

The simplest approach to model finding is the naive search, which is simply trying all possible assignments over the set of variables contained in a formula. This consists of creating a complete assignment and testing if it is a model for the formula before moving to the next complete assignment, repeating until either a model has been found or all possible assignments have been checked. However, this approach is very inefficient, as it does not retain information of prior conflicts, which amounts to reevaluating similar assignments multiple times. The *backtracking* method improves the naive search by constructing a partial assignment step by step, going one step back in the construction if the current assignment cannot be extended to a model. In addition to this it also simplifies the remaining formula by applying the partial assignment. Those two modifications cause the search algorithm to abandon partial assignments that caused a simplified formula to contain the empty clause, making it unsatisfiable, instead of further pursuing it. Furthermore, the backtrack algorithm stops whenever the simplified formula is empty, as the original formula is already satisfied by the current partial assignment.

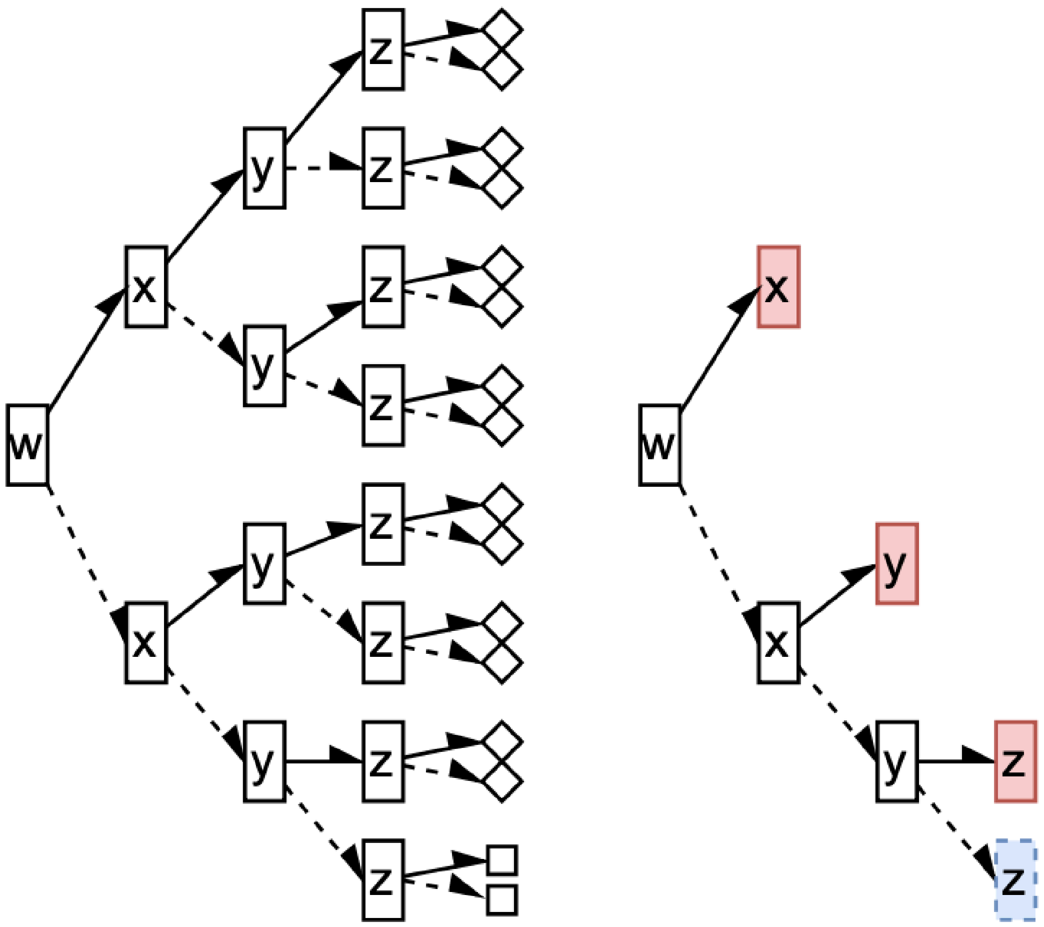


Figure 2. Example search trees for naive search (left) and backtracking (right) for the formula $\phi = \neg w \wedge \neg x \wedge \neg y \wedge (\neg y \vee z)$. Solid arrow: assign 1, dotted arrow: assign 0, box: satisfying assignments, diamond: unsatisfying assignment

Figure 2 shows the different search trees for a naive search and backtracking if both always look at assigning 1 to a variable first. The red boxes in the backtrack tree are the places where the backtrack method has found the empty clause in the simplified formula which caused it to backtrack one step, thus abandoning this path. The dotted blue box in the backtrack tree is the place where the simplified formula became empty after assigning 0 to y, thus signaling the partial assignment constructed up to that point being satisfying.

It is important to note that both methods' efficiency depends on the order in which the variables are assigned. To counter this and further accelerate the search method, the class of DPLL algorithms [17] has emerged, which add further heuristics to the standard backtracking, in addition to utilizing unit clauses and pure literals. Another class of algorithms, which is derived from the DPLL algorithms, is called Conflict Driven Clause Learning (CDCL) [18]. The main difference between CDCL and DPLL is that CDCL uses non-chronological backtracking and can learn clauses during the search process. Informally, clause learning adds a conflict clause (a clause encoding the cause of a conflict found during assignment construction) to the formula in order to avoid the same conflict in later constructions. After adding the conflict clause to the formula, a non-chronological backtrack step is done, which returns to the earliest point in the search tree where one of the conflict-generating variables got assigned.

As mentioned before, the other possible approach to determine if a formula is satisfiable is by refutation proof. By using a sound and complete calculus, the approach tries to infer the empty clause $\square$, which can not be satisfied by definition. The naive approach is to subsequently generate all inferences while adding the results to the formula via conjunction, yielding a modified but equisatisfiable formula. A common calculus used in propositional logic is the resolution calculus, which uses the following inference rule:

Definition 2.9. Let $C = l_1 \vee .. \vee l_n \vee b$ and $D = r_1 \vee .. \vee r_m \vee \neg b$ be two clauses, then their resolvent $R_{C,D}$ over the variable b is defined as $R_{C,D} = l_1 \vee .. \vee l_n \vee r_1 \vee .. \vee r_m$.

