Introducing the Black-and-White SAT Problem and the CSFLOC SAT Solver

Habilitation dissertation

Gábor Kusper

University of Debrecen

Faculty of Informatics

Debrecen
2020

Contents

1 Introduction

The intention of this dissertation is to introduce the Black-and-White SAT problem, the CSFLOC SAT solver, and some predecessor of those results from my earlier works. This work is organized around theorems from my selected papers.

Selected Papers

My selected papers are attached to this dissertation. These are the following ones:

- Selected papers which are summarized in Chapter 3:
  - G. Kusper, Cs. Biró, *Convert a Strongly Connected Directed Graph to a Black-and-White 3-SAT Problem by the Balatonboglár Model*, submitted to Theory of Computing, 17 pages, arrived on 24.08.2019, status: under review, [87].
  - G. Kusper, Cs. Biró, T. Balla, *Investigation of the Efficiency of Conversion of Directed Graphs to 3-SAT Problems*, Proceedings of SACI-2020, DOI: 10.1109/SACI49304.2020.9118786, 2020, [93].
  - Cs. Biró, G. Kusper, *Equivalence of Strongly Connected Graphs and Black-and-White 2-SAT Problems*, Miskolc Mathematical Notes, Vol. 19, No. 2, pp. 755-768, 2018, [88].
- Selected papers which are summarized in Chapter 4:
  - Cs. Biró, G. Kusper, T. Tajti, *How to generate weakly nondecisive SAT instances*, Proceedings of SISY 2013, DOI: 10.1109/SISY.2013.6662583, pp. 265–269, 2013, [99].
  - G. Kusper, L. Csőke, G. Kovásznai, *Simplifying the propositional satisfiability problem by sub-model propagation*, Annales Mathematicae et Informaticae, Vol. 35, ISSN 1787-5021, pp. 75–94, 2008, [90].
- Selected papers which are summarized in Chapter 5:
  - G. Kusper, Cs. Biró, Gy. B. Iszály, *SAT solving by CSFLOC, the next generation of full-length clause counting algorithms*, Proceedings of IEEE International Conference on Future IoT Technologies 2018, 2018, [97].
  - G. Kusper, Cs. Biró, *Solving SAT by an Iterative Version of the Inclusion-Exclusion Principle*, Proceedings of SYNASC 2015, pp. 189–190, 2015, [98].

Selected Theorems

The dissertation contains all definitions and lemmas, which are needed to prove my selected theorems, which are the following ones:

- The Transitions Theorem, which is the main result of my selected paper: [87], and which is presented in Chapter 3 as Theorem 4.
- The Sharp Threshold of WnDGen, which is the main result of my selected paper: [99], and which is presented in Chapter 4 as Theorem 11.
- The Soundness and Completeness of CSFLOC, which is the main result of my selected paper: [97], and which is presented in Chapter 5 as Theorem 13.

This dissertation contains also some more results and remarks which was not present in the original papers. These remarks reflect how are those results interconnected, what is my current understanding about them.

This dissertation is based on published results but it contains also a massive new part [87], which is submitted to ToC (Theory of Computing) journal, which is a Q1 journal.

In my selected papers we, me and my coauthors, studied the propositional satisfiability problem, and almost all of our works have some contacts to the Black-and-White SAT problem. This dissertation highlights those connections.

Propositional satisfiability is the problem of determining, for a formula of the propositional logic, if there is an assignment of truth values to its variables for which that formula evaluates to true. By SAT we mean the problem of propositional satisfiability for formulae in conjunctive normal form (CNF). SAT is one of the most-researched NP-complete problems [21]. SAT plays an important role in several fields of computer science, including theoretical computer science, artificial intelligence, hardware design, and formal verification [10]. A Black-and-White SAT problem is a SAT problem which has only two solutions: where each variable is true (the white assignment), and where each variable is false (the black one) [88]. Strongly connected directed graphs can be represented by Black-and-White SAT problems using one of our models:

- strong model [88, 87],
- weak model [87],
- Balatonboglár model [87], and
- simplified Balatonboglár model [94, 93].

The WnDGen SAT problem generator [99] can generate Black-and-White SAT problem, see Theorem 12, which is one of our new, unpublished result of this dissertation.

The CSFLOC SAT solver [97] is an iterative version of the inclusion-exclusion principle. It counts those full-length clauses which are subsumed by the input SAT problem. This

SAT solver can solve efficiently those SAT problems which are generated by WnDGen, and those which are generated from directed graph, and which are Black-and-White SAT problems, if the directed graph is strongly connected.

Organization

In the following, the dissertation is divided into chapters.

In Chapter 2, we give a short survey on the field of SAT problem, and locates my work in the big pictures. Some of my results are cited here. These are those results which are not selected but which are SAT related.

In Chapter 3, we present how can we represent a directed graph as a SAT problem. We present 4 models. We show that if the directed graph is strongly connected then its representation in those models are Black-and-White SAT problems.

In Chapter 4, we present the Weakly-Nondecensive SAT problem. We show how can one generate Weakly-Nondecensive SAT problems. The generator algorithm, WnDGen, has two parameters: C - the generator clause, k - the length of clauses. We show that if C is the white clause or the black clause, and $|C| \geq 2k + 3$, then the generated Weakly-Nondecensive SAT problem is a Black-and-White SAT problem, see Theorem 12.

In Chapter 5, we present `CSFLOC`, a novel SAT solver algorithm. We show that it is sound and complete, see Theorem 13. We compare its performance to state-of-the-art SAT solvers on problems which are generated by our models.

Chapter 6 concludes and summarizes the dissertation.

2 Background and Related Work

This dissertation focuses on my results in the field of the propositional satisfiability problem, known as SAT problem in the literature. This chapter gives a short overview of this wide field, although very good and broad surveys are available [43, 49, 10, 51, 42].

Propositional satisfiability is the problem of determining, for a formula of the propositional logic, if there is an assignment of truth value to its variables for which that formula evaluates to true. By *SAT* we mean the problem of propositional satisfiability for formulae in conjunctive normal form (*CNF*).

SAT is one of the most-researched **NP**-complete problems [21] in several fields of computer science, including theoretical computer science, artificial intelligence, hardware design, and formal verification [10].

2.1 *Pure literal, blocked clause, nondecisive clause*

In this section we give some basic notions and overview the field of blocked clauses. This section is the extended version of the Introduction from our paper [90].

Given a boolean variable x, there exist two *literals*, the positive literal x and the negative literal $\neg x$. A *clause* is a disjunction of literals, an *assignment* or *cube* is a conjunction of literals. Either a clause or a cube can be considered as a finite set of literals. A *clause set* or *SAT problem* is a conjunction of clauses. An assignment is a solution of a clause set, if its intersection with each clause from the clause set is not empty.

Clause A is subsumed by a clause set if and only if (iff) the clause set contains a clause, say B, such that B is a subset of A.

A clause is entailed by a clause set iff the clause is the logical consequence of the clause set. Note that subsumed clauses are also entailed. A clause is independent in a clause set iff it is not entailed. A full-length clause is independent in a clause set iff it is not subsumed.

Resolution on the clauses $a \vee A$ and $\neg a \vee B$ is equal to $A \vee B$, and the result is called resolvent.

A literal is pure in a clause set iff its negation does not occur in the clause set. A literal is blocked in a clause set iff each resolution on it in the clause set results in a tautology. A clause is blocked iff it contains a blocked literal. A literal is nondecisive in a clause set iff each resolution on it in the clause set results in a tautology or the union of the literal (as a set) and the resolvent is subsumed by the clause set. A clause is nondecisive iff it

contains a nondecisive literal. Note that pure literals are blocked, and blocked literals are nondecisive.

A more formal definitions can be found in [90].

A clause with a pure literal can be added or deleted from a clause set without changing its satisfiability. The notion of blocked [52, 53] and nondecisive clause [39] was introduced by O. Kullmann and A. V. Gelder. They showed that a blocked or nondecisive clause can be added or deleted from a clause set without changing its satisfiability, too.

I contributed to this field by defining two new notions, the notion of weakly nondecisive clause, and the notion of strongly nondecisive clause [86, 90].

A literal is weakly nondecisive in a clause set iff each resolution on it in the clause set results in a tautology or the resolvent is subsumed by the clause set. A clause is weakly nondecisive iff it contains a weakly nondecisive literal.

A literal is strongly nondecisive in a clause set iff each resolution on it in the clause set results in a tautology or the union of the literal (as a set) and the resolvent is entailed by the clause set. A clause is strongly nondecisive iff it contains a strongly nondecisive literal.

I was able to prove that if we have a full-length independent blocked clause, say $(a \vee A)$, where a is the blocked literal and A is the rest of the clause, then $a \vee \neg A$ is a solution of the clause set, see the Independent Blocked Clause Rule in [90]. I was able to prove that the same property holds for full-length independent (weakly / strongly) nondecisive clauses.

2.2 Polynomial time solvable SAT problems

This section overviews some restricted SAT problems which are solvable in polynomial time. This section is the extended version of the Introduction from my paper [91].

Some particular cases of *SAT* are solvable in polynomial time:

1. The restriction of SAT to instances where all clauses have length at most k is denoted by k-SAT. Of special interest are *2-SAT* and *3-SAT*: 3 is the smallest value of k for which k-SAT is **NP**-complete, while 2-SAT is solvable in linear time [2].

2. *Horn SAT* is the restriction to instances where each clause has at most one positive literal. Horn SAT is solvable in linear time [26, 75], as are a number of generalizations such as *renamable Horn SAT* [3], *extended Horn SAT* [16] and *q-Horn SAT* [14].

3. The hierarchy of *tractable* satisfiability problems [22], which is based on Horn SAT and 2-SAT, is solvable in polynomial time. An instance on the k-th level of the hierarchy is solvable in $O(n^{k+1})$ time.

4. *Nested SAT*, in which there is a linear ordering on the variables and no two clauses overlap with respect to the interval defined by the variables they contain [50].

5. SAT in which no variable appears more than twice. All such problems are satisfiable if they contain no unit clauses [81].

6. *r,r-SAT*, where r,s-SAT is the class of problems in which every clause has exactly r literals and every variable has at most s occurrences. All r,r-SAT problems are satisfiable in polynomial time [81].

7. A formula is *SLUR* (Single Lookahead Unit Resolution) *solvable* if, for all possible sequences of selected variables, algorithm SLUR does not give up. Algorithm SLUR is a nondeterministic algorithm based on unit propagation. It eventually gives up the search if it starts with, or creates, an unsatisfiable formula with no unit clauses. The class of SLUR solvable formulae was developed as a generalization including Horn SAT, renamable Horn SAT, extended Horn SAT, and the class of CC-balanced formulae [76].

8. I have contributed to this list in 2005. *Resolution-Free* SAT Problem, where every resolution results in a tautology, is solvable in linear time [92].

9. I have contributed to this list also in 2007. *Blocked* SAT Problem is also a restriction of SAT to instances where each clause of the clause set is blocked, i.e., we have at least one blocked literal in each clause. It is a generalization of the Resolution-Free SAT problem, where all literals are blocked. The notion of blocked clause was introduced by Kullmann in [52, 53]. He showed that a blocked clause can be added or deleted from a clause set without changing its satisfiability. From this it is easy to show that any blocked clause set is satisfiable, because if we remove all blocked clauses then we obtain the trivially satisfiable clause set. But this process do not show how to find a model for blocked clause sets. The Blocked SAT Solver algorithm [91] provides a model for Blocked SAT problems in linear time, if we know at least one blocked literal per clauses. It uses heavily the notion of sub-model [90].

10. *Linear autarkies* can be found in polynomil time [54]. A partial assignment is an *autarky* if it satisfies all clauses such that they have a common variable. For example, a pure literal is an autarky. Linear autarkies include q-Horn formulae, and incomparable with the SLUR [63].

11. *Matched expressions* are recognized by creating a bipartite graph $(V1, V2, E)$, such that vertices of $V1$ represent clauses, vertices of $V2$ represent variables, and there is an edge from clause C to variable v if and only if C contains v or $\neg v$. If there is a total matching in this graph, i.e., there is a subset of edges, such that each clause and each variable are present but only once, then we say that the formula is matched. Matched formulae are satisfiable [28]. Total matching can be constructed, if it exists, in polynomial time. The class of matched formulae is incomparable with the q-Horn and SLUR classes.

2.3 SAT solvers

This section overviews the state-of-the-art SAT solvers. This section is the extended version of the Introduction from [89].

The *DIMACS CNF* format Each SAT solver uses the *DIMACS CNF* file format [1]. It represents a boolean variable by its 1-based index. A positive number corresponds to a positive literal, a negative number to a negative literal. Each line is a clause, and each line is terminated by a "0".

A DIMACS CNF file must contain a problem line which starts by *p cnf* which tells the number of variables and the number of clauses information:

```
p cnf #variables #clauses
```

It can contain also comments, which start by the letter "c". An example for DIMACS CNF file:

```
c This is a SAT problem with 3 variables and 5 clauses.
c It is unsatisfiable.
p cnf 3 5
-1 2 0
-2 3 0
1 -3 0
1 2 3 0
-1 -2 -3 0
```

Sequential SAT solvers Modern sequential *SAT* solvers are based on the Davis-Putnam-Logemann-Loveland (*DPLL*) [23] algorithm. This algorithm performs Boolean Constraint Propagation (*BCP*) and backtrack search, i.e., at each node of the search tree it selects a decision variable and assigns a truth value to it, then steps back when a conflict occurs. Conflict-driven clause learning, see *Chapter 4* in [10], is based on the idea that conflicts can be exploited to reduce the search space. If the method finds a conflict, then it analyzes this situation, determines a sufficient condition for this conflict to occur, in form of a learned clause, which is then added to the formula, and thus avoids that the same conflict occurs again. This form of clause learning was first introduced in the *SAT* solver `GRASP` [65] in 1996. Besides clause learning, lazy data structures are one of the key techniques for the success of *CDCL SAT* solvers, such as "watched literals" as pioneered in 2001, by the *CDCL* solver `Chaff` [69, 62] . Another important technique is the use of the `VSIDS` heuristics and the first-UIP backtracking scheme. In the state-of-the-art *CDCL* solvers, like `Lingeling` [11, 12], `Glucose` [6], `COMiniSatPS` [20] and `MapleCOMSPS` [58], several other improvements are applied. Besides enhanced preprocessing techniques, like e.g. failed literal

[1] `http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf`

detection, variable elimination, and blocked clause elimination, clause deletion strategies and restart policies have a great impact on the performance of the *CDCL* solver. In 2016 Liang et al. [59] introduced the conflict history-based (*CHB*) branching heuristic which seems to be better than *VSIDS*. In the same year, the same group introduced an even more efficient heuristics, called learning rate branching (*LRB*) [58]. The newest branching and restart policies are based on machine learning [60].

Lookahead-based SAT solvers Lookahead-based SAT solvers, see Chapter 5 in [10], combine the *DPLL* algorithm with lookaheads, which are used in each search node to select a decision variable and at the same time to simplify the formula. One popular way of lookahead measures the effect of assigning a certain variable to a certain truth value: *BCP* is applied, and then the difference between the original clause set and the reduced clause set is measured (by using heuristics). In general, the variable for which the lookahead on both truth values results in a large reduction of the clause set is chosen as the decision variable. The first lookahead *SAT* solver was `posit` [29] in 1995. It already applied important heuristics for pre-selecting the "important" variables, for selecting a decision variable, and for selecting a truth value for it. The lookahead solvers `satz` [57] and `OKsolver` [55] further optimized and simplified the heuristics, for example `satz` does not use heuristics for selecting a truth value *(rather prefers* true*)*, and `OKsolver` does not apply any pre-selection heuristics. Furthermore, `OKsolver` added improvements like local learning and autarky reasoning. In 2002, the solver `march` [33] further improved the data structures and introduced preprocessing techniques. As a variant of `march`, `march_cc` [34] can be considered as a case splitting tool. It produces a set of cubes, where each cube represents a branch cutoff in the *DPLL* tree constructed by the lookahead solver. It is also worth to mention that `march_cc` outputs learnt clauses as well, which represent refuted branches in the *DPLL* tree. The resulting set of cubes represents the remaining part of the search tree, which was not refuted by the lookahead solver itself.

Parallel SAT solvers Since multi-core architectures are common today, the need for parallel *SAT* solvers using multiple cores has increased considerably.

In essence, there are two approaches to parallel *SAT* solving [31]. The first group of solvers typically follow a divide-and-conquer approach. They split the search space into several subproblems, sequential *DPLL* workers solve the subproblems, and then these solutions are combined in order to create a solution to the original problem. This first group uses relatively intensive communication between the nodes. They do for example load balancing and dynamic sharing of learned clauses.

The second group apply portfolio-based *SAT* solving. The idea is to run independent sequential *SAT* solvers with different restart policies, branching heuristics, learning heuristics, etc. `ManySAT` [30] was the first portfolio-based parallel *SAT* solver. `ManySAT` applies several strategies to the sequential *SAT* solver `MiniSAT`. `Plingeling` [11, 12] follows a

similar approach, and uses the sequential SAT solver `Lingeling`. In most of the state-of-the-art portfolio-based parallel *SAT* solvers (e.g. `ppfolio`, `pfolioUZK`, `SATzilla`) not only different strategies, but even different sequential solvers compete and, to a limited extent, cooperate on the same formula. In such approaches there is no load balancing and the communication is limited to the sharing of learned clauses.

`GridSAT` [19, 18] was the first complete and parallel *SAT* solver employing a grid. It belongs to the divide-and-conquer group. It is based on the sequential *SAT* solver `zChaff`. Besides achieving significant speedup in the case of some (satisfiable and even unsatisfiable) instances, `GridSAT` is able to solve some problems for which sequential `zChaff` exceed time out. `GridSAT` distributes only the short learned clauses over the nodes, therefore it minimizes the communication overhead. Search space splitting is based on the selection of a so-called pivot variable x on the second decision level, and then creating two subproblems by adding a new decision on x respectively $\neg x$ to the first decision level. If sufficient resources are available, the subproblems can be further partitioned recursively. Each new subproblem is defined by a clause set, including learned clauses, and a decision stack.

[37] proposes a more sophisticated approach, based on using "partition functions", in order to split a problem into a fixed number of subproblems. Two partition functions were compared, a scattering-based and a *DPLL*-based one with lookahead. A partition function can be applied even in a recursive way, by repartitioning difficult subproblems *(e.g. the ones that exceeds time out)*. For some of the experiments, an open source grid infrastructure called `Nordugrid` was used.

`SAT@home` [72] is a large volunteer SAT-solving project on grid, which involves more than 2000 clients. The project is based on the Berkeley Open Infrastructure for Network Computing `BOINC` [4], which is an open source middleware system for volunteer grid computing. On top of `BOINC`, the project was implemented by using the `SZTAKI Desktop Grid` [47], which provides the Distributed Computing Application Programming Interface (`DC-API`), in order to simplify the development, and then also to deploy and distribute applications to multiple grid environments. [72] proposes a rather simple partitioning approach: given a set of n selected variables, called a decomposition, a set of 2^n subproblems is generated. The key issue is how to select a decomposition. One way to solve this issue is to derive the set of "important" decomposition variables from the original problem formulation, which, however, then is problem-specific, and needs human guidance. For instance, in the context of *SAT*-based cryptoanalysis of keystream generators, a decomposition set can be obtained from the encoding of the initial state of the linear feedback shift registers [72]. `SAT@home` uses no data exchange among clients.

***DIMACS iCNF* format** There is an extension of DIMACS CNF called DIMACS *iCNF* [2] format, which extends the standard with so called assignment lines. In this case we write `inccnf` instead of *cnf* in the problem line. In this case the problem line contains no more

[2] `http://www.siert.nl/icnf/`

details.

An assignment line starts with the letter a fallowed by an assignment which has the same syntax as a clause. An example for an *iCNF* file:

```
p inccnf
-1 2 0
-2 3 0
a -1 0
...
```

The *Cube and Conquer* approach The assignment lines together are called the cube set, and allows us to use the Cube and Conquer approach. The Cube and Conquer approach was introduced in [34]. This technique allows to target very large SAT problems. It is a two-phase approach. In the first phase a Lookahead-based SAT solver partitions a problem into many thousands subproblems with a cube. In the second phase a CDCL SAT solver solves those problems guided by the cube.

We have also a contribution in this field, called CCGrid [89], which uses BOINC and the SZTAKI Desktop Grid. For partitioning the input problem, we use `march_cc` [34]. Our approach differs from the previous ones in the fact that it uses a parallel SAT solver `iLingeling` [3] in the second phase.

Recent results of this field are reported in [35, 36]. They were able to solve the largest known SAT problem which is 200 terabytes in size using the Cube and Conquer approach.

2.4 *Signed Logic* and *Multi–Domain Logic*

This section overviews some multivalued logics and their connections to the SAT problem. This section is the extended version of the Introduction from our paper [101].

Signed logic [8, 64] is a special type of multivalued logic in which the set of satisfying values for the variables may differ in different clauses. Namely, a *signed formula* is a conjunction of *signed clauses*, and a signed clause is a disjunction of *signed literals* of the form $S : p$, where S is a set (the *sign*) and p is a variable. (S is called the *support* of the variable in the respective clause.) The union of all signs constitutes the *domain* N of the formula. An interpretation I is a mapping from the set of variables P to the set of truth values N, and it satisfies a literal $S : p$ if $I(p) \in S$. We may assume that each variable occurs at most once in each clause (otherwise we merge the corresponding literals by union of their supports). When $S = \emptyset$ the literal may be omitted from the clause, and when $S = N$ then the whole clause is redundant.

The classical methods from boolean logic generalize in a natural way to signed logic, see [8], which also describes the generalization of the DPLL algorithm [23], including a specific aspect of it for signed logic, namely the elimination from of branches corresponding to

[3] `http://fmv.jku.at/cnc/`

certain redundant truth values. We will describe this strategy in the sequel under the name of *elimination of weak assignments.*

However, we found only few implementations of a direct method for solving signed logic problems, e.g., [64], which is targeted at a restricted class of formulae. Rather, most approaches are based on translating signed logic into boolean logic and using some version of a SAT algorithm. For instance, [1] uses the information from the original signed logic problem in order to guide the SAT search.

We also contributed to this field. We have introduced the *Multi–Domain Logic (MDL)*, the generalization of signed logic in which the domains of the variables may differ [102, 101, 100]. Although from the theoretical point of view the expressivity of MDL does not differ from signed logic, in practice this distinction leads to more efficient solving methods. This is because the domains can be reduced during the solving process, and finding a contradiction is expressed as the reduction of the domain of a variable to the empty set. Initially the domain of a variable which does not occur in unit clauses is the union of all its supports. Otherwise, the domain is the intersection of all the supports from the unit clauses containing the respective variable.

Unit resolution is done by intersecting the support of a non-unit clause with the respective domain - when this is empty then the literal disappears. If a support includes the corresponding domain, then the whole clause can be deleted (unit subsumption). Whenever a new unit is obtained, this will reduce the respective domain. When no new unit can be obtained, then one must branch on one of the variables, by splitting its domain. We have experimented some splitting strategies in [101].

While the operations above are straightforward generalizations of boolean constraint propagation, MDL also benefits from specific strategies. Similarly to signed-logic, one may detect certain redundant elements in domains, which we call *weak assignments*: If an element a of a domain occurs in all supports together with another element b, then we say a *is weaker then* b and a can be eliminated from the domain.

Novel in our approach is the use of a technique which we call *variable merging.* This consists in replacing two or more variables by a new one, whose domain is the cartesian product of the old domains. In each clause, the disjunction of the literals containing the old variables is replaced by one literal whose support is constructed in a straightforward way. Variable merging is used at the beginning of the solving process in order to translate a boolean SAT problem into signed logic, but also during the solving process in order to reduce the number of variables, when some domains become relatively small.

We generalized the DPLL method in [101] which exhibits two novel aspects:

- *separation of the domains* of the variable,
- *dynamic merging* of the domains of the variables.

In contrast to the current approaches, we have demonstrated how to solve boolean SAT problems by transforming them into signed logic problems and then applying our direct

solver. This may constitute an efficient alternative to the current SAT solvers based on unit propagation, because, in the context of signed logic, more boolean constraints can be propagated simultaneously. For the representation of the signs we use strings of bits in the current implementation, but this is not essential for the main algorithm.

2.5 Strong Connectivity

Since we uses the notion of strongly connected directed graph, we overview its history. This section is the extended version of the Introduction from our paper [96].

In the 1950s and 1960s, mathematicians and computer scientists began to study the complexity of algorithms. Thus, in the case of graph-based algorithms, time and space complexity analysis became an important factor. It was Roy, who, in 1959, first considered the famous problem of computing the transitive closure *(finding all reachable vertices from each vertex)* of a directed graph [74]. In the 60s, a variety of sequential algorithms [82, 73] were proposed to solve this problem. Warshall solved the problem of computing the transitive closure of a binary relation represented as an adjacency matrix of the directed graph and considered as a boolean matrix. The time complexity of the Warshall algorithm was $O(|\mathcal{V}|^3)$, then Purdom gave an $O(|\mathcal{V}|^2)$ algorithm. Munro [70] optimized Purdom's work in 1971 by using a more efficient data structure for merging the vertices. Instead of using an adjacency matrix for representing the edges, Munro proposed the usage of adjacency lists.

As we know, the best algorithm [78] that solve this problem has $O(|\mathcal{V}| + |\mathcal{E}|)$ time complexity. The first "depth-first search" based algorithm for finding the strongly connected components of a directed graph is presented in Tarjan's paper [80]. In the same year, R. Karp [48] introduced the intractability of the traveling-salesperson problem. Sharir [77] presented an algorithm, called Kosaraju's algorithm, for constructing all strongly-connected components of a directed graph. Like Tarjan, he used a linear time depth-first spanning tree algorithm. But this algorithm differed from Tarjan's in that it produced these components in reverse post-order of their roots, and also ordered the nodes within each component in reverse post-order. In both Tarjan's and Kosaraju's algorithms, there are fundamental techniques of efficient algorithm design for graphs.

In 1982 Dijkstra proposed [25] a different variation of Tarjan's algorithm in order to find the maximum strong components in a directed graph. Instead of keeping track of low degree vertices, he maintains a stack of possible root candidates. On finding a last edge, the algorithm pops vertices from the stack until the 'root' of the cycle is found. At backtracking, the current flag of reachable states is set to false so that these states do not interfere with a future search. This algorithm also runs in linear time. Nearly forty years later H. Gabow [38] found a third linear-time algorithm for this problem in 2000. Contrary to earlier theories he presented a one-pass algorithm that only maintain a representation of the depth-first search path. This gives a simplified view of depth-first search without

sacrificing efficiency.

We proved that the Black-and-White 2-SAT representation of a strongly connected directed graph is solvable in linear time [88], and we contracted also a special SAT solver for this type of problem, called BaW 1.0 [96].

3 The Black-and-White SAT Problem

In this part we introduce the Black-and-White SAT problem [88, 87, 95, 94, 93]. We introduce 4 models, the strong one, the weak one, the Balatonboglár one, and the simplified Balatonboglár one, which can model a directed graph as a SAT problem. We show that if the directed graph is strongly connected, then the corresponding SAT representation is a Black-and-White SAT problem.

This part of the dissertation is based on our following papers: [87, 93, 88]

3.1 Introduction

In logic the most natural representation of an edge of a directed graph, say $a \to b$, is to use implication, i.e., $a \implies b$. This means that the edge $a \to b$ can be represented by the binary clause: $(\neg a \vee b)$. If a graph contains two edges: $a \to b$, and $a \to c$, then those can be represented by the formula: $(a \implies b) \wedge (a \implies c)$, which is equivalent to two 2-clauses $(\neg a \vee b) \wedge (\neg a \vee c)$.

We used that representation in the paper [88] and we were able to prove that the SAT representation of a strongly connected directed graph is a Black-and-White 2-SAT problem. It is not a surprise that the SAT representation is a 2-SAT problem, since each edge is represented by a binary clause.

The other property, Black-and-White, means that the SAT problem is "nearly" unsatisfiable, because it has only two solutions, the white assignment, and the black one. The white assignment assigns true to each variable, the black one assigns false to each variable.

This representation helps us to translate a graph into a 2-SAT problem. We can also translate a Black-and-White 2-SAT problem into a directed graph. We can also translate any other 2-SAT problem into a directed graph, if it does not subsume neither the black nor the white clause, i.e., it does not contain a clause which contains only positive or negative literals, like $(a \vee b)$ or $(\neg a \vee \neg b)$

Actually, there is no point to translate a 2-SAT problem into a directed graph, because the 2-SAT problem is solvable in linear time [2].

Preferably, we should translate 3-SAT problems into directed graphs, because then we could use graph tools to study it.

Our second paper aims to link the field of directed graph and the field of 3-SAT problem [87]. In the second paper the idea is the following: If a graph contains two edges: $a \to b$, and $a \to c$, then those can be represented by the formula: $(a \implies b) \vee (a \implies c)$, which

is equivalent to $a \implies (b \vee c)$, which is equivalent to a 3-clause $(\neg a \vee b \vee c)$.

This idea is not enough to be able to generate a Black-and-White SAT formula from a strongly connected graph. We need to represent cycles of the graph, too. If $a_1 \to a_2 \to \cdots \to a_n \to a_1$ is a cycle with exit points $b_1, b_2, \ldots, b_m$, then this cycle can be represented by the clause: $(\neg a_1 \vee \neg a_2 \vee \cdots \vee \neg a_n \vee b_1 \vee b_2 \vee \cdots \vee b_m)$.

The Balatonboglár model uses the trick that instead of detecting each cycle, it generates from each path $a \to b \to c$ the following 3-clause: $(\neg a \vee \neg b \vee c)$ even if there is no cycle which contains the vertices a and b. This simplification allows very fast 3-SAT problem generation from a directed graph, and the SAT instance will be a Black-and-White 3-SAT if and only if (iff) the input directed graph is strongly connected. On the other hand this simplification does not tell us how to generate from a 3-SAT problem a directed graph.

In case of Balatonboglár model we show that if the directed graph is strongly connected, then the corresponding SAT representation is a Black-and-White 3-SAT problem.

We can construct a more compact model with fewer clauses which has the same property, i.e., if the represented direct graph is strongly connected, then the corresponding SAT representation is a Black-and-White 3-SAT problem. This more compact model is called the simplified Balatonboglár model.

3.2 Related works

These days one of the most promising branch of mathematics is the idea that we try to unify different mathematical theories, like in case of Langlands program [56], which relates algebraic number theory to automorphic forms and representation theory. Another nice example is the modularity theorem [83] (formerly called the Taniyama–Shimura–Weil conjecture), which states that elliptic curves over the field of rational numbers are related to modular forms. Without the modularity theorem Andrew Wiles could not prove Fermat's Last Theorem [84].

In this part, we show a link between directed graphs and propositional logic formulae. We prove a theorem which allows to use an algorithm from the field of propositional logic to check a graph property. Namely, we transform a directed graph into a SAT problem to check whether the graph is strongly connected or not.

The most prominent graph representations are:

- Implication graph [2] is a is a skew-symmetric directed graph, where vertices are literals (boolean variables, and their negation), edges represents implication. Note that the binary clause $x \vee y$ is represented by two implications in the implication graph: $\neg x \supset y$, and $\neg y \supset x$, and so the implication graph is skew-symmetric, i.e., it is isomorphic to its own transpose graph.

- AIG, And-Inverter Graph [44] is directed acyclic graph where vertices are logical conjunction with to input edges, a marked edge means logical negation, the boolean

variables are the input, the formula itself is the output.

- BDD, Reduced Ordered Binary Decision Diagram [15], which is a rooted, directed, acyclic graph, which consists of vertices, which are boolean variables, and terminal vertices, called 0-terminal, which terminates paths, where the formula evaluates to false; and 1-terminal, which terminates paths, where the formula evaluates to true. Each non-terminal vertex has two child vertices called low child, corresponding edge is called 0-edge; and high child, corresponding edge is called 1-edge; which are possible values of the parent vertex. One has to merge any isomorphic subgraphs and eliminate any vertex whose two children are isomorphic.

- ZDD (called also ZBDD in the literature), Zero-Suppressed Binary Decision Diagram [67], is a kind of binary decision diagram, where instead of the rule "eliminate any vertex whose two children are isomorphic" we use the rule "eliminate those vertices whose 1-edge points directly to 0-terminal". If a SAT problem has only a few solutions then ZDD is a better representation than BDD.

- Graph representation of logical games is a well-knows connection between graphs and logical formulae. Some examples are [32, 71].

As we can see a great effort has been done in the direction from formulae to graphs. In this part of the dissertation, we study the other way, the direction from graphs to formulae.

In our first paper [88] we showed how to convert a directed graph into a 2-SAT problem. In our second paper [87] we showed how to convert a directed graph into a 3-SAT problem. In our third paper [93] we showed how to convert a directed graph into a more compact 3-SAT problem. In this part of the dissertation we summarize the result of these papers. Note that the second paper is under review. We have submitted it to a high quality journal, called Theory of Computing, see `https://theoryofcomputing.org/`.

3.3 Definitions

A *literal* is a boolean variable, called positive literal, or the negation of a boolean variable, called negative literal. Examples for literals are: $a, \neg a, b, \neg b, \ldots$.

A *clause* is a set of literals. A *clause set* is a set of clauses. A *SAT problem* is a clause set. An *assignment* is a set of literals. In a clause or in an assignment, a variable may occur either as a positive literal or as a negative literal, but not as both, or it may not occur at all.

Clauses are interpreted as disjunction of their literals. Assignments are interpreted as conjunction of their literals. Clause sets are interpreted as conjunction of their clauses.

If a clause or an assignment contains exactly k literals, then we say it is a *k-clause* or a *k-assignment*, respectively. A 1-clause is called to be a *unit*, a 2-clause is called to be a *binary clause*. A *k-SAT problem* is a clause set where its clauses have at most k literals.

A clause from a clause set is a *full-length clause* iff it contains all variables from the clause set.

We use two intuitive notions: NNP clause, and NPP clause. A clause is an NNP clause iff it contains exactly one positive literal. A clause is an NPP clause iff it contains exactly one negative literal.

If a is a literal in clause set S, and $\neg a$ is not a literal in S, then we say that a is a pure literal in S.

Negation of a set H is denoted by $\neg H$ which means that all elements in H are negated. Note that $\neg\neg H = H$.

Let V be the set of variables of a clause set. We say that WW is the *white clause* or the *white assignment* iff $WW = V$. We say that BB is the *black clause* or the *black assignment* iff $BB = \neg V$. For example if $V = \{a, b, c\}$, then $WW = \{a, b, c\}$, and $BB = \{\neg a, \neg b, \neg c\}$.

We say that clause C *subsumes* clause D iff C is a subset of D.

We say that clause set S *subsumes* clause C iff there is a clause in S which subsumes C. Formally: S *subsumes* $C \iff \exists D(D \in S \wedge D \subseteq C)$.

We say that assignment M is a *solution* for clause set S iff for all $C \in S$ we have $M \cap C \neq \{\ \}$.

We say that the clause set S is a *Black-and-White SAT problem* iff it has only two solutions, the white assignment (WW) and the black one (BB).

We say that clause sets A and B are equivalent iff they have the same set of solutions. We say that clause set A entails clause set B iff the set of solutions of A is a subset of the set of solutions of B, i.e., A may have no other solutions than B. This notion is denoted by $A \geq B$. Note that if A subsumes all clauses of B, then $A \geq B$.

We say that A is stronger than B iff $A \geq B$ and A and B are not equivalent. This notion is denoted by $A > B$.

The construction $\mathcal{D} = (\mathcal{V},\mathcal{E})$ is a directed graph, where $\mathcal{V}$ is the set of vertices, and $\mathcal{E}$ is the set of edges. An edge is an ordered pair of vertices. The edge (a, b) is depicted by $a \to b$, and we can say that a has a child b. If (a, b) is an element of $\mathcal{E}$, then we may say that (a, b) is an edge of $\mathcal{D}$.

We say that $\mathcal{D}$ is a communication graph iff for all a in $\mathcal{V}$ have that (a, a) is not in $\mathcal{E}$, and if x is an element of $\mathcal{V}$ then $\neg x$ must not be an element of $\mathcal{V}$. We need this constraint because we generate a logical formula out of $\mathcal{D}$. If we speak about a communication graph then we may use the word node as a synonym of vertex.

A path from a_1 to a_j in directed graph $\mathcal{D}$ is a sequence of vertices $a_1, a_2, \ldots, a_j$ such that for each $i \in \{1, \ldots, j-1\}$ we have that (a_i, a_{i+1}) is an edge of $\mathcal{D}$. A path from a_1 to a_j in directed graph $\mathcal{D}$ is a *cycle* iff (a_j, a_1) is an edge of $\mathcal{D}$. The cycle $a_1, a_2, \ldots, a_j, a_1$ is represented by the following tuple: $(a_1, a_2, \ldots, a_j)$. This tuple can be used as a set of its elements. Note that in the representation of a cycle the first and the last element must not be the same vertex. Figure 3.1 shows as example for a path and a cycle.

If we have a cycle $(a_1, a_2, \ldots, a_n)$ then b is an exit point of it iff for some $j \in \{1,2, \ldots, m\}$

we have that (a_j, b) is an edge and $b \notin \{a_1, a_2, \ldots, a_n\}$.

A directed graph is complete iff every pair of distinct vertices is connected by a pair of unique edges (one in each direction). A directed graph is strongly connected iff there is a path from each vertex to each other vertex. Note that a complete graph is also strongly connected. Note that a strongly connected graph contains a cycle which contains all vertices.

Figure 3.1: A path and a cycle.

3.4 The Strong Model of Communication Graphs

In our first paper [88] we used to call this model as "2-SAT representation". In our second paper [87] we used to call this model as "the strong model".

Let us assume that the graph $\mathcal{D}$ is $(\{a,b,c\},\{(a,b),(a,c)\})$, i.e., $\mathcal{D}$ is a graph with 3 vertices: a, b, c, and with two edges: $a \to b$, and $a \to c$. Since the most natural representation of an edge is implication in case of propositional logic, a possible model of $\mathcal{D}$ is $(a \implies b) \wedge (a \implies c)$. Actually, this the strong model of $\mathcal{D}$.

The interpretation of this formula is the following in the field of Wireless Sensors Networks: If node a is able to send a message to nodes b and c, then node a sends the message to both nodes b and c. But this might result in an error, because both nodes have to send an acknowledgement that the message is arrived, but using the same frequency at the same time results in interference. A more natural way of communication is that we use some protocol, which decides where to send the message, either to node b or to node c, see the next section.

The strong model was defined formally in our first paper [88]. We recall its definition.

Let $\mathcal{D} = (\mathcal{V},\mathcal{E})$ be a communication graph, then the strong model of $\mathcal{D}$ is denoted by $\mathcal{SM}$, and defined as follows:

$$\mathcal{SM} := \{\{\neg a,\ b\} \mid (a,b) \in \mathcal{E}\}.$$

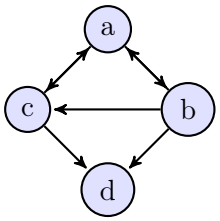

Figure 3.2: A communication graph with 4 vertices, 3 cycles.

As an example we show the strong model of the communication graph of Figure 3.2:

$$\begin{aligned}\mathcal{SM} = \{&\{\neg a, b\}, \{\neg a, c\}, \{\neg b, a\}, \{\neg b, c\},\\ &\{\neg b, d\}, \{\neg c, a\}, \{\neg c, d\}\}.\end{aligned} \tag{3.4.1}$$

Note that since the communication graph in Figure 3.2 is not strongly connected, its strong model (3.4.1) is not a Black-and-White SAT problem. For example $\{\neg a, \neg b, \neg c, d\}$ is a solution of it.

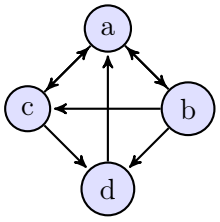

Figure 3.3: A strongly connected communication graph with 4 vertices, 6 cycles.

As the second example we show the strong model of the communication graph in Figure 3.3. This graph is almost the same as the previous one (Figure 3.2), but here we have an edge from d to a, which makes this graph strongly connected. Its strong model is:

$$\begin{aligned}\mathcal{SM} = \{&\{\neg a, b\}, \{\neg a, c\}, \{\neg b, a\}, \{\neg b, c\},\\ &\{\neg b, d\}, \{\neg c, a\}, \{\neg c, d\}, \{\neg d, a\}\}.\end{aligned} \tag{3.4.2}$$

Note that since the communication graph in Figure 3.3 is strongly connected, its strong model is a Black-and-White SAT problem, i.e., the SAT problem in (3.4.2) has only these two solutions: $\{a, b, c, d\}$, and $\{\neg a, \neg b, \neg c, \neg d\}$.

It is not easy to check that this SAT problem is indeed a Black-and-White SAT problem. Therefore, first we show the set of all full-length clauses on the variables $\{a, b, c, d\}$ with an index number (each entry has the form: index number: full-length clause):

$$\begin{aligned}\{&0 : \{\neg a, \neg b, \neg c, \neg d\},\ 1 : \{\neg a, \neg b, \neg c, d\},\ 2 : \{\neg a, \neg b, c, \neg d\},\ 3 : \{\neg a, \neg b, c, d\},\\ &4 : \{\neg a, b, \neg c, \neg d\},\ 5 : \{\neg a, b, \neg c, d\},\ 6 : \{\neg a, b, c, \neg d\},\ 7 : \{\neg a, b, c, d\},\\ &8 : \{a, \neg b, \neg c, \neg d\},\ 9 : \{a, \neg b, \neg c, d\},\ 10 : \{a, \neg b, c, \neg d\},\ 11 : \{a, \neg b, c, d\},\\ &12 : \{a, b, \neg c, \neg d\},\ 13 : \{a, b, \neg c, d\},\ 14 : \{a, b, c, \neg d\},\ 15 : \{a, b, c, d\}\},\end{aligned} \tag{3.4.3}$$

As second step, we show which clause from (3.4.2) subsumes which full-length clauses (each entry has the form: clause: indices of subsumed full-length clauses):

$$\begin{aligned}\mathcal{SM} = \{&\{\neg a, b\} : 4,5,6,7,\ \{\neg a, c\} : 2,3,6,7,\ \{\neg b, a\} : 9,10,11,12,\\ &\{\neg b, c\} : 2,3,10,11,\ \{\neg b, d\} : 1,3,9,11,\ \{\neg c, a\} : 8,9,13,14,\\ &\{\neg c, d\} : 1,5,9,13,\ \{\neg d, a\} : 8,10,12,14\}.\end{aligned} \tag{3.4.4}$$

From this one can check that (3.4.2) subsumes all full-length clauses, except the black and the white clause, hence, it has only two solutions, the white and the black assignment.

3.5 The Weak Model of Communication Graphs

In this section we define the weak model of communication graphs.

Let us assume that $\mathcal{D} = (\{a,b,c\},\{(a,b),(a,c)\})$, as in the previous section. The weak model of $\mathcal{D}$ is the formula: $(a \implies b) \vee (a \implies c)$, which means that if a node can send a message to more than one nodes, then it can send the message only to one of them. Note that $(a \implies b) \vee (a \implies c)$ is equivalent to $(\neg a \vee b \vee c)$.

The only problem with this representation is that the message can be trapped if a graph contains cycles. Let us assume that we have a cycle with two nodes, n_1 and n_2. Then n_1 may send the message to n_2, and n_2 to n_1, and so on, which means that other nodes could never get the message. This is not good, since our goal is to send a message to each node, if it is possible.

Let us add a new edge to the graph $(\{a,b,c\},\{(a,b),(a,c)\})$. The new edge should go from b to a, so the new graph is this: $(\{a,b,c\},\{(a,b),(a,c),(b,a)\})$. Now we have a cycle with the nodes a and b. Now we have to add a clause to our model which ensures that if b sends a message to a, and a can send a message to b or c, then a should not send back the message to b, but it has to send it to c, which can be formalized like this: $((b \implies a) \wedge (a \implies (b \vee c)) \implies (a \implies c)$. Note that this is equivalent to the clause: $(\neg a \vee \neg b \vee c)$.

In a more general way, node a which has outgoing edges to nodes $b_1, b_2, \ldots, b_k$ is represented by the clause: $(\neg a \vee b_1 \vee b_2 \vee \cdots \vee b_k)$; and the cycle $a_1, a_2 \ldots a_n, a_1$ with exit points $b_1, b_2, \ldots, b_m$ is represented by the clause: $(\neg a_1 \vee \neg a_2 \vee \cdots \vee \neg a_n \vee b_1 \vee b_2 \vee \cdots \vee b_m)$. We call this model as the weak model of communication graphs.

We define it also formally: Let $\mathcal{D}$ be a communication graph. Let $\mathcal{V}$ be the set of vertices of $\mathcal{D}$ and $\mathcal{E}$ the set of edges of $\mathcal{D}$. Since $\mathcal{D}$ is a communication graph, we know that elements of $\mathcal{V}$ can be used as positive literals. Then we define the following notions:

$$OutE(a) := \{b \mid (a,b) \in \mathcal{E}\}.$$

$$NodeRep(a) := \{\neg a\} \cup OutE(a).$$

$$NodeRep := \{NodeRep(a) \mid a \in \mathcal{V} \wedge OutE(a) \neq \emptyset\}.$$

Figure 3.4: Two simple communication graphs.

$$Cycles := \{(a_1, a_2, \ldots, a_k) \mid k = 1 \vee \forall_{i=1\ldots k}(a_{(i \bmod k)+1} \in OutE(a_i)\},$$

and for any two elements of $Cycles$ they cannot be equal as a set.

$$ExitPoints(\ (a_1, a_2, \ldots a_k)\ ) := \{b \mid \exists_{i=1\ldots k}(b \in OutE(a_i)) \wedge \neg\exists_{j=1\ldots k}(b = a_j)\}.$$

$$CycleRep := \{\neg C \cup ExitPoints(C) \mid C \in Cycles \wedge ExitPoints(C) \neq \emptyset\}.$$

$$\mathcal{WM} := NodeRep \cup CycleRep.$$

$\mathcal{WM}$ is the weak model of $\mathcal{D}$.

Note that $NodeRep$ is a subset of $CycleRep$, because we take each node itself as a cycle, see the constraint $k = 1$ in the definition of $Cycles$ which is connected by an disjunction to the rest of the constraint.. So alternatively we can define $\mathcal{WM}$ as follows: $\mathcal{WM} := CycleRep$. We do not use this observation in this work, although, some proofs would be shorter.

Please note that each clause in $\mathcal{WM}$ contains at least one positive and one negative literal, because a node is only represented if it has an outgoing edge, and a cycle is only represented if it has an exit point, therefore, $\mathcal{WM}$ is satisfiable for any communication graph $\mathcal{D}$.

Figure 3.4 shows two simple communication graphs. Their $NodeRep$ values are: $\{\{\neg a, b, c\}\}$, and $\{\{\neg a, b, c, d\}\}$, respectively.

Figure 3.2 shows a communication graph with 3 cycles:

$$(a, b), (a, b, c), (a, c).$$

So its weak model is (after each clause we list the indices of subsumed full-length clauses from (3.4.3)):

$$\begin{aligned}\mathcal{WM} = \{&\{\neg a, b, c\} : 6{,}7,\ \{\neg b, a, c, d\} : 11,\ \{\neg c, a, d\} : 9{,}14,\\ &\{\neg a, \neg b, c, d\} : 3,\ \{\neg a, \neg b, \neg c, d\} : 1,\ \{\neg a, \neg c, b, d\} : 5\}.\end{aligned} \tag{3.5.1}$$

Note that since d has no child node, it does not occur as a negative literal in the model. Note that since the communication graph in Figure 3.2 is not strongly connected, its weak model (3.5.1) is not a Black-and-White SAT problem. For example $\{\neg a, \neg b, \neg c, d\}$ is a solution of the SAT problem in (3.5.1),

As the second example we show the weak model of the communication graph in Figure 3.3. In that graph we have 6 cycles:

$$(a,b),(a,b,c),(a,c),(a,b,d),(a,c,d),(a,b,c,d).$$

The last cycle contains all vertices, so it has no exit point, therefore no clause generated for it. Note that in that graph we have an edge from d to a. The corresponding clauses are the last 3 clauses in this example, the last 2 clauses corresponds to the new cycles. The rest of the clauses are the same as in the previous model. So the weak model is (after each clause we list the indices of subsumed full-length clauses from (3.4.3)):

$$\begin{aligned}\mathcal{WM} = \{&\{\neg a,b,c\}:6,7,\ \{\neg b,a,c,d\}:11,\ \{\neg c,a,d\}:9,14,\\ &\{\neg a,\neg b,c,d\}:3,\ \{\neg a,\neg b,\neg c,d\}:1,\ \{\neg a,\neg c,b,d\}:5,\\ &\{\neg d,a\}:8,10,12,14,\ \{\neg a,\neg b,\neg d,c\}:2,\ \{\neg a,\neg c,\neg d,b\}:4\}.\end{aligned} \tag{3.5.2}$$

Note that since the communication graph in Figure 3.3 is strongly connected, its weak model is a Black-and-White SAT problem, i.e., the SAT problem in (3.5.2) has only these two solutions: $\{a,b,c,d\}$, and $\{\neg a,\neg b,\neg c,\neg d\}$.

Each cycle was a simple cycle in the above 2 examples. A cycle is simple iff only the first and last vertices are repeated, so we might think that it is enough to consider only simple cycles. The next example shows a case when we need to consider also a non-simple cycle.

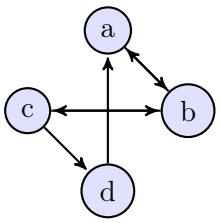

Figure 3.5: A strongly connected communication graph with 4 vertices, 3 simple cycles, 1 non-simple cycle.

On Figure 3.5 we can see a strongly connected communication graph which contains 3 simple cycles: $(a,b),(a,b,c,d),(b,c)$, and 1 non-simple cycle: (a,b,c,b). As the third example we give the weak model of this communication graph (after each clause we list the indices of subsumed full-length clauses from (3.4.3)):

$$\begin{aligned}\mathcal{WM} = \{&\{\neg a,b\}:4,5,6,7,\ \{\neg b,a,c\}:10,11,\ \{\neg c,b,d\}:5,13,\\ &\{\neg d,a\}:8,10,12,14,\ \{\neg a,\neg b,c\}:2,3,\ \{\neg b,\neg c,a,d\}:9,\ \{\neg a,\neg b,\neg c,d\}:1\}.\end{aligned} \tag{3.5.3}$$

Note that the last clause is generated from the non-simple cycle, and that full-length clause is not subsumed by any other clause.

3.6 The Balatonboglár Model of Communication Graphs

Since the weak model does not result in a 3-SAT problem we introduce also the Balatonboglár model, which is a simplified version of the weak model, which uses the trick that

instead of detecting each cycle in the graph, it generates from each path a, b, c the following 3-clause: $(\neg a \vee \neg b \vee c)$ even if there is no cycle which contains the vertices a and b. This simplification allows very fast 3-SAT model generation from a directed graph, and the model will be Black-and-White 3-SAT iff the input directed graph is strongly connected.

We define the Balatonboglár model of communication graphs as follows. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{V}$ be the set of vertices of $\mathcal{D}$ and $\mathcal{E}$ the set of edges of $\mathcal{D}$. Since $\mathcal{D}$ is a communication graph, we know that elements of $\mathcal{V}$ can be used as positive literals. Then we define the following notions:

$$NodeRep2 := \{\{\neg a, b\} \mid a \in \mathcal{V} \wedge \{b\} = OutE(a)\}.$$

$$NodeRep3(a) := \{\neg a, b, c\}, where\ a \in \mathcal{V} \wedge b \in OutE(a) \wedge c \in OutE(a) \wedge b \neq c.$$

$$NodeRep3 := \{\ NodeRep3(a) |\ a \in \mathcal{V}\}.$$

$$TagForward := \{\{\neg a, \neg b, c\} \mid a \in \mathcal{V} \wedge b \in OutE(a) \wedge c \in OutE(b) \wedge a \neq b \neq c\}.$$

$$\mathcal{BM} := NodeRep2 \cup NodeRep3 \cup TagForward.$$

$\mathcal{BM}$ is a Balatonboglár model of $\mathcal{D}$.

Note that we assume that $NodeRep3(a)$ is a deterministic function, but it is defined in a nondeterministic way, i.e., it is a 3 length subset of $NodeRep(a)$, but its implementation must be deterministic, which means that it should give back always the same subset.

The name $TagForward$, comes from the child game Tag: there is a child who has to catch someone else by touching he or she. This action is called tag. There is a rule, the tagged one cannot tag back, he or she must tag someone else, so must tag forward.

As the first example, we give the Balatonboglár model of the graph in Figure 3.2 (after each clause we list the indices of subsumed full-length clauses from (3.4.3)):

$$\begin{aligned}\mathcal{BM} = \{&\{\neg a, b, c\} : 6{,}7,\ \{\neg b, c, d\} : 3{,}11,\ \{\neg c, a, d\} : 9{,}13,\\ &\{\neg a, \neg b, c\} : 2{,}3,\ \{\neg a, \neg b, d\} : 1{,}3,\ \{\neg a, \neg c, b\} : 4{,}5,\\ &\{\neg a, \neg c, d\} : 1{,}5,\ \{\neg b, \neg c, a\} : 8{,}9,\ \{\neg b, \neg c, d\} : 1{,}9\}.\end{aligned} \tag{3.6.1}$$

Note that b has 3 child nodes, so instead of $\{\neg b, c, d\}$, we could use either $\{\neg b, a, c\}$ or $\{\neg b, a, d\}$. Note also that Figure 3.2 is not strongly connected, so its Balatonboglár model is not a Black-and-White SAT problem. For example $\{\neg a, \neg b, \neg c, d\}$ is a solution of the SAT problem in (3.6.1).

As the second example, we show the Balatonboglár model of the communication graph in Figure 3.3. Note that in that graph we have an edge from d to a. The corresponding clauses are the last 3 clauses in this Balatonboglár model (after each clause we list the indices of subsumed full-length clauses from (3.4.3)):

$$\begin{aligned}\mathcal{BM} = \{&\{\neg a, b, c\} : 6{,}7,\ \{\neg b, c, d\} : 3{,}11,\ \{\neg c, a, d\} : 9{,}13,\\ &\{\neg d, a\} : 8{,}10{,}12{,}14,\ \{\neg a, \neg b, c\} : 2{,}3,\ \{\neg a, \neg b, d\} : 1{,}3,\ \{\neg a, \neg c, d\} : 1{,}5,\\ &\{\neg b, \neg c, a\} : 8{,}9,\ \{\neg b, \neg c, d\} : 1{,}9,\ \{\neg b, \neg d, a\} : 8{,}10,\\ &\{\neg c, \neg a, b\} : 4{,}5,\ \{\neg c, \neg d, a\} : 8{,}12,\ \{\neg d, \neg a, b\} : 4{,}6,\ \{\neg d, \neg a, c\} : 3{,}7\}.\end{aligned} \tag{3.6.2}$$

Note that since the communication graph in Figure 3.3 is strongly connected, its Balatonboglár model is a Black-and-White SAT problem, i.e., the SAT problem in (3.6.2) has only these two solutions: $\{a, b, c, d\}$, and $\{\neg a, \neg b, \neg c, \neg d\}$.

As the third example, we show the Balatonboglár model of the communication graph in Figure 3.5 (after each clause we list the indices of subsumed full-length clauses from (3.4.3)):

$$\begin{aligned}\mathcal{BM} = \{&\{\neg a, b\} : 4,5,6,7,\ \{\neg b, a, c\} : 10,11,\ \{\neg c, b, d\} : 5,13,\\ &\{\neg d, a\} : 8,10,12,14,\ \{\neg a, \neg b, c\} : 2,3,\ \{\neg b, \neg c, d\} : 1,9,\\ &\{\neg c, \neg b, a\} : 8,9,\ \{\neg c, \neg d, a\} : 8,12,\ \{\neg d, \neg a, b\} : 4,6\}.\end{aligned} \tag{3.6.3}$$

Since there is no node which has more than two child nodes, there is no other possible Balatonboglár model for this graph.
Note that since the communication graph in Figure 3.5 is strongly connected, its Balatonboglár model (3.6.3) is a Black-and-White SAT problem.

Our long term goal is to find out, how to represent a 3-SAT problem as a directed graph. This model does not tell us how to do that, it gives only a link from the field of directed graphs to the field of 3-SAT problems, which might help us to find out in the future, how to represent a 3-SAT problem as a directed graph. So this remains an open question, which seems to be a very difficult one, because if we were able to translate a 3-SAT problem into a directed graph, then we could translate it to a 2-SAT problem using our strong model, which means that we would have a 3-SAT to 2-SAT converter.

3.7 The Simplified Balatonboglár Model of Communication Graphs

Our previous model, the Balatonboglár model uses the trick that instead of detecting each cycle, it generates from each path $a \to b \to c$ the following 3-clause: $(\neg a \vee \neg b \vee c)$, which is a negative-negative-positive clause, or for short an NNP clause, even if there is no cycle which contains the vertices a and b. This simplification allows very fast 3-SAT problem generation from a directed graph, and the SAT instance will be a Black-and-White 3-SAT iff the input directed graph is strongly connected. On the other hand this trick generates a lot of superfluous clauses.

By other words, the Balatonboglár model generates lots of so called NNP clauses. This kind of clauses contains two negative literals and a positive one. These NNP clauses are generated as part of $TagForward$. These clauses are used instead of the clauses from $CycleRep$ of the weak model. But $TagForward$ contains also clauses which do not subsumes any clause from $CycleRep$.

For example the clause $\{\neg c, \neg d, a\}$ from 3.6.3 does not subsumes any clause from 3.5.3, where 3.6.3 is the Balatonboglár model and 3.5.3 is the weak model of the directed graph depicted on Figure 3.5.

So $TagForward$ is over-constrained. Therefore, we construct a new model, the simplified Balatonboglár model, which generates much less NNP clauses [94, 93].

The main idea is the following: We create the strongly connected components (SCC) of the graph [80, 77]. For each component we generate a "big-cycle", a cycle which contains all the nodes of the component. Then for each such big-cycle we generate NNP clauses along it, see $BigRep$. For example, for the cycle $(n_1, n_2, \dots n_k)$ we generate the clauses

$$\{\{\neg n_1, \neg n_2, n_3\}, \{\neg n_2, \neg n_3, n_4\}, \dots, \{\neg n_k, \neg n_1, n_2\}\}.$$

After giving the main idea, we need also some more ideas. We should also generate NNP clauses which link the components, see $LinkRep$. After representing the big-cycles and the links between them, we can delete the edges which build them up, and also the opposite of those edges. The rest of the edges are "inner-edges" which may form cycles, so called "inner-cycle". We represent them one by one by an NNP clause, such that the positive literal should be a neighbour node on the big-cycle of one of the negative ones. Then we delete that edge which is represented by the negative literals, and the opposite of it. We do this till their is no more inner-cycle, see $RestRep$. Other parts of the model are just the same as in case of the Balatonboglár model, see $NodeRep2$ and $NodeRep3$.

We define the simplified Balatonboglár model of communication graphs as follows. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{V}$ be the set of vertices of $\mathcal{D}$ and $\mathcal{E}$ the set of edges of $\mathcal{D}$. Since $\mathcal{D}$ is a communication graph, we know that elements of $\mathcal{V}$ can be used as positive literals. Then we define the following notions:

Let $BigCycle(\ (V,E)\ )$ be defined as a cycle which contains all vertices from V, or UnDef if there exists no such cycle. Note that it is not UnDef if (V,E) is a strongly connected component. Note that if a, b are consecutive vertices in $BigCycle(\ (V,E)\ )$ then $(a,b) \in E$.

Let $BigCycles$ be defined as the set of $BigCycle(Comp)$, where $Comp$ is a strongly connected component, i.e., $BigCycles := \{BigCycle(Comp) \mid Comp \in SCC(\mathcal{D})\}$.

Let $CycEdges(\ (a_1, a_2, \dots a_k)\ )$ be defined as the set of pairs of consecutive vertices, i.e., $CycEdges(\ (a_1, a_2, \dots a_k)\ ) := \{(a_i, a_j) \mid i \in \{1, \dots, k\} \wedge j = (i\ mod\ k) + 1\}$.

Let $BigRep(Cyc)$ be defined as the set of the following NPP clauses: $\{\neg a, \neg b, c\}$, where a, b, c are conscutive vertices in Cyc, i.e., $BigRep(Cyc) := \{\{\neg a, \neg b, c\} \mid (a, b) \in CycEdges(Cyc) \wedge (b, c) \in CycEdges(Cyc)\}$, where Cyc is a cycle.

Let $BigRep$ be defined as the union of the representation of all strongly connected components which have at least 3 vertices, i.e.,

$$BigRep := \bigcup_{Cyc \in BigCycles \wedge |Cyc| > 2} BigRep(Cyc).$$

Let $LinkRep(Cyc)$ be defined as the following NNP clause: $\{\neg a, \neg b, c\}$, where c is an exit point of Cyc and (b,c) is an edge and a,b are consecutive vertices in Cyc, i.e., $LinkRep(Cyc) := \{\neg a, \neg b, c\}$, where $(a,b) \in CycEdges(Cyc) \wedge c \in ExitPoints(Cyc) \wedge c \in OutE(b)$, where Cyc is a big-cycle of a strongly connected component.

Let $LinkRep$ be defined as the union of the representation of links between strongly connected components which have at least 2 vertices, i.e.,

$$LinkRep := \bigcup_{Cyc \in BigCycles \wedge |Cyc|>1} LinkRep(Cyc).$$

Let $ReverseEdges(Cyc)$ be defined as the set of reverse edges of Cyc, i.e.,
$ReverseEdges(Cyc) := \{(b,a) \mid (a,b) \in CycEdges(Cyc)\}$, where Cyc is a cycle.

Let $ToDel$ be defined as the union of edges of all strongly connected components and their reverse edges, i.e.,
$ToDel := \bigcup_{Cyc \in BigCycles \wedge |Cyc|>1} (CycEdges(Cyc) \cup ReverseEdges(Cyc))$.

Let $Rest_0$ be defined as the set of edges of the represented communication graph subtraction $ToDel$, i.e., $Rest_0 := \mathcal{E}cToDel$.

Let ACs_i be the set of cycles in $Rest_i$, i.e.,
$ACs_i := \{(a_1, a_2, \ldots, a_k) \mid \forall_{j=1\ldots k}(a_j, a_{(j \bmod k)+1}) \in Rest_i\}$.

Let AC_i be a cycle from ACs_i with at least 2 vertices, if there is no such cycle, then it is UnDef.

Let $RestRep_0$ be defined as the empty set, i.e., $RestRep_0 := \{\}$.

Let $RestRep_i$ be the union of $RestRep_{i-1}$ and the singleton set of the following NPP clause: $\{\neg a, \neg b, c\}$, where a,b are consecutive vertices in AC_i and b, c are consecutive vertices in the representation of one of the strongly connected components, i.e., $RestRep_i := RestRep_{i-1} \cup \{\{\neg a, \neg b, c\}\}$, where $(a,b) \in CycEdges(AC_i) \wedge \exists_{C \in BigCycles}(b \in C \wedge (b,c) \in CycEdges(C))$, and let $A_i := a$, and let $B_i := b$, because in the next step we have to delete them.

Let $Rest_i$ be defined as $Rest_{i-1}$ subtraction the edge (A_i, B_i) and its reverse, i.e., $Rest_i := Rest_{i-1} \setminus \{(A_i, B_i), (B_i, A_i)\}$.

Let $RestRep$ be defined as the last one of its sequence, i.e.,

$$RestRep := RestRep_{i-1},\ where\ AC_i\ is\ UnDef.$$

Let $\mathcal{SBM}$ be defined as the union of the representation of the nodes, the representation of the strongly connected components (SCCs), the representation of the links between SCCs and the representation of the rest, i.e., $\mathcal{SBM} := NodeRep2 \cup NodeRep3 \cup BigRep \cup LinkRep \cup RestRep$. We say that $\mathcal{SBM}$ is a simplified Balatonboglár model of $\mathcal{D}$.

As the first example, we give the simplified Balatonboglár model of the graph in Figure 3.2 (after each clause we list the indices of subsumed full-length clauses from (3.4.3)):

$$\begin{aligned}\mathcal{SBM} = \{&\{\neg a, b, c\} : 6{,}7,\ \{\neg b, c, d\} : 3{,}11,\ \{\neg c, a, d\} : 9{,}13,\ \{\neg a, \neg b, c\} : 2{,}3,\\ &\{\neg b, \neg c, a\} : 8{,}9,\ \{\neg c, \neg a, b\} : 4{,}5,\ \{\neg a, \neg b, d\} : 1{,}3\}.\end{aligned} \tag{3.7.1}$$

Note that b has 3 child nodes, so instead of $\{\neg b, c, d\}$, we could use either $\{\neg b, a, c\}$ or $\{\neg b, a, d\}$. Note also that Figure 3.2 is not strongly connected, so its simplified Balatonboglár model is not a Black-and-White SAT problem. For example $\{\neg a, \neg b, \neg c, d\}$ is a solution of the SAT problem in (3.7.1). Note also that Figure 3.2 consists of two strongly

connected components (SCC), which are: $\{a, b, c\}$ and $\{d\}$. The first SCC is represented by the clauses: $\{\neg a, \neg b, c\}, \{\neg b, \neg c, a\}, \{\neg c, \neg a, b\}$, see *BigRep*. The second one cannot be represented, because it has less than 3 nodes. But we can represent the link between then, see *LinkRep*. All the following clauses can represent the link, but we need only one. We used the first one from the possible ones: $\{\neg a, \neg b, d\}, \{\neg a, \neg c, d\}, \{\neg b, \neg c, d\}$.

As the second example, we show the simplified Balatonboglár model of the communication graph in Figure 3.3 (after each clause we list the indices of subsumed full-length clauses from (3.4.3)):

$$\begin{aligned}\mathcal{SBM} = \{&\{\neg a, b, c\} : 6{,}7,\ \{\neg b, c, d\} : 3{,}11,\ \{\neg c, a, d\} : 9{,}13,\\ &\{\neg d, a\} : 8{,}10{,}12{,}14,\ \{\neg a, \neg b, c\} : 2{,}3,\ \{\neg b, \neg c, d\} : 1{,}9,\\ &\{\neg c, \neg d, a\} : 8{,}12,\ \{\neg d, \neg a, b\} : 4{,}6,\ \{\neg a, \neg c, d\} : 1{,}5\}.\end{aligned} \tag{3.7.2}$$

Note that since the communication graph in Figure 3.3 is strongly connected, its simplified Balatonboglár model is a Black-and-White SAT problem, i.e., the SAT problem in (3.7.2) has only these two solutions: $\{a, b, c, d\}$, and $\{\neg a, \neg b, \neg c, \neg d\}$. Since the graph is strongly connected it contains only one strongly connected component (SCC) which can be represented by this big-cycle: (a, b, c, d), which means that this SCC is represented by the clauses: $\{\neg a, \neg b, c\}, \{\neg b, \neg c, d\}, \{\neg c, \neg d, a\}, \{\neg d, \neg a, b\}$. After removing the corresponding edges, see $ToDel$, it remains a cycle: (a, c), which is represented by the clause: $\{\neg a, \neg c, d\}$, see *RestRep*.

As the third example, we show the simplified Balatonboglár model of the communication graph in Figure 3.5 (after each clause we list the indices of subsumed full-length clauses from (3.4.3)):

$$\begin{aligned}\mathcal{SBM} = \{&\{\neg a, b\} : 4{,}5{,}6{,}7,\ \{\neg b, a, c\} : 10{,}11,\ \{\neg c, b, d\} : 5{,}13,\\ &\{\neg d, a\} : 8{,}10{,}12{,}14,\ \{\neg a, \neg b, c\} : 2{,}3,\ \{\neg b, \neg c, d\} : 1{,}9,\\ &\{\neg c, \neg d, a\} : 8{,}12,\ \{\neg d, \neg a, b\} : 4{,}6\}.\end{aligned} \tag{3.7.3}$$

Note that since the communication graph in Figure 3.5 is strongly connected, its simplified Balatonboglár model (3.7.3) is a Black-and-White SAT problem.

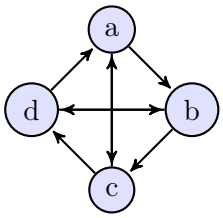

Figure 3.6: A strongly connected communication graph with big-cycle (a, b, c, d), and with two inner cycles: (a, c), and (b, d).