Finding a refutation by resolution consists of inferring the empty clause $\square$ by repeatedly applying the resolution rule until no new resolvent can be added to the formula. This process similarly can be augmented by heuristics, like linear resolution [19], for additional computational gain.

3. Structures and Measures

In this section we introduce and examine the concrete structures and hardness measures selected for this survey. The selection is mostly based on their adjacency to DPLL/CDCL algorithms, used by most state of the art solvers.

For this section let $\phi = (x) \wedge (\neg x \vee y) \wedge (y \vee z) \wedge (\neg p \vee p \vee x \vee \neg y)$ be a recurring example. Figure 3 depicts the VIG for ϕ with and without standard weights as defined in Section 2.1.

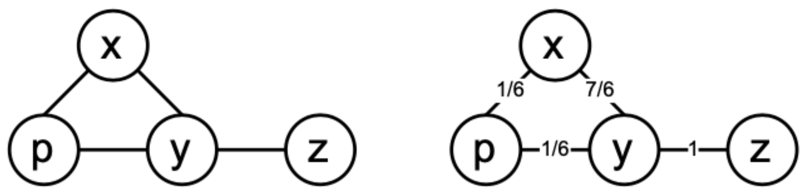
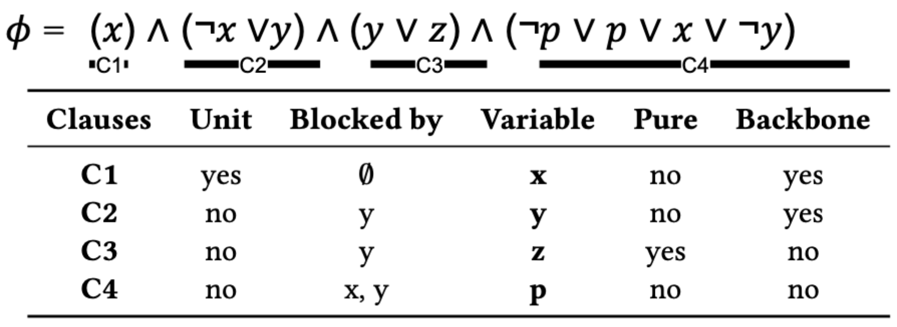


Figure 3. Variable Incidence Graph of ϕ. Left: without weights, right: with standard weights

To tie a more concrete meaning to the variables of ϕ, consider the following interpretation of the variables as simple propositions: x = "it is raining", y = "it is cloudy", z = "it is windy" and p = "it is noon". The formulas clauses now encode the following: (x) = it is raining, $(\neg x \vee y)$ = if it is raining then it is cloudy, $(y \vee z)$ = it is cloudy or it is windy and, $(\neg p \vee p \vee x \vee \neg y)$ = if it is noon or cloudy then it is noon or raining.

3.1. Structures

First we introduce the selected structures and show where they occur in ϕ, which we summarize in Figure 4.



$$\phi = \underbrace{(x)}_{C1} \wedge \underbrace{(\neg x \vee y)}_{C2} \wedge \underbrace{(y \vee z)}_{C3} \wedge \underbrace{(\neg p \vee p \vee x \vee \neg y)}_{C4}$$

Clauses	Unit	Blocked by	Variable	Pure	Backbone
C1	yes	∅	**x**	no	yes
C2	no	y	**y**	no	yes
C3	no	y	**z**	yes	no
C4	no	x, y	**p**	no	no

Figure 4. Overview of structure found in ϕ

3.1. Unit Clauses

The first and simplest structure in the selection is the *unit clause*, which are clauses of size 1 [20][21]. Since all clauses of a formula need to be satisfied in order to satisfy the entire formula, and there is only one possibility to satisfy a clause of size 1, they can be used as a heuristic in constructing a satisfying assignment. As such they play an important role in the aforementioned class of DPLL algorithms where unit clauses are given high priority as they allow the formula to be simplified subsequently, which may create new unit clauses in the process. The prioritization of unit clauses and subsequent simplification of a formula is known as *unit propagation*. Besides their important role in DPLL, they also are of high significance for resolution refutations as they are needed to infer the empty clause $\square$. In addition to that, prioritizing resolutions with unit clauses only results in small clauses, which makes it more likely to approach a possible refutation proof. In ϕ only (x) is a unit clause, however, if unit propagation is used as a simplification mechanism it would turn the second clause $(\neg x \vee y)$ to (y), creating a new unit clause. In terms of the concrete propositions this means that "it is raining" has to be true if ϕ is to be satisfied.

3.1.2. Pure Literals

A literal is called *pure* in a formula ϕ, if its negation is not in ϕ [20][21]. For a non pure literal, both assignments with 0 and 1 may lead to an overall satisfying assignment, which requires both to be examined in a search tree. Compared to that, a pure literal is guaranteed to only benefit the search for a model by assigning the value that satisfies it, preventing a branch in the search tree. Just like unit clauses, pure literals are a main heuristic in the class of DPLL algorithms as they are easy to determine. For instance, counting both positive and negative occurrences of each variable and updating those counters upon simplifications would make checking for purity a matter of checking if exactly one of the counters is zero. Similarly to unit clauses, simplification steps may cause a literal to become pure in a subformula. In addition, a literal that is pure in a formula is not useful for a resolution refutation, which therefore cannot contribute towards inferring $\square$. In ϕ only z is (positive) pure as all other three variables occur both positive and negative. In the context of feature models, pure literals are always representing optional leafs in the tree representation, which can be freely selected or deselected. This means that the validity of the feature selection is not dependent on such optional leaf features.