As the last example, we show the simplified Balatonboglár model of the communication graph in Figure 3.6 (after each clause we list the indices of subsumed full-length clauses

from (3.4.3)):

$$\begin{aligned}\mathcal{SBM} = \{&\{\neg a, b, c\} : 6{,}7,\ \{\neg b, c, d\} : 3{,}11,\ \{\neg c, d, a\} : 9{,}13,\\ &\{\neg d, a, b\} : 12{,}14,\ \{\neg a, \neg b, c\} : 2{,}3,\ \{\neg b, \neg c, d\} : 1{,}9,\\ &\{\neg c, \neg d, a\} : 8{,}12,\ \{\neg d, \neg a, b\} : 4{,}6,\\ &\{\neg a, \neg c, d\} : 1{,}5,\ \{\neg b, \neg d, a\} : 8{,}10\}.\end{aligned} \tag{3.7.4}$$

Note that since the communication graph in Figure 3.6 is strongly connected, its simplified Balatonboglár model (3.7.4) is a Black-and-White SAT problem. Furthermore, if we add the black and the white clause to this clause set, then we get a MIN-UNSAT problem [24].

3.8 Theoretical results

This section is the main contribution of this part. We prove that if we use one of the previous models then the model of a communication graph is a Black-and-White SAT problem iff the graph is strongly connected. To be able to prove this we need some auxiliary lemmas.

Lemma 1. *Let F be a Black-and-White SAT problem. Then $F \cup \{BB, WW\}$ is unsatisfiable.*

Proof. From the definition of Black-and-White SAT problem we know that F has only two solutions: BB and WW. Since $\neg BB = WW$, and $\neg WW = BB$ we have that $F \cup \{BB, WW\}$ is unsatisfiable. □

The next lemma states the following: there is a path from vertex x_i to x_j iff $\neg x_i \vee x_j$ is subsumed by the strong model of the graph.

Since the proof is rather technical, we give its high-level trace: We know that the strong model is a special SAT instance where each clause contains exactly one positive and one negative literal, hence, each resolvent of clauses from it is a binary clause with one positive and one negative literal. The main idea of the proof is that if we have two clauses $\neg v_1 \vee v_2$ and $\neg v_2 \vee v_3$, i.e., there is a path in the graph from v_1 to v_3, then by resolution we can generate $\neg v_1 \vee v_3$.

Lemma 2. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Then $\mathcal{SM}$ implies the formula $\neg x_i \vee x_j$ iff there is path from vertex x_i to x_j in $\mathcal{D}$.*

Proof. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$.

(From left to right:) We assume $\mathcal{SM}$ implies the formula $\neg x_i \vee x_j$, we show that there is path from vertex x_i to x_j in $\mathcal{D}$. We know that $\mathcal{SM}$ is a 2-SAT formula. Since $\mathcal{D}$ is a communication graph we know that $\mathcal{SM}$ is a special SAT instance where each clause contains exactly one positive and one negative literal, hence, each resolvent of clauses from $\mathcal{SM}$ is a 2-clause with one positive and one negative literal. From this, and since

resolution refutation is a sound and complete method for SAT solving we know that $\mathcal{SM}$ implies $\neg x_i \vee x_j$ iff we can obtain $\neg x_i \vee x_j$ from $\mathcal{SM}$ by resolution. Without loss of generality let us assume that we obtain $\neg x_i \vee x_j$ from $\mathcal{SM}$ by resolution on the following clauses from $\mathcal{SM}$: $\neg v_1 \vee v_2, \neg v_2 \vee v_3, \ldots \neg v_{k-1} \vee v_k$, where $v_1 = x_i$, and $v_k = x_j$ and v_z is one of the variables from $x_1 \ldots x_n$, where $z = 1 \ldots k$. These clauses represent a path in $\mathcal{D}$ from vertex x_i to x_j, and this is what we wanted to find.

(From right to left:) We assume there is path from vertex x_i to x_j in graph $\mathcal{D}$, we show $\mathcal{SM}$ implies the formula $\neg x_i \vee x_j$. Let us assume that the path from x_i to x_j is the following one: $v_1, v_2, \ldots v_k$, where $v_1 = x_i$ and $v_k = x_j$ and each v_z is one the vertices from $x_1 \ldots x_n$, where $z = 1 \ldots k$. From definition of $\mathcal{SM}$ we know that it contains the following clauses: $(\neg v_1 \vee v_2) \wedge (\neg v_2 \vee v_3) \wedge \cdots \wedge (\neg v_{k-1} \vee v_k)$. From this, by resolution, we get $(\neg v_1 \vee v_k)$. From this we obtain that $\mathcal{SM}$ implies $\neg x_i \vee x_j$, because $v_1 = x_i$ and $v_k = x_j$. □

Based on this lemma we can prove the following theorem, which states that the strong model of a strongly connected graph is a Black-and-White 2-SAT problem, and the other way around, the a directed graph representation of a Black-and-white 2-SAT problem is a strongly connected graph.

Theorem 1. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Then $\mathcal{SM}$ is a Black-and-White 2-SAT problem iff the graph $\mathcal{D}$ is strongly connected.*

Proof. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$.

(From left to right:) The main idea of the proof if the following, if a formula is satisfied by all solutions of a SAT problem, then it is implied by the problem. We know that $\mathcal{SM}$ has only two solutions, the white and the black assignments, both of them satisfy all $\neg x_i \vee x_j$ shaped formulae. From this and from Lemma 2 we obtain that in $\mathcal{D}$ there is a path from any vertex to any other one, i.e., $\mathcal{D}$ is strongly connected.

(From right to left:) The main idea of the proof is the following, since $\mathcal{D}$ is strongly connected we know that there is a path $v_1, v_2, \ldots, v_{z-1}, v_z, v_1$ which is cyclic and contains all vertices from $\mathcal{D}$, the corresponding clause set is $\{(\neg v_1 \vee v_2), (\neg v_2 \vee v_3), \ldots, (\neg v_{z-1} \vee v_z), (\neg v_z \vee v_1)\}$, which is a subset of $\mathcal{SM}$, and which can be satisfied only by the white and the black assignments. From this and since each clause in $\mathcal{SM}$ is a binary clause with one negative and one positive literal, we obtain that $\mathcal{SM}$ is a Black-and-White 2-SAT problem. □

To check whether a directed graph is strongly connected or not, we need linear time [77]. We show that in case of the strong model the SAT representation of a strongly connected directed graph can be solved also in linear time.

Theorem 2. *Let F be a Black-and-White 2-SAT problem. Then we need linear time to show that $F \cup \{BB, WW\}$ is unsatisfiable.*

Proof. The main idea of the proof is that any SAT solver which uses variable branching and BCP, can show that $F \wedge BB \wedge WW$ is unsatisfiable by using 1 variable branching and 2 BCP steps as follows: Variable branching will result in a unit. Without lost of generality let us assume that it is a positive one, say a. Then BCP will generate other positive units, because by Theorem 1 we know that F represents a strongly connected graph, therefore, vertex a has at least one outgoing edge, and each outgoing edge is represented by a binary clause, where variable a appears as a negative literal, and it contains also a positive literal, because of the construction of the strong model. BCP will finally result in a conflict, because BB is the negation of the white assignment. Then the other branch will result in a negative unit. This enables a BCP step which will generate negative units, and which will terminate in a conflict because WW is the negation of the black assignment. Since BCP is a linear time method [85] we need linear time to show that $F \cup \{BB, WW\}$ is unsatisfiable. □

Note that DPLL algorithm is a suitable choice to solve the above problem, because it uses variable branching and BCP.

We developed also a special algorithm to solve Black-and-White 2-SAT instances. This solver got the name BaW 1.0 [96]. It uses the idea of of the above theorem. Currently we work on BaW 2.0, which will be a special solver for models generated by the Balatonboglár model.

The question arises, while should we transform a graph into a SAT problem to check a property which can be checked in linear time in both fields? We think that our model is a new link between the two fields which might help to visualize SAT problems. This is not a problem in case of a 2-SAT problem, but in case of 3-SAT it is a problem.

The next lemma states that the weak model has at least two solutions, the white assignment and the black one.

Lemma 3. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Then $\mathcal{WM}$ has at least two solutions, namely the white assignment (WW) and the black assignment (BB). The same is true for $\mathcal{SM}$.*

Proof. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Since there is a positive and also a negative literal in each clause of $\mathcal{WM}$, the white assignment (WW) and also the black assignment (BB) are solutions for $\mathcal{WM}$. Since there is a positive and also a negative literal in each clause of $\mathcal{SM}$, it has the same property. □

The next lemma states that if the weak model has only two solutions, then each cycle in the communication graph has at least one exit point, except the cycle which contains all vertices of the graph.

Lemma 4. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Assume that $\mathcal{WM}$ has only two solutions. Then for each cycle $C \in Cycles$ we have that $ExitPoints(C)$ is not empty or C contains all vertices of $\mathcal{D}$.*

Proof. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Assume that $\mathcal{WM}$ has only two solutions. Lemma 3 makes sure that the solutions are WW and BB. Assume that our goal is not true, i.e., there is a cycle $C = (a_1, a_2, \ldots, a_k)$, which has no exit point and does not contain all vertices of $\mathcal{D}$. We show that this assumption leads to a contradiction. Let us construct the following assignment: $\{a_1, \ldots, a_k, \neg b_1, \ldots \neg b_m\}$, where $\{b_1, b_2, \ldots, b_m\} = \mathcal{V} \setminus C$, where $\mathcal{V}$ is the set vertices of $\mathcal{D}$. Since C has no exit point there is no corresponding clause generate by *CycleRep*. Since there is no clause which would falsify the constructed assignment, so this is a solution of $\mathcal{WM}$. But this contradicts the assumption that $\mathcal{WM}$ has only two solutions, namely WW and BB. So our original statement is true. □

The next lemma states that a Black-and-White SAT problem may not contain pure literals.

Lemma 5. *If $\mathcal{S}$ is a Black-and-White SAT problem, then it contains no pure literal.*

Proof. Assume $\mathcal{S}$ is a Black-and-White SAT problem. Then, by definition of a Black-and-White SAT problem, we know that $\mathcal{S}$ has only two solutions, WW and BB. Since there is no other solution of $\mathcal{S}$ and $\neg WW = BB$, we obtain that $\mathcal{S}$ may not contain a pure literal. □

The next lemma states that if we have a complete graph, then its weak model is a Black-and-White SAT problem. This result is somehow natural, since the weak model of a complete graph is the biggest possible what we can generate and the weak model has always at least two solutions.

Lemma 6. *If $\mathcal{D}$ is a complete communication graph, and $\mathcal{WM}$ is the weak model of $\mathcal{D}$, then $\mathcal{WM}$ is a Black-and-White SAT problem.*

Proof. Assume $\mathcal{D}$ is a complete communication graph. Assume $\mathcal{WM}$ is the weak model of $\mathcal{D}$. Since $\mathcal{D}$ is complete, for any vertex a we have that $OutE(a)$ contains all other vertices, so *NodeRep* contains all full-length clauses with 1 negative literal. Since any two vertices create a cycle where the exit points are the rest of the vertices, *CycleRep* contains all full-length clauses with 2 negative literals. The same is true for any $k = 2 \ldots n-1$ vertices, so *CycleRep* contains all full-length clauses with $2, \ldots, n-1$ negative literals. But $\mathcal{WM}$ do not subsumes the full-length clause with 0 negative literal, i.e., the white clause (WW), neither the full-length clause with n negative literal, i.e., the black clause (BB). This means that $\mathcal{WM}$ has only two solutions, the white and the black assignment, i.e., $\mathcal{WM}$ is a Black-and-White SAT problem. □

Now comes one of our main contributions, the theorem which states that the weak model of a strongly connected graph is a Black-and-White SAT problem. Which means that if the graph is strongly connected, then the weak model has the same power as the strong model. Since the proof is rather technical, we give also a high-level trace of it.

From left to right we prove the theorem in a constructive way: Let us have a sequence of vertices which can be built by Lemma 5 and by the construction of *NodeRep*. Eventually, we will find a cycle at the end of this sequence. Lemma 4 ensures that there is an exit point from this cycle, which either results in a bigger cycle, or in a longer sequence. Using these two steps eventually we construct a strongly connected component. From Lemma 4 it follows that it is also maximal (otherwise there would be an exit point from it which would allow to do the above steps), i.e., the graph is strongly connected.

From right to left we use induction: We start the proof from a complete graph. We know that complete graphs are also strongly connected. As the induction step, we drop an edge from this graph such that it remains strongly connected. We assume that before the drop the weak model was a Black-and-White SAT problem, which is true by Lemma 6. We show that it remains a Black-and-White SAT problem also after the drop. To show this, we show that each clause in the model after the drop is a subset of some clause from the model before the drop. So the new model may not have more solutions then the old one. But the new model has to have at least two solutions by Lemma 3. So the new model is a Black-and-White SAT problem. Since our graph can be constructed by dropping edges from a complete graph, which has the same set of vertices, its weak model is also a Black-and-White SAT problem.

Theorem 3. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Then $\mathcal{WM}$ is a Black-and-White SAT problem iff $\mathcal{D}$ is strongly connected.*

Proof. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. We show that $\mathcal{WM}$ is a Black-and-White SAT problem iff $\mathcal{D}$ is strongly connected.

(From left to right:) We assume $\mathcal{WM}$ is a Black-and-White SAT problem, we show by construction that $\mathcal{D}$ is strongly connected.

(R0) Let a_1 be a vertex from $\mathcal{D}$. Let $j = 1$.

(R1) Now we have a sequence of vertices: $a_1, \ldots, a_j$. We work with its last vertex: a_j. From Lemma 5 we know that literal a_j is not pure in $\mathcal{WM}$, so $\neg a_j$ is present in $\mathcal{WM}$. So there must be a clause in $\mathcal{WM}$ generated by *NodeRep* for vertex a_j. Since *NodeRep* is a subset of *CycleRep*, therefore, by Lemma 4, there is a clause $\{\neg a_j, b_1, \ldots, b_k\}$ in $\mathcal{WM}$. Let a_{j+1} be a vertex from $b_1, \ldots, b_k$ such that a_{j+1} is not in $\{a_1, \ldots, a_j\}$. If this is possible, then let $j = j + 1$ and goto (R1). Note that this is not always possible because $\mathcal{D}$ is a final graph, so eventually each vertex from $b_1, \ldots, b_k$ must be in the set $\{a_1, \ldots, a_j\}$. In this case let a_{j+1} be a vertex from $b_1, \ldots, b_k$ such that $a_{j+1} = a_i$, where $i < j$ and i is the smallest possible. This means that we have i such that $i < j$ and $a_i = a_{j+1}$, i.e., $(a_i, \ldots, a_j)$ is a cycle. Goto (R2).

(R2) Now we have a sequence of vertices: $a_1, \ldots, a_j$, and we a cycle at the end of this sequence from a_i to a_j. If $i = 1$ and the cycle contains all the vertices from $\mathcal{D}$, then goto (R3). Otherwise by creation of *CycleRep* and by Lemma 4 we know that $\mathcal{WM}$ contains a clause $\{\neg a_i, \ldots, \neg a_j, b_1, \ldots, b_m\}$, where $b_1, \ldots, b_m$ are exit points of the cycle $(a_i, \ldots, a_j)$.

Let a_{j+1} be a vertex from $b_1, \ldots, b_m$ such that a_{j+1} is not present in $a_1, \ldots, a_j$. If this is possible, then let $j = j+1$, and goto (R1). Note that this is not always possible because $\mathcal{D}$ is a final graph. so eventually each vertex from $b_1, \ldots, b_k$ must be in the set $\{a_1, \ldots, a_j\}$. In this case let a_{j+1} be a vertex from $b_1, \ldots, b_m$, such that $a_{j+1} = a_{i'}$ and i' is the smallest possible. Since $b_1, \ldots, b_m$ are exit points of the cycle $(a_i, \ldots, a_j)$, we know that i' is smaller than i because each vertex from $b_1, \ldots, b_m$ is different from $a_i, \ldots, a_j$. This means that we have i' such that $i' < j$, and $i' < i$ and $a_{i'} = a_{j+1}$, i.e., $(a'_i, \ldots, a_j)$ is a cycle. Let $i = i'$, and goto (R2).
(R3) Eventually (R2) exists to (R3) because, by Lemma 4, each cycle has at least one exit point, which result in a bigger cycle, or in a longer $a_1, \ldots, a_j$ sequence of vertices. So eventually the sequence $a_1, \ldots, a_j$ will contain all vertices, and eventually i will be 1, so eventually we will find a cycle which contains all vertices. Hence, $\mathcal{D}$ is strongly connected.

(From right to left:) We assume $\mathcal{D}$ is strongly connected, we show by induction that $\mathcal{WM}$ is a Black-and-White SAT problem. Let H be the complete communication graph which has the same set of vertices as $\mathcal{D}$. We know from Lemma 6 that the weak model of H is a Black-and-White SAT problem. This is our induction base. It is well known that complete graphs are also strongly connected. We create G from H by deleting edges from H such that G remains strongly connected. Let Z be the weak model of G. Our induction hypothesis is that Z is a Black-and-White SAT problem. Our induction step is that we delete an edge from G such that it remains strongly connected. Let G' be this graph. Let Z' be the weak model of G'. We show that Z' a Black-and-White SAT problem. To show this, we show that $Z' \geq Z$, i.e., Z' subsumes all clauses from Z. Without loss of generality let us assume that we deleted the edge from vertex a to vertex b. All clause in Z are the same as the clauses in Z' excepted the ones generated by *CycleRep* for those cycles which contain a. From those clauses let $D = \{\neg a_1, \ldots, \neg a_k, b_1, \ldots, b_m\}$ be an arbitrary but fixed one, where $(a_1, \ldots, a_k)$ is a cycle in Z and $b_1, \ldots, b_m$ are its exit points. We know that $k >= 1$ and $a = a_i$ for some $i \in \{1, \ldots, k\}$, and b is one of the other vertices from this clause. There are the following cases:

- If b is one of the exit points and some other vertex from the cycle has an edge to b, then Z' contains the same clause: D.
- If b is one of the exit points, say $b = b_p$, where $p \in \{1, \ldots, m\}$, and no other vertex from the cycle has an edge to b, then Z' contains the following clause: $\{\neg a_1, \ldots, \neg a_k, b_1, \ldots, b_{p-1}, b_{p+1}, \ldots b_m\}$, which is a subset of D. Note that the vertex sequence $b_1, \ldots, b_{p-1}, b_{p+1}, \ldots b_m$ contains some vertices, because G' is a strongly connected graph, i.e., by Lemma 4, each cycle has an exit point, unless it contains all vertices.
- If b is one of the other vertices of the cycle, but b is not the next vertex after a in this cycle, then Z' contains the same clause, D.

- If b is one of the other vertices of the cycle, and b is the next vertex after a in this cycle, then $(a_1, \ldots, a_k)$ is not a cycle in Z'. If there is no other child vertex of a in $\{a_1, \ldots, a_k\}$, then the clause generated by $NodeRep$ for a in G' is a subset of D. If there are other child vertices of a in $\{a_1, \ldots, a_k\}$, then there must be a subset of $\{a_1, \ldots, a_k\}$ which contains all other child vertices of a except b, and which is a cycle in G'. The clause generated by $CycleRep$ for this cycle in G' is a subset of D.

We showed that Z' subsumes all clauses from Z. This means that $Z' \geq Z$, i.e., Z' may have no other solutions than Z. By our induction hypothesis Z has only two solutions BB and WW. We know from Lemma 3 that BB and WW are solutions for Z', because Z' is a weak model. In addition, since Z' may have no other solution, Z' is a Black-and-White SAT problem. This means that the weak model of any strongly connected communication graph is a Black-and-White SAT problem. Hence, $\mathcal{WM}$ is a blacked-and-white SAT problem, because $\mathcal{D}$ is strongly connected and $\mathcal{WM}$ is its weak model. □

The following lemma states that $\mathcal{SM}$ entails $\mathcal{WM}$, i.e., $\mathcal{SM} \geq \mathcal{WM}$ for any communication graph. This means that any solution of $\mathcal{SM}$ is also a solution of $\mathcal{WM}$. To show this, it is enough to show that $\mathcal{SM}$ subsumes all clauses from $\mathcal{WM}$.

Lemma 7. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Then $\mathcal{SM} \geq \mathcal{WM}$.*

Proof. Assume $\mathcal{D}$ is a communication graph, $\mathcal{SM}$ is the strong model of $\mathcal{D}$, and $\mathcal{WM}$ is the weak model of $\mathcal{D}$. We show that $\mathcal{SM} \geq \mathcal{WM}$. To show this it is enough to show that $\mathcal{SM}$ subsumes all clauses from $\mathcal{WM}$. Let C be an arbitrary but fixed element of $\mathcal{WM}$. If C is an element of $NodeRep$, then, without loss of generality, we may assume that $C = \{\neg a, b_1, b_2, \ldots, b_k\}$, where $k > 0$. From the construction of $NodeRep$ we know that in $\mathcal{D}$ there are the following edges: $(a, b_1), (a, b_2), \ldots, (a, b_k)$. Since $\mathcal{SM}$ is generated from the same graph, by its definition we know that $\mathcal{SM}$ contains the clauses $\{\neg a, b_1\}, \{\neg a, b_2\}, \ldots, \{\neg a, b_k\}$ and all of these ones subsume C.

If C is an element of $Cycles$, then, without loss of generality, we may assume that $C = \{\neg d_1, \neg d_2, \ldots, \neg d_m, e_1, e_2, \ldots, e_n\}$, where $m > 0$, and $n > 0$. From the construction of $Cycles$ we know that in $\mathcal{D}$ there is an edge: (d, e) such that d is one of the vertices from $d_1, d_2, \ldots, d_m$, and e is one of the vertices from $e_1, e_2, \ldots, e_m$. Since $\mathcal{SM}$ is generated from the same graph, by its definition we know that $\mathcal{SM}$ contains the clauses $\{\neg d, e\}$, which subsumes C.

Hence, $\mathcal{SM} \geq \mathcal{WM}$. □

The following lemma states that $\mathcal{SM}$ and $\mathcal{WM}$ are equivalent if the represented communication graph is strongly connected or there are no branches in the graph. If the communication graph is strongly connected, then both $\mathcal{SM}$ and $\mathcal{WM}$ are Black-and-White SAT problems, so they are equivalent.

Lemma 8. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$.*
If
(I) $\mathcal{D}$ is strongly connected, or
(II) each vertex in $\mathcal{D}$ has only one child,
then $\mathcal{SM}$ and $\mathcal{WM}$ are equivalent.

Proof. In case (I) we know that $\mathcal{D}$ is strongly connected, so by Theorem 1, and by Theorem 3 we know that $\mathcal{SM}$ and $\mathcal{WM}$ are Black-and-White SAT problems, so by the definition of clause set equivalent-ness they are equivalent. In case (II) we have that $NodeRep = \mathcal{SM}$ and $NodeRep \subseteq \mathcal{WM}$. From this we obtain that $\mathcal{WM} \geq \mathcal{SM}$. From Lemma 7 we know that $\mathcal{SM} \geq \mathcal{WM}$. Hence, $\mathcal{SM}$ and $\mathcal{WM}$ are equivalent. □

The following lemma gives a criterion for the communication graph which makes the strong model stronger than the weak model. This lemma is not needed to prove the following theorems but it is still interesting.

Lemma 9. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Then $\mathcal{SM} > \mathcal{WM}$ iff $\mathcal{D}$ is not strongly connected, and there is at least one vertex in it which has more than one child vertex.*

Proof. Assume $\mathcal{D}$ is a communication graph. Assume $\mathcal{SM}$ is the strong model of $\mathcal{D}$. Assume $\mathcal{WM}$ is the weak model of $\mathcal{D}$. We show that $\mathcal{SM} > \mathcal{WM}$ iff $\mathcal{D}$ is not strongly connected, and there is at least one vertex in it which has more than one child vertex.

(From left to right:) We assume $\mathcal{SM} > \mathcal{WM}$, so $\mathcal{SM}$ and $\mathcal{WM}$ are not equivalent. Since the right side of Lemma 8 is negated, the negation of the left side of Lemma 8 is implied. Hence, $\mathcal{D}$ is not strongly connected, and there is at least one vertex in it which has more than one child vertex.

(From right to left:) Assume $\mathcal{D}$ is not strongly connected, and there is at least one vertex in it, say a, which has more than one child vertex, say $b_1, b_2, \ldots, b_k$, where $k > 1$. In this case $(WW \setminus \{b_1\}) \cup \{\neg b_1\}$ and $(BB \setminus \{\neg a, \neg b_k\}) \cup \{a, b_k\}$ are solutions for $\mathcal{WM}$ but they are not solutions of $\mathcal{SM}$, since they do not satisfies the clause $\{\neg a, b_1\}$ from $\mathcal{SM}$. So $\mathcal{SM}$ and $\mathcal{WM}$ are not equivalent. From this and from Lemma 7 we obtain that $\mathcal{SM} > \mathcal{WM}$. □

The following theorem, the so called Transitions Theorem, states that any "transition" between $\mathcal{SM}$ and $\mathcal{WM}$ is a Black-and-White SAT problem iff the communication graph is strongly connected. This is one of the main results of this work.

Theorem 4 (Transitions Theorem)**.** *If for all communication graphs $\mathcal{D}$ we have that $\mathcal{SM} \geq \mathcal{MM} \geq \mathcal{WM}$, where $\mathcal{SM}$ is the strong model of $\mathcal{D}$, and $\mathcal{WM}$ is the weak model of $\mathcal{D}$, and $\mathcal{MM}$ is an arbitrary but fixed model of $\mathcal{D}$, then $\mathcal{MM}$ is a Black-and-White SAT problem iff $\mathcal{D}$ is strongly connected.*

Proof. Assume that $\mathcal{D}$ is a communication graph. Assume that $\mathcal{SM}$ is the strong model of $\mathcal{D}$, and $\mathcal{WM}$ is the weak model of $\mathcal{D}$, and $\mathcal{MM}$ is an arbitrary but fixed model of $\mathcal{D}$. We show that $\mathcal{MM}$ is a Black-and-White SAT problem iff $\mathcal{D}$ is strongly connected.

(From left to right:) Assume $\mathcal{MM}$ is a Black-and-White SAT problem. From this and from the assumption that $\mathcal{SM} \geq \mathcal{MM}$, we have that $\mathcal{SM}$ is a Black-and-White SAT problem, since any strong model has always at least two solutions, the white assignment and the black one, see Lemma 3. From this and from Theorem 1 we have that $\mathcal{D}$ is strongly connected.

(From right to left:) Assume $\mathcal{D}$ is strongly connected. Then by Lemma 8 we know that $\mathcal{SM}$ and $\mathcal{WM}$ are equivalent. From this and from the assumption that $\mathcal{SM} \geq \mathcal{MM} \geq \mathcal{WM}$ it follows that $\mathcal{SM}$ and $\mathcal{MM}$ and $\mathcal{WM}$ are equivalent. Since $\mathcal{D}$ is strongly connected by Theorem 3 we know that $\mathcal{WM}$ is a Black-and-White SAT problem. We know that $\mathcal{MM}$ and $\mathcal{WM}$ are equivalent, hence, $\mathcal{MM}$ is a Black-and-White SAT problem. □

The following theorem states that the Balatonboglár model of a communication graph is a transition between the strong and the weak model. Actually, Balatonboglár is a small city in Hungary next to lake Balaton. I have spent quite a few time next to the lake thinking on a model which is 3-SAT and Black-and-White iff the directed graph is strongly connected. As the next two theorems show I was successful. As the respect of that great time we gave the name Balatonboglár to that model.

Theorem 5. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{BM}$ be a Balatonboglár model of $\mathcal{D}$. Then $\mathcal{SM} \geq \mathcal{BM} \geq \mathcal{WM}$.*

Proof. Assume $\mathcal{D}$ is a communication graph. Assume $\mathcal{BM}$ is a Balatonboglár model of $\mathcal{D}$. We show that $\mathcal{SM} \geq \mathcal{BM} \geq \mathcal{WM}$.

(I) As the first step, we show that $\mathcal{SM} \geq \mathcal{BM}$. To show this it is enough to show that $\mathcal{SM}$ subsumes all clauses from $\mathcal{BM}$. Let C be an arbitrary but fixed element of $\mathcal{BM}$. This means that C is an element either of $NodeRep2$, or $NodeRep3$, or $TagForward$. If C is an element of $NodeRep2$, then C is also element of $\mathcal{SM}$, so this case is trivial. If C is an element of $NodeRep3$, then, without loss of generality, we may assume that $C = \{\neg a, b, c\}$, where a is a vertex in $\mathcal{D}$ and $b \in OutE(a)$ and $c \in OutE(a)$ and $b \neq c$. From the construction of $NodeRep3$ we know that in $\mathcal{D}$ there are the following edges: $(a, b), (a, c)$. Since $\mathcal{SM}$ is generated from the same graph, by its definition we know that $\mathcal{SM}$ contains the clauses $\{\neg a, b\}, \{\neg a, c\}$ and both of them subsume C. If C is an element of $TagForward$, then, without loss of generality, we may assume that $C = \{\neg d, \neg e, f\}$, where d is a vertex in $\mathcal{D}$, and $e \in OutE(d)$ and $f \in OutE(e)$. From the construction of $TagForward$ we know that in $\mathcal{D}$ there are the following edges: $(d, e), (e, f)$. Since $\mathcal{SM}$ is generated from the same graph, by its definition we know that $\mathcal{SM}$ contains the clause $\{\neg e, f\}$ which subsumes C.

(II) As the second step, we show $\mathcal{BM} \geq \mathcal{WM}$. To show this it is enough to show that $\mathcal{BM}$ subsumes all clauses from $\mathcal{WM}$. Let C be an arbitrary but fixed element of $\mathcal{WM}$.

This means that C is an element either of $NodeRep$ or $Cycles$. If C is an element of $NodeRep$, then, without loss of generality, we may assume that $C = \{\neg a, b_1, b_2, \ldots, b_k\}$, where $k > 0$. From the construction of $NodeRep$ we know that in $\mathcal{D}$ there are the following edges: $(a, b_1), (a, b_2), \ldots, (a, b_k)$. Since $\mathcal{BM}$ is generated from the same graph, by its definition we know that $\mathcal{BM}$ contains the clause $\{\neg a, b_1\}$, if $k = 1$, or $\{\neg a, b, c\}$, where b and c are different vertices from $b_1, b_2, \ldots, b_k$, and that clause subsumes C. If C is an element of $Cycles$, then, without loss of generality, we may assume that $C = \{\neg d_1, \neg d_2, \ldots, \neg d_m, e_1, e_2, \ldots, e_n\}$, where $m > 0$, and $n > 0$. If $m = 1$ then C is an element of $NodeRep$, which case is already considered, so we may assume that $m > 1$. From the construction of $Cycles$ and from $m > 1$ we know that in $\mathcal{D}$ there are these edges: (d_i, d_{i+1}) and (d_{i+1}, e) such that d_i and d_{i+1} are one of the vertices from $d_1, d_2, \ldots, d_m$, and e is one of the vertices from $e_1, e_2, \ldots, e_m$. Since $\mathcal{BM}$ is generated from the same graph, by its definition we know that $\mathcal{BM}$ contains the clauses $\{\neg d_i, \neg d_{i+1}, e\}$ which subsumes C.

Hence, $\mathcal{SM} \geq \mathcal{BM} \geq \mathcal{WM}$. □

The next theorem states that the Balatonboglár model is a Black-and-White 3-SAT problem iff the represented directed graphs is strongly connected. This result is an immediate consequence of the above result, see Theorem 5, and Transitions Theorem (Theorem 4),

Theorem 6. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{BM}$ be a Balatonboglár model of $\mathcal{D}$. Then $\mathcal{BM}$ is a Black-and-White 3-SAT problem iff $\mathcal{D}$ is strongly connected.*

Proof. Assume $\mathcal{D}$ is a communication graph. Assume $\mathcal{BM}$ is a Balatonboglár model of $\mathcal{D}$. From this, from Theorem 5 and from the Transitions Theorem (Theorem 4) we know that $\mathcal{BM}$ is a Black-and-White SAT problem iff $\mathcal{D}$ is strongly connected. From the construction of $\mathcal{BM}$ we know also that it is a 3-SAT problem for any communication graph. Hence, $\mathcal{BM}$ is a Black-and-White 3-SAT problem iff $\mathcal{D}$ is strongly connected. □

The above result is very nice. It took us several years to find it. We have submitted this result to Theory of Computing [87] and we are still waiting for its acceptance. We hope that it will be published soon.

As the next step we tried to make a model which is more compact than the Balatonboglár model. This new model got the name: simplified Balatonboglár model. We show that this is also a transition between the strong model and the weak one.

Theorem 7. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Let $\mathcal{BM}$ be a Balatonboglár model of $\mathcal{D}$. Let $\mathcal{SBM}$ be a simplified Balatonboglár model of $\mathcal{D}$. Assume that $NodeRep3$ is a deterministic method. Then we have also that $\mathcal{SM} \geq \mathcal{BM} \geq \mathcal{SBM} \geq \mathcal{WM}$.*

Proof. It is easy to see that $\mathcal{BM} \geq \mathcal{SBM}$, because by the construction of $\mathcal{SBM}$ it is a subset of $\mathcal{BM}$ if we assume that $NodeRep3$ is a deterministic method. From Theorem 5 we know that $\mathcal{SM} \geq \mathcal{BM}$. We show that $\mathcal{SBM} \geq \mathcal{WM}$. To show this it is enough to show

that $\mathcal{SBM}$ subsumes all clauses from $\mathcal{WM}$. Let C be an arbitrary but fixed element of $\mathcal{WM}$. This means that C is an element either of $NodeRep$ or $Cycles$. If C is an element of $NodeRep$, then, this case is already covered in the proof of Theorem 5 because the simplified Balatonboglár model and the Balatonboglár model represent nodes in the same way, see $NodeRep2$ and $NodeRep3$.

If C is an element of $Cycles$, then, without loss of generality, we may assume that $C = \{\neg d_1, \neg d_2, \ldots, \neg d_m, e_1, e_2, \ldots, e_n\}$, where $m > 1$, and $n > 0$. From the definition of strongly connected components there is a $DC \in BigCycles$ such that for all $i \in 1, \ldots, m$ we have that $d_i \in DC$. There are two cases: (a) either there is $j \in 1, \ldots, n$ such that $e_j \in DC$ or (b) there is no such j. In case (a) let a, b such that $a, b \in \{d_1, d_2, \ldots, d_m\}$ and $b \in OutE(a)$ and $e_j \in OutE(b)$. There are two cases: (aa) either a and b are consecutive nodes in DC or (ab) not.