3.1.3. Autark Assignments

A partial assignment α is called *autark*, if all clauses that contain a variable assigned by α are satisfied by it [22][23]. This property allows to have an easier simplification step when applying an autark assignment to a formula, as all affected clauses can be removed while everything else is kept as is. In essence simplification becomes a clause filtering operation for them. In contrast to that, applying a non-autark assignment may lead to clauses needing to be simplified by taking out unsatisfied literals. Therefore autark assignments capture a kind of independence of the assigned variables w.r.t. a formula [24]. The Monien-Speckenmeyer algorithm [25], which is a DPLL algorithm with an additional heuristic, utilizes autark assignment in addition to the standard unit propagation and pure literals. An example of an autark assignment for ϕ would be (p=0) or (p=1) as they would fulfill clause C4 while not touching the others. Simplification thus would just remove the last clause. Further autark assignments for ϕ are (x=1,y=1) and (z=1), which would cause simplification to filter out all clauses or just the third clause respectively.

3.1.4. Backbone Variables

A variable is called *backbone variable* if it is assigned the same truth value in all satisfying assignments [26]. This means that there is no choice for these variables in a satisfying assignment. The set of all backbone variables is called backbone of a formula. While ϕ has 16 possible assignments in total, only half of these need to be considered when it comes to backbones, since the last clause C4 is tautological and thus always true. Therefore, looking at the remaining 8 assignments there are only two of them that are satisfying, namely (x=1,y=1,z=0) and (x=1,y=1,z=1). Since x and y are both assigned the same values in all satisfying assignments they are backbone variables of ϕ. In terms of the concrete propositions for ϕ this means that "it is raining" and "it is cloudy" are both true in all satisfying assignments whereas "it is windy" can be either true or false. In the context of feature models, the backbone variables resemble the core features of a model, which always need to be part of a valid feature selection. In turn this also means that only the non backbone variables, or the features associated with them, are able to cause variability in a system.

3.1.5. Blocked Clauses

A *blocked clause* is defined in terms of resolution calculus as follows: A clause P is called blocked by a literal l if for all other clauses C containing ¬l, the resolvent $R_{P,C}$ over l is a tautology [27][28]. The literal causing C to be blocked is called blocking literal. A clause may be blocked by multiple of its literals. A blocked clause cannot contribute to finding a refutation proof, as a sound calculus cannot infer a contradiction from tautologies, thus allowing it to be removed from the formula, therefore reducing the number of clauses that need to be combined through resolution. The elimination of blocked clauses has been studied more in depth by Järvisalo et al. [29] and Kiesl [30]. ϕ has three clauses that are blocked by at least one literal. Both clauses C2 and C3 are blocked by y, since the only clause they can be resolved with over y is C4, which result in tautologies. Conversely C4 is also blocked by y as it can only be resolved over y with C2 and C3. In addition to this C4 is blocked by x due to the same reason, as the only available resolvent with C2 is also tautological. However C2 is not blocked by x, as resolving with C1 results in a unit clause which is not tautological, thus having at least one non-tautological resolvent.

3.2. Hardness Measures based on Structure

Second we introduce the selected measures which can be used as indicators of hardness. Here hardness generally refers to the difficulty of a given instance compared to other instances of same problem. The amount of steps a solving algorithm like DPLL/CDCL has to take in order to find a solution is often used as the hardness value. The longer an algorithm will take to find a solution the more difficult the instance.

3.2.1. Clause to Variable Ratio

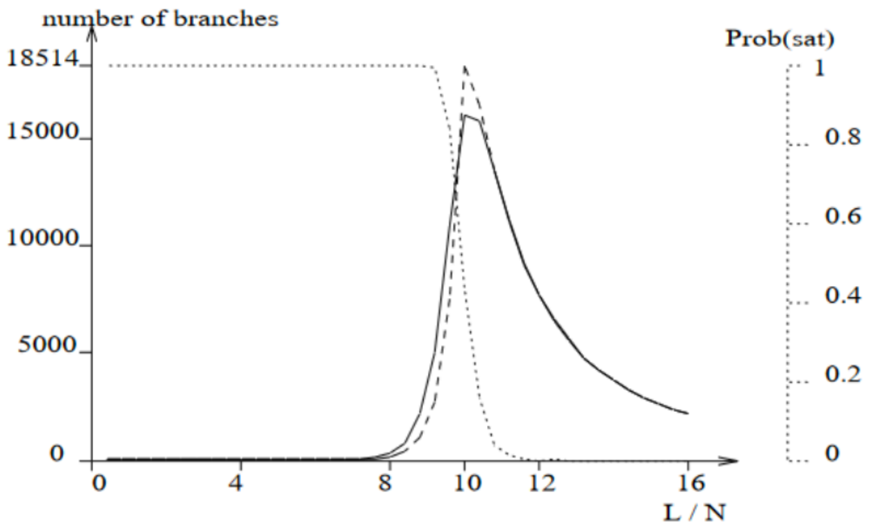


Figure 5. Phase transition plot (taken from [31]). The transition point is found at an L/N ratio between 10 and 11.

The ratio of clauses to variables of an instance can serve as an indicator of hardness of said instance as determined in an experiment about phase transition by Walsh [31]. Phase transition here describes the easy-hard-easy pattern of randomly generated k-SAT instances that arises from plotting the amount of branches, or number of function calls, over the aforementioned ratio. The peak of this plot is the transition point at which the probability for a random instance being satisfiable is about 0.5. This pattern captures the fact that instances with many variables but comparably few clause are very likely to be satisfiable as they are underconstrained on the variables. On the other extreme end, instances with a lot more clauses than variables are overconstrained problems that are very likely to be unsatisfiable. The difficulty arises somewhere in between, where there is just the right amount of clauses to variables that neither over- nor underconstrain the problem and where the probability of being satisfiable is about 0.5. Figure 5 plots the number of branches needed to solve a problem of a certain clause to variable ratio with fixed number of variables. The dotted line plots the probability of the instances to be satisfiable, showing a sharp fall of probability around the transition point while it is near constant outside of this area. For randomized 4-SAT problems, the phase transition point is found at a ratio value between 10 and 12. This means that for 4-SAT formulas with 75 variables specifically, the hardest instances are those that have between 750 to 900 clauses. This ratio will be referred to as CVR. ϕ, which is a 4-SAT instance, has four clauses and four variables, resulting in a CVR of 1 classifying it as an easy instance with near 1 probability of being satisfiable, as seen by the dotted line of Figure 5.