In case (aa) there is a clause $D \in BigRep$ such that D is a subset of C. In case (ab) there is a clause $D \in RestRep$ such that D is a subset of C. Case (b) is only possible if the cycle represented by C corresponds to a strong connected component, i.e., if $\{d_1, d_2, \ldots, d_m\} = DC$. In this case there is a clause $D \in LinkRep$ such that D is a subset of C.

Hence, $\mathcal{SM} \geq \mathcal{BM} \geq \mathcal{SBM} \geq \mathcal{WM}$. □

The next theorem states that the simplified Balatonboglár model is a Black-and-White 3-SAT problem iff the represented directed graphs is strongly connected. This result is an immediate consequence of the above result, see Theorem 7, and Transitions Theorem (Theorem 4).

Theorem 8. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SBM}$ be a simplified Balatonboglár model of $\mathcal{D}$. Then $\mathcal{SBM}$ is a Black-and-White 3-SAT problem iff $\mathcal{D}$ is strongly connected.*

Proof. Assume $\mathcal{D}$ is a communication graph. Assume $\mathcal{SBM}$ is a simplified Balatonboglár model of $\mathcal{D}$. From this, from Theorem 7 and from the Transitions Theorem (Theorem 4) we know that $\mathcal{SBM}$ is a Black-and-White SAT problem iff $\mathcal{D}$ is strongly connected. From the construction of $\mathcal{SBM}$ we know also that it is a 3-SAT problem for any communication graph. Hence, $\mathcal{SBM}$ is a Black-and-White 3-SAT problem iff $\mathcal{D}$ is strongly connected. □

3.9 Unpublished results

In this section we introduce some theoretical results which are not published yet. First, we recall some definitions from Chapter 2.1: A clause is entailed by a clause set iff the clause is the logical consequence of the clause set. Subsumed clauses are also entailed. A clause is independent in a clause set iff it is not entailed. A full-length clause is independent in a clause set iff it is not subsumed.

We define also some auxiliary function. If C is a clause, then let $V(C)$ be the set of variables which occur in C, let $N(C)$ be the set of negative literals from C, and let $P(C)$ be

the set of positive literals from C. We have that $C = N(C) \cup P(C)$, $V(N(C)) \cap V(P(C)) = \emptyset$, and $P(C) = V(P(C))$.

The next lemma is interesting, but an idea in its proof is even more interesting.

Lemma 10. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Then for all C in $\mathcal{WM}$ we have that C is not subsumed by $\mathcal{WM} \setminus \{C\}$.*

Proof. Assume $\mathcal{D}$ is a communication graph. Assume $\mathcal{WM}$ is the weak model of $\mathcal{D}$. We show that for all C in $\mathcal{WM}$ we have that C is not subsumed in $\mathcal{WM} \setminus \{C\}$. To show this let C' be an arbitrary but fixed clause from $\mathcal{WM}$, we show that C' is not subsumed in $\mathcal{WM} \setminus \{C'\}$, which means that for all B in $\mathcal{WM}$, such that $B \neq C'$ we have that $B \not\subseteq C'$. Let B' be an arbitrary but fixed clause from $\mathcal{WM} \setminus \{C'\}$, we show that $B' \not\subseteq C'$. We know that $C' = N(C') \cup P(C')$, and $B' = N(B') \cup P(B')$. Because of the construction of $\mathcal{WM}$ neither of these sets are empty. There are two cases: (a) $N(B') \not\subseteq N(C')$ or $P(B') \not\subseteq P(C')$, (b) $N(B') \subseteq N(C')$ and $P(B') \subseteq P(C')$. In case (a) we trivially have that $B' \not\subseteq C'$. In case (b) we have that $V(N(B'))$ represents a cycle, and $V(N(C'))$ represents also a cycle. From the construction of $\mathcal{WM}$ we know that $N(B') \neq N(C')$, because each cycle is represented only by one clause and $B' \neq C'$. Therefore, we have that $N(B') \subset N(C')$. From this, and from $P(B') \subseteq P(C')$ we know that there is a variable v such that v is not present in B' neither as a positive nor as a negative literal, but v is present as a negative literal in C'. This variable, v, is also a node in $\mathcal{D}$. Now we have the following situation: we have a small circle, $V(N(B'))$, and a bigger one, $V(N(C'))$, such that $N(B') \subset N(C')$. In the graph there is no exit point to v from $V(N(B'))$, so there must be a path from $V(N(B'))$ to v such that this bigger cycle is created. Without loss of generality, let us assume that this path is the following: $b, d, v_1 \ldots, v_q$, such that $q \geq 1$, $v_q = v$, b is in $V(N(B'))$, d is not in $V(N(B'))$. Note that d must be in $P(B')$ and in $V(N(C'))$. Therefore, it is not true that $B' \subseteq C'$, because there is variable, d, which occurs as a positive literal in B' but as a negative literal in C', i.e., $B' \not\subseteq C'$. Hence, C is not subsumed in $\mathcal{WM} \setminus \{C\}$. □

Now let us prove the generalization of this lemma. We do not use the previous lemma to prove the next theorem, but we use its main idea: clause C' is not subsumed by clause B' because there must be a path $b, d, \ldots, v$ from $V(N(B'))$ to $V(N(C'))$, such that d occurs as a positive literal in B', but as a negative literal in C', hence, C' is not subsumed by B'.

Theorem 9. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Then for all C in $\mathcal{WM}$ we have that C is independent in $\mathcal{WM} \setminus \{C\}$.*

Proof. Assume $\mathcal{D}$ is a communication graph. Assume $\mathcal{WM}$ is the weak model of $\mathcal{D}$. We show that for all C in $\mathcal{WM}$ we have that C is independent in $\mathcal{WM} \setminus \{C\}$. To show that let C' be an arbitrary but fixed clause from $\mathcal{WM}$, we show that C' is independent in $\mathcal{WM} \setminus \{C'\}$. To show this it is enough to show that there exists a full-length clause C'' such that C' subsumes C'', i.e., $C' \subseteq C''$, and there is no other clause in $\mathcal{WM}$ which

subsumes C'', i.e., C'' is independent in $\mathcal{WM} \setminus \{C'\}$, i.e., for all B in $\mathcal{WM}$, such that $B \neq C'$ we have that $B \not\subseteq C''$. To show this let B' be an arbitrary but fixed clause from $\mathcal{WM}$, such that $B' \neq C'$, we show that $B' \not\subseteq C''$. To show this let us construct C'' as follows: C'' contains all literals from C' and the rest of variables as positive literals, i.e., $C'' := C' \cup V \setminus V(C')$, where V is the set of variables in $\mathcal{D}$. Because of this construction we have that $N(C'') = N(C')$. In the followings we use the idea from the proof of Lemma 10. We know that $C'' = N(C'') \cup P(C'')$, and $B' = N(B') \cup P(B')$. Because of the construction of $\mathcal{WM}$ neither of these sets are empty. There are two cases: either (a) $N(B') \not\subseteq N(C'')$ or $P(B') \not\subseteq P(C'')$; or (b) $N(B') \subseteq N(C'')$ and $P(B') \subseteq P(C'')$. In case (a) we trivially have that $B' \not\subseteq C''$. In case (b) we have that $V(N(B'))$ represents a cycle, and $V(N(C''))$ represents also a cycle. From the construction of $\mathcal{WM}$ we know that $N(B') \neq N(C'')$, because each cycle is represented only by one clause and $B' \neq C'$. From this and from $N(C'') = N(C')$, we have that $N(B') \subset N(C'')$. From this, and from $P(B') \subseteq P(C'')$ we know that there is a variable v such that v is not present in B' neither as a positive nor as a negative literal, but v is present as a negative literal in C''. This variable, v, is also a node in $\mathcal{D}$. Now we have the following situation: we have a small circle, $V(N(B'))$, and a bigger one, $V(N(C''))$, such that $N(B') \subset N(C'')$. In the graph there is no exit point to v from $V(N(B'))$, so there must be a path from $V(N(B'))$ to v such that this bigger cycle is created. Without loss of generality, let us assume that this path is the following: $b, d, v_1 \ldots, v_q$, such that $q \geq 1$, $v_q = v$, b is in $V(N(B'))$, d is not in $V(N(B'))$. Note that d must be in $P(B')$ and in $V(N(C''))$. Therefore, it is not true that $B' \subseteq C''$, because there is variable, d, which occurs as a positive literal in B' but as a negative literal in C'', i.e., $B' \not\subseteq C''$. Hence, C is independent in $\mathcal{WM} \setminus \{C\}$. □

This is a very powerful theorem. It states that the weak model is the weakest possible one. This theorem has several corollaries.

Corollary 1. *Let $\mathcal{D}$ be a strongly connected communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Then $\mathcal{WM} \cup \{BB, WW\}$ is a minimal unsatisfiability SAT instance.*

Proof. From the construction of the weak model we know that neither BB nor WW is entailed by $\mathcal{WM}$. From this, from Lemma 1 and from Theorem 9 follows this corollary. □

Corollary 2. *Let $\mathcal{D}$ be a strongly connected communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Then there is no blocked clause in $\mathcal{WM} \cup \{BB, WW\}$.*

Proof. This is an immediate consequence of Corollary 1 and the property of blocked clauses that they can be removed from a clause set without changing its satisfiability. □

We hope that we can find more nice corollaries. We hope that this powerful theorem helps us to understand how to use resolution to create shorter clauses if we know the input communication graph. We already know this in case of the strong model, see the proof of Lemma 2, but we do not know yet how to do the same in case of the weak model.

The next lemma is motivated by the following idea: the restrictions which are forced by the nature of Black-and-White SAT problems apply also to strongly connected directed graphs. Such a restrictions is that Black-and-White SAT problems subsumes n full-length clauses with exactly one positive literal if $n > 1$, where n is the number of variables. Since those clauses represent cycles (or vertices if the length of the clause is 2) with one exit point, we have that each strongly connected directed graph must contain at least n cycles or vertices which have only one exit point.

Lemma 11. *Let $\mathcal{D}$ be a strongly connected directed graph. Let n be the number of vertices of $\mathcal{D}$. If $n > 1$, then $\mathcal{D}$ contains at least n cycles or vertices which have only one exit point.*

Proof. Let $\mathcal{D}$ be a strongly connected directed graph. Let n be the number of vertices of $\mathcal{D}$. Let us assume that $n > 1$.

Without loss of generality we may assume that the set of vertices of $\mathcal{D}$ are boolean variables, which means that if x is a vertex, then $\neg x$ must not be a vertex of $\mathcal{D}$. Let $\mathcal{D}'$ be communication graph created from $\mathcal{D}$ by deleting each self loop, i.e., edges (a,a), where a is element of vertices of $\mathcal{D}$.

It is easy to see that $\mathcal{D}$ contains at least n cycles or vertices which have only one exit point if $\mathcal{D}'$ has the the same property, because the self loop (b,b) has only one exit point in $\mathcal{D}$, only if b has exactly one exit point in $\mathcal{D}'$. Furthermore, $\mathcal{D}'$ is also strongly connected.

Let $\mathcal{WM}$ be the weak model of $\mathcal{D}'$. From Theorem 3 we know that $\mathcal{WM}$ is a Black-and-White SAT problem. From this we know that it subsumes all full-length clauses whit exactly one positive literal. We know that there are n such full-length clauses. There must be at least n such clauses in $\mathcal{WM}$ which subsume one of those full-length clauses. Theorem 9 makes sure that those clauses have a unique counter part in $\mathcal{D}'$. The construction of the weak model makes sure that those counter parts are cycles or vertices with one exit point. Therefore, $\mathcal{D}'$ contains at least n cycles or vertices which have only one exit point. Hence, if $n > 1$, then $\mathcal{D}$ contains at least n cycles or vertices which have only one exit point. □

We give a high level example which helps us to better understand the main step of the proof: Let us assume that $\neg n_1, \ldots, \neg n_k, p$ is a clause form $\mathcal{WM}$ which subsumes a full-length clause with one positive liter, where $\neg n_1, \ldots, \neg n_k$ are negative literals, p is a positive literal, and $1 \leq k < n$. We know that this clause represents a cycle $(n_1, \ldots, n_k)$ in $\mathcal{D}'$ with one exit point: p.

Maybe this lemma is a known one in graph theory, maybe not. We tried to find it in these textbooks: [7, 17], but we could not locate it. We wrote an email to Prof. Gregory Gutin, one of the author of the book on directed graphs [7]. He wrote us that this lemma seems to be new. If it is a known one, then at least we found a new proof for it. If not, i.e., it is a new piece of knowledge, then this is the first result of us in graph theory which is independent from our models, and the new structures are used to prove something interesting.

The next lemma is a very easy one. It states that there is no unit in a Black-and-White clause set.

Lemma 12. *Let S be Black-and-White SAT problem. Then there is no unit in S.*

Proof. A unit either subsumes WW, if it is a positive literal, or BB, if it is a negative literal. But a Black-and-White SAT problem, by its definition, does not subsumes neither of them. So there must be no unit in S. □

We recall the definition of NPP clause: A clause is an NPP clause iff it contains exactly one negative literal.

The next theorem is a property of Black-and-White SAT problems, which states that if we generate a graph out of its NPP clauses, then there is at least one cycle in this graph. This theorem is a little step forward which helps us to create a directed graph out of SAT problem. This theorem does not assume that we use one of our models, but it uses the common element of the models that the input graph can be reconstructed using only the NPP clauses.

We use the following functions:

$$NPPGraphNodes(S) := \{n \mid \{n, p_1, \ldots, p_k\} \textit{ is an } NPP \textit{ clause in } S\}.$$

$$NPPGraphEdges(S) := \{(n, p_1), \ldots, (n, p_k) \mid \{n, p_1, \ldots, p_k\} \textit{ is an } NPP \textit{ clause in } S\}.$$

$$NPPGraph(S) := (NPPGraphNodes(S), NPPGraphEdges(S)).$$

Theorem 10. *Let S be Black-and-White SAT problem. Then there exists a cycle in $NPPGraph(S)$.*

Proof. We prove this theorem by induction on the number of variables in S. Let n be the number of variables in S. We assume that our variables are ordered by some total order. We construct the following matrix. Let M be a $n \times n$ matrix, where the i-th column corresponds to the i-th variable, the j-th row corresponds to the clause from S which subsumes the full-length NPP clause, where the j-th variable occurs as a negative literal (any other variables occur as a positive literal, since it is a full-length NPP clause). Note that S must contain such a clause, because it is a Black-and-White clause set. The intersection of a row and a column is a '-' if the represented variable occur as a negative literal in the represented clause, '+' if it is a positive occurrence, and 'X' if the variable does not occur neither as a positive nor as a negative literal in the represented clause. Note that because of the construction we have that $M(i, i)$ is '-' for all $i = 1, \ldots, n$. Any other entries are either '+' or 'X', there are no more '-' entries, because NPP clauses contains exactly one negative literal, and each clause in S there must be at least one positive and one negative literals, because S is a Black-and-White SAT problem, and because of Lemma 12. To show that our theorem is true, we have to find indices $i_1, i_2, \ldots, i_z$, such that $z \leq n$, and $M(i_1,i_1)$ = '-', $M(i_1,i_2)$ = '+', $M(i_2,i_2)$ = '-', $M(i_2,i_3)$ = '+' , ,..., $M(i_z,i_z)$ = '-',

$M(i_z,i_1)$ = '+', i.e., there is a cycle $(v_{i_1}, v_{i_2}, \ldots, v_{i_z})$ in $NPPGraph(S)$, where v_i is the i-th variable in S. We show this by induction. We show that this is true for $n = 2$. Assume that the two variables are v_1 and v_2. Assume that we use the lexicographic order. Since S is a Black-and-White SAT problem, it subsumes each full-length NPP clause. Since there must be no unit in S, see Lemma 12, therefore we have that $\{\neg v_1, v2\} \in S$ and $\{v_1, \neg v2\} \in S$. So our matrix is $M(1,1)$ = '-', $M(1,2)$ = '+', M(2,1) = '+', and M(i,2) = '-', so $i_1 = 1$ and $i_2 = 2$ is a suitable choice. Note that $i_1 = 2$ and $i_2 = 1$ is also a suitable choice. Our induction hypothesis is that for $n = k$ there is a suitable choice for $i_1, i_2, \ldots, i_z$, such that $z \leq k$, and $M(i_1,i_1)$ = '-', $M(i_1,i_2)$ = '+', $M(i_2,i_2)$ = '-', $M(i_2,i_3)$ = '+', ,... $M(i_z,i_z)$ = '-', $M(i_z,i_1)$ = '+'. As the induction step, we show that it is also true for $n = k + 1$. There are two cases: (a) either there is a column which contains k pieces of 'x', (b) or there is no such column. In case (a) let us assume the the j-th column is the one which contains k pieces of 'x'. This column must contain only one not 'x' symbol, a '-' in the j-th row. Let us inactivate the j-th row and j-th column. This matrix without the inactivated row and column fulfils all requirement of our induction hypothesis, so there must be a suitable choice for $i_1, \ldots, i_{j-1}, i_{j+1}, \ldots, i_z$. In case (b) each column has at most $k-1$ pieces of 'x' symbols, so there must be at least '+' symbol in each column. Since there must be no unit in S, see Lemma 12, there are also one '+' symbol in each row. There are two cases: (ba) either their is a suitable choice for $i_1, i_2, \ldots, i_z$ in the first k rows of M, (bb) or not. Case (ba) is trivial. In case (bb) we can find an other variable ordering such that in the first k rows we have that $M(i,i)$ = '-', $M(i,i+1)$ = '+' for each $i = 1, \ldots k$. In the last row we have that $M(k+1,k+1)$ = '-', and $M(k+1,j)$ = '+', where $j < k + 1$. Here $i_1 = j, i_2 = j + 1, \ldots, i_z = k + 1$ is a suitable choice, where $z = k + 1 - j + 1$. Hence, there is a cycle in $NPPGraph(S)$. □

This theorem guaranties at least one cycle even if S is randomly generated. Our empirical investigation shows that usually we have the easiest case, i.e., we have a cycle with two nodes. We plan to use this observation in BaW 2.0, which will be our next SAT solver. We already have its prototype, but we need more time to prove the underlying theorems and test the implementation.

Unfortunately, there is no guaranty that there is any other cycle. The next lemma says that we also need to consider non-NPP clauses in order to find the graph representation of a clause set. To be more precise, the next lemma states that we can construct a Black-and-White SAT problem in such a way that its $NPPGraph$ is not strongly connected, although, our previous results suggest that a strongly connected graph should be represented by a Black-and-White SAT problem.

Lemma 13. *There exists a Black-and-White SAT problem, let us call it S, such that $NPPGraph(S)$ is not strongly connected.*

Proof. Let S be the following clause set: $\{\{\neg a, b, c\}, \{\neg b, c\}, \{b, \neg c\}, \{a, \neg b, \neg c\}\}$. It is easy to see that S is Black-and-White, because it does not subsumes the black (BB), and the

white clause (WW), but subsumes any other full-length clauses on the variables: a, b, c. Furthermore, S consists of 3 NPP clauses: $\{\neg a, b, c\}, \{\neg b, c\}, \{b, \neg c\}$, and a non-$NNP$ clause: $\{a, \neg b, \neg c\}$. $NPPGraph(S)$, by its definition, constructs the following graph: $(\{a, b, c\}, \{(a, b), (a, c), (b, c), (c, b)\})$. In this graph there is no vertex which goes to a, therefore, this graph is not strongly connected. □

The next lemma is a variant of the Clear Clause View [86] for Black-and-White problems. It states that a Black-and-White SAT problem subsumes all full-length clauses, except the "first" one, and the "last" one.

Lemma 14. *Let S be a SAT problem. Let n be the number of variables in S. Let T be a total order on the variables of S. Let us represent full-length clauses with bit arrays, where the first bit, i.e., the highest bit, corresponds to the first variable, and so on. The i-th bit is 0 if the i-th variable occurs as a negative literal in the represented full-length clause, 1 otherwise. Then S is a Black-and-White SAT problem iff it subsumes the full-length clauses which are represented from 1 till $2^n - 2$.*

Proof. We know from the definition of the black (BB) and the white clause (WW) that they are full-length clauses. Since BB contains only negative literals, it is represented by 0. Since WW contains only positive literals, it is represented by $2^n - 1$. From the definition of Black-and-White SAT problem we know that it has only two solutions: BB and WW, which are the negation of each other. Therefore, by Lemma 21 (Clear Clause Rule), a Black-and-White SAT problem subsumes any other full-length clauses. Hence, S is a Black-and-White SAT problem iff it subsumes the full-length clauses which are represented from 1 till $2^n - 2$. □

The next lemma shows an interesting property of Black-and-White SAT problems: if we combine two variables into one, then the resulting problem is Black-and-White. The motivation of this lemma is the following: We can consider a node to be a "very small" cycle to make definitions more generic, or we can consider a cycle to be a "very big" node to reduce the complexity of the graph. This lemma tells us what happens if we really do so.

We use the following functions in the next lemma:

$$ReduceXX(S,a,b) := \{C \mid C \in S \wedge a \notin C \wedge b \notin C \wedge \neg a \notin C \wedge \neg b \notin C\},$$

$$ReduceNX(S,a,b,z) := \{\{z\} \cup C \mid \{\neg a\} \cup C \in S \wedge b \notin C \wedge \neg b \notin C\},$$

$$ReduceXN(S,a,b,z) := \{\{z\} \cup C \mid \{\neg b\} \cup C \in S \wedge a \notin C \wedge \neg a \notin C\},$$

$$ReduceNN(S,a,b,z) := \{\{z\} \cup C \mid \{\neg a, \neg b\} \cup C \in S\},$$

$$ReducePX(S,a,b,z) := \{\{z\} \cup C \mid \{a\} \cup C \in S \wedge b \notin C \wedge \neg b \notin C\},$$

$$ReduceXP(S,a,b,z) := \{\{z\} \cup C \mid \{b\} \cup C \in S \wedge a \notin C \wedge \neg a \notin C\},$$

$$ReducePP(S,a,b,z) := \{\{z\} \cup C \mid \{a,b\} \cup C \in S\},$$

$$ReduceN(S,a,b,z) := ReduceNX(S,a,b,z) \cup ReduceXN(S,a,b,z) \cup ReduceNN(S,a,b,z),$$

$$ReduceP(S,a,b,z) := ReducePX(S,a,b,z) \cup ReduceXP(S,a,b,z) \cup ReducePP(S,a,b,z),$$

$$Reduce(S,a,b,z) := Reduce(S,a,b) \cup ReduceN(S,a,b,z) \cup ReduceP(S,a,b,z),$$

where z is a new variable which does not occur in S.

Note that $Reduce(S,a,b,z)$ drops all clauses which are subsumed by $\{\neg a, b\}$ or by $\{a, \neg b\}$.

Intuitively, this $Reduce(S,a,b,z)$ works as follows: $+,+ \to +$, i.e., if a occurs as a positive literal, and b occurs also as a positive literal, then z will occur as a positive literal; $+,X \to +$, i.e., b does not occur neither as a positive literal nor as a negative literal; $X,+ \to +$; $-,- \to -$; $-,X \to -$; $X,- \to -$; $X,X \to X$; and we drop the rest of the clauses.

Lemma 15. *Let S be a Black-and-White SAT problem. Let a, b two variables in S. Let z be a new variable which does not occur in S. Let n be the number of variables in S. Then $Reduce(S,a,b,z)$ is a Black-and-White SAT problem, or empty if $n = 2$.*

Proof. Let S be a Black-and-White SAT problem. Let n be the number of variables in S. From Lemma 12 we know that $n > 1$. Let T be a total order on variables of S, where a is the first variable, and b is the second one. We represent full-length clauses with bit arrays, where the first bit, i.e., the highest bit, corresponds to the first variable, and so on. The i-th bit is 0 if the i-th variable occurs as a negative literal in the represented full-length clause, 1 otherwise. In this representation, by Lemma 14, S subsumes the full-length clauses which are represented from 1 till $2^n - 2$. Note that the number of variables of $Reduce(S,a,b,z)$ is $n - 1$. Now, let us create T', a total order on variables of $Reduce(S,a,b,z)$, which is the same as T except that it does not contain a and b, and z is the first variable. It is clear that in T' we have that $Reduce(S,a,b,z)$ subsumes full-length clauses from 1 till $2^{n-1} - 2$. Hence, by Lemma 14, $Reduce(S,a,b,z)$ is a Black-and-White SAT problem, or empty if $n = 2$. □

The last theorem, i.e., Theorem 10, and the last lemma, i.e., Lemma 15, suggest that we can create an algorithm to represent a SAT problem as a directed graph. The input SAT problem may not subsume the black and the white clause. The algorithm is the following: First we use Theorem 10 to find a cycle. Then we reduce the SAT problem using the variables of the cycle, which guaranties, because of Lemma 15, a new cycle. We repeat these two steps until the reduced problem becomes empty. This algorithm works, but will not use all the information from the input SAT problem, because the reduction steps drops some clauses.

Our research stays now at this point. We have some more results, but we do not see yet where they fit into the puzzle, where the ultimate goal is to find a meaningful way to represent a SAT problem as a directed graph.

3.10 Future work

This part was purely theoretical, there is no usage described here. This does not mean that this is not possible. In this section we list some possible usage and future work.

The BaW 1.0 SAT solver solves the SAT problems generated by our strong model in linear time [96]. The proof of Theorem 3 suggests that we might find a similar algorithm for SAT problems generated by our weak model.

We are very close to generalize BaW 1.0, called BaW 2.0, for the Balatonboglár model. The main idea is to propagate two units in the first run to earn units. This means that we have 4 branches in the highest level. We have found a way which guaranties no decisions in the first 3 branches, but we still do not know what to do in the last branch.

It is an interesting question how to represent a 3-SAT problem as a directed graph. Most probably it is not feasible, because then we could translate a 3-SAT problem into a directed graph, and then that directed graph into a 2-SAT problem. Although this direction seems to be unrealistic, it is still interesting and very challenging for us.

In this direction, one of the problems is what to do with those clauses in a 3-SAT problem, which subsumes the white clause, or the black one. To solve this problem we need a technique which creates a Black-and-White SAT problem, if the input SAT problem is unsatisfiable. We give some possible answers to this question in of our newest work, see [95].

The examples in (3.6.1), (3.6.2), and (3.6.3) suggest a question: Can we define alternatively $TagForward$, which is a part of the Balatonboglár model, in a following way?

$$TagForward(a,b) := \{\neg a, \neg b, c\}, where\ a \in \mathcal{V} \wedge b \in OutE(a) \wedge c \in OutE(b) \wedge a \neq c.$$

$$TagForward := \{\ TagForward(a,b) |\ a \in \mathcal{V} \wedge b \in OutE(a)\}.$$

This means that if there are more than one ways of tagging forward, then is it enough to use one of them? We do not know.

4 Properties of WnDGen

In this part we introduce a very simple algorithm, called WnDGen [99], which generates weakly nondecisive clause sets [86, 90, 99]. Its input is a clause and a number, its output is a clause set. WnDGen(C, k) works as follows: It generates all k-length subsets of C. For each subset it adds k clauses to the output, negating every time another literal in the subset. Then it does the same with the negation of C. We show that the resulting clause set is always weakly nondecisive and satisfiable. Actually, C and negation of C are solutions of the SAT instance generated by WnDGen(C, k). We also show that if $n \geq 2k-3$, then WnDGen(C, k) together with C and negation of C is unsatisfiable, where n is the length of C. If $n \geq 2k-3$, then WnDGen(BB, k) and WnDGen(WW, k) are Black-and-White SAT instances, where WW is the white clause, and BB is the black clause, and n is their length. We have generated several clause sets by WnDGen.

This part of the dissertation is based on our following papers: [90, 99].

4.1 Introduction

In this part we study the notion of weakly nondecisive literal. It is a generalization of blocked literal but a specialization of nondecisive literal. The notion of blocked literal and clause was introduced by O. Kullman in [52, 53]. A.V. Gelder generalized this idea and introduced the notion of nondecisive literal and clause in [39]. The definitions of these notions are given in the next section. Here we give a general overview.

O. Kullmann showed that a blocked clause can be removed or added without changing the satisfiability of the clause set. He studied the blocked clauses, because he wanted to have a better worst-case upper bound for solving the SAT problem.

A.V. Gelder generalized further this idea and introduced the notion of nondecisive clause. He showed that a nondecisive clause can be removed or added without changing the satisfiability of the clause set.

I showed in [91] that blocked clause sets (a clause set is blocked if all its clauses are blocked) are satisfiable. The same is note true for nondecisive clause sets. I wanted to find between these two notions a new one which still ensures satisfiablity. Therefore, I introduced the notion of weakly nondecisive clause set in my thesis [86]. I showed that not all weakly nondecisive clause sets are satisfiable. In my thesis there was an open question: How to generate weakly nondecisive clause sets? We answered this question in [99]. This part of the dissertation recalls the theoretical results from that work and some definitions

from [90].

4.2 Overview

To answer that open question we introduced WnDGen1(C, k) and WnDGen(C, k).

WnDGen1(C, k) works as follows (C is a clause, k is a number): It generates all k-length subsets of C. For each subset it adds k clauses to the output, negating every time another literal in the subset. WnDGen is the union of WnDGen1 on C and WnDGen1 on the negation of C.

We see that these functions are very simple ones, but it was very difficult to find them. First we created a function which tests whether a clause set is weakly nondecisive or not. Then we generated lots of random clause sets and we were lucky to find some weakly nondecisive ones. We did not do this blindly, because we already had a weakly nondecisive clause set from [86]. Then we started to analyze the structure of the new ones. Finally, we have found these two functions after several months of work. There should exists some more ways to generate weakly nondecisive clause sets, because these two functions are unable to generate the example from [86].

We have started to work with the generated clause sets and it turned out that they have very interesting properties, see Section 4.4.

4.3 Definitions

In this section we give the definitions. We do not repeat the basic definitions which are already defined in Section 3.3.

Let V be a finite *set of boolean variables*. *Variables of a set* C is denoted by $\mathbf{V}ar(C)$ and defined as $\mathbf{V}ar(C) := (C \cup \neg C) \cap V$.

The *length* of a set U is its cardinality, denoted by $|U|$. Let $n := |V|$. Clause C is a full-length clause iff its length is n.

C is *subsumed by* S, denoted by $C \supseteq\in S$, iff

$$C \supseteq\in S :\iff \exists[B \in S]B \subseteq C.$$

Note that if a clause set subsumes all full-length clauses, then it is unsatisfiable.

The *clause difference* of C and D, denoted by $diff(C, D)$, is defined as $diff(C, D) := C \cap \neg D$. If $diff(C, D) \neq \emptyset$ then we say that C *differs from* D.

Resolution can be performed on two clauses iff they differ only in one variable. If resolution can be performed then the *resolvent*, denoted by $Res(C, D)$, is defined as $Res(C,D) := (C \cup D) \setminus (diff(C,D) \cup diff(D,C))$.

We define the notion of *blocked, nondecisive* and *weakly nondecisive* literal, denoted by

$Blck(c, C, S)$, $NonD(c, C, S)$, $WnD(c, C, S)$, respectively.

$$\begin{aligned} Blck(c, C, S) :\Longleftrightarrow\ & \forall[B \in S][\neg c \in B] \\ & \exists[b \in B][b \neq \neg c]\neg b \in C \end{aligned} \tag{4.3.1}$$

$$\begin{aligned} NonD(c, C, S) :\Longleftrightarrow\ & \forall[B \in S] \\ & [\neg c \in B](\exists[b \in B][b \neq \neg c]\neg b \in C \vee \\ & Res(C, B) \cup \{c\} \supseteq \in S \setminus \{C\}) \end{aligned} \tag{4.3.2}$$

$$\begin{aligned} WnD(c, C, S) :\Longleftrightarrow\ & \forall[B \in S][\neg c \in B] \\ & (\exists[b \in B][b \neq \neg c]\neg b \in C \vee Res(C, B) \supseteq \in S) \end{aligned} \tag{4.3.3}$$

In other words, a literal is weakly nondecisive in C and S iff it is blocked or any resolvent of C and a non-blocking B is subsumed by S.

C is a *weakly nondecisive clause* in S iff it has a weakly nondecisive literal. S is a *weakly nondecisive clause set* iff its clauses are weakly nondecisive.

In this part of the dissertation we generate weakly nondecisive clause sets. The first generator function is *WnDGen1*. If C is a clause, k is a natural number, and $2 \leq k \leq |C|$, then

$$WnDGen1(C,k) := \{D \mid \mathbf{V}ar(D) \subseteq \mathbf{V}ar(C) \wedge |D| = k \wedge |diff(D,C)| = 1\}. \tag{4.3.4}$$

This means that WnDGen1(C, k) generates all k-clauses which differ only in one literal from C. C is called the generator clause.

We give two examples for WnDGen1:

$$\begin{aligned} WnDGen1(\{a, b, c\}, 2) = \{&\{a,\neg b\}, \\ &\{\neg a,b\},\{a,\neg c\},\{\neg a,c\},\{b,\neg c\},\{\neg b,c\}\}. \end{aligned} \tag{4.3.5}$$

$$WnDGen1(\{a, b, c\}, 3) = \{\{a,b,\neg c\}, \{a,\neg b,c\},\{\neg a,b,c\}\}. \tag{4.3.6}$$

The second generator function is *WnDGen*. If C is a clause, k is a natural number, and $2 \leq k \leq |C|$, then

$$WnDGen(C, k) := WnDGen1(C,k) \cup WnDGen1(\neg C, k). \tag{4.3.7}$$

4.4 Properties of WnDGen1 and WnDGen

In this section we prove that WnDGen1 and WnDGen generate weakly nondecisive clause sets. We give a criterion on the shape of clauses which are subsumed by WnDGen1. Based on this result we show how to use WnDGen to generate unsatisfiable clause sets. First we give the shape criterion.

Lemma 16 (Shape Criterion)**.** *Assume C and D are clauses and $\boldsymbol{Var}(D) \subseteq \boldsymbol{Var}(C)$. Assume k is a natural number and $2 \leq k \leq |C|$. If $|diff(D,C)| \geq 1$ and $|D| \geq |diff(D,C)| + k - 1$, then D is subsumed by WnDGen1(C, k).*

Proof. Without loss of generality we may assume that C contains only positive literals. Then WnDGen1(C, k) contains all clauses with $k-1$ positive literals and 1 negative one. Since $|diff(D,C)|$ is the number of negative literals in D and $|D| - |diff(D,C)|$ is the number of positive literals in D we have that D is subsumed by WnDGen1(C, k) if $|diff(D,C)| \geq 1$ and $|D| - |diff(D,C)| \geq k - 1$. □

Note that there is a special case. If $|diff(D,C)| = 1$ then it is enough to show that $|D| \geq k$ to prove that D is subsumed by WnDGen1(C, k). We will use this case in Lemma 17. We use another case in Lemma 18 when $|diff(D,C)| = k - 1$. In this case it is enough to show that $|D| \geq 2k - 2$.

Lemma 17. *Assume C is a clause, k is a natural number, and $2 \leq k \leq |C|$. Let S = WnDGen1(C, k). Then S is a weakly nondecisive clause set.*

Proof. Let S = WnDGen1(C, k). The main idea of the proof is that for any A in S (this means that $|A| = k$ and $|diff(A,C)| = 1$) there is a literal a, which occurs positively (negatively) in A but negatively (positively) in C and this literal is a weakly nondecisive literal in A and S. To show this, we show that for any B in S in which a occurs negatively (positively) we have that either $Res(A,B)$ is not a clause, i.e., B blocks A; or $Res(A,B)$ is a clause (this means that $|diff(A,B)| = 1$), but in this case, by Lemma 16, $Res(A,B)$ is subsumed by S, because $|Res(A,B)| \geq k$ (in general we have $|Res(A,B)| \geq k - 1$, but $|Res(A,B)| = k - 1$ would contradict the assumption $|diff(A,C)| = 1$) and $|diff(Res(A,B),C)| = 1$ (which comes from that $diff(A,C) = \{a\}$, $|diff(B,C)| = 1$, and $a \notin Res(A,B)$). Hence, S is weakly nondecisive. □

Actually, we could state a more powerful statement that in WnDGen1(C, k) all literals are weakly nondecisive, see Lemma 19. But this weaker lemma helps us to prove that WnDGen(C,k) is also weakly nondecisive.

Lemma 18. *Assume C is a clause, k is a natural number, and $2 \leq k \leq |C|$. Let S = WnDGen(C, k). Then S is a weakly nondecisive clause set.*

Proof. Let S = WnDGen(C, k). We show that S is weakly nondecisive. To show this we show that any A in S is weakly nondecisive. Without loss of generality we may assume that A is an element of WnDGen1(C, k), this means that $|A| = k$ and $|diff(A,C)| = 1$. The main idea of the proof is that there is a literal a, which occurs positively (negatively) in A, but negatively (positively) in C and this literal is a weakly nondecisive literal in A. To show this, we show that for any B in S we have that either B blocks A or $Res(A,B)$ is subsumed by S. We know from Lemma 17 that this is true if B is an element of WnDGen1(C, k). If B is in WnDGen1($\neg C$, k) then we have $|diff(B,\neg C)| = 1$, i.e., $|diff(B,C)| = k - 1$ and

we also know that $|diff(A,C)| = 1$ which means that there is only one case when B does not block A, when $|\mathbf{V}ar(A) \cap \mathbf{V}ar(B)| = 1$, but in this case, by Lemma 16 $Res(A,B)$ is subsumed by S, because $|Res(A,B)| = 2k-2$ and $|diff(Res(A,B), C)| = k-1$. Hence, S is weakly nondecisive. □

Now we prove that in WnDGen1(C, k) all literals are weakly nondecisive.

Lemma 19. *Assume C is a clause, k is a natural number, and $2 \leq k \leq |C|$. Let S = WnDGen1(C, k). Then all literals in S are weakly nondecisive.*

Proof. Without loss of generality we may assume that C contains only positive literals. From definition of WnDGen1 we know that in each clause in S there is only one negative literal, this is the literal in which each clause differs from C. From the proof of Lemma 17 we know that these literals are weakly nondecisive. This means that all negative literals in S are weakly nondecisive. From this we can obtain that all positive literals are also weakly nondecisive, because if a positive literal is involved in a resolution, there must be involved also a negative one, and the resolvent will be subsumed because the negative literal is weakly nondecisive. Hence, all literals in S are weakly nondecisive. □

We proof now that $WnDGen(C, k)$ is always satisfiable, which means naturally that $WnDGen1(C, k)$ is always satisfiable.

Lemma 20. *Assume C is a clause, k is natural number, and $2 \leq k \leq |C|$. Then $WnDGen(C, k)$ is satisfiable.*

Proof. It is easy to see that for all $D \in WnDGen(C, k)$ we have that $D \cap C \neq \emptyset$ and $D \cap \neg C \neq \emptyset$. This means that C and $\neg C$ are models for $WnDGen(C, k)$. Hence, $WnDGen(C, k)$ is satisfiable. □

We have seen that WnDGen(C, k) is satisfiable. Now we show that there is a sharp threshold $(|C| + 3)/2$ such that if k is greater than this threshold, then $WnDGen(C, k) \cup \{C, \neg C\}$ is satisfiable, otherwise unsatisfiable.

Theorem 11 (Sharp Threshold of WnDGen)**.** *Assume C is a clause, k is natural number, and $2 \leq k \leq |C|$. Then $WnDGen(C, k) \cup \{C, \neg C\}$ is unsatisfiable iff $|C| \geq 2k - 3$.*

Proof. Let $V = \mathbf{V}ar(C)$, i.e., $n = |C|$. Without loss of generality we may assume that C contains only positive literals.
(From left to right:) We assume $WnDGen(C, k) \cup \{C, \neg C\}$ is unsatisfiable. We show $n \geq 2k - 3$. From the assumption we obtain that $WnDGen(C, k) \cup \{C, \neg C\}$ subsumes all full-length clauses. This means that $WnDGen(C, k)$ subsumes all full-length clauses with 1, 2, ..., or $n-1$ negative literals. Note that for an arbitrary clause D the number $|diff(D,C)|$ is the number of negative literals in D since C contains only positive literals. From Lemma 16 we can obtain that WnDGen1(C, k) subsumes those full-length clauses which contain $1, 2, \ldots, n-k$, or $n-k+1$ negative literals. We can also obtain that WnDGen1($\neg C$, k)

subsumes those full-length clauses which contain $n-1, n-2, \ldots, n-(n-k)$, or $n-(n-k+1)$ negative literals. This means that both WnDGen1(C, k) and WnDGen1($\neg C$, k) subsumes $n-k+1$ cases from the sequence $1, 2, \ldots,$ or $n-1$, but start from different ends. From our assumption we know that there is no such case which is not subsumed from this sequence. This means that the number of subsumed cases must be greater or equal than the number of cases in the sequence, i.e., $2 * (n - k + 1) \geq n - 1$. After simplification we obtain $n \geq 2k - 3$. Hence, $|C| \geq 2k - 3$.
(From right to left:) We assume that $n \geq 2k-3$. We show that $WnDGen(C, k) \cup \{C, \neg C\}$ is unsatisfiable. To show this, it is enough to show that $WnDGen(C, k) \cup \{C, \neg C\}$ subsumes all full-length clauses. Any full-length clause has either $0, 1, \ldots, n-1$, or n negative literals. C subsumes those full-length clauses which have 0 negative literals (actually there is only one such clause: C). Furthermore, $\neg C$ subsumes those full-length clauses which have n negative literals (because it subsumes itself). Therefore, we have to show that $WnDGen(C, k)$ subsumes all full-length clauses with 1, 2, ..., $n-2$, or $n-1$ negative literals. Let D be a full-length clause such that $D \neq C$ and $D \neq \neg C$. This means that $|diff(D, C)| \geq 1$, and, by definition of full-length clause, $|D| = n$. From Lemma 16 we know that D is subsumed by WnDGen1(C, k) if $|D| \geq |diff(D, C)| + k - 1$, i.e., $n - k + 1 \geq |diff(D, C)|$. We show that if D is not subsumed by WnDGen1(C, k), then it is subsumed by subsumed by WnDGen1($\neg C$, k). Note that the number $|diff(D,C)|$ is the number of negative literals in D since C contains only positive literals. This means that WnDGen1(C, k) subsumes those full-length clauses which contain $1, 2, \ldots, n-k$, or $n-k+1$ negative literals. From this, by definition of WnDGen, we can obtain that we have to show that WnDGen1($\neg C$, k) subsumes those full-length clauses which contain $n-k+2, n-k+3, \ldots, n-2$, or $n-1$ negative literals; on the other way around, we have to show that WnDGen1($\neg C$, k) subsumes those full-length clauses which contain $n-(n-k+2), n-(n-k+3), \ldots, n-(n-2)$, or $n-(n-1)$ positive literals, i.e., $k-2, k-3, \ldots, 2$, or 1 ones. So there are $k-2$ such cases. From Lemma 16 we know that WnDGen1($\neg C$, k) subsumes those full-length clauses which contain $1, 2, \ldots, n-k$, or $n-k+1$ positive literals. So there are $n-k+1$ such cases. This means that we have to show that the number of cases which are subsumed by WnDGen1($\neg C$, k) is greater or equal than the number of cases which remain un-subsumed by WnDGen1(C, k), i.e., $n-k+1 \geq k-2$. We already know that $n \geq 2k-3$, if we add $-k+1$ to both sides, then we obtain $n-k+1 \geq k-2$. This means that D is subsumed by WnDGen1(C, k) or by WnDGen1($\neg C$, k), i.e., $WnDGen(C, k)$ subsumes all full-length clauses with 1, 2, ..., $n-2$, or n-1 negative literal. Hence, $WnDGen(C, k) \cup \{C, \neg C\}$ is unsatisfiable. □

This is our main theorem. This theorem was a kind of surprise for us, because we had a feeling that WnDGen with the generator clause and with the negation of the generator clause should be always unsatisfiable. It turned out that sometimes this structure is satisfiable. This theorem gives as a sharp threshold, $2k-3$, such that if the generator

clause is shorter them this threshold, then this structure is satisfiable, otherwise not. It is a common experience that around a threshold there are interesting things, see Section 4.6.

4.5 WnDGen and the Black-and-White SAT Problem

We recall from Section 3.3 that BB is the black clause, and WW is the white clause. The black clause is the full-length clause which contains only negative literals, and the white clause is the full-length clause which contains only positive literals.

Now we can prove a connection between the notions of WnDGen and Black-and-White SAT problem.

Theorem 12. *$WnDGen(BB, k)$ and $WnDGen(WW, k)$ are Black-and-White SAT problems iff $|BB| \geq 2k - 3$.*

Proof. From Theorem 11 we know that $WnDGen(BB, k)$ is a SAT problem, and has only two solutions BB and WW (since $\neg BB = WW$) iff $|BB| \geq 2k - 3$. From this, by definition of Black-and-White SAT problem, we obtain that $WnDGen(BB, k)$ is a Black-and-White SAT problems iff $|BB| \geq 2k - 3$. In the same way we can prove that $WnDGen(WW, k)$ is a Black-and-White SAT problems iff $|WW| \geq 2k - 3$. Note that by definition $|BB| = |WW|$. □

Actually, $WnDGen(WW, 2)$ creates the strong model of the complete communication graph with n nodes, where $n = |WW|$.

For example, in case of $n = 5$ we have that $WnDGen(WW, 2)$ equals to (without redundant elements) to this clause set:

```
{{-1, 2},
{1, -2},
{-1, 3},
{1, -3},
{-1, 4},
{1, -4},
{-1, 5},
{1, -5},
{-2, 3},
{2, -3},
{-2, 4},
{2, -4},
{-2, 5},
{2, -5},
{-3, 4},
{3, -4},
```

```
{-3, 5},
{3, -5},
{-4, 5},
{4, -5}}
```

This represents the communication graph depicted on Figure 4.1.

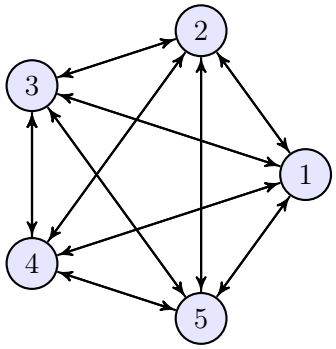

Figure 4.1: The communication graph represented by WnDGen($\{1, 2, 3, 4, 5\}$, 2), and also by WnDGen($\{1, 2, 3, 4, 5\}$, 3).

Interestingly enough, $WnDGen(WW, 3)$ creates a Balatonboglár model of the complete communication graph with n nodes, where $n = |WW|$.

For example, in case of $n = 5$ we have that $WnDGen(WW, 3)$ equals to (without redundant elements) to this clause set:

```
{{-1, 2, 3},
{1, -2, 3},
{1, 2, -3},
{1, -2, -3},
{-1, 2, -3},
{-1, -2, 3},
{-1, 2, 4},
{1, -2, 4},
{1, 2, -4},
{1, -2, -4},
{-1, 2, -4},
{-1, -2, 4},
{-1, 2, 5},
{1, -2, 5},
{1, 2, -5},
{1, -2, -5},
{-1, 2, -5},
{-1, -2, 5},
{-1, 3, 4},
```

```
{1, -3, 4},
{1, 3, -4},
{1, -3, -4},
{-1, 3, -4},
{-1, -3, 4},
{-1, 3, 5},
{1, -3, 5},
{1, 3, -5},
{1, -3, -5},
{-1, 3, -5},
{-1, -3, 5},
{-1, 4, 5},
{1, -4, 5},
{1, 4, -5},
{1, -4, -5},
{-1, 4, -5},
{-1, -4, 5},
{-2, 3, 4},
{2, -3, 4},
{2, 3, -4},
{2, -3, -4},
{-2, 3, -4},
{-2, -3, 4},
{-2, 3, 5},
{2, -3, 5},
{2, 3, -5},
{2, -3, -5},
{-2, 3, -5},
{-2, -3, 5},
{-2, 4, 5},
{2, -4, 5},
{2, 4, -5},
{2, -4, -5},
{-2, 4, -5},
{-2, -4, 5},
{-3, 4, 5},
{3, -4, 5},
{3, 4, -5},
{3, -4, -5},
{-3, 4, -5},
```

```
{-3, -4, 5}}
```

This represents the same communication graph as the previous example, so the represented communication graph is depicted on Figure 4.1.

4.6 Test results

It turned out that on the threshold, i.e., if $|C| = 2k - 3$, the generated SAT instances are hard to solve for SAT solvers. We developed a so called *WnDGen* tool, which can generate *WnD* SAT instances with the *WnDGen* algorithm. One can download this tool from *http://fmv.ektf.hu/*. The tool has a switch, $-unsat$, which puts two extra clauses at the end of the generated file, the generator clause and its negation, i.e., it creates the structure used in Theorem 11. This case is called *unsat* in the rest of this part. By *sat* we mean those SAT instances which where generated without the $-unsat$ switch. In the rest of this part let $n = |C|$, i.e., the length of the generator clause.

Usage of the WnDGen tool: `java WnDGen switches N K`. Generates a *WnD* clause set with N variables. Each clause has K literals. If K is not given, then each clause has 3 literals.
Switches:

- *-sat*: Generates a satisfiable clause set. It is the default switch.
- *-unsat*: Tries to generate an unsatisfiable clause set, which is *WnD* upto the two last clauses.
- $-help$: Prints how to use *WnDGen*.

Our tests were done on Intel quadcore machine with 6 GB of RAM running at 2.66 Ghz. For testing, we used the MiniSat[1] 2.2.0. The MiniSat is a modern CDCL solver, has four major features of these: - Conflict-driven clause learning [65], [66], - Random search restarts [40], - Boolean constraint propagation using lazy data structures [68], - Conflict-based adaptive branching [68]. For each instance, we used a timeout of 1 hour. The result are presented in Tab. 4.1 and Tab. 4.2. The first column presents the WnDGen instances. In the second column, shows number of restarts. The next four columns gives information for number of conflicts, decisions, propagations and conflict literals. Finally, the last column, shows CPU times.

4.7 Conclusions

We have showed that around this threshold there are SAT instances, which are difficult for a well-known SAT solver, i.e., they are good for testing SAT solvers. We have introduced

[1] http://www.minisat.se

WnDGen	restarts	conflicts	decisions	propagations	conflict literals	CPU time (s)
n=5 k=4	1	1	4	7	3	0.00
n=7 k=5	1	9	12	30	30	0.00
n=9 k=6	1	31	34	90	124	0.00
n=11 k=7	2	105	111	304	524	0.01
n=13 k=8	3	283	297	796	1812	0.04
n=15 k=9	6	1502	1533	4320	11028	13.64
n=17 k=10	9	6335	6406	18510	52287	9.61
n=19 k=11	12	24960	25089	72661	229711	157.14
n=21 k=12	15	78298	78472	222964	812344	2779.43

Table 4.1: Runtimes and results (all instances is *sat*)

two SAT instance generators, $WnDGen1$ and $WnDGen$, which generate highly structured (hard combinatorical) SAT instances, which have very interesting properties.

We showed that if the generator clause is either the black or the white clause, and we are not under the threshold, then $WnDGen$ generates a Black-and-White SAT problem. This suggests that we might be able to use either the Shape Criterion Lemma, or the Sharp Threshold for WnDGen Theorem to create better models for directed graphs.

WnDGen	restarts	conflicts	decisions	propagations	conflict literals	CPU time (s)
n=5 k=4	1	8	7	18	12	0.00
n=7 k=5	1	22	21	54	49	0.00
n=9 k=6	1	72	71	192	228	0.00
n=11 k=7	3	256	257	705	1056	0.01
n=13 k=8	5	935	945	2651	4805	0.11
n=15 k=9	8	3481	3514	10148	21544	1.92
n=17 k=10	11	12960	13029	37004	92650	30.45
n=19 k=11	14	49168	49296	140618	399670	448.03
n=21 k=12	TIME OUT					

Table 4.2: Runtimes and results (all instances is *unsat*)

5 The CSFLOC SAT Solver

In this chapter we introduce the `CSFLOC` algorithm [97] which solves the SAT problem by counting full-length clauses. It is the successor of the `Optimized CCC` algorithm [98]. By studying `Optimized CCC` we observed that its full-length clause counter can be increased on its last 1 bit in the best case. As a main contribution we prove the lemma that this observation is generally true for `Optimized CCC`. The new algorithm, `CSFLOC`, uses this lemma. It uses also a data structure in which the clauses are ordered by the index of their last literal. These two improvements result in a faster algorithm which can compete with a state-of-the-art SAT solver on problems like Black-and-White SAT problems, see Chapter 3, and weakly nondecisive SAT problems, see Chapter 4.

This part of the dissertation is based on our following papers: [97, 98].

5.1 Introduction

An interesting question is how many solutions does a SAT problem instance have. This problem is known as the #SAT problem. In other words, by #SAT we mean the problem of counting the solutions of a SAT instance [41]. The most popular way to solve this problem is to use a variant of the DPLL method [23] which does not stop if it finds a solution but keep exploring the whole search space. One of the first examples is CDP [13].

Another way to count the solutions is to use the inclusion-exclusion principle. It is well known that this principle can solve the #SAT problem [45, 61, 9].

Let $A_1, A_2, \ldots, A_m$ be sets, where $m > 0$. The inclusion-exclusion principle states that $|\bigcup_{i=1}^{m} A_i| = \sum_{i=1}^{m} |A_i| - \sum_{1 \leq i < j \leq m} |A_i \cap A_j| + \ldots (-1)^{m+1} |A_1 \cap A_2 \cap \cdots \cap A_m|$.

We recall that clause C subsumes clause D if and only if C is a subset of D. To be able to use the inclusion-exclusion principle on SAT we need the so called Clear Clause View [86]. In this view we have to see the clauses as sets of the subsumed full-length clauses. For example if we have 4 variables, a, b, c, d, then the clause $\{a, b\}$ should be seen as the set of subsumed full-length clauses: $\{\{a, b, \neg c, \neg d\}, \{a, b, \neg c, d\}, \{a, b, c, \neg d\}, \{a, b, c, d\}\}$. Now, let $C_1, \ldots, C_m$ be the clauses of a SAT problem instance. Let A_i be the set of full-length clauses which are subsumed by C_i, $i = 1 \ldots m$. Then $|A_i|$ means the number of full-length clauses subsumed by C_i, and $|A_i \cap A_j|$ means the number of full-length clauses subsumed by both C_i and C_j. Note that $|A_i| = 2^{n-k}$, where n is the number of variables of the clause set, and k is the number of literals in C_i. Also note that the number of solution of a clause set is 2^n minus the number of subsumed full-length clauses, see [9].

Instead of using the inclusion-exclusion principle one can use our algorithm, the Counting Subsumed Full-Length Ordered Clauses algorithm, for short CSFLOC. It counts the subsumed full-length clauses, as the inclusion-exclusion principle, but in an iterative way. It is easy to see that there are 2^n different full-length clauses. A SAT problem is unsatisfiable if it subsumes all full-length clauses. If a SAT problem subsumes fewer, then it is satisfiable. This means that by counting the subsumed full-length clauses we can decide satisfiability.

In a previous chapter, see Chapter 3, we introduced how to translate a directed graph into a SAT problem. The motivation was to check whether the communication graph of a wireless sensor network (WSN) is strongly connected or not. We showed that if the graph is strongly connected, then the generated SAT instance is a Black-and-White SAT problem, which can be also generated by WnDGen, see Chapter 4. In this chapter we introduce a SAT solver algorithm, called CSFLOC, which is very good at solving Black-and-White SAT problems and WnD problems generated by the WnDGen SAT problem generator.

The CSFLOC algorithm is the successor of the Optimized CCC algorithm, which counts those full-length clauses which are subsumed by the input clause set. To do this, it uses a counter. By studying Optimized CCC we have observed that in the best case it increases the counter on its last 1 bit. For example, if the counter is 24, which is 16+8, then the biggest possible increment is 8.

As a main contribution we prove the lemma that this observation is generally true for Optimized CCC. The new algorithm, CSFLOC, uses this lemma. It uses also a data structure in which the clauses are ordered by the index of their last literal.

These two improvements result in a faster algorithm which is faster than a state-of-the-art SAT solver on problems with lots of clauses but few variables, like WnD problems. Otherwise it is slower, but it uses no fine-tuned heuristics, and it is built on different ideas then a state-of-the-art SAT solver, so it is still interesting.

Our experience shows that over-constrained problems with less than 100 variables, like WnD problems, are easy for CSFLOC. Our experience shows also that if we cluster the variables of a SAT problems then CSFLOC runs faster. Since CSFLOC is good at solving WnD problems, it might be a useful tool to check whether the communication graph of a WSN is strongly connected or not.

We describe first the CCC algorithm, then Optimized CCC, finally we describe the new idea in CSFLOC.

CCC works as follows: It sets its counter to be 0. It converts 0 to be a full-length clause, called C, which is the one with only negative literals. In general, C is created from the counter in the following way: Each bit of the counter is represented by a literal, the literal is positive if the corresponding bit is 1, negative otherwise. It checks whether this full-length clause is subsumed by the input problem. This means that it looks for a clause D from the input SAT problem that is a subset of C. If such a clause D is found, then it adds 1 to the counter and repeats the process until it finds a full-length clause which is not subsumed, which means that the input is satisfiable, or all possible full-length clauses

are visited, i.e., the input is unsatisfiable.

It is easy to see that this algorithm stops for any SAT problem and can decide whether the SAT instance is satisfiable or not. This means that this simple algorithm is sound and complete, but in case of an unsatisfiable SAT instance it visits all the 2^n full-length clauses.

The next version is called `Optimized CCC`, which uses the observation that a clause subsumes 2^{n-i} consecutive ordered full-length clauses, where i is the index of its last literal. This means that if we find a clause D which subsumes C, then we can increase the counter by 2^{n-i} instead of 1. So we do not need to visit all full-length clause to decide satisfiability. `Optimized CCC` always selects that D in the loop which grants the biggest step, i.e., where the number 2^{n-i} is the greatest. This optimization does not affect the soundness and the completeness of the algorithm.

The next version of this algorithm family is `CSFLOC`. Here we use the observation that if we always use the best D in the loop, as in `Optimized CCC`, then the biggest possible step is 2^{n-j}, where j is the index of the last 1 bit in the counter. Note that this bit always comes from the previous run of the loop. It is so, because if there were an even better D in the input clause set, then the algorithm would select that D even in the previous loop and setting a bit to 1 which has smaller index than j. This means that we do not have to scan the whole input clause set to find the best D, we have to consider only those clauses where the index of the last literal is greater than or equal to j. This optimization does not affect the soundness and the completeness of the algorithm.

Counting full-length clauses is not a new idea. Already Iwama used this idea in 1989 [45]. Later Lozinskii introduced an approximation algorithm for counting solutions [61]. They count full-length clauses following the inclusion-exclusion principle more closely. This means that they also decrement the counter, see Algorithm 3 in [9]. In case of the `CCC` algorithm family we always increment the counter. So we can use the above observation about the biggest possible step, while the classical algorithms cannot do that.

Full-length clauses are called also maximal clauses in the literature. A good overview of these papers can be found in [5]. In this paper Andrei introduce an inverse resolution to generate maximal clauses. He also overviews the field of maximal clause counting. We could not access the old papers, like the one written by Tanaka [79] and by Dubois [27]. From Andrei's overview it seems that Tanaka's algorithm is similar to `Optimized CCC`. We could not find a similar one to `CSFLOC`.

The test results show that `CSFLOC` is always faster than `Optimized CCC` and it can compete with a state-of-the-art SAT solver on problems like Black-and-White SAT problems and weakly nondecisive SAT problems.

5.2 Definitions

In this section we give the definitions. We do not repeat the basic definitions which are already defined in Section 3.3 or in Section 4.3.

We say that the injective function I from variables to the natural numbers $1 \dots n$ is a *variable index function*, where n is the number of variables. We can also define I on literals such that $I(lit) = I(\neg lit)$, where lit is a literal. If otherwise is not stated I is the lexicographical ordering of variables.

If we use the same variable index function on the clauses of a clause set, then we call them *ordered clauses*. In an ordered clause we can select the *first literal*, which is the one with the smallest index, and the *last literal*, which is the one with the greatest index, in the ordered clause.

We also define two functions: IndexOfLastLiteral$(C) := \max(\{I(lit) \mid lit \in C\})$, and IndexOfLastPositiveLiteral$(C) := \max(\{I(lit) \mid lit \in C \wedge lit \; is \; a \; positive \; literal\} \cup \{1\})$.

We also define how to convert a bit array into a clause:
FullLengthClauseRepresentation$(bits) := \{I^{-1}(i) \mid i \in \{1 \dots n\} \wedge bits_i = 1\} \cup \{\neg I^{-1}(i) | i \in \{1 \dots n\} \wedge bits_i = 0\}$, where $bits_i$ is the i-th bit in the integer $bits$, and I^{-1} is the inverse of I.

5.3 Theory

The `CCC` algorithm is based on the following lemmas.

Lemma 21. *[Clear Clause Rule, see Lemma 4.3.1. in [86]] Let S be a clause set and C be a full-length clause. Then*

$$(a) \;\; S \; subsumes \; C \Leftrightarrow \neg(\neg C \; is \; a \; solution \; for \; S);$$

$$(b) \;\; \neg(S \; subsumes \; C) \Leftrightarrow \neg C \; is \; a \; solution \; for \; S.$$

The proof of this lemma can be found in [86]. This lemma states that if S subsumes a full-length clause, say C, then its negation is not a solution. But if S does not subsume C, then its negation, $\neg C$, is a solution of S. Hence, it is enough to find a not subsumed full-length clause to find a solution of S.

The next lemma is a trivial consequence of the previous one.

Lemma 22. *Let S be a clause set with n variables. Then S has m different full-length solutions iff S subsumes $2^n - m$ different full-length clauses.*

This lemma has two important corollaries.

Corollary 3. *(a) Let S be a clause set with n variables. Then S is satisfiable iff S subsumes less than 2^n full-length clauses; and S is unsatisfiable iff S subsumes 2^n full-length clauses. (b) Let C be a k-clause. Then C subsumes 2^{n-k} full-length clauses.*

We use the following representation: In a full-length clause all variables are present either as a positive literal, or as a negative one. This means that we can represent them by an n bit wide unsigned integer. If it is an ordered full-length clause then we can use the following representation: bit value 0 corresponds to negative literal, 1 to a positive one on the same index. We give an example:

Example 1. *The clause* $\{a, \neg b, c, d, \neg e\}$ *is represented by* `10110`.

With this representation we can order full-length clauses.

Lemma 23. *Let n be the number of boolean variables in use. Let I be a variable index function of these variables. Let D be a non-empty clause. Let i be the index of the last literal of D, i.e., let i =IndexOfLastLiteral(D). Then D subsumes 2^{n-i} consecutive full-length clauses.*

Proof. We know that D does not contain any literal with index greater than i. Let $\mathbf{C}$ be the set of full-length clauses which are subsumed by D. Let C_0 be the element of $\mathbf{C}$ which has the smallest integer representation among elements of $\mathbf{C}$. We show that D subsumes C_0 and the next $2^{n-i} - 1$ full-length clauses. It is easy to see that the last $n - i$ literals of C_0 are negative ones. The next full-length clause is the same as C_0 except that its last literal is positive. Let us denote this clause, i.e., the next one according to its integer representation, by C_1. D subsumes C_1 unless $i = n$, i.e., unless the case that the last variable is present in D. We can create C_2 by adding 1 to the integer representation of C_1 and translate it back to a full-length clause. In this way we can construct the consecutive full-length clauses. We can see that D subsumes each full-length clause in this sequence until the i-th literal is not affected, i.e., it subsumes $C_0, C_1, \ldots, C_{2^{n-i}-1}$, but not $C_{2^{n-i}}$. Hence, D subsumes 2^{n-i} consecutive full-length clauses. □

To visualize the above lemma, we give an example.

Example 2. *Let us assume that we have* 6 *variables, $a, b, \ldots, f$. This means that $n = 6$. Let $I(a) = 1, I(b) = 2, \ldots, I(f) = 6$. Let $D = \{\neg b, d\}$, i.e., $i = 4$. Then D subsumes the following 2^{n-i} consecutive full-length clauses (we give in brackets also the bit representation): $C_0 = \{\neg a, \neg b, \neg c, d, \neg e, \neg f\}$ `(000100)`, $C_1 = \{\neg a, \neg b, \neg c, d, \neg e, f\}$ `(000101)`, $C_2 = \{\neg a, \neg b, \neg c, d, e, \neg f\}$ `(000110)`, $C_3 = \{\neg a, \neg b, \neg c, d, e, f\}$ `(000111)`, but does not subsume the next one, which is: $C_4 = \{\neg a, \neg b, c, \neg d, \neg e, \neg f\}$ `(001000)`.*

5.3.1 The CCC Algorithm

We present two versions of CCC. The first counts one by one. The second is more optimized.

Algorithm 1 works as follow: It sets *count* to zero, then starts a loop. In the loop, in Line 4, it converts *count* to a full-length clause. In Line 5 it checks whether this full-length clause is subsumed or not. If yes, it increases *count* by one in Line 6. If not, by point (b) of Lemma 21, a solution is found, which is returned in Line 8. Eventually *count* either

Algorithm 1 `CCC`(S)

Require: S is a non-empty clause set.

Ensure: If S is satisfiable it returns a solution of S, otherwise returns the empty set.

```
1: n := number of variables in S;
2: count := 0;
3: while count < 2^n do
4:     C :=FullLengthClauseRepresentationOf(count);
5:     if ∃D ∈ S(D subsumes C) then
6:         count := count + 1;
7:     else
8:         return ¬C;
9:     end if
10: end while
11: return {};
```

grows beyond $2^n - 1$ or a solution is found. In the first case we know from point (a) of Corollary 3 that S is unsatisfiable.

Lemma 24. *[Soundness of `CCC`] Algorithm 1 is sound.*

Proof. There are two cases, the input clause set is either satisfiable (I), or unsatisfiable (II).
(I:) Let us assume that the input clause set S is satisfiable. From point (a) of Corollary 3 we know that S subsumes less than 2^n full-length clauses, i.e., there is a full-length clause which is not subsumed by S. Since Algorithm 1 iterates over all full-length clauses, eventually it encounters a full-length clause, called C in Line 5, which is not subsumed by S and returns its negation in Line 8. We know from point (b) of Lemma 21 that $\neg C$ is a solution of S, which means that the input clause set is satisfiable, which is the correct answer.
(II:) Let us assume that the input clause set S is unsatisfiable. From point (a) of Corollary 3 we know that S subsumes all 2^n full-length clauses. Therefore, the **if** expression in Line 5 will be always true, so *count* will be increased by 1 in Line 6. Eventually *count* will be 2^n and, hence, the empty set will be returned in Line 11, which means that the input clause set is unsatisfiable, which is the correct answer. □

Lemma 25. *[Soundness and completeness of `CCC`] Algorithm 1 is sound and complete.*

Proof. We already know from Lemma 24 that Algorithm 1 is sound. So it is enough to show that it stops for each valid input. The algorithm has one **while** loop, which terminates either if $count = 2^n$ or if we found a full-length clause which is not subsumed by S. In the **while** loop there is one **if** statement. In the **else** branch we have a **return** statement, i.e., the **while** loop terminates in that branch. If it does not terminate in the **else** branch, then in the **then** branch, in Line 6, it increases *count* by one. This means that if it does

not terminate in the **else** branch, then *count* will be eventually equal to 2^n, i.e., the **while** loop eventually terminates. Therefore, Algorithm 1 is complete. Hence, Algorithm 1 is sound and complete. □

Lemma 26. *[Worst case complexity of `CCC`] The worst case complexity of `CCC` is $O(2^n \cdot Check \cdot Conversion)$, where Check is the complexity of a subsumption check, and Conversion is the complexity of converting a number to a full-length clause.*

Proof. In the worst case the algorithm has to visit all the 2^n full-length clauses. □

5.3.2 The `Optimized CCC` Algorithm

Now we give a more optimized version of `CCC`, which is called `Optimized CCC`.

First we present it in a very intuitive way. The `Optimized CCC` algorithm uses the observation that a clause subsumes 2^{n-i} consecutive ordered full-length clauses, where i is the index of its last literal. This means that if we find a clause D which subsumes C, then we can increase the counter by 2^{n-i} instead of 1. So we do not need to visit all full-length clause to decide satisfiability. `Optimized CCC` always selects that D in the loop which grants the biggest step, i.e., where the number 2^{n-i} is the greatest. This optimization does not affect the soundness and the completeness of the algorithm.

Now we give it in a more formal way, see Algorithm 2.