3.2.2. Backbone Variable to Variable Ratio

The hardness of purely satisfiable instances is dependent on the ratio of backbone variables to regular variables. If plotting said ratio together with the number of branches of a solving algorithm run on only satisfiable instances, it shows a similar relationship between the two as seen in the branch and probability plots of Figure 5. This relationship, as analyzed in Achlioptas et al. [32] indicates, that instances with over 90% and instances with under 10% of variables belonging to the backbone are the easy areas left and right of the transition zone, whereas the peak of the branch curve is around the point where 50% of variables belong to the backbone. Intuitively this follows from the observation that overconstrained instances have many backbone variables whereas underconstrained instances have very little of them. It also coincides with the intuition that instances that are not over- or underconstrained, which also have roughly equal amounts of normal and backbone variables, are the hardest to solve.

This ratio will be referred to as BVR. This relationship of hardness and BVR was originally discovered and discussed in Achlioptas et al. [32] which proposed a generator for purely satisfiable but hard instances by using Latin squares, which then can be translated into CNF formulas. ϕ has the two backbone variables x and y, which are half of all variables found in ϕ thus resulting in a BVR of 0.5. If trying to use the plots and numbers from the aforementioned work, the BVR would indicate ϕ to be in the area of hardest instances, which would contradict the observation made with the CVR of ϕ. The problem stems from ϕ not being comparable to the instances generated and examined in the work, the results therefore cannot be directly applied to general CNF formulas.

3.2.3. Centrality

Centrality itself is a measure from graph theory to measure the importance of a vertex within a graph [33]. There are various ways to define a centrality measure, like using a betweenness measure or utilizing eigenvectors and adjacency matrices. The encoding of a formula ϕ as a graph opens it up to those graph-theory-based measures. Applied to a VIG, the centrality of a variable encapsulates the importance of it in a formula, or rather, how much it contributes to its difficulty. As mentioned above, one way to define centrality is by using the adjacency matrix and its eigenvectors, resulting in the equation $A\nu = \lambda x$, where A is the adjacency matrix with $a_{i,j} = 1$ if variable vertex i and j are adjacent in the graph, otherwise $a_{i,j} = 0$, ν is the centrality vector, x the variable vector, and λ the eigenvalue.

Another centrality definition is based on betweenness [34]. One simple way to define the betweenness of a vertex v is to count the number of shortest paths between any other vertices x and y, that contain v. An augmented version of this definition also considers the weights of the graph which opens the measure up to weighted VIGs. The vertex with highest betweenness centrality in ϕ is y as it lies on two shortest paths (i.e., path x-y-z and path p-y-z). All other shortest paths consist of exactly one edge which do not contribute to any vertices' betweenness, putting x, p and z at a betweenness of 0.

3.2.4. Modularity

Modularity is a measure on decompositions of a graph into communities, which are disjoint subsets of the vertices [35]. That is to say, it serves as a kind of quality measure for a division of vertices of a graph into communities. Informally a community decomposition is of good modularity, if it exhibits low coupling and high cohesion, meaning that each community is highly connected within itself while only having few connections to other communities. A simple definition of the modularity measure Q, which is to be maximized, is defined in Newman [36] as Q = $\sum_i (e_{ii} - (\sum_j e_{ij})^2)$, Where e_{ij} is the fraction of edges in the graph that connect vertices of community i with vertices of community j. The squared sum is the expected value a random decomposition would have. Newman [36] has found that in practice a Q value of about 0.3 or higher is indicating significant community structure, meaning that decompositions with a Q value under 0.3 are as good as a randomly picked decomposition in terms of high cohesion and low coupling. For ϕ consider an exemplary decomposition ***D*** into $D_1 = \{p, x, y\}$ and $D_2 = \{z\}$. Using the Q formula above, ***D*** and the VIG of ϕ, Q of ***D*** is calculated as follows: $Q = e_{D_1,D_1} - e^2_{D_1,D_2} + e_{D_2,D_2} - e^2_{D_2,D_1} = \frac{3}{4} - \frac{1}{16} + 0 - \frac{1}{16} = 0.625$. The chosen decomposition ***D*** has therefore significant community structure according to the findings of Newman, as its modularity value is above 0.3 . Comparing this to a different decomposition ***E*** into $E_1 = \{x, p\}$ and $E_2 = \{y, z\}$ and its modularity of 0.25 shows ***E*** to have no significant community structure, as both communities are connected by two edges while each community itself contains only one edge. Figure 6 depicts both decompositions on the VIG of ϕ.

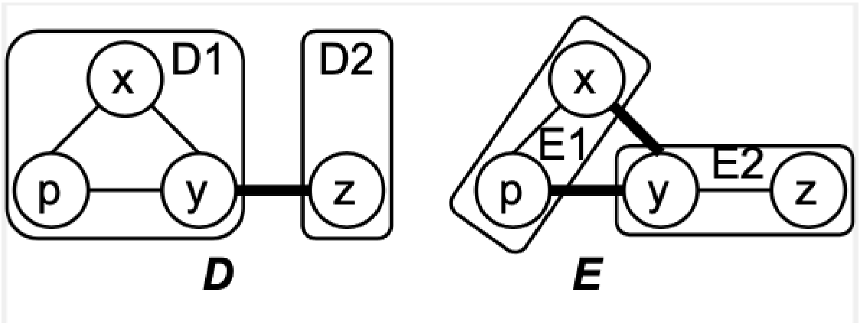


Figure 6. Decompositions ***D*** and ***E*** on the VIG of ϕ

3.2.5. Self-Similarity

The property of self-similarity in general refers to a structure or shape containing itself. In terms of graphs, this means that groups of vertices can be replaced by single vertices resulting in a graph of similar structure [37]. Applying this observation to VIGs allows to explore the self-similarity of a propositional formula. The self-similarity of a formula ϕ can be defined over the function $\beta(s)$ which describes the minimum number of boxes of size s needed to cover the graph of ϕ. If $\beta(s)$ is proportional to $s^{-\lambda}$ for some λ, meaning the function decreases polynomially, then ϕ is called self similar, λ being its fractal dimension. Ansótegui et al. [38] have shown that low fractal dimensions are found in crafted and industrial instances, meaning high self similarity. They also found that similar types of instances have similar fractal dimensions which would allow to classify them by their fractal dimension.

This concludes our overview of the selected structures and measures.

4. Relationships between Structures and Measures

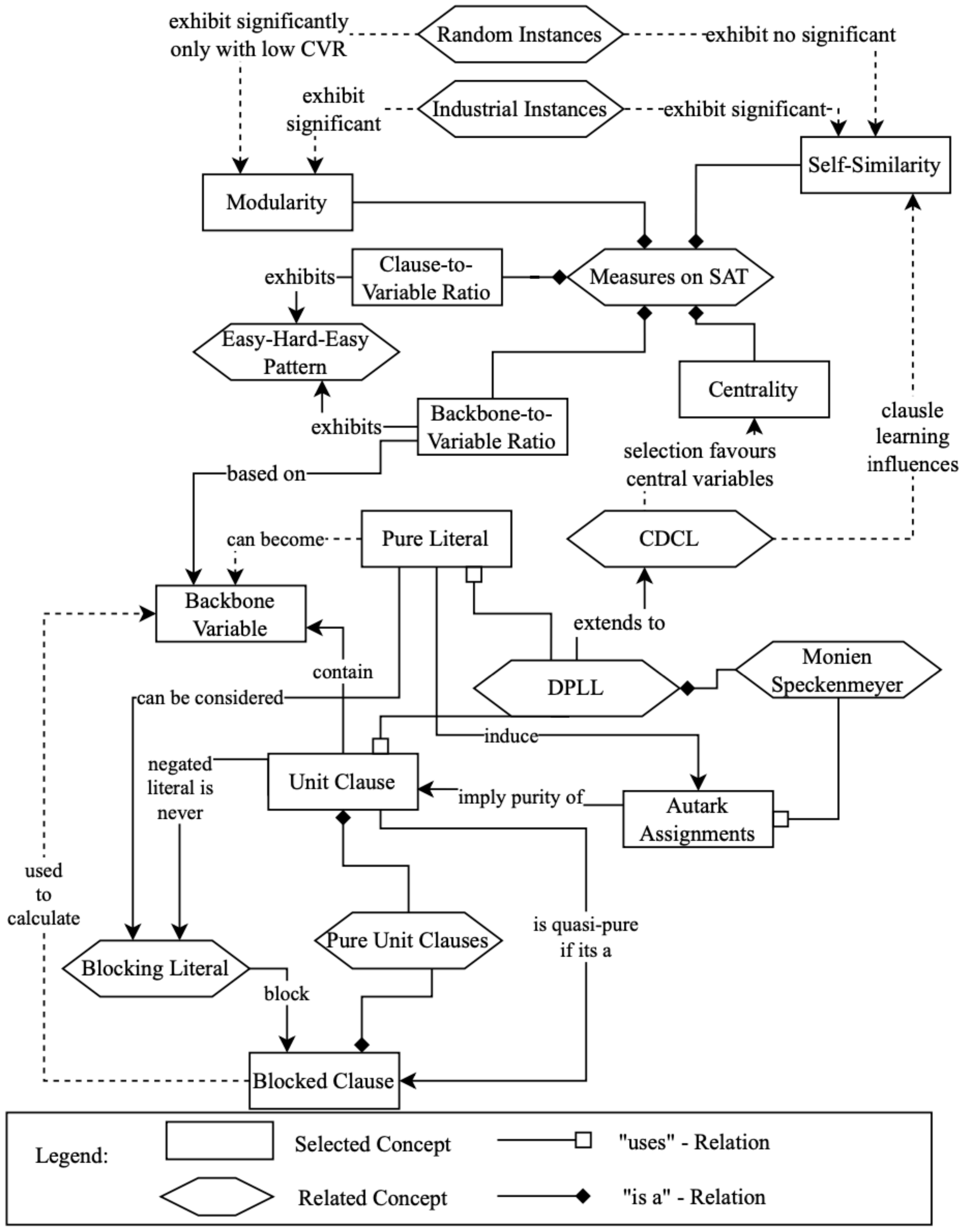


Figure 7. Overview of identified relationships between selected structures/measures.

After we introduced the various structures and measures in Section 3 we will now identify relationships between them. Figure 7 shows the most significant relationships that we have found. Square vertices represent the selected metrics and structures, hexagonal vertices represent related but not explicitly covered concepts. Solid lines indicate certain relationships while dotted lines symbolize uncertain relationships or speculations. Empty box shaped arrowheads symbolize the target being used in the source of the connection while diamond shaped arrows symbolize "is a" relations, e.g., modularity is a measure on SAT. The illustrated relations are covered in more detail in this section.

Pure literals, autark assignments, and unit clauses.

The first relation that can be extracted out of their definition is that pure literals induce an autark assignments. As a pure literal only appears without its negation, a satisfying partial assignment for it will automatically satisfy all clauses containing the literal. This in turn means that they will likely be used by the Monien-Speckenmeyer algorithm [25] as it utilizes autark assignments as its main heuristic. In addition, if an assignment of a unit clause variable is autark it implies the literal of the unit clause to be pure. If this were not the case, there would be a clause containing the unit clause's negated literal, which then would require simplification of the unsatisfied clause, contradicting the definition of an autark assignment.

Backbone, unit clauses, and pure literals.

Another notable relation is that variables of unit clauses are backbone variables of the formula containing them. This also extends to variables that become unit clauses through unit propagation if no other simplification is used beforehand. For instance, looking at the example $\phi = (x) \wedge (\neg x \vee y) \wedge (y \vee z) \wedge (\neg p \vee p \vee x \vee \neg y)$ again.