Algorithm 2 works as follow: It counts full-length clauses in the local variable *count*. In Line 2 it sets *count* to zero, then starts a loop. In the loop, in Line 4, it converts *count* to a full-length clause, called C. In the second loop, see Line 7-14, it computes i, which is the smallest of the last literal indices of clauses which subsumes C. If no such i exists, i.e., there exists no clause in S, which subsumes C, then a solution, $\neg C$ is found, and returned in Line 16. If we cannot find a not subsumed full-length clause, then *count* will be eventually greater or equal than 2^n. In this case it returns the empty set in Line 21.

The algorithm suggests that in case of an unsatisfiable input clause set *count* will be eventually greater or equal than 2^n, but actually in this case *count* will be eventually equal to 2^n. We give only an intuitive reason to support this observation: If we would have more than n variables (we might run the algorithm on a subproblem), then the integer 2^n would represent the next possible clause to be check.

Line 9 implies that we have to index the variables. Assume that S contains a clause $\{a, b, c\}$ with indices i_a, i_b, and i_c, where $i_a < i_b < i_c$, then `Optimized CCC` selects this clause in the worst case, i.e., if *count* is increased always by one, in $2^{i_a-1} \cdot 2^{i_b-i_a-1} \cdot 2^{i_c-i_b-1} = 2^{i_c-3}$ times in Line 8. This means that different indexing results in different running time.

The following lemma states that `Optimized CCC` is sound and complete.

Lemma 27. *[Soundness and completeness of `Optimized CCC`] Algorithm 2 is sound and complete.*

Algorithm 2 `Optimized CCC`(S)

Require: S is a non-empty clause set.

Ensure: If S is satisfiable it returns a solution of S, otherwise returns the empty set.

```
1: n := number of variables in S;
2: count := 0;
3: while count < 2^n do
4:     C :=FullLengthClauseRepresentationOf(count);
5:     increment := 0;
6:     i := n + 1;
7:     for all D ∈ S do
8:         if D subsumes C then
9:             lastI :=IndexOfLastLiteral(D)
10:            if lastI < i then
11:                i := lastI; increment := 2^(n−i);
12:            end if
13:        end if
14:    end for
15:    if increment = 0 then
16:        return ¬C;
17:    else
18:        count := count + increment;
19:    end if
20: end while
21: return {};
```

Proof. Since the structure of the optimized algorithm is the same as `CCC` we do not need to prove that it stops for every valid input. Its soundness is also clear from soundness of Algorithm `CCC`, see Lemma 24, and from the fact that *count* is incremented by 2^{n-i}, which is the number of consecutive full-length clauses which are subsumed by D, see Lemma 23. □

While we investigated this algorithm, we made the observation that if we always select the best D in the for loop, then the biggest possible step is 2^{n-j}, where j is the index of the last 1 bit in the counter. Note that this bit always comes from the previous run of the loop. It is so, because if there were an even better D in the input clause set, then the algorithm would select that D even in the previous run of the while loop and setting a bit to 1 which has smaller index than j. We use this observation in the next version of the algorithm, called `CSFLOC`.

5.3.3 The `CSFLOC` Algorithm

Now we introduce the `Counting Subsumed Full Length Ordered Clauses` algorithm, for short `CSFLOC`. It is based on `Optimized CCC`, see Algorithm 2. In `Optimized CCC` we would like to increase the counter by the biggest possible 2^{n-i} step, i.e., we try to find a clause D which subsumes the counter and the index of its last literal is the smallest. While we studied `Optimized CCC` we observed that the biggest possible step is 2^{n-j}, where j is the index of the last 1 bit in the counter. So it is useless to search the whole clause set to find the best clause. It is enough to check only those clauses where the index of the last literal is greater than equal to j.

The best solution is to use ordered clauses. Then we can define the following data structure: $S[i] := \{C \mid C \in S \wedge \text{IndexOfLastLiteral}(C) = i\}$, where $i = 1..n$, and where S is a clause set. In this case $S[k]$ is a set which contains those clauses from S where the last literal has variable index k.

With the help of this data structure we can redefine `Optimized CCC`. This version does not have to visit all clauses in each run. It runs until it finds the first clause which subsumes the clause representation of the counter searching from $S[1]$ to $S[n]$.

Algorithm 3 works as follow: It counts full-length clauses in the local variable *count*. In Line 2 it creates the $S[]$ data structure. In Line 3 it sets *count* to zero. In Line 4 it sets i to zero, which has no function but it is used in the proofs. It starts a loop in Line 5. In the loop, in Line 6, it converts *count* to a full-length clause, called C. In Line 7 it sets *increment* to zero. In the second loop, see Line 8-14, it computes j, which is the smallest of the last literal indices of clauses which subsumes C. If j is found then it sets the *increment* to be 2^{n-j} in Line 11. Line 12 is a kind of break statement, it stops the loop, because the smallest j value is found. If no such j is found, i.e., there exists no clause in S, which subsumes C, then a solution, $\neg C$ is found, and returned in Line 16. If we cannot find a not subsumed full-length clause, then *count* will be eventually greater or equal than 2^n. In this case it returns the empty set in Line 21.

Algorithm 3 `Optimized CCCv2`(S)

Require: S is a non-empty list of ordered clauses with variable index function I.

Ensure: If S is satisfiable it returns a solution of S, otherwise returns the empty set.

1: $n :=$ number of variables in S;
2: $S[i] := \{C | C \in S \wedge \text{IndexOfLastLiteral}(C) = i\}$, where $i = 1..n$;
3: $count := 0$;
4: $i := 0$;
5: **while** $count < 2^n$ **do**
6: $C :=$FullLengthClauseRepresentationOf($count$);
7: $increment := 0$;
8: **for** $j := 1$; $j <= n$; $j := j + 1$ **do**
9: **if** $\exists D \in S[j]$ such that D subsumes C **then**
10: $C_i := C$; $D_i := D$; $i := i + 1$;
11: $increment := 2^{n-j}$;
12: $j := n + 1$; // it stops the for loop
13: **end if**
14: **end for**
15: **if** $increment = 0$ **then**
16: return $\neg C$;
17: **else**
18: $count := count + increment$;
19: **end if**
20: **end while**
21: return {};

Algorithm 3 is the same as Algorithm 2 except it uses an auxiliary data structure. From this we can obtain that it is sound and complete.

Lemma 28. *[Soundness and completeness of `Optimized CCCv2`] Algorithm 3 is sound and complete.*

Proof. To show this we have to show that Algorithm 3 has the same behaviour as Algorithm 2. We know that Algorithm 3 uses the following data structure: $S[i] := \{C \mid C \in S \wedge \text{IndexOfLastLiteral}(C) = i\}$, where $i = 1..n$, and where S is the input clause set. This means that $S[k]$ is a set which contains those clauses from S where the last literal has variable index k.
Line 1 of Algorithm 3 is the same as Line 1 of Algorithm 2.
Line 2 of Algorithm 3 creates the $S[]$ data structure.
Line 3 of Algorithm 3 is the same as Line 2 of Algorithm 2.
Line 4 of Algorithm 3 initializes variable i which is used in Lemma 29 and Lemma 30. It has no functional purpose.
Lines 5-7 of Algorithm 3 are the same as Lines 3-5 of Algorithm 2.
Lines 8-9 and 11-14 of Algorithm 3 select the best subsuming clause, called D, and set *increment* according to it. To do that it uses the $S[]$ data structure and the loop variable j. Lines 6-14 of Algorithm 2 do the same. To do that it uses the minimum value selection algorithm.
Line 10 of Algorithm 3 is used in Lemma 29 and Lemma 30. It has no functional purpose.
Lines 15-21 of Algorithm 3 are the same as Lines 15-21 of Algorithm 2.
Therefore, Algorithm 3 has the same behaviour as Algorithm 2. From this and from Lemma 27 we obtain that Algorithm3 is sound and complete. □

Now we prove two properties of this algorithm, which helps us to create the `CSFLOC` algorithm. First, we show that the last 1 bit of the counter is always set by the previous run of the outer loop.

Lemma 29. *[Observation 1.] In Algorithm 3 the following observation is true: if C_x has a value and k =IndexOfLastPositiveLiteral(C_x), then the k-th literal is negative in C_{x-1}, where $x >= 1$ and x is a natural number.*

Proof. We prove this by induction. This is true for $i = 1$, because C_0 is the clause where all literal is negative, because of Lines 3-4, Line 6 and Line 10. The induction hypothesis is that this is true for C_{x-1}. We show that this is true for C_x. Let k =IndexOfLastPositiveLiteral(C_x). We have two cases, either the k-th literal is negative in C_{x-1} or positive. We have to show that this later case is not possible. Let us assume that the k-th literal in C_{x-1} is positive. Let m =IndexOfLastPositiveLiteral(C_{x-1}). Then there are 3 cases: (a) either $m < k$, (b) or $m > k$, (c) or $m = k$. Case (a) contradicts that the k-th literal in C_{x-1} is positive. In case (b) we have that C_{x-1} and C_x are clause representation of $counter_{x-1}$ and $counter_x$, respectively, such that $counter_x = counter_{x-1} + increment$,

see Lines 10-11 and Line 18. We know that the m-th bit is 1 in $counter_{x-1}$ but 0 in $counter_x$, so we cannot have $counter_x = counter_{x-1} + increment$, because $increment$ has only one 1 bit all others are 0, see Line 11. This is a contradiction. In case (c) we know that the k-th literal is the last positive literal in C_{x-1} and also in C_x. From this we know that $j > k$, where j =IndexOfLastPositiveLiteral(D_{x-1}). From this we know that (I) from the k-th to n-th literals are not present in D_{x-1}. We also know from the induction hypothesis that in $count_{x-2}$ the k-th bit is 0, which implies that (II) from the 1-st to the $k-1$-th bits $count_{x-1}$ and $count_{x-2}$ are the same, and (III) the index of the last literal in D_{x-2} is less then k, because $counter_{x-1} = counter_{x-2} + increment$ and $increment$ has only one 1 bit all others are 0, see Lines 3-4, 9-11, and Line 18. From (I) and (II) it follows that D_{i-1} would be a suitable choice in Line 9 for the case $i = x - 2$, i.e., in the $x - 2$-th run of the outer loop, but this contradicts (III). □

As the second step, we show that the biggest possible $increment$ value is 2^{n-k}, where k is the index of the last 1 bit in the counter.

Lemma 30. *[Observation 2.] In Line 9 of Algorithm 3 it is true that* $2^{n-j} <= 2^{n-k}$*, where* k *=IndexOfLastPositiveLiteral(C)., i.e.,* $j >= k$*.*

Proof. Assume k =IndexOfLastPositiveLiteral(C) in Line 9 of Algorithm 3. Our goal is to show that $j >= k$ in Line 9 of Algorithm 3. We assume that our goal is not true, i.e., $j < k$. We show that this assumption leads to a contradiction, i.e., our goal is true. From the assumption $j < k$ it follows that there is a clause, say D, in $S[j]$, such that D subsumes C. From Lemma 29 we know that the k-th bit of the *counter* is set in the previous run of the outer loop. It means that the previous *counter* and the recent *counter* differs only in $z >= 1$ bits on the indices from k to $k + z - 1$, such that in the previous *counter* the k.-th bit was 0 and now it is 1, $k+1$.-th bit was 1 and now it is 0, ... $k + z - 1$.-th bit was 1 and now it is 0. In this case clause D is a suitable choice in Line 9 in the previous run of the outer loop, because the k-th ... $k + z$-th literals are not present in D, because the index of its last literal is j and $j < k$. Since the last literal index of D is j and $j < k$ the algorithm had to select this clause in the previous run of the outer loop, which would cause that the index of last 1 in the *counter*, i.e., the value of k should be greater than equal to j which contradicts that $j < k$. □

The two observation lemmas Lemma 29 and Lemma 30 mean that if we always use the best D in the inner loop, as in `Optimized CCC`, then the biggest possible step is 2^{n-j}, where j is the index of the last 1 bit in the counter.

Algorithm 3 serves as the basis of `CSFLOC` which is introduced as Algorithm 4 in this chapter. To get the new algorithm we need to change only one assignment in Line 7: from $j := 1$ to $j := IndexOfLastPositiveLiteral(C)$, i.e., we use the observation that in case of `Optimized CCC` the biggest possible step is 2^{n-j}, where j is the index of the last 1 bit in the counter. So the new algorithm is defined as follows:

Algorithm 4 `CSFLOC`(S)

Require: S is a non-empty list of ordered clauses with variable index function I.

Ensure: If S is satisfiable it returns a solution of S, otherwise returns the empty set.

1: $n :=$ number of variables in S;
2: $S[i] := \{C | C \in S \wedge \text{IndexOfLastLiteral}(C) = i\}$, where $i = 1..n$;
3: $count := 0$;
4: **while** $count < 2^n$ **do**
5: $increment := 0$;
6: $C :=$FullLengthClauseRepresentationOf($count$);
7: **for** $j :=$IndexOfLastPositiveLiteral(C); $j <= n$; $j := j + 1$ **do**
8: **if** $\exists D \in S[j]$ such that D subsumes C **then**
9: $increment := 2^{n-j}$;
10: $j := n + 1$;
11: **end if**
12: **end for**
13: **if** $increment = 0$ **then**
14: return $\neg C$;
15: **else**
16: 0 $count := count + increment$;
17: **end if**
18: **end while**
19: return {};

The next theorem states that CSFLOC is sound and complete.

Theorem 13. *[Soundness and Completeness of CSFLOC] Algorithm 4 is sound and complete.*

Proof. From Lemma 29 and Lemma 30 we know that Algorithm 4 has the same behaviour as Algorithm 3, from this and from Lemma 28 we obtain that Algorithm 4 is sound and complete. □

5.4 Implementation

We have implemented Algorithm 2 in Java and C#. The implementation can be downloaded from https://github.com/cccsat/cccsat/.

We have implemented Algorithm 3 and 4 in ANSI C. The implementation can be downloaded from http://aries.ektf.hu/~gkusper/opt_ccc.c and http://aries.ektf.hu/~gkusper/csfloc.c. Actually the two files are the same except that in csfloc.c a flag called 'doIndexing' is set. Our goal was to demonstrate that the CSFLOC algorithm works. We did not want to come up with a fine-tuned algorithm, but we wanted to have a nice, readable code. The main idea of the implementation is to use arrays of integers for the bit representation of clauses.

The old Java implementation can handle SAT problems with at most 62 variable, the C# one is good with at most 64 variables. The new ANSI C implementation can handle any number of variables. This implementation uses an integer array to represent a clause and the counter. Since we use integers we can use fast bit operations to implement the methods of a clause.

We have also a new Java implementation, called CSFLOC6. This can be downloaded from http://aries.ektf.hu/~gkusper/CSFLOC6.java. This version uses a BitSet to represent the counter and an ArrayList to represent a clause. This implementation can handle also any number of variables.

The newest implementation, CSFLOC18, also written in Java. It contains lots of variable ordering strategies and variable clustering. It uses array of boolean variables instead of BitSet, because it gives 20-30% speed-up in most of the cases. This version can be downloaded from http://aries.ektf.hu/~gkusper/CSFLOC18.java. This implementation can handle also any number of variables. This version has the following usage:

```
Usage: java CSFLOC18 cnf_file_name variable_renaming_strategy
use_learned_clauses add_long_tail_clauses clustering_factor
clause_list_ordering.
Example: java CSFLOC18 hole7.cnf BC 1 1 5 1 .
The parameter 'cnf_file_name' is the name of a file in DIMACS format
containing a CNF SAT problem.
```

```
The parameter 'variable_renaming_strategy' may contain the letters
in any order: B, C, H, I, R, S, W, or the same lower case letters;
or it is 'NON' which means that there is no strategy.
If it is not set, then no strategy is used.
Letter 'B' means that black clauses (all literals are negative) will
contain low variable indices.
Letter 'C' means that it clusters variables, i.e., it moves those
variables closer, which occurs frequently together in the clauses.
Letter 'H' means that definite horn clauses (exactly one positive
literal) will contain low variable indices. It renames the literals
of a definite horn clause only if non of them are renamed yet.
Letter 'I' means that definite horn clauses will contain low variable
indices. The positive literal will get the biggest index inside the
definite horn clause.
Letter 'R' means that it renames variables by their frequency, the most
frequent variable will get the least index.
Letter 'S' means that 'strait' clauses (exactly one negative literal)
will contain low variable indices. The negative literal will get the
least index inside the strait clause.
Letter 'W' means that white clauses (all literals are positive) will
contain low variable indices.
List of some useful strategies:
    In case of random 3-SAT problems: 'IWCR', 'HWCR', or 'BHWCR'.
    In case of pigeon-holes problems: 'B'.
    In case of Black-and-White SAT problems generated by the
    Balatonbogl\'{a}r model: 'I 1 2 0 0'.
The parameter 'use_learned_clauses' may be 0 or 1.
If it is not set, then it is 1.
Number '0' means that it does not use learned clauses
to speed up the search.
Number '1' means that it generates for each variable upto 2*3
learned clauses, and it uses them to speed up the search.
The parameter 'add_long_tail_clauses' may be 0, 1, or 2.
If it is not set, then it is 0.
Number '0' means that it does not generate resolvents
in the preprocessing steps.
Number '1' means that it generates resolvents, which contains big
index variables, and it adds them to the initial clause set.
Number '2' means that it generates resolvents, which contains big
index variables, and it adds them to the learned clauses.
```

```
The parameter 'clustering_factor' may be 2 or a bigger number.
If it is not set, then it is 5.
Clustering computes the variable pair frequency.
It groups the most frequent pairs in the first cluster until it
becomes full, and so on.
It works only if the variable_renaming_strategy contains 'C'.
Usually 5 is the best option.
The parameter 'clause_list_ordering' may be 0 or 1.
If it is not set, then it is 1.
Number '0' means that it does not sort clauses inside a clause list.
Number '1' means that it sorts the clauses inside each clause list
by the index of their first literal. For example {-1, 5, 10}
precedes {2, 4, -8}. Sorting almost always results in less
'numberOfRunsOfTheMainLoop', but sorting needs
O(numberOfClauses*numberOfClauses/numberOfVariables) time, so
if you have lots of clauses, you might consider to switch it off.
```

5.5 Test Results

We tested the `ANSI C` version of the `Optimized CCC` and `CSFLOC`. The tests were done on `iMac macOS Sierra` (CPU: 2,5GHz Intel Core i5, Memory: 4GB 1333MHz DDR3). We tested our algorithm against `Glucose`[1] `3.0`, which is a widely used standard *SAT* solver. We used `Glucose 3.0` with simplification mode turned on. The *SAT* instances were downloaded from the *SATLIB* Benchmark Problems [2] page. The rest is generated by the *WnDGen SAT* problem generator [89].

For each instance we used a timeout of 900 seconds. The results are presented in table format. The first column is the name of the tested *SAT* instance. In the 2nd to 3rd columns, we show the number of variables (n), the number of clauses (m). The last three columns are *CPU* times in seconds for `Glucose 3.0`, `Optimized CCC` and `CSFLOC`.

One of our goals was to countercheck the completeness of `CSFLOC` on lots of problems. Table 5.1 shows that we have run it for all `uf20*`, `uf50*`, and `uuf50*` instances from *SATLIB* Benchmark Problems. For all the 3000 instances it returned the right result. The same is true for all the other tests we have performed.

Table 5.1 shows that random *SAT* instance, like `uf50*` and `uuf50*`, are difficult for `CSFLOC`, but very easy for `Glucose3`. We can also see that the `CSFLOC` outperforms `Optimized CCC`. We can see on lines `hole 6-9` that on pigeon hole problems, where we place $n+1$ pigeons in n holes without placing 2 pigeons in the same hole, the `CSFLOC` can compete with the `Glucose3` if the number of variables is relatively small. It is so because

[1] `http://www.labri.fr/perso/lsimon/glucose/`
[2] `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

in pigeon hole problems there are lots of short clauses, which help CSFLOC to do big steps.

	n	m	Glucose3	CSFLOC	OptCCC
uf20*	20	91	0,0042s	0,0046s	0,0043s
uf50*	50	218	0,0044s	1,0477s	5,1632s
uuf50*	50	218	0,0041s	5,4807s	26,2165s
hole6	42	133	0,0085s	0,0061s	0,0284s
hole7	56	204	0,0920s	0,0758s	0,5499s
hole8	72	297	1,1338s	1,1734s	12,1701s
hole9	90	415	13,1981s	19,7952s	TIME OUT

Table 5.1: Runtimes on problems from *SATLIB* Benchmark Problems, uf* are *SAT*, others are *UNSAT*

Since the CSFLOC uses a variable ordering, a natural question arises, whether a "better" variable ordering results in a better runtime result? To answer this question we used the variable clustering tool written by Prof. Tudor Jebelean [102]. The tool is available at http://aries.ektf.hu/~gkusper/Clustering-8-Jul-2010.tar.gz. Our experience shows, see Table 5.2 that if we cluster the variables of SAT problems then CSFLOC runs faster. Especially it can solve random *SAT* problems with 75 or 100 variables, which was otherwise very difficult for CSFLOC.

Instances	n	m	clustering factor	CSFLOC
uf20*	20	91	no clustering	0,0046 s
uf20*	20	91	clusters of 2 vars	0.0033s
uf20*	20	91	clusters of 3 vars	0.0031s
uf50*	50	218	no clustering	1,0477s
uf50*	50	218	clusters of 2 vars	0.2161s
uf50*	50	218	clusters of 3 vars	0.7917s
uf75*	75	325	no clustering	TIME OUT
uf75*	75	325	clusters of 2 vars	147,7269s
uf75*	75	325	clusters of 3 vars	129,1973s
uf100*	100	430	no clustering	TIME OUT
uf100*	100	430	clusters of 2 vars	TIME OUT
uf100*	100	430	clusters of 3 vars	25,0288s

Table 5.2: Runtimes on problems after clustering, all instances are *SAT*

5.6 IoT Inspired Test Results

In Chapter 3 several models was introduced to translate a directed graph into a *SAT* problem. The motivation was to check whether the communication graph of a *WSN* is strongly connected or not. We showed that if a directed graph is strongly connected, then

the generated *SAT* instance is a Black-and-White *SAT* problem. A *Black-and-White SAT* problem has only two solutions, the white assignment in which each variable is true, and the black assignment in which each variable is false.

5.6.1 Test on Black-and-White 2-SAT Problems

Now we use this result to generate test problems. We generated *WSN*s with 100 sensors with different density and translated them into *Black-and-White 2-SAT* problems by the strong model presented in Section 3.4. We also added the black and the white assignments to the tested *SAT* instances to make the strongly connected ones to be unsatisfiable. The less dense *WSNs* are not strongly connected, i.e., the resulting *SAT* problems are satisfiable. The more dense *WSNs* are strongly connected, i.e., the corresponding *SAT* instances are unsatisfiable.

We tested `Glucose3` and `CSFLOC6` on these instances. Table 5.3 shows the test results. The name of the instance indicates whether it is satisfiable (*SAT*), or unsatisfiable (*UNSAT*). We can see that `Glucose3` is around 10 times faster than `CSFLOC6`.

	n	**m**	`Glucose3`	`CSFLOC6`
`WSN_100_SAT_1.cnf`	100	676	0,0017s	0,0181s
`WSN_100_SAT_2.cnf`	100	852	0,0021s	0,0202s
`WSN_100_SAT_3.cnf`	100	837	0,0022s	0,0201s
`WSN_100_UNSAT_1.cnf`	100	1464	0,0026s	0,0241s
`WSN_100_UNSAT_2.cnf`	100	1458	0,0030s	0,0230s
`WSN_100_UNSAT_3.cnf`	100	2263	0,0037s	0,0271s
`WSN_100_UNSAT_4.cnf`	100	4631	0,0060s	0,0370s
`WSN_100_UNSAT_5.cnf`	100	6172	0,0093s	0,0461s
`WSN_100_UNSAT_6.cnf`	100	8134	0,0047s	0,0481s
`WSN_100_UNSAT_7.cnf`	100	842	0,0021s	0,0240s

Table 5.3: Runtimes on *Black-and-White 2-SAT* problems

We also generated *WSNs* with more sensors: 10-10 instances with 200, 500, 1000, 2000 sensors and with different density. We used the same method described in the previous test case, i.e., we translated the *WSN*s into *Black-and-White 2-SAT* problems. Table 5.4 shows the average test results. Here the number of clauses, i.e., column **m** is an average number. We can see that from 1000 sensors the `CSFLOC` algorithm is better to check whether the communication graph of a *WSN* is strongly connected or not. This result was a big surprise for us because these SAT problems have the same number of variables as the number of sensors, see column **n**. It shows that a random *SAT* instance is very difficult for the `CSFLOC` already with 100 variables but not a *Black-and-White 2-SAT* problem.

	n	m	Glucose3	CSFLOC6
WSN_100*	100	2732,9	0,0038s	0,0317s
WSN_200*	200	5171,6	0,0108s	0,0494s
WSN_500*	500	50310,0	0,1499s	0,1652s
WSN_1000*	1000	271103,6	2,4919s	0,5336s
WSN_2000*	2000	1006531,6	20,1476s	3,7704s

Table 5.4: Average runtimes on *Black-and-White 2-SAT* problems

5.6.2 Test on Weakly Nondecisive SAT Problems

The results in this subsection are generated by CSFLOC6. Weakly nondecisive SAT problems (*WnD* problems) can be generated by our tool, called WnDGen tool, available here: `http://fmv.ektf.hu/tools.html`. We recall that these problems have an interesting property, they have only two solutions, and the two solutions are the opposite of each other, as in the case of *Black-and-White 2-SAT* problems.

It seems that `CSFLOC` is very good at solving *WnD* problems generated by the `WnDGen` *SAT* problem generator, see Table 5.5.

To create the *WnD* problem instances we used the "-unsat" switch of WndGen tool, which adds the negation of the two solutions of the *WnD* problems to the instance, making it unsatisfiable. This shows that overconstrained problems with few variables, like *WnD* problems, are easier for the `CSFLOC`, than general *SAT* problems.

5.6.3 Test on Black-and-White 3-SAT Problems

The results in this subsection are generated by CSFLOC18. This is the newest version, which contains lots of variable ordering strategies. We tested CSFLOC18 on models generated by our strong model ($\mathcal{SM}$-based benchmarks), on models generated by the Balatonboglár model ($\mathcal{BM}$-based benchmarks), and on models generated by the simplified Balatonboglár model ($\mathcal{SBM}$-based benchmarks). We also wanted to test our weak model, but it takes too long time to generate it even for 1000 nodes, so we decided to left out that model.

The tests were done on Intel quadcore machine (CPU: i7-6700HQ CPU 2.60GHz, 2601 Mhz, Memory: 32GB 3000MHz DDR4). For testing, we used Glucose 4.2.1 [3] *(Glucose)*, MapleLCMDistChrono BT-DL v3 [4] *(Maple)* and the latest version of CSLFOC [98, 97] (CSFLOC 18) *(CSFLOC)*.

Performance Results

We ran the solvers on three categories of benchmarks: $\mathcal{BM}$-, $\mathcal{SBM}$-, and $\mathcal{SM}$-based. For each instance we used a timeout of 900 seconds.

[3] `https://www.labri.fr/perso/lsimon/glucose/`

[4] `https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/`

Instances	**n**	**m**	**Glucose3**	CSFLOC6
WndGen_unsat_10_3.cnf	10	722	0,0017s	0,0281s
WndGen_unsat_10_4.cnf	10	1682	0,0112s	0,0242s
WndGen_unsat_10_5.cnf	10	2522	0,0276s	0,0321s
WndGen_unsat_10_6.cnf	10	2522	0,0316s	0,0532s
WndGen_unsat_15_3.cnf	15	2732	0,0027s	0,0281s
WndGen_unsat_15_4.cnf	15	10922	0,2088s	0,0893s
WndGen_unsat_15_5.cnf	15	30032	1,9055s	0,1282s
WndGen_unsat_15_6.cnf	15	60062	9,3856s	0,2062s
WndGen_unsat_15_7.cnf	15	90092	25,2388s	0,2991s
WndGen_unsat_15_8.cnf	15	102962	38,8714s	0,4343s
WndGen_unsat_15_9.cnf	15	90092	36,3287s	3,1751s
WndGen_unsat_20_3.cnf	20	6842	0,0063s	0,0532s
WndGen_unsat_20_4.cnf	20	38762	1,8473s	0,1271s
WndGen_unsat_20_5.cnf	20	155042	37,5251s	0,3022s
WndGen_unsat_20_6.cnf	20	465122	TIME OUT	0,7083s
WndGen_unsat_20_7.cnf	20	1085282	TIME OUT	3,5827s
WndGen_unsat_20_8.cnf	20	2015522	TIME OUT	8,7173s
WndGen_unsat_20_9.cnf	20	3023282	TIME OUT	19,7737s

Table 5.5: Runtimes on WnD problems *(all instances are UNSAT)*

We examined the growth of runtime as a function of $|V| + |E|$, where $|V|$ is the number of vertices in the graph, and $|E|$ is the number of edges.

For this performance measurement the benchmarks were generated from directed graphs of different densities, with 10-10000 vertices. In all $\mathcal{BM}$-based benchmarks the Glucose won. In the case of $\mathcal{SBM}$, the individual solvers won alternately. These results will be the subject of deeper analysis in the future. In case of $\mathcal{SM}$-based benchmarks for smaller graphs ($|V| < 750$), Glucose and Maple won alternately. Results for larger problems ($|V| > 1000$) are shown in Table 5.6 and in Table 5.7. In Table 5.6, all benchmark were generated from sparse graphs (the density of graph was smaller than 10%). We can see that Maple won in all cases.

However, CSFLOC wins each case where the graph density was greater than 10%. These results are shown in Table 5.7. This result suggest that CSFLOC provides a faster answer to the question whether the communication network is strongly connected or not than the most relevant SAT solvers, if the network consists of a large number of sensors and many direct connections.

In Figures 5.1-5.3 we show the runtimes of CSFLOC on various benchmarks. Each graph consists of 100 vertices and the densities are between 0.03 and 1. If the density is 1 then

$\|V\|$	**CSFLOC**	**Maple**	**Glucose**
1000	0,1754s	**0,01216s**	0,01552s
1500	0,2013s	**0,0184s**	0,0191s
2000	0,2927s	**0,02422s**	0,0252s
2500	0,3986s	**0,02709s**	0,0275s
3000	0,4853s	**0,0450s**	0,0492s
5000	0,6238s	**0,0652s**	0,0723s
10000	0,7815s	**0,0825s**	0,0906s

Table 5.6: Average runtimes on Strong Model-based problems

$\|V\|$	**CSFLOC18**	**Maple**	**Glucose**
1000	**0,5235s**	1,0457s	1,0903s
1500	**1,5541s**	5,7417s	6,9023s
2000	**2,4236s**	9,6167s	9,8691s
2500	**2,787s**	10,987s	12,4136s
3000	**3,102s**	12,94s	13,23s
5000	**4,4807s**	17,6777s	18,6290s
10000	**6,1337s**	25,2252s	27,3703s

Table 5.7: Average runtimes on Strong Model-based problems

the graph is complete, i.e, if we have 100 vertices, then we have $100 * 99 = 9900$ edges, i.e., $|V| + |E| = 10^4$.

Analyzing the runtimes of the three solvers, it can be generally stated that their time complexities are close to linear on $\mathcal{SM}$-based problems, polynomial on $\mathcal{BM}$-based problems and logarithmic on $\mathcal{SBM}$-based problems.

File sizes

We examined the growth of the file sizes as a function of $|V|+|E|$, where $|V|$ is the number of vertices in the graph, and $|E|$ is the number of edges. For this examination, we used graphs with density from 0.03 to 1. As before, each instance consists of 100 vertices. Figure 5.4-5.5 show that the file size in the case of Balatonboglár Model grows polynomial, like the previously mentioned time complexity.

We examined the file size ratio, too. Figure 5.6 shows that above 20% graph density, the ratio of $\mathcal{SM}/\mathcal{BM}$ and $\mathcal{SBM}/\mathcal{BM}$ is below 1%, and $\mathcal{SM}$ is bit smaller than $\mathcal{SBM}$.

This means that $\mathcal{BM}$ seems to be very redundant. To check this we also measured the number of unused clauses.

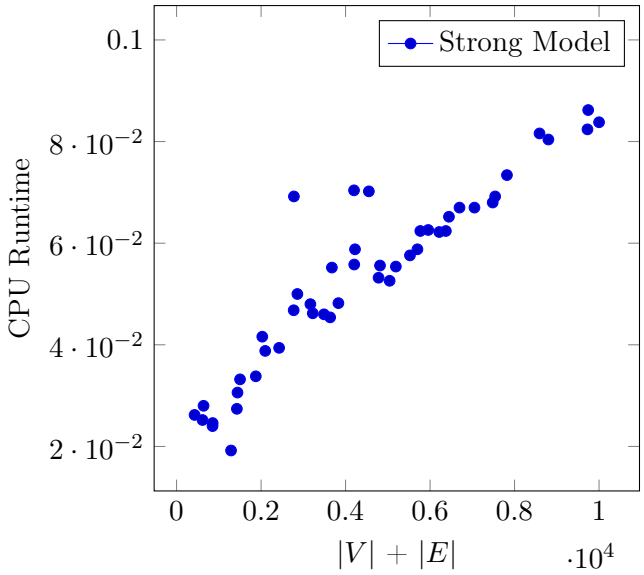

Figure 5.1: CPU runtimes of CSFLOC as a function of $|V| + |E|$ on Strong Model-based benchmarks

Unused Clauses

The latest version of CSFLOC includes an option that it monitors the unused clauses (actually the redundancy in the model). In Figures 5.6-5.7 we show the ratio of number of unused clauses to the number of all clauses in the various models. In the case of Figure 5.6, the graph size ($|V| + |E|$) is from 20 to 60000.

Figure 5.8 shows the ratio of unused clauses to all clause as the function of the density of the represented directed graph.

Figures 5.7-5.8 show that $\mathcal{SBM}$ gives the most concise description. $\mathcal{SBM}$ may provide the theoretical minimum, for some very small graphs, but not for bigger ones.

5.7 How to Create a Parallel Version?

From `CSFLOC` we can create naturally a parallel algorithm. Assume we have q clients, then the j.-th `CSFLOC` instance has to count from $j \cdot 2^{n/q}$ till $(j+1) \cdot 2^{n/q-1}$, where $j = 0 \ldots q-1$ and n is the number of variables in the input SAT problem.