As has been determined in Section 3.1.4, x and y are both backbone variables of ϕ. Another way to determine this is by looking at the effects of unit propagation while building a satisfying assignment. Since x is a unit in ϕ, it is required to be assigned 1 in any satisfying assignment, thus making it a backbone variable. Consequently simplifying ϕ using this partial assignment reduces it to $\phi' = (y) \wedge (z \vee y)^1$ which similarly requires y now to be assigned 1, also rendering y a backbone variable. On the contrary, the variable of a pure literal is not necessarily a backbone variable. While pure literals can only be assigned one value to benefit the construction of a satisfying assignment of a formula, it does not necessarily require them to be assigned this value. For instance, z is pure (positive) in ϕ and ϕ', however, the satisfying assignments for ϕ are (x=1,y=1,z=1,p=?) and (x=1,y=1,z=0,p=?)^2, hence showing that z is not a backbone variable. Despite that, one could say they still have backbone potential, that is if a unit clause containing a pure literal emerges through unit propagation the variable of a pure literal becomes a backbone variable. For instance, if ϕ would contain $(\neg y \vee z)$ as its third clause instead, doing two simplifications of unit propagation would turn (z) into a pure unit clause, thus adding it to the backbone of ϕ.

Blocked clauses, unit clauses, and pure literals.

Another relation regarding unit clauses is one with blocked clauses. Any literal of a clause, whose negation exists in a unit clause, can not be a blocking literal. In other words, if a literal x exists as a unit clause, then ¬x can never block a clause as a blocking literal. Since resolving with a unit clause does not add literals to the resolvent, the result can not become tautological by such a resolution step, whereas resolving with a non-unit clause will add the remaining literals, causing tautologies to emerge whenever a literal and its negation were present. Using already-inferred unit clauses can therefore speed up looking for blocking literals within a clause. Instead of having to test all literals for being blocking or not, those with negated unit clauses can be safely filtered as non blocking. Furthermore, if a unit clause is blocked, it is quasi-pure, meaning it will become a pure literal after removing all tautologies. In order to block a unit clause x in a formula ϕ, all its possible resolvents in ϕ need to be tautological. This can only happen by resolution with clauses that contain ¬x, w and $\neg w$ for any other variable w found in ϕ. Therefore ¬x only occurs in tautological clauses, thus making x pure if removing all tautological clauses from the formula.

In addition to that, the notion of a literal blocking a clause by only resulting in tautological resolvents can be extended by including literals that do not produce any resolvents at all.

This would notably classify pure literals as blocking literals, as a resolvent over such a pure literal would require its negation to be present in other clauses, which by definition of pure literals is not possible. This classification as being blocking is justified as resolving with a clause containing a pure literal will at most end with the pure literal in a unit clause, but never with the empty clause as a resolvent. They therefore cannot contribute to finding a refutation by resolution as all resolvents will always contain the pure literal.

Combining the observations about pure literals and unit clauses alters the notion that literals of unit clauses are never blocking, as they would be considered blocking if they are pure. It would also mean that unit clauses are blocked clauses if they are pure.

Blocked clauses also seem to have relations to backbone variables as suggested in Parkes [39] which use blocked clauses to find the backbone variables of a formula.

Phase transition

The next considered relations are the ones between the BVR and CVR, the most obvious is that both exhibit a similar easy-hard-easy pattern, as described in Achlioptas et al. [32]. However, the difference lies in the fact that the BVR is a measure on exclusively satisfiable instances whereas CVR is a measure over both satisfiable and unsatisfiable instances suggesting they may not be quite comparable. In addition to that, the CVR can be applied to any CNF formula whereas the BVR can not, as it is an observation based on the specific formula type used in the generator approach. A separate study on the link between hardness and the BVR on general instances is needed in order to directly compare it to the CVR of a formula. However, in Zhang [40], which looked at phase transition patterns of a variety of ratios of asymmetric traveling salesmen problems, the authors concluded that all examined ratios exhibited a similar phase transition pattern.

Centrality, decision variables, and propagation variables.

Centrality is mostly indirectly related to other presented measures and structures. In Katsirelos and Simon [41] it was shown that decision variables, which are variables picked by a solver after no further simplification steps like unit propagation can be applied, have a higher average centrality compared to the average centrality of all variables. Their observations suggest that CDCL procedures and their heuristics favor picking variables with higher centrality over those with lower centrality, which may be an explanation as to why CDCL procedures tend to concentrate on certain parts of the search space.

Modularity and phase transition.

For the modularity measure and community structure Ansótegui et al. [42] has found that industrial instances have high modularity compared to randomized instances. They also found that for random instances, modularity only has relevance for instances with low CVR. Similar observations have been found in Ansótegui et al. [43] where they also show a clear decreasing trend of the modularity for increasing CVR values of random instances. They also found that the modularity is smallest for those instances at the peak of the phase transition, implying that the hardest instances have comparably the lowest community structure. They conjecture that CDCL methods are relatively efficient and successful when used on industrial instances as they utilize the community structure found within them.

Self-similarity.

For self-similarity, Ansótegui et al. [38] showed that industrial and crafted instances exhibit a low fractal dimension, indicating self-similarity. However, opposite to modularity and centrality, self similarity decreases upon clause learning. Their work also observed a connection between the decay used to determine fractality and the CVR of random instances, noting that the decay is exponential at the phase transition point, resulting in a larger value for the fractal dimension and therefore smaller self-similarity.

5. Related Work

Besides the selection of structures and measures in this survey, there are many other measures and structures that could be examined, compared, and put into relation to those already covered. Among them are further structural properties like scale-free [44], small-world [45] and entropy [46]. Ansótegui et al. [47] has shown experimentally that industrial instances tend to be scale-free, similarly Walsh [45] has found small-world topology in SAT problems, whereas Zhang et al. [46] found the entropy measure to be inverse proportional to problem hardness. Besides these, a more in depth review of the various concrete measures for centrality could lead to a better understanding of potential relations to modularity and the behavior of CDCL solvers. In addition examining combined measures could be of relevance, as it has been found in [48] that measures on their own tend to not correlate with hardness. Combined hardness measures also find application in portfolio type solvers like SATzilla [7]. Here the general idea is to use multiple solvers that are efficient for different classes of instances and picking the best suited to solve a specific instance according to its properties. More precisely, first a presolver is run for a fixed amount of time. If the presolver did not find a solution, a set of measures and features on the instance, like the CVR or vertex degree statistics of the VIG, are computed in order to pick the solver best suited to finally solve the instance. HordeSat is another portfolio approach that has been proposed by Balyo et al. [8] which runs several CDCL type solvers in parallel while periodically collecting conflict clauses from each solver before passing them to all others in order to avoid redundant inferences.

Another potential avenue to pursue is to find parallels and differences between tree decompositions of a VIGs and their community structures, and if their associated measures treewidth [49] and modularity behave similar or different. Besides that, the treewidth of industrial instances and its relation to CDCL runtime have been investigated by Mateescu [50]. A further topic potentially related to those covered in this surveys are redundancy properties and redundant clauses. Kiesl [30] has examined the notions of locally redundant clauses, globally redundant clauses, and proof systems that are based on them. The work also identified blocked clauses as a form of locally redundant clauses. In addition Fourdrinoy et al. [51] has examined the usefulness of eliminating unit-propagation-redundant clauses in speeding up solution algorithms as a form of utilizing redundancy properties. In addition to adding these other structures and measures to the picture and finding relations between them, a more in depth or experimental examination of the covered structures and measures could provide further and better insight into the mentioned relations or potentially prove or disprove the more speculative relations that we have identified.

6. Conclusion

SAT is a central and difficult problem, finding application in industry and academia alike. While algorithms cannot solve general instances efficiently, many have emerged that can solve certain classes of instances efficiently by exploiting structure within the formula. This is reflected by the observation that industrial instances can be solved more efficiently than randomized instances due to the additional information encoded by their structure. In this paper, we have presented a selection of structures and measures for SAT formulas that are related or adjacent to state of the art solvers and examined relations between them.

Among those are pure literals inducing autark assignments, autark assignments over unit clauses implying their purity and the relationship of blocked literals to unit clauses and pure literals. In addition, we highlighted the similarity of CVR and BVR and the general relationship of easy-hard-easy pattern exhibiting ratios, industrial instances exhibiting properties like modularity and self-similarity more than randomized instances, as well as high-centrality variables being favored by CDCL procedures. In addition to identifying relationships between the selected measures and structures, we also gave an overview over other related concepts and their application. For instance, portfolio type solvers like SATzilla or HordeSat select the solver method best-suited for an instance according to measures on them.

Footnotes

[1] p=? being shorthand for that both assignments with p=1 and p=0 are satisfying

[2] $(\neg p \lor p \lor x \lor \neg y)$ has been assumed removed by simplification as it is tautological

References

1. ^*Richard M. Karp. 1972. Reducibility among Combinatorial Problems. Springer US, Boston, MA, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9*
2. ^*Stephen A. Cook. 1971. The complexity of theorem-proving procedures. Proceedings of the third annual ACM symposium on Theory of computing (1971).*

3. ^*Armin Biere and W. Kunz. 2002. SAT and ATPG: Boolean engines for formal hardware verification. IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, 782– 785. https://doi.org/10.1109/ICCAD.2002. 1167620*
4. ^*Thomas Ball, Byron Cook, Vladimir Levin, and Sriram Rajamani. 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. Integrated Formal Methods, Volume 2999 of Lecture Notes in Computer Science 2999, 1–20. https://doi.org/10.1007/978-3-540-24756-2_1*
5. ^*Kyo Chul Kang, Sholom Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study.*
6. a, b*Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. FeatureOriented Software Product Lines. Springer-Verlag.*
7. a, b*Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. SATzilla: Portfolio-based Algorithm Selection for SAT. CoRR abs/1111.2249 (2011). arXiv:1111.2249 http://arxiv.org/abs/1111.2249*
8. a, b*Tomas Balyo, Peter Sanders, and Carsten Sinz. 2015. HordeSat: A Massively Parallel Portfolio SAT Solver. https://doi.org/10.1007/978-3-319-24318-4_12*
9. ^*Domagoj Babic and Frank Hutter. 2008. Spear theorem prover. (01 2008).*
10. ^*William F. Dowling and Jean H. Gallier. 1984. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. J. Log. Program. 1 (1984), 267– 284.*
11. ^*Marcílio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. 2009. SATbased analysis of feature models is easy. SPLC, 231–240. https://doi.org/10.1145/1753235.1753267*
12. ^*Martin Kreuzer and Stefan Kühling. 2006. Logik für Informatiker.*
13. ^*Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. 2022. Tseitin or not Tseitin? The Impact of CNF Transformations on Feature-Model Analyses. 110:1–110:13.*
14. ^*G. S. Tseitin. 1983. On the Complexity of Derivation in Propositional Calculus. Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483. https://doi.org/10.1007/978-3-642-81955-1_28*
15. ^*Melven R. Krom. 1967. The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary. Mathematical Logic Quarterly 13 (1967), 15–20.*
16. ^*Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. 1979. A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. Inf. Process. Lett. 8 (1979), 121–123.*
17. ^*J. A. Robinson. 1967. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. Communications of the ACM, vol. 5 (1962), pp. 394–397. The Journal of Symbolic Logic 32, 1 (1967), 118–118. https: //doi.org/10.2307/2271269*

18. ^J. Marques-Silva and Karem Sakallah. 2003. GRASP - A New Search Algorithm for Satisfiability.

19. ^Robert Kowalski and Donald Kuehner. 1971. Linear Resolution with Selection Function. Artificial Intelligen ce 2 (12 1971), 227–260. https://doi.org/10.1016/0004-3702(71)90012-9

20. a, bArmin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2009. Handbook of Satisfiability. F rontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press. http://dblp.uni-trier.de/db/series/faia/f aia185.html

21. a, bJacobo Torán Uwe Schöning. 2013. The Satisfiability Problem, Algorithms and Analyses. Lehmanns Medi a GmbH, Berlin.

22. ^Oliver Kullmann. 2001. On the use of autarkies for satisfiability decision. Electronic Notes in Discrete Math ematics 9 (06 2001), 231–253. https://doi.org/10.1016/S1571-0653(04)00325-7

23. ^Mark Liffiton and Karem Sakallah. 2008. Searching for Autarkies to Trim Unsatisfiable Clause Sets. In The ory and Applications of Satisfiability Testing – SAT 2008, Hans Kleine Büning and Xishun Zhao (Eds.). Spri nger Berlin Heidelberg, Berlin, Heidelberg, 182–195.