The algorithm is so simple that there is no necessary communication between the clients, except the signal that one of the clients has found the solution, which stops also the other clients.

Each node has to maintain only a counter, they can share the input *SAT* problem because it does not change during the execution. This means that the memory usage is minimal.

We have a preliminary Java implementation of the parallel version. It is available here: `http://aries.ektf.hu/~gkusper/CCCBomberv1Dot0.java`, or here: `http://fmv.ektf.hu/files/CCCv1.0.zip`.

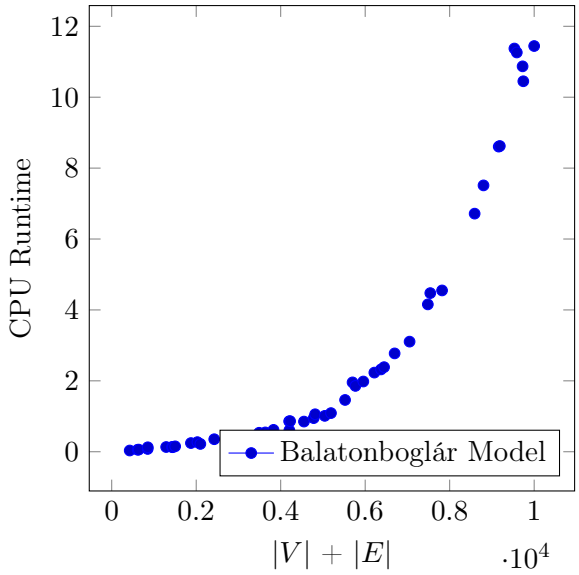

Figure 5.2: CPU runtimes of CSFLOC as a function of $|V| + |E|$ on Balatonboglár Model-based benchmarks

5.8 Future Work

We could find a closer connection between the inclusion-exclusion principle and `CSFLOC`. Let us assume that we have the following situation:

Let us have 3 variables, $\{a, b, c\}$, i.e., $n = 3$. Let $I(a) = 1, I(b) = 2, I(c) = 3$. Let the input clause set be $\{C_1, C_2\}$, where $C_1 = \{\neg a, \neg b\}$, $C_2 = \{\neg a, \neg c\}$. Let *count* be 0, so $C = \{\neg a, \neg b, \neg c\}$. We can see that both C_1 and C_2 subsumes C. By the `CSFLOC` algorithm we have to increase *count* by 2^{3-2} because the best subsuming clause is C_1 and its last literal index is 2, because $I(b) = 2$.

Now $count = 2$ and $D = \{\neg a, b, \neg c\}$. This is subsumed only by C_2. Its last variable is c and $I(c) = 3$. So we can increase *count* by 2^{3-3}. Now $count = 3$ and $D = \{\neg a, b, c\}$. It is not subsumed, so its negation is a solution.

In the above example, we can see that we find the same clause, C_2, in the first and also in the second loop, but we used it only in the second one. So it seems that in the first loop we could use both C_1 and C_2. We believe that the inclusion-exclusion principle could help us to find a better solution here.

We would like to also implement the parallel version to *GPU* architecture.

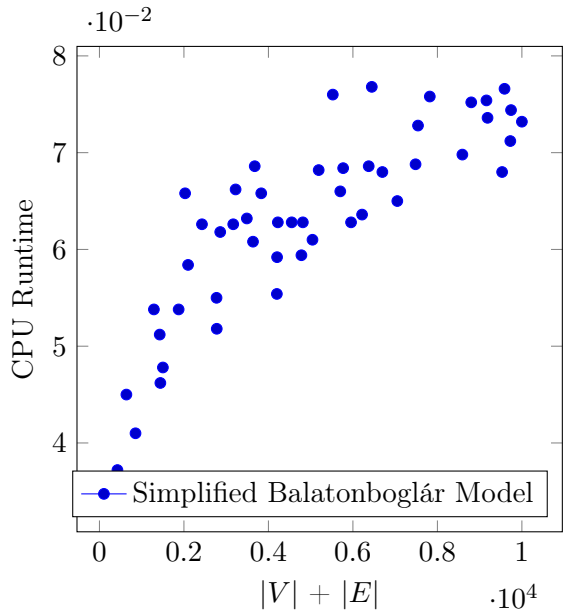

Figure 5.3: CPU runtimes of CSFLOC as a function of $|V|+|E|$ on Simplified Balatonboglár Model-based benchmarks

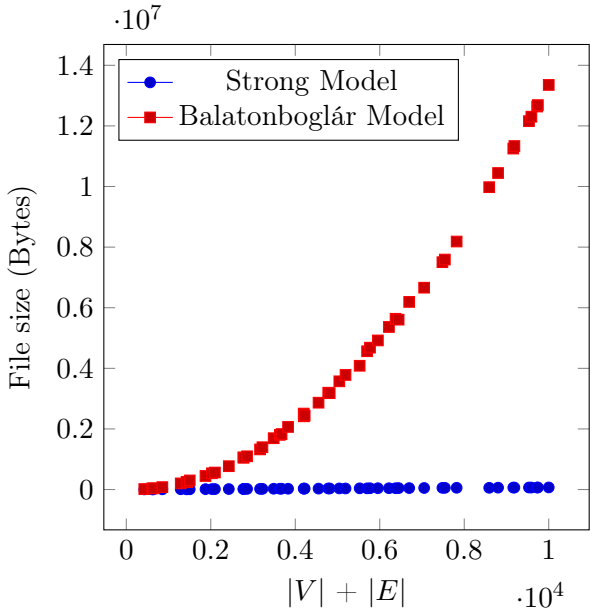

Figure 5.4: File sizes as a function of $|V| + |E|$

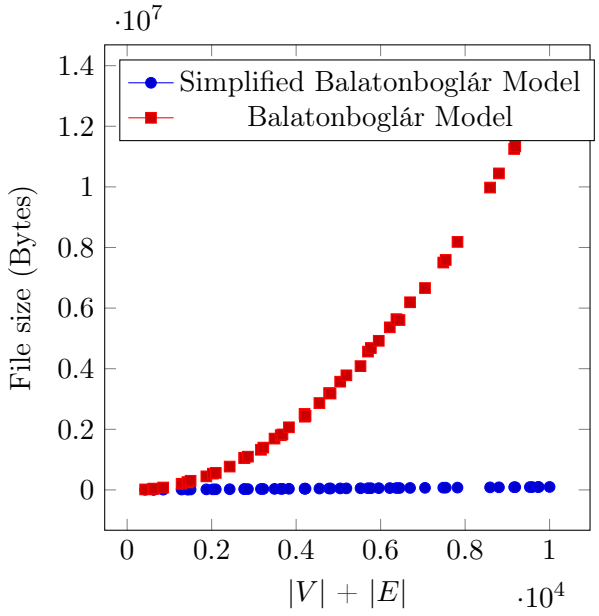

Figure 5.5: File sizes as a function of $|V| + |E|$

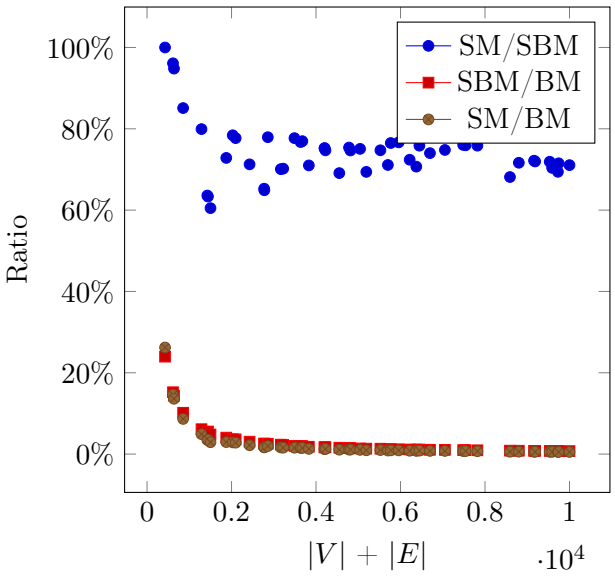

Figure 5.6: Ratio of file sizes as a function of $|V| + |E|$

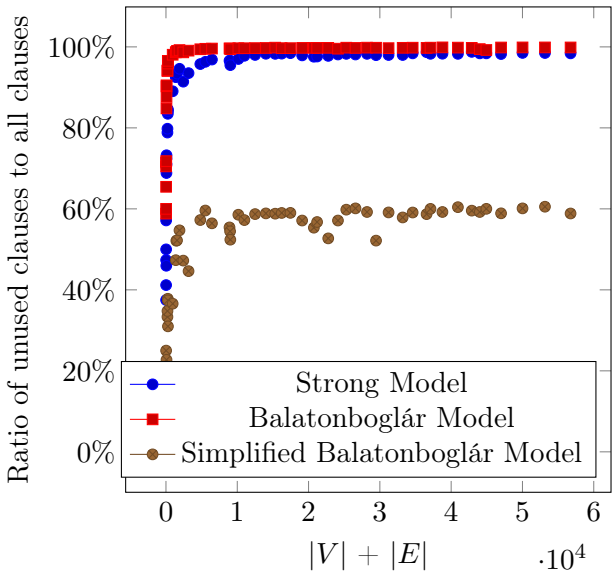

Figure 5.7: Ratio of unused clauses to all clauses

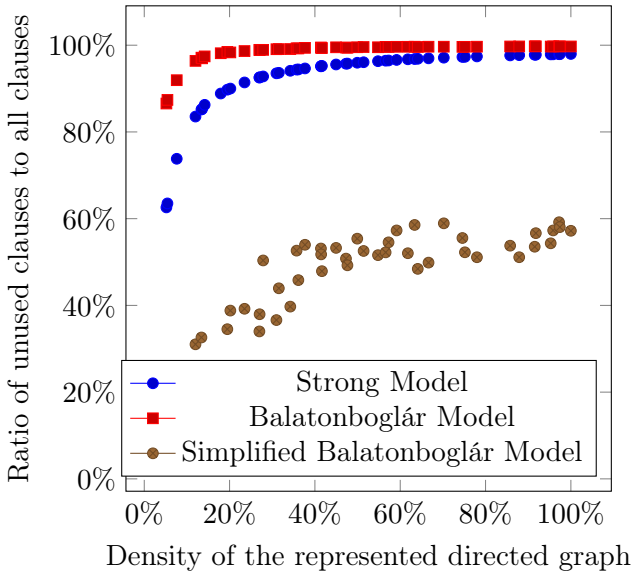

Figure 5.8: Ratio of unused clauses to all clauses

6 Summary

In this dissertation, we have described the Black-and-White SAT problem, and the CS-FLOC SAT solver, which are novel results of mine and my research team.

In the **Introduction**, see Chapter 1, we have summarized our results in these fields, In **Background and Related Work**, see Chapter 2, we gave a short survey of the SAT problem. This chapter is based on the introductions of our papers in this field.

The following 3 chapters contains our main results. All of them are based on our original papers.

The chapter entitled **The Black-and-White SAT Problem**, see Chapter 3, contains our recent results how to represent a directed graph as a SAT problem. The main theoretical result is the so called Transitions Theorem, see Theorem 4, which states that any model between the strong and the weak one has the same property, i.e., the SAT representation of the communication graph is Black-and-White if and only if the communication graph is strongly connected.

It seems that this theorem is important, because the strong model is the most natural model of communication graph: we use implication (which is intuitively an arrow) to represent an edge (which is intuitively an arrow). The weak model is an other kind of natural representation: if we can send a message to more than one neighbor then we send it only to one, but we do not allow the message to be trapped in a cycle, the message must leave cycles. Any model between them is a "good" one.

We could find two such models, but it seems that is not that difficult to find more ones.

To find the first one, the Balatonboglár model, was a highly non-trivial task. We needed a high level intuition of the child game called tag. Tag uses the tag forward rule: the tagged one cannot tag back, he or she must tag someone else, so must tag forward. This means that if node A sends a message to node B, and B has the possibility to send this back to A, then this is forbidden, B has to send the message to someone else. This rule cancels the need of cycle detection in directed graph, which means that this model can be constructed in a very fast way.

On the other hand this process generates lots of clauses which are unnecessary, which means that lots of clauses are implied by the rest of the model. To overcome this problem, we introduced the simplified Balatonboglár model which generates much less clauses. Its file size is only 1% of the file size of the Balatonboglár model.

The simplified Balatonboglár model uses the same trick, i.e., the tag forward rule instead of cycle detection, but it generates these clauses only for so called big-cycles. A big-cycle is

a cycle which contains all nodes from a strongly connected component. It represents also in the same way the so called inner-cycles and links between the components. In this way we generate much less clauses.

We study these models to be able to create better SAT solvers. The first special one, BaW 1.0 [96] SAT solver is already created, but it is specialized only to accept SAT problems created by the strong model, but for those it is a linear time solver. We work on it to generalize it to the other less restrictive models.

The chapter entitled **Propoerties of WnDGen**, see Chapter 4, contains our results on weakly nondecisive clause sets (WnD), and on the algorithm WnDGen which can generate such instances. The notion of weakly nondecisive clause set was introduced in my thesis. When I started to work with Csaba Biró, who is my formal PhD student, we decided to study an open question from my thesis: How to generate weakly nondecisive clause sets?

The answer is not easy, because if we have a weakly nodecisive clause set, i.e., all clauses are weakly nondecisive, then if we remove one clause, then the rest will be not necessary weakly nondecisive anymore. But after a half year thinking we could come up with an algorithm, which can create such SAT problems. The new algorithm got the name $WnDGen$. It has two parameters: $WnDGen(C, k)$, where is C is the generator clause, and k is the length of the generated clauses.

We showed that $WnDGen(C, k)$ generates a Black-and-Write SAT instance if and only if C is either the black clause or the white clause and $|C| \geq 2k - 3$, see Theorem 12. This shows that there is a link between $WnDGen$ and Black-and-White SAT problem.

The chapter entitled **The CSFLOC SAT Solver**, see Chapter 5, contains our results on our novel SAT solver. The main result in this part is the theorem which states that `CSFLOC` is a sound and complete SAT solver for the general SAT problem, see Theorem 13.

The `CSFLOC` algorithm uses an idea from the field of #SAT to solve the SAT problem. It is an iterative version of the well-known inclusion-exclusion principle.

It seems that `CSFLOC` is very good at solving Black-and-White SAT problems and WnD problems generated by the `WnDGen` SAT problem generator. Chapter 5 also discusses implementation issues and presents lots of tests results.

Although, this dissertation is finished here, the work goes on. Currently we have 3 research directions: We would like extend the BaW 1.0 SAT solver to the Balatonboglár model. The second goal is to find new models, where the number of clauses generated by the Balatonboglár model are reduced without loosing its nice properties. Finally, we work on an incomplete SAT solver which tries to fit the input SAT problem into one of our models, creates the corresponding directed graph, and suggests solutions based on the graph.

There is a small team which works on these ideas. I am pretty sure that this work will result in nice articles and PhD dissertations.

7 Bibliography

7.1 Related Bibliography

[1] C. Ansotegui, J. Larrubia, C. M. Li, F. Manya, *Mv-Satz: A SAT solver for many-valued clausal forms*, Proceedings of the 4th International Conference on Knowledge discovery and discrete mathematics, pp. 143–150, 2003.

[2] B. Aspvall, M. F. Plass, R. E. Tarjan, *A Linear-Time Algorithm For Testing The Truth Of Certain Quantified Boolean Formulas*, Information Processing Letters, pp. 121–123, 1979.

[3] B. Aspvall, *Recognizing Disguised NR(1) Instances of the Satisfiability Problem*, J. of Algorithms, 1, pp. 97–103, 1980.

[4] D. P. Anderson, *BOINC: A System for Public-Resource Computing and Storage*, Proc. of GRID 2004, pp. 4–10, 2004.

[5] S. Andrei, *Counting for satisfiability by inverting resolution*, Artificial Intelligence Review, Volume 22, Issue 4, pp. 339–366, 2004.

[6] G. Audemard, L. Simon, *Glucose 2.3 in the SAT 2013 Competition*, In: Proceedings of SAT Competition 2013, pp. 42—43, 2013.

[7] J. Bang-Jensen, G.Z. Gutin, *Digraphs: Theory, Algorithms and Applications*, book, 2nd edition, ISBN: 978-1-84800-997-4, Springer Publishing Company, 2008.

[8] B. Beckert, R. Hähnle, F. Manyà, *The SAT problem of signed CNF formulas*, Labelled Deduction, vol. 17 of Applied Logic Series, pp. 61–82, 2000.

[9] H. Bennett and S. Sankaranarayanan, *Model Counting Using the Inclusion-Exclusion Principle*, Theory and Applications of Satisfiability Testing - SAT 2011 Lecture Notes in Computer Science, Volume 6695, pp. 362–363, 2011.

[10] A. Biere, M. Heule, H. van Maaren, T. Walsh, *Handbook of Satisfiability*, IOS Press, Amsterdam, 2009.

[11] A. Biere, *Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010*, Technical Report 10/1, FMV Reports Series, JKU, 2010.

[12] A. Biere, *Lingeling and Friends Entering the SAT Challenge 2012*, Haifa Verification Conference, Department of Computer Science Series of Publications B, vol. B-2012-2, pp. 33–34, 2012.

[13] E. Birnbaum and E. L. Lozinskii, *The good old Davis-Putnam procedure helps counting models*, Journal of Artificial Intelligence Research, Volume 10, 457–477, 1999.

[14] E. Boros, P. L. Hammer, X. Sun, *Recognition of q-Horn Formulae in Linear Time.* Discrete Applied Mathematics, v. 55, pp. 1–13, 1994.

[15] R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Trans. Comput., pp. 677–691, 1986.

[16] V. Chandru, J. Hooker, *Extended Horn Sets in Propositional Logic*, J. of the ACM, 38(1), pp. 205–221, 1991.

[17] G. Chartrand, L. Lesniak, P. Zhang, *Graphs & Digraphs*, book, 6th edition, ISBN: 978-1-4987-3576-6, Chapman and Hall/CRC, 2015.

[18] W. Chrabakh, R. Wolski, *GridSAT: A Chaff-based Distributed SAT Solver for the Grid*, Proc. of SC'03, pp. 37–49, 2003.

[19] W. Chrabakh, R. Wolski, *GrADSAT: A Parallel SAT Solver for the Grid.*, Technical report, UCSB Computer Science, 2003.

[20] O. Chanseok, *Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL, PhD thesis*, New York University, 2016.

[21] S. A. Cook, *The Complexity of Theorem-Proving Procedures*, Proc of STOC'71, pp. 151–158, 1971.

[22] M. Dalal, D. W. Etherington, *A Hierarchy of Tractable Satisfiability Problems.* Information Processing Letters, v.44, pp. 173–180, 1992.

[23] M. Davis, G. Logemann, D. Loveland, *A Machine Program for Theorem Proving*, Commun. ACM, vol. 5, no. 7, pp. 394–397, 1962.

[24] G. Davydov, I. Davydova, and H.K. Büning, *An efficient algorithm for the minimal unsatisfiability problem for a subclass of CNF*, Annals of Mathematics and Artificial Intelligence 23, pp. 229-–245, `https://doi.org/10.1023/A:1018924526592`, 1998.

[25] E. W. Dijkstra, *Finding the Maximum Strong Components in a Directed Graph*, In Selected Writings on Computing: A personal Perspective, Texts and Monographs in Computer Science, Springer New York, pp. 22—30, 1982.

[26] W. F. Dowling, J. H. Gallier, *Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae*. J. of Logic Programming, 1(3), pp. 267–284, 1984.

[27] O. Dubois, *Counting the Number of Solutions for Instances of Satisfiability*, Theoretical Computer Science, Volume 81, pp. 49-–64, 1991.

[28] J. Franco, A. V. Gelder *A perspective on certain polynomial-time solvable classes of satisfiability* Discrete Applied Mathematics, v. 125, no. 2–3, pp. 177-–214, https://doi.org/10.1016/S0166-218X(01)00358-4, 2003.

[29] J.W. Freeman, *Improvements to Propositional Satisfiability Search Algorithms. Ph.D. Dissertation*, Department of Computer and Information Science, University of Pennsylvania, 1995.

[30] Y. Hamadi S. Jabbour, L. Sais, *ManySAT: a Parallel SAT Solver*, Journal on Satisfiability, Boolean Modeling and Computation, vol. 6, pp. 245–262, 2009.

[31] Y. Hamadi, S. Jabbour, C. Piette, L. Saïs, *Deterministic Parallel DPLL*, JSAT, vol. 7. no. 4, pp. 127–132, 2011.

[32] R. A. Hearn, *Games, Puzzles, and Computation*, PhD thesis, MIT, June 2006.

[33] M. Heule, *March: Towards a Look-ahead SAT Solver for General Purposes*, Master thesis, TU Delft, The Netherlands, 2004.

[34] M. Heule, O. Kullmann, S. Wieringa, A. Biere, *Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads*, Lecture Notes in Computer Science, vol. 7261, pp. 50–65, 2011.

[35] M. Heule, O. Kullmann, V. W. Marek, *Solving and verifying the boolean Pythagorean Triples problem via Cube-and-Conquer*, Applications of Satisfiability Testing - SAT 2016, Lecture Notes in Computer Science, vol. 9710, pp. 228—245, 2016.

[36] M. Heule, O. Kullmann, V. W. Marek, *Solving Very Hard Problems: Cube-and-Conquer, a Hybrid SAT Solving Method*, Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence Best Sister Conferences, https://doi.org/10.24963/ijcai.2017/683, pp. 4864–4868, 2017.

[37] A. E. J. Hyvärinen, T. Junttila, I. Niemelä, *Partitioning SAT instances for distributed solving*, Proc. of LPAR'10, pp. 372–386, 2010.

[38] H. N. Gabow, *Path-based depth-first search for strong and biconnected components*, Information Processing Letters, Volume 74, Issues 3–4, pp: 107-114, 2000.

[39] A. V. Gelder, *Propositional Search with k-Clause Introduction Can be Polynomially Simulated by Resolution*, Proceedings of the 5th International Symposium on Artificial Intelligence and Mathematics, 1998.

[40] C.P. Gomes, B. Selman, H. Kautz, *Boosting combinatorial search through randomization*, National Conference on Artificial Intelligence, pp. 431-–437, 1998.

[41] C. P. Gomes, A. Sabharwal, B. Selman, *Model Counting*, Chapter 20 of Handbook of Satisfiability, IOS Press, Amsterdam, 2009.

[42] W. Gong, X. Zhou, *A survey of SAT solver*, AIP Conference Proceedings 1836, 020059, https://doi.org/10.1063/1.4981999. 2017.

[43] J. Gu, P. W. Purdom, J. Franco, B. W. Wah, *Algorithms for the Satisfiability (SAT) Problem: A Survey*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 19–152, 1996.

[44] L. Hellerman, *A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits*, IEEE Transactions on Electronic Computers, pp. 198–223, 1963.

[45] K. Iwama, *CNF-satisfiability test by counting and polynomial average time*, SIAM Journal on Computing, Volume 18, Issue 2, pp. 385–391, 1989.

[46] J. Johannsen, *The Complexity of Pure Literal Elimination*, In: Giunchiglia E., Walsh T. (eds) SAT 2005. Springer, Dordrecht 2005.

[47] P. Kacsuk, J. Kovács, Z. Farkas, A. C. Marosi, G. Gombás, Z. Balaton, *SZTAKI Desktop Grid (SZDG): A Flexible and Scalable Desktop Grid System*, Journal of Grid Computing, vol. 7, no. 4, pp. 439–461, 2009.

[48] R. M. Karp, *Reducibility among Combinatorial Problems*, Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, pp. 20–22, 1972.

[49] H. Kautz, BSelman, *The State of SAT*, Discrete Appl. Math. 155:12, pp. 1514–1524, http://dx.doi.org/10.1016/j.dam.2006.10.004, 2007.

[50] D. E. Knuth, *Nested Satisfiability*, Acta Informatica, v. 28, pp. 1–6, 1990.

[51] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability* book, ISBN-10: 0134397606, Addison-Wesley Professional, 1st Edition, 2015.

[52] O. Kullmann, *New methods for 3-SAT decision and worst-case analysis*, Theoretical Computer Science, 223(1-2): pp. 1–72, 1999.

[53] O. Kullmann, *On a generalization of extended resolution*, Discrete Applied Mathematics, 96-97(1-3) pp. 149–176, 1999.

[54] O. Kullmann, *Investigations on autark assignments*, Discrete Applied Mathematics, v. 107, pp. 99—137, 2000.

[55] O. Kullmann, *Investigating the Behaviour of a SAT Solver on Random Formulas*, Technical Report CSR 23-2002, Swansea University, Computer Science Report Series, 2002.

[56] R. P. Langlands, *Problems in the theory of automorphic forms to Salomon Bochner in gratitude*, Lectures in Modern Analysis and Applications III, pp. 18–61, 1970.

[57] C.M. Li and Anbulagan, *Look-Ahead versus Look-Back for Satisfiability Problems*, Lecture Notes in Computer Science, vol. 1330, pp. 342–356, 1997.

[58] J. H. Liang, V. Ganesh, P. Poupart, K. Czarnecki, *Learning Rate Based Branching Heuristic for SAT Solvers*, 19th International Conference on Theory and Applications of Satisfiability Testing SAT 2016, 2016.

[59] J. H. Liang, V. Ganesh, P. Poupart, K. Czarnecki, *Exponential Recency Weighted Average Branching Heuristic for SAT Solvers*, In: Proceedings of AAAI-16, 2016.

[60] J. H. Liang, Chanseok Oh, M. Mathews, C. Thomas, C. Li, V. Ganesh, *Machine Learning-based Restart Policy for CDCL SAT Solvers*, 21st International Conference on Theory and Applications of Satisfiability Testing (SAT 2018), 2018.

[61] E. L. Lozinskii, *Counting propositional models*, Information Processing Letters, Volume 41, pp. 327–332, 1992.

[62] Y. S. Mahajan, Z. Fu, S. Malik, *Zchaff2004: An Efficient SAT Solver*, Lecture Notes in Computer Science: Theory and Applications of Satisfiability Testing, vol. 3542, pp. 360–375, 2005.

[63] H. van Maaren, *A Short Note on Some Tractable Cases of the Satisfiability Problem* Information and Computation, v. 158:2, pp. 125–130, DOI:10.1006/inco.2000.2867, 2000.

[64] *F. Manyà, R. Béjar, G. Escalada-Imaz*, The satisfiability problem in regular CNF-formulas, Soft Computing: A Fusion of Foundations, Methodologies and Applications, Vol. 2(3), pp. 116–123, 1998.

[65] J. Marques-Silva, K. A. Sakallah, *GRASP: A new search algorithm for satisfiability*, Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 220—227, 1996.

[66] J. Marques-Silva, K. A. Sakallah, *GRASP-A search algorithm for propositional satisfiability*, IEEE Transactions on Computers 48(5), pp. 506-–521, 1999.

[67] S. Minato, *Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems*, Proceedings of the 30th International Design Automation Conference, pp.272–277, 1993.

[68] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, *Engineering an efficient SAT solver*, Design Automation Conference, pp. 530-–535, 2001.

[69] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, *Chaff: Engineering an Efficient SAT Solver*, Proc. of DAC'01, pp. 530–535, 2001.

[70] I. Munro, *Efficient Determination of the Transitive Closure of a Directed Graph*, Information Processing Letters, 1(2), pp: 56–58, 1971.

[71] B. Nagy, *Truth-teller-liar puzzles and their graphs*, Central European Journal of Operations Research - CEJOR 11, pp. 57–72, 2003.

[72] M. Posypkin, A. Semenov, O. Zaikin, *Using BOINC Desktop Grid to Solve Large Scale SAT Problems*, Computer Science, vol. 13, no. 1, pp. 25–34, 2012.

[73] P. Purdom, *A transitive closure algorithm*, BIT Numerical Mathematics 10.1, pp. 76-94, 1970.

[74] B. Roy *Transitivité et connexité*, C R Acad Sci Paris 249, pp: 216–218, 1959.

[75] M. G. Scutella, *A Note on Dowling and Gallier's Top-Down Algorithm for Propositional Horn Satisfiability*, J. of Logic Programming, v. 8(3), pp. 265–273, 1990.

[76] J. S. Schlipf, F. Annexstein, J. Franco, R. P. Swaminathan, *On finding solutions for extended Horn formulas*, Information Processing Letters, v. 54, pp. 133–137, 1995.

[77] M. Sharir, *A strong-connectivity algorithm and its applications in data flow analysis*, Computers And Mathematics with Applications, pp. 67–72, 1981.

[78] K. Simon, *An improved algorithm for transitive closure on acyclic digraphs*, Theor Comput Sci 58, pp. 325–346, 1988.

[79] Y. Tanaka, *A Dual Algorithm for the Satisfiability Problem*, Information Processing Letters, Volume 37, pp. 85—89, 1991.

[80] R. Tarjan,, *Depth-First Search and Linear Graph Algorithms*, SIAM Journal on Computing, Vol. 1, No. 2, pp. 146-160, 1972.

[81] C. A. Tovey, *A Simplified NP-complete Satisfiability Problem*, Discrete Applied Mathematics, v. 8, pp. 85–89, 1984.

[82] S. Warshall, *A theorem on boolean matrices*, J Assoc Comput Mach pp. 11–12, 1962.

[83] A. Weil, *Uber die Bestimmung Dirichletscher Reihen durch Funktionalgleichungen*, Mathematische Annalen, 149–156, 1967.

[84] A. J. Wiles, *Modular elliptic curves and Fermat's Last Theorem*, ANNALS OF MATH, pp. 443–551, 1995.

[85] H. Zhang, M. E. Stickel, *An Efficient Algorithm for Unit Propagation*, In Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96), pp. 166–169, 1996.

7.2 Author Bibliography

7.2.1 PhD Thesis

[86] G. Kusper *Solving and Simplifying the Propositional Satisfiability Problem by Sub-Model Propagation*, PhD thesis, Supervisor: Tudor Jebelean, Johannes Kepler University Linz, RISC Institute, pp. 1–146, 2005.

7.2.2 Journal Papers

[87] G. Kusper, Cs. Biró, *Convert a Strongly Connected Directed Graph to a Black-and-White 3-SAT Problem by the Balatonboglár Model*, submitted to Theory of Computing, 17 pages, submitted on 24.08.2019, status: under review.

[88] Cs. Biró, G. Kusper, *Equivalence of Strongly Connected Graphs and Black-and-White 2-SAT Problems*, Miskolc Mathematical Notes, Vol. 19, No. 2, pp. 755-768, 2018.

[89] Cs. Biró, G. Kovásznai, A. Biere, G. Kusper, G. Geda, *Cube-and-Conquer approach for SAT solving on grids*, Annales Mathematicae et Informaticae, Vol. 42, ISSN 1787-5021, pp. 9–21, 2013.

[90] G. Kusper, L. Csőke, G. Kovásznai, *Simplifying the propositional satisfiability problem by sub-model propagation*, Annales Mathematicae et Informaticae, Vol. 35, ISSN 1787-5021, pp. 75–94, 2008.

[91] G. Kusper, *Finding Models for Blocked 3-SAT Problems in Linear Time by Systematical Refinement of a Sub-Model*, Lecture Notes in Computer Science 4314, pp. 128–142, 2007.

[92] G. Kusper, *Solving the Resolution-Free SAT Problem by Hyper-Unit Propagation in Linear Time*. Annals of Mathematics and Artificial Intelligence, v. 43(1-4), pp. 129–136, 2005.

7.2.3 Conference Papers and Talks

[93] G. Kusper, Cs. Biró, T. Balla, *Investigation of the Efficiency of Conversion of Directed Graphs to 3-SAT Problems*, Proceedings of SACI-2020, DOI: 10.1109/SACI49304.2020.9118786, pp. 227–234, 2020.

[94] G. Kusper, Cs. Biró, T. Balla, *Representing Directed Graphs as 3-SAT Problems using the Simplified Balatonboglár Model*, ICAI-2020, poster, `https://icai.uni-eszterhazy.hu/2020/abstracts/ICAI_2020_abstract_128.pdf`, 2020.

[95] T. Balla, Cs. Biró, G. Kusper, *The BWConverter Toolchain: An Incomplete Way to Convert SAT Problems into Directed Graphs*, ICAI-2020, the paper version is accepted, the poster version is available: `https://icai.uni-eszterhazy.hu/2020/abstracts/ICAI_2020_abstract_133.pdf`, 2020.

[96] Cs. Biró, G. Kusper, *BaW 1.0 - A Problem Specific SAT Solver for Effective Strong Connectivity Testing in Sparse Directed Graphs*, Proceedings of CINTI 2018, pp. 160–165, 2018.

[97] G. Kusper, Cs. Biró, Gy. B. Iszály, *SAT solving by CSFLOC, the next generation of full-length clause counting algorithms*, Proceedings of IEEE International Conference on Future IoT Technologies 2018, DOI: 10.1109/FIOT.2018.8325589, 2018.

[98] G. Kusper, Cs. Biró, *Solving SAT by an Iterative Version of the Inclusion-Exclusion Principle*, Proceedings of SYNASC 2015, DOI: 10.1109/SYNASC.2015.38, pp. 189–190, 2015.

[99] Cs. Biró, G. Kusper, T. Tajti, *How to generate weakly nondecisive SAT instances*, Proceedings of SISY 2013, DOI: 10.1109/SISY.2013.6662583, pp. 265–269, 2013.

[100] G. Kusper and T. Jebelean, *SAT Solving Experiments in Multi-Domain Logic*, Proceedings of SCSS 2010 Symbolic Computation in Software Science, pp. 105–117, 2010.

[101] T. Jebelean and G. Kusper, *SAT Solving Experiments in Multi-Domain Logic*, Proceedings of ICAI-2010, ISBN: 978-963-9894-72-3, Volume I., pp. 95–105, 2010.

[102] T. Jebelean and G. Kusper, *Multi–Domain Logic and its Applications to SAT*, Proceedings of SYNASC 2008, DOI: 10.1109/SYNASC.2008.93, pp. 3–8, 2008.

8 Annex 1. - Convert a Strongly Connected Directed Graph to a Black-and-White 3-SAT Problem by the Balatonboglár Model

This is the submitted version of G. Kusper, Cs. Biró, *Convert a Strongly Connected Directed Graph to a Black-and-White 3-SAT Problem by the Balatonboglár Model*, submitted to Theory of Computing, 17 pages, submitted on 24.08.2019, status: under review. See also [87].

Convert a Strongly Connected Directed Graph to a Black-and-White 3-SAT Problem by the Balatonboglár Model

Gábor Kusper, Csaba Biró

Eszterházy Károly University, Eger, Hungary
{kusper.gabor, biro.csaba}@uni-eszterhazy.hu

Abstract. In a previous paper we defined the Black-and-White SAT problem which has exactly two solutions, where each variable is either true or false. We showed that Black-and-White 2-SAT problems represent strongly connected directed graphs. In this work we would like to extend that result. We introduce the Balatonboglár model which can represent a directed graph as a 3-SAT problem. We show that if the directed graph is strongly connected, then its Balatonboglár model is a Black-and-White 3-SAT problem. In our previous work the main idea was the following: If a graph contains two edges: $a \to b$, and $a \to c$, then those can be represented by the formula: $(a \implies b) \wedge (a \implies c)$, which is equivalent to two 2-clauses $(\neg a \vee b) \wedge (\neg a \vee c)$. The new idea is the following: If a graph contains two edges: $a \to b$, and $a \to c$, then those can be represented by the formula: $(a \implies b) \vee (a \implies c)$, which is equivalent to $a \implies (b \vee c)$, which is equivalent to a 3-clause $(\neg a \vee b \vee c)$. This idea is not enough to be able to generate a Black-and-White SAT formula from a strongly connected graph. We need to represent cycles of the graph, too. If $a_1 \to a_2 \to \cdots \to a_n \to a_1$ is a cycle with exit points $b_1, b_2, \ldots, b_m$, then this cycle can be represented by the clause: $(\neg a_1 \vee \neg a_2 \vee \cdots \vee \neg a_n \vee b_1 \vee b_2 \vee \cdots \vee b_m)$. The Balatonboglár model uses the trick that instead of detecting each cycle, it generates from each path $a \to b \to c$ the following 3-clause: $(\neg a \vee \neg b \vee c)$ even if there is no cycle which contains the vertices a and b. This simplification allows very fast 3-SAT problem generation from a directed graph, and the SAT instance will be a Black-and-White 3-SAT if and only if the input directed graph is strongly connected. On the other hand this simplification does not tell us how to generate from a 3-SAT problem a directed graph.

1 Motivation

In logic the most natural representation of an edge of a directed graph, say $a \to b$, is to use implication, i.e., $a \implies b$, i.e., the edge $a \to b$ can be represented by the binary clause: $(\neg a \vee b)$. We used that representation in a previous paper [2] and we were able to prove that the SAT representation of a strongly connected directed graph is a Black-and-White 2-SAT

problem. It is not a surprise that the SAT representation is a 2-SAT problem, since each edge can be represented by a binary clause.

The other property, Black-and-White, means that the resulting SAT problem is "nearly" unsatisfiable, that means, it has only two solutions, the white assignment, and the black one. The white assignment assigns true to each variable, the black one assigns false to each variable.

This representation helps to translate a graph into a 2-SAT problem. We can also translate a Black-and-White 2-SAT problem into a directed graph. We can also translate any other 2-SAT problem into a directed graph, if it does not subsume neither the black nor the white clause, i.e., it does not contain a clause which contains only positive or negative literals, like $(a \vee b)$ or $(\neg a \vee \neg b)$

There are some natural questions. Why shall we translate a 2-SAT problem into a direct graph? What to do with clauses which contain only positive or negative literals?

Actually, there is no point to translate a 2-SAT problem into a directed graph, because the 2-SAT problem is solvable in linear time [1]. Preferably, we should translate 3-SAT problems into directed graphs, because then we could use graph tools to study it, or we can translate them into 2-SAT problems. To be able to do so, the first step is to create a 3-SAT representation for directed graphs. The second step should answer the second question, what to do with clauses which contain only positive or negative literals?

This paper aims to help to answer the first question. We present several models which can translate a directed graph into a SAT problem such that the SAT problem is Black-and-White if and only if the directed graph is strongly connected.

2 Introduction

These days one of the most promising branch of mathematics is the idea, that we try to unify different mathematical theories, like in case of Langlands program [7], which relates algebraic number theory to automorphic forms and representation theory. Another nice example is the modularity theorem [10] (formerly called the Taniyama–Shimura–Weil conjecture), which states that elliptic curves over the field of rational numbers are related to modular forms. Without the modularity theorem Andrew Wiles could not prove Fermat's Last Theorem [11].

In this paper, we show a link between directed graphs and propositional logic formulas. We prove a theorem which allows to use an al-

gorithm from the field of propositional logic to check a graph property. Namely, we transform a directed graph into a SAT problem to check whether the graph is strongly connected or not.

The most prominent graph representations are:

- Implication graph [1] is a is a skew-symmetric directed graph, where vertices are literals (boolean variables, and their negation), edges represents implication. Note, that the binary clause $x \vee y$ is represented by two implications in the implication graph: $\neg x \supset y$, and $\neg y \supset x$, and so the implication graph is skew-symmetric, i.e., it is isomorphic to its own transpose graph.
- AIG, And-Inverter Graph [6] is directed acyclic graph where vertices are logical conjunction with to input edges, a marked edge means logical negation, the boolean variables are the input, the formula itself is the output.
- BDD, Reduced Ordered Binary Decision Diagram [4], which is a rooted, directed, acyclic graph, which consists of vertices, which are boolean variables, and terminal vertices, called 0-terminal, which terminates paths, where the formula evaluates to false; and 1-terminal, which terminates paths, where the formula evaluates to true. Each non-terminal vertex has two child vertices called low child, corresponding edge is called 0-edge; and high child, corresponding edge is called 1-edge; which are possible values of the parent vertex. One has to merge any isomorphic subgraphs and eliminate any vertex whose two children are isomorphic.
- ZDD (called also ZBDD in the literature), Zero-Suppressed Binary Decision Diagram [8], is a kind of binary decision diagram, where instead of the rule "eliminate any vertex whose two children are isomorphic" we use the rule "eliminate those vertices whose 1-edge points directly to 0-terminal". If a SAT problem has only a few solutions then ZDD is a better representation than BDD.
- Graph representation of logical games is a well-knows connection between graphs and logical formulas. Some examples are [5,9].

As we can see a great effort has been done in the direction from formulas to graphs. In this paper, we study the other way, the direction from graphs to formulas.

In a previous paper [2] we showed how to convert a directed graph into a 2-SAT problem. In this paper, we aim to translate directed graphs into 3-SAT problems.

3 Definitions

A *literal* is a boolean variable, called positive literal, or the negation of a boolean variable, called negative literal. Examples for literals are: $a, \neg a, b, \neg b, \ldots$.

A *clause* is a set of literals. A *clause set* is a set of clauses. A *SAT problem* is a clause set. An *assignment* is a set of literals.

Clauses are interpreted as disjunction of their literals. Assignments are interpreted as conjunction of their literals. Clause sets are interpreted as conjunction of their clauses.

If a clause or an assignment contains exactly k literals, then we say it is a *k-clause* or a *k-assignment*, respectively. A *k-SAT problem* is a clause set where its clauses have at most k literals. A clause from a clause set is a *full-length clause* if and only if (iff) it contains all variables from the clause set.

If a is a literal in clause set S, and $\neg a$ is not a literal in S, then we say that a is a pure literal in S.

Negation of a set H is denoted by $\neg H$ which means that all elements in H are negated. Note, that $\neg\neg H = H$.

Let V be the set of variables of a clause set. We say that WW is the *white clause* or the *white assignment* iff $WW = V$. We say that BB is the *black clause* or the *black assignment* iff $BB = \neg V$. For example if $V = \{a, b, c\}$, then $WW = \{a, b, c\}$, and $BB = \{\neg a, \neg b, \neg c\}$.

We say that clause C *subsumes* clause D iff C is a subset of D.

We say that clause set S *subsumes* clause C iff there is a clause in S which subsumes C. Formally: S *subsumes* $C \iff \exists D(D \in S \wedge D \subseteq C)$.

We say that assignment M is a *solution* for clause set S iff for all $C \in S$ we have $M \cap C \neq \{\ \}$.

We say that the clause set S is a *Black-and-White SAT problem* iff it has only two solutions, the white assignment (WW) and the black one (BB).

We say that clause sets A and B are equivalent iff they have the same set of solutions. We say that clause set A entails clause set B iff the set of solutions of A is a subset of the set of solutions of B, i.e., A may have no other solutions than B. This notion is denoted by $A \geq B$. Note, that if A subsumes all clauses of B, then $A \geq B$.

We say that A is stronger than B iff $A \geq B$ and A and B are not equivalent. This notion is denoted by $A > B$.

The construction $\mathcal{D} = (\mathcal{V},\mathcal{E})$ is a directed graph, where $\mathcal{V}$ is the set of vertices, and $\mathcal{E}$ is the set of edges. An edge is an ordered pair of vertices.

The edge (a, b) is depicted by $a \to b$, and we can say that a has a child b. If (a, b) is an element of $\mathcal{E}$, then we may say that (a, b) is an edge of $\mathcal{D}$.

We say that $\mathcal{D}$ is a communication graph iff for all a in $\mathcal{V}$ have that (a, a) is not in $\mathcal{E}$, and if x is an element of $\mathcal{V}$ then $\neg x$ must not be an element of $\mathcal{V}$. We need this constraint because we generate a logical formula out of $\mathcal{D}$. If we speak about a communication graph then we may use the word node as a synonym of vertex.

A path from a_1 to a_j in directed graph $\mathcal{D}$ is a sequence of vertices $a_1, a_2, \ldots, a_j$ such that for each $i \in \{1, \ldots, j-1\}$ we have that (a_i, a_{i+1}) is an edge of $\mathcal{D}$. A path from a_1 to a_j in directed graph $\mathcal{D}$ is a *cycle* iff (a_j, a_1) is an edge of $\mathcal{D}$. The cycle $a_1, a_2, \ldots, a_j, a_1$ is represented by the following tuple: $(a_1, a_2, \ldots, a_j)$. This tuple can be used as a set of its elements. Note, that in the representation of a cycle the first and the last element must not be the same vertex. Figure 1 shows as example for a path and a cycle.

If we have a cycle $(a_1, a_2, \ldots, a_n)$ then b is an exit point of it iff for some $j \in \{1,2, \ldots, m\}$ we have that (a_j, b) is an edge and $b \notin \{a_1, a_2, \ldots, a_n\}$.

A directed graph is complete iff every pair of distinct vertices is connected by a pair of unique edges (one in each direction). A directed graph is strongly connected iff there is a path from each vertex to each other vertex. Note, that a complete graph is also strongly connected. Note, that a strongly connected graph contains a cycle which contains all vertices.

Fig. 1. A path and a cycle

4 The Strong Model of Communication Graphs

In our previous model [2], which we call in this paper the strong model of communication graphs, edges are represented by logical implication: Let us assume, that the graph $\mathcal{D}$ is ($\{a,b,c\}$,$\{(a,b),(a,c)\}$), i.e., $\mathcal{D}$ is a graph with 3 vertices: a, b, c, and with two edges: from a to b, and from a to c. So the strong model of $\mathcal{D}$ is $(a \implies b) \wedge (a \implies c)$.

The interpretation of this formula is the following in the field of Wireless Sensors Networks: If node a is able to send a message to nodes b and c, then node a sends the message to both nodes b and c. But this might result in an error, because both nodes have to send an acknowledgement, that the message is arrived, but using the same frequency at the same time results in interference. A more natural way of communication is that we use some protocol, which decides where to send the message, either to node b or to node c, see the next section.

The strong model was defined formally in our previous work [2]. We recall its definition.

Let $\mathcal{D} = (\mathcal{V},\mathcal{E})$ be a communication graph, then the strong model of $\mathcal{D}$ is denoted by $\mathcal{SM}$, and defined as follows:

$$\mathcal{SM} := \{\{\neg a,\ b\} \mid (a,b) \in \mathcal{E}\}.$$

Note, that $\mathcal{SM}$ is a 2-SAT problem and has a nice property, each clause in it has exactly one positive and one negative literal, therefore $\mathcal{SM}$ is satisfiable for any communication graph $\mathcal{D}$.

Furthermore, $\mathcal{SM}$ is a Black-and-White SAT problem iff $\mathcal{D}$ is strongly connected, see Theorem 1 in [2]. Since we use this theorem several times, we recall it:

Theorem 1 (Theorem 1 from [2]). *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Then $\mathcal{SM}$ is a Black-and-White 2-SAT problem if and only if $\mathcal{D}$ is strongly connected.*

As an example we show the strong model of the communication graph of Figure 2:

$$\begin{aligned}\mathcal{SM} = \{&\{\neg a, b\}, \{\neg a, c\}, \{\neg b, a\}, \{\neg b, c\},\\ &\{\neg b, d\}, \{\neg c, a\}, \{\neg c, d\}\}.\end{aligned} \tag{1}$$

Note, that since the communication graph from Figure 2 is not strongly connected, its strong model (1) is not a Black-and-White SAT problem. For example $\{\neg a, \neg b, \neg c, d\}$ is a solution of it.

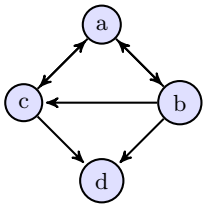

Fig. 2. A communication graph with 4 vertices, 3 cycles.

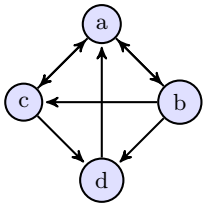

Fig. 3. A strongly connected communication graph with 4 vertices, 6 cycles.

As a second example we show the strong model of the communication graph from Figure 3. This graph is almost the same as the previous one (Figure 2), but here we have an edge from d to a, which makes this graph strongly connected. Its strong model is:

$$\begin{aligned}\mathcal{SM} = \{&\{\neg a, b\}, \{\neg a, c\}, \{\neg b, a\}, \{\neg b, c\},\\ &\{\neg b, d\}, \{\neg c, a\}, \{\neg c, d\}, \{\neg d, a\}\}.\end{aligned} \tag{2}$$

Note, that since the communication graph from Figure 3 is strongly connected, its strong model is a Black-and-White SAT problem, i.e., the SAT problem in (2) has only these two solutions: $\{a, b, c, d\}$, and $\{\neg a, \neg b, \neg c, \neg d\}$.

It is not easy to check, that this SAT problem is indeed a Black-and-White SAT problem. Therefore, first we show the set of all full-length clauses on the variables $\{a, b, c, d\}$ with an index number (index number: full-length clause:

$$\begin{aligned}\{0 : \{\neg a, \neg b, \neg c, \neg d\},\ 1 : \{\neg a, \neg b, \neg c, d\},\ 2 : \{\neg a, \neg b, c, \neg d\},\\ 3 : \{\neg a, \neg b, c, d\}, 4 : \{\neg a, b, \neg c, \neg d\},\ 5 : \{\neg a, b, \neg c, d\},\\ 6 : \{\neg a, b, c, \neg d\},\ 7 : \{\neg a, b, c, d\}, 8 : \{a, \neg b, \neg c, \neg d\},\\ 9 : \{a, \neg b, \neg c, d\},\ 10 : \{a, \neg b, c, \neg d\},\ 11 : \{a, \neg b, c, d\},\\ 12 : \{a, b, \neg c, \neg d\},\ 13 : \{a, b, \neg c, d\},\ 14 : \{a, b, c, \neg d\},\\ 15 : \{a, b, c, d\}\},\end{aligned} \tag{3}$$

As second step, we give which clause from (2) subsumes which full-length clauses (clause: indices of subsumed full-length clauses):

$$\begin{aligned}\mathcal{SM} = \{&\{\neg a, b\} : 4,5,6,7,\ \{\neg a, c\} : 2,3,6,7,\ \{\neg b, a\} : 9,10,11,12,\\ &\{\neg b, c\} : 2,3,10,11,\ \{\neg b, d\} : 1,3,9,11,\ \{\neg c, a\} : 8,9,13,14,\\ &\{\neg c, d\} : 1,5,9,13,\ \{\neg d, a\} : 8,10,12,14\}.\end{aligned} \tag{4}$$

From this one can check that (2) subsumes all full-length clauses, except the black and the white clause, hence, it has only two solutions, the white and the black assignment.

5 The Weak Model of Communication Graphs

We define the weak model of communication graphs. Let us assume, that $\mathcal{D} = (\{a,b,c\},\{(a,b),(a,c)\})$, as in the previous section. The weak model of $\mathcal{D}$ contains the clause $(a \implies b) \vee (a \implies c)$, which means that if a node can send a message to more than one nodes, then it can send the message only to one of them. Note, that $(a \implies b) \vee (a \implies c)$ is equivalent to $(\neg a \vee b \vee c)$.

The only problem with this representation is that the message can be trapped if a graph contains a cycle. Let us assume that we have a cycle with two nodes, n_1 and n_2. Then n_1 may send the message to n_2, and n_2 to n_1, and so on, which means that other nodes could never get the message. This is not good, since our goal is to send a message to each node, if it is possible.

Let us add a new edge to the graph $(\{a,b,c\},\{(a,b),(a,c)\})$. The new edge should go from b to a. Now we have a cycle with the nodes a and b. Now we have to add a clause to our model which ensures that if b sends a message to a, and a can send a message to b or c, then a should not send back the message to b, but it has to send it to c: $((b \implies a) \wedge (a \implies (b \vee c)) \implies (a \implies c)$. Note, that this is equivalent to the clause: $(\neg a \vee \neg b \vee c)$.

In a more general way, node a which has outgoing edges to nodes $b_1, b_2, \ldots, b_k$ is represented by the clause: $(\neg a \vee b_1 \vee b_2 \vee \cdots \vee b_k)$; and the cycle $a_1, a_2 \ldots a_n, a_1$ with exit points $b_1, b_2, \ldots, b_m$ is represented by the clause: $(\neg a_1 \vee \neg a_2 \vee \cdots \vee \neg a_n \vee b_1 \vee b_2 \vee \cdots \vee b_m)$. We call this model as the weak model of communication graphs.

We define it also formally: Let $\mathcal{D}$ be a communication graph. Let $\mathcal{V}$ be the set of vertices of $\mathcal{D}$ and $\mathcal{E}$ the set of edges of $\mathcal{D}$. Since $\mathcal{D}$ is a communication graph, we know that elements of $\mathcal{V}$ can be used as positive

literals. Then we define the following notions:

$$OutE(a) := \{b \mid (a,b) \in \mathcal{E}\}.$$

$$NodeRep(a) := \{\neg a\} \cup OutE(a).$$

$$NodeRep := \{NodeRep(a) \mid a \in \mathcal{V} \wedge OutE(a) \neq \emptyset\}.$$

$$Cycles := \{(a_1, a_2, \ldots, a_k) \mid k = 1 \vee \forall_{i=1\ldots k}(a_{(i \bmod k)+1} \in OutE(a_i)\},$$

and for any two elements of $Cycles$ they cannot be equal as a set.

$$ExitPonts(\ (a_1, a_2, \ldots a_k)\ ) := \{b \mid \exists_{i=1\ldots k}(b \in OutE(a_i)) \wedge \neg\exists_{j=1\ldots k}(b = a_j)\}.$$

$$CycleRep := \{\neg C \cup ExitPonts(C) \mid C \in Cycles \wedge ExitPonts(C) \neq \emptyset\}.$$

$$\mathcal{WM} := NodeRep \cup CycleRep.$$

$\mathcal{WM}$ is the weak model of $\mathcal{D}$.

Note, that $NodeRep$ is a subset of $CycleRep$, because we take each node itself as a cycle, see the constraint $k = 1 \vee \ldots$ in the definition of $CycleRep$. So alternatively we can define $\mathcal{WM}$ as follows: $\mathcal{WM} := CycleRep$. We do not use this observation in this work, although, some proofs would be shorter.

Please note, that each clause in $\mathcal{WM}$ contains at least one positive and one negative literal, because a node is only represented if it has an outgoing edge, and a cycle is only represented if it has an exit point, therefore, $\mathcal{SM}$ is satisfiable for any communication graph $\mathcal{D}$.

Fig. 4. Two simple communication graphs.

Figure 4 shows two simple communication graphs. Their $NodeRep$ values are: $\{\{\neg a, b, c\}\}$, and $\{\{\neg a, b, c, d\}\}$, respectively.

Figure 2 shows a communication graph with 3 cycles:

$$(a, b), (a, b, c), (a, c)$$

. So its weak model is (after each clause we list the indices of subsumed full-length clauses from (3)):

$$\begin{aligned}\mathcal{WM} = \{&\{\neg a, b, c\} : 6{,}7,\ \{\neg b, a, c, d\} : 11,\ \{\neg c, a, d\} : 9{,}14,\\ &\{\neg a, \neg b, c, d\} : 3,\ \{\neg a, \neg b, \neg c, d\} : 1,\ \{\neg a, \neg c, b, d\} : 5\}.\end{aligned} \tag{5}$$

Note, that since d has no child node, it does not occur as a negative literal in the model.
Note, that since the communication graph from Figure 2 is not strongly connected, its weak model (5) is not a Black-and-White SAT problem. For example $\{\neg a, \neg b, \neg c, d\}$ is a solution of the SAT problem in (5),

As a second example we show the weak model of the communication graph from Figure 3. In this graph we have 6 cycles:

$$(a, b), (a, b, c), (a, c), (a{,}b{,}d), (a, c, d), (a, b, c, d)$$

. The last cycle contains all vertices, so it has no exit point, therefore no clause generated for it. Note, that in this graph we have an edge from d to a. The corresponding clauses are the last 3 clauses in this example, the last 2 clauses corresponds to the new cycles. The rest of the clauses is the same as in the previous model. So the weak model is (after each clause we list the indices of subsumed full-length clauses from (3)):

$$\begin{aligned}\mathcal{WM} = \{&\{\neg a, b, c\} : 6{,}7,\ \{\neg b, a, c, d\} : 11,\ \{\neg c, a, d\} : 9{,}14,\\ &\{\neg a, \neg b, c, d\} : 3,\ \{\neg a, \neg b, \neg c, d\} : 1,\ \{\neg a, \neg c, b, d\} : 5\\ &\{\neg d, a\} : 8, 10, 12, 14,\ \{\neg a, \neg b, \neg d, c\} : 2,\ \{\neg a, \neg c, \neg d, b\} : 4\}.\end{aligned} \tag{6}$$

Note, that since the communication graph from Figure 3 is strongly connected, its weak model is a Black-and-White SAT problem, i.e., the SAT problem in (6) has only these two solutions: $\{a, b, c, d\}$, and $\{\neg a, \neg b, \neg c, \neg d\}$.

Each cycle was a simple cycle In the above 2 examples. A cycle is simple iff only the first and last vertices are repeated, So we might think that it is enough to consider simple cycles. The next example shows a case when we need to consider also a non simple cycle.

On Figure 5 we can see a strongly connected communication graph which contains 3 simple cycles: $(a{,}b)$, $(a{,}b{,}c{,}d)$, $(b{,}c)$, and 1 non-simple cycle: $(a{,}b{,}c{,}b)$. As a third example we give the weak model of this communication graph (after each clause we list the indices of subsumed full-length clauses from (3)):

$$\begin{aligned}\mathcal{WM} = \{&\{\neg a, b\} : 4{,}5{,}6{,}7,\ \{\neg b, a, c\} : 10{,}11,\ \{\neg c, b, d\} : 5{,}13,\\ &\{\neg d, a\} : 8, 10, 12, 14,\ \{\neg a, \neg b, c\} : 2, 3,\ \{\neg b, \neg c, a, d\} : 9,\\ &\{\neg a, \neg b, \neg c, d\} : 1\}.\end{aligned} \tag{7}$$

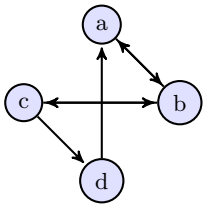

Fig. 5. A strongly connected communication graph with 4 vertices, 3 simple cycles, 1 non simple cycle.

Note, that the last clause is generated from the non-simple cycle, and that full-length clause is not subsumed by any other clause.

6 The Balatonboglár Model of Communication Graphs

Since the weak model does not result in a 3-SAT problem we introduce also the Balatonboglár model, which is a simplified version of the weak model, which uses the trick that instead of detecting each cycle in the graph, it generates from each path a, b, c the following 3-clause: $(\neg a \vee \neg b \vee c)$ even if there is no cycle which contains the vertices a and b. This simplification allows very fast 3-SAT model generation from a directed graph, and the model will be Black-and-White 3-SAT if and only if the input directed graph is strongly connected.

We define the Balatonboglár model of communication graphs as follows. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{V}$ be the set of vertices of $\mathcal{D}$ and $\mathcal{E}$ the set of edges of $\mathcal{D}$. Since $\mathcal{D}$ is a communication graph, we know that elements of $\mathcal{V}$ can be used as positive literals. Then we define the following notions:

$$NodeRep2 := \{\{\neg a, b\} \mid a \in \mathcal{V} \wedge \{b\} = OutE(a)\}.$$

$$NodeRep3(a) := \{\neg a, b, c\}, where\ a \in \mathcal{V} \wedge b \in OutE(a) \wedge c \in OutE(a) \wedge b \neq c.$$

$$NodeRep3 := \{\ NodeRep3(a) |\ a \in \mathcal{V}\}.$$

$$TagForward := \{\{\neg a, \neg b, c\} \mid a \in \mathcal{V} \wedge b \in OutE(a) \wedge c \in OutE(b) \wedge a \neq b \neq c\}.$$

$$\mathcal{BM} := NodeRep2 \cup NodeRep3 \cup TagForward.$$

$\mathcal{BM}$ is a Balatonboglár model of $\mathcal{D}$.

Note, that $NodeRep3(a)$ is not a deterministic function, it is just any 3 length subset of $NodeRep(a)$.

The name $TagForward$, comes from the child game Tag: there is a child who has to catch someone else by touching he or she. This action is called tag. There is a rule, the tagged one cannot tag back, he or she must tag someone else, so must tag forward.

As a first example, we give the Balatonboglár model of the graph from Figure 2 (after each clause we list the indices of subsumed full-length clauses from (3)):

$$\begin{aligned}\mathcal{BM} = \{&\{\neg a, b, c\} : 6{,}7,\ \{\neg b, c, d\} : 3{,}11,\ \{\neg c, a, d\} : 9{,}13,\\ &\{\neg a, \neg b, c\} : 2{,}3,\ \{\neg a, \neg b, d\} : 1{,}3,\ \{\neg a, \neg c, b\} : 4{,}5,\\ &\{\neg a, \neg c, d\} : 1{,}5,\ \{\neg b, \neg c, a\} : 8{,}9,\ \{\neg b, \neg c, d\} : 1{,}9\}.\end{aligned} \tag{8}$$

Note, that b has 3 child nodes, so instead of $\{\neg b, c, d\}$, we could use either $\{\neg b, a, c\}$ or $\{\neg b, a, d\}$. Note also, that Figure 2 is not strongly connected, so its Balatonboglár model is not a Black-and-White SAT problem. For example $\{\neg a, \neg b, \neg c, d\}$ is a solution of the SAT problem in (8).

As a second example, we show the Balatonboglár model of the communication graph from Figure 3. Note, that in this graph we have an edge from d to a. The corresponding clauses are the last 3 clauses in this Balatonboglár model: (after each clause we list the indices of subsumed full-length clauses from (3)):

$$\begin{aligned}\mathcal{BM} = \{&\{\neg a, b, c\} : 6{,}7,\ \{\neg b, c, d\} : 3{,}11,\ \{\neg c, a, d\} : 9{,}13,\\ &\{\neg d, a\} : 8{,}10{,}12{,}14,\ \{\neg a, \neg b, c\} : 2{,}3,\ \{\neg a, \neg b, d\} : 1{,}3,\ \{\neg a, \neg c, d\} : 1{,}5,\\ &\{\neg b, \neg c, a\} : 8{,}9,\ \{\neg b, \neg c, d\} : 1{,}9,\ \{\neg b, \neg d, a\} : 8{,}10,\\ &\{\neg c, \neg a, b\} : 4{,}5,\ \{\neg c, \neg d, a\} : 8{,}12,\ \{\neg d, \neg a, b\} : 4{,}6,\ \{\neg d, \neg a, c\} : 3{,}7\}.\end{aligned} \tag{9}$$

Note, that since the communication graph from Figure 3 is strongly connected, its Balatonboglár model is a Black-and-White SAT problem, i.e., the SAT problem in (9) has only these two solutions: $\{a, b, c, d\}$, and $\{\neg a, \neg b, \neg c, \neg d\}$.

As a third example, we show the Balatonboglár model of the communication graph from Figure 5 (after each clause we list the indices of subsumed full-length clauses from (3)):

$$\begin{aligned}\mathcal{BM} = \{&\{\neg a, b\} : 4{,}5{,}6{,}7,\ \{\neg b, a, c\} : 10{,}11,\ \{\neg c, b, d\} : 5{,}13,\\ &\{\neg d, a\} : 8{,}10{,}12{,}14,\ \{\neg a, \neg b, c\} : 2{,}3,\ \{\neg b, \neg c, a\} : 8{,}9,\\ &\{\neg b, \neg c, d\} : 1{,}9,\ \{\neg c, \neg d, a\} : 8{,}12,\ \{\neg a, \neg d, b\} : 4{,}6\}.\end{aligned} \tag{10}$$

Since there is no node which has more than two child nodes, there is no other possible Balatonboglár model for this graph.

Note, that since the communication graph from Figure 5 is strongly connected, its Balatonboglár model (10) is a Black-and-White SAT problem.

Our long term goal is to find out, how to represent a 3-SAT problem as a directed graph. This model does not tell us how to do that, it gives only a link from the field of directed graphs to the field of 3-SAT problems, which might help us to find out in the future, how to represent a 3-SAT problem as a directed graph. So this remains an open question, which seems to be a very difficult one, because if we were able to translate a 3-SAT problem into a directed graph, then we could translate it to a 2-SAT problem using the result of our previous work, which means that we would have a 3-SAT to 2-SAT converter.

7 Theoretical results

This section is the main contribution of this work. We prove that the new models, the weak one and the Balatonboglár one, have the same property as the strong one, i.e., the model of a communication graph is a Black-and-White SAT problem iff the graph is strongly connected. To be able to prove this we need some auxiliary lemmas. The first one states that the weak model has at least two solutions, the white one and the black one.

Lemma 1. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Then $\mathcal{WM}$ has at least two solutions, namely the white assignment (WW) and the black assignment (BB).*

Proof. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Since there is a positive and also a negative literal in each clause of $\mathcal{WM}$, the white assignment (WW) and also the black assignment (BB) are solutions for $\mathcal{WM}$. □

The second lemma states that if the weak model has only two solutions, then each cycle in the communication graph has at least one exit point, except the cycle which contains all vertices of the graph.

Lemma 2. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Assume that $\mathcal{WM}$ has only two solutions. Then for each cycle $C \in Cycles$ we have that $ExitPonts(C)$ is not empty or C contains all vertices of $\mathcal{D}$.*

Proof. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Assume that $\mathcal{WM}$ has only two solutions. Lemma 1 makes sure that

the solutions are WW and BB. Assume that our goal is not true, i.e., there is a cycle $C = (a_1, a_2, \ldots, a_k)$, which has no exit point and does not contain all vertices of $\mathcal{D}$. We show that this assumption leads to a contradiction. Let us construct the following assignment: $\{a_1, \ldots, a_k, \neg b_1, \ldots \neg b_m\}$, where $\{b_1, b_2, \ldots, b_m\} = \mathcal{V} \setminus C$, where $\mathcal{V}$ is the set vertices of $\mathcal{D}$. Since C has no exit point there is no corresponding clause generate by $CycleRep$. Since there is no clause which would falsify the constructed assignment, so this is a solution of $\mathcal{WM}$. But this contradicts the assumption that $\mathcal{WM}$ has only two solutions, namely WW and BB. So our original statement is true. □

The next lemma states that a Black-and-White SAT problem may not contain pure literals.

Lemma 3. *If $\mathcal{S}$ is a Black-and-White SAT problem, then it contains no pure literal.*

Proof. Assume $\mathcal{S}$ is a Black-and-White SAT problem. Then, by definition of a Black-and-White SAT problem, we know that $\mathcal{S}$ has only two solutions, WW and BB. Since there is no other solution of $\mathcal{S}$ and $\neg WW = BB$, we obtain that $\mathcal{S}$ may not contain a pure literal. □

The next lemma states that if we have a complete graph, then its weak model is a Black-and-White SAT problem. This result is somehow natural, since the weak model of a complete graph is the biggest possible what we can generate and the weak model has always at least two solutions.

Lemma 4. *If $\mathcal{D}$ is a complete communication graph, and $\mathcal{WM}$ is the weak model of $\mathcal{D}$, then $\mathcal{WM}$ is a Black-and-White SAT problem.*

Proof. Assume $\mathcal{D}$ is a complete communication graph. Assume $\mathcal{WM}$ is the weak model of $\mathcal{D}$. Since $\mathcal{D}$ is complete, for any vertex a we have that $OutE(a)$ contains all other vertices, so $NodeRep$ contains all full-length clauses with 1 negative literal. Since any two vertices create a cycle where the exit points are the rest of the vertices, $CycleRep$ contains all full-length clauses with 2 negative literals. The same is true for any $k = 2 \ldots n-1$ vertices, so $CycleRep$ contains all full-length clauses with $2, \ldots, n-1$ negative literals. But $\mathcal{WM}$ do not subsumes the full-length clause with 0 negative literal, i.e., the white clause (WW), neither the full-length clause with n negative literal, i.e., the black clause (BB). This means that $\mathcal{WM}$ has only two solutions, the white and the black assignment, i.e., $\mathcal{WM}$ is a Black-and-White SAT problem. □

Now comes one of our main contributions, the theorem which states, that the weak model of a strongly connected graph is a Black-and-White SAT problem. Which means that if the graph is strongly connected, then the weak model has the same power as the strong model. Since the proof is rather technical, we give also a high-level trace of it.

From left to right we prove the theorem in a constructive way: Let us have a sequence of vertices which can be built by Lemma 3 and by the construction of $NodeRep$. Eventually we will find a cycle at the end of this sequence. Lemma 2 ensures that there is an exit point from this cycle, which either results in a bigger cycle, or in a longer sequence. Using these two steps eventually we construct a strongly connected component. From Lemma 2 it follows that it is also maximal (otherwise there would be an exit point from it which would allow to do the above steps), i.e., the graph is strongly connected.

From right to left we use induction: We start the proof from a complete graph. We know that complete graphs are also strongly connected. As the induction step, we drop an edge from this graph such that it remains strongly connected. We assume that before the drop