24. ^Mark Liffiton and Karem Sakallah. 2008. Searching for Autarkies to Trim Unsatisfiable Clause Sets. In The ory and Applications of Satisfiability Testing – SAT 2008, Hans Kleine Büning and Xishun Zhao (Eds.). Spri nger Berlin Heidelberg, Berlin, Heidelberg, 182–195.

25. a, bBurkhard Monien and Ewald Speckenmeyer. 1985. Solving satisfiability in less than 2n steps. Discrete Ap plied Mathematics 10 (03 1985), 287–295. https://doi.org/10.1016/0166-218X (85)90050-2

26. ^Philip Kilby, John Slaney, Sylvie Thiebaux, and Toby Walsh. 2005. Backbones and Backdoors in Satisfiabili ty. Proceedings of the National Conference on Artificial Intelligence 3, 1368–1373.

27. ^Tomas Balyo, Andreas Fröhlich, Marijn Heule, and Armin Biere. 2014. Everything You Always Wanted to K now about Blocked Sets (But Were Afraid to Ask). https://doi.org/10.1007/978-3-319-09284-3_24

28. ^Matti Järvisalo, Armin Biere, and Marijn Heule. 2010. Blocked Clause Elimination. 129–144. https://doi.org/ 10.1007/978-3-642-12002-2_10

29. ^Matti Järvisalo, Armin Biere, and Marijn Heule. 2010. Blocked Clause Elimination. 129–144. https://doi.org/ 10.1007/978-3-642-12002-2_10

30. a, bBenjamin Kiesl. 2019. Structural Reasoning Methods for Satisfiability Solving and Beyond. Ph. D. Dissert ation. Technische Universität Wien.

31. a, bIan P Gent and Toby Walsh. 1994. The SAT phase transition. In ECAI, Vol. 94. PITMAN, 105–109.

32. a, b, cDimitris Achlioptas, Carla Gomes, Henry Kautz, and Bart Selman. 2000. Generating Satisfiable Proble m Instances. Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00) (05 2

000).

33. ^*Phillip Bonacich. 1987. Power and Centrality: A Family of Measures. Amer. J. Sociology 92 (1987), 1170 – 1182.*
34. ^*Sima Jamali and David G. Mitchell. 2017. Improving SAT Solver Performance with Structure-based Preferential Bumping. In Global Conference on ArtificialIntelligence.*
35. ^*Mark E. J. Newman. 2006. Modularity and community structure in networks. Proceedings of the National Academy of Sciences of the United States of America 103 23 (2006), 8577–82. https://api.semanticscholar.org/CorpusID:2774707*
36. a, b*M Newman. 2004. Fast algorithm for detecting community structure in networks. Phys. Rev. E Stat. Nonlin. Soft. Matter. Phys. 69(6 Pt 2), 066133. Physical review. E, Statistical, nonlinear, and soft matter physics 69 (07 2004), 066133. https: //doi.org/10.1103/PhysRevE.69.066133*
37. ^*Kiran Chilakamarri, M. Khan, C. Larson, and Cj Tymczak. 2013. Self-Similar Graphs. (10 2013).*
38. a, b*Carlos Ansótegui, Maria Bonet, Jesús Giráldez-Cru, and Jordi Levy. 2013. The Fractal Dimension of SAT Formulas. https://doi.org/10.1007/978-3-319-08587-6_8*
39. ^*Andrew J. Parkes. 1997. Clustering at the Phase Transition. In AAAI/IAAI. 340– 345. http://www.aaai.org/Library/AAAI/1997/aaai97-053.php*
40. ^*Weixiong Zhang. 2004. Phase Transitions and Backbones of the Asymmetric Traveling Salesman Problem. J. Artif. Intell. Res. (JAIR) 21 (04 2004), 471–497. https://doi.org/10.1613/jair.1389*
41. ^*George Katsirelos and Laurent Simon. 2012. Eigenvector Centrality in Industrial SAT Instances. 348–356. https://doi.org/10.1007/978-3-642-33558-7_27*
42. ^*Carlos Ansótegui, Maria Bonet, Jesús Giráldez-Cru, and Jordi Levy. 2016. Community Structure in Industrial SAT Instances. Journal of Artificial Intelligence Research 66 (06 2016). https://doi.org/10.1613/jair.1.11741*
43. ^*Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. 2012. The Community Structure of SAT Formulas. 410–423. https://doi.org/10.1007/978-3-642-316128_31*
44. ^*Rita Albert, Hawoong Jeong, and Albert-Laszlo Barabasi. 1999. Diameter of the World-Wide Web. Nature 401 (09 1999), 130–131. https://doi.org/10.1038/43601*
45. a, b*Toby Walsh. 2002. Search in a Small World. IJCAI International Joint Conference on Artificial Intelligence 2 (04 2002).*
46. a, b*Zaijun Zhang, Daoyun Xu, and Jincheng Zhou. 2021. A Structural Entropy Measurement Principle of Propositional Formulas in Conjunctive Normal Form. Entropy 23 (03 2021), 303. https://doi.org/10.3390/e23030303*

47. ^Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. 2009. On the Structure of Industrial SAT Instances. In International Conference on Principles and Practice of Constraint Programming.
48. ^Edward Zulkoski, Ruben Martins, Christoph M. Wintersteiger, Jia Hui Liang, K. Czarnecki, and Vijay Ganes h. 2018. The Effect of Structural Measures and Merges on SAT Solver Performance. In International Confere nce on Principles and Practice of Constraint Programming.
49. ^Neil Robertson and P. D. Seymour. 1983. Graph minors. I. Excluding a forest. Journal of Combinatorial Theo ry. Series B 35, 1 (Aug. 1983), 39–61. https://doi.org/10.1016/0095-8956(83)90079-5
50. ^Robert Mateescu. 2011. Treewidth in Industrial SAT Benchmarks. Technical Report MSR-TR-2011-22. http s://www.microsoft.com/en-us/research/publication/treewidth-in-industrial-sat-benchmarks/
51. ^Olivier Fourdrinoy, Eric Gregoire, Bertrand Mazure, and Lakhdar Sais. 2007.Eliminating Redundant Claus es in SAT Instances. 71–83. https://doi.org/10.1007/978-3-540-72397-4_6

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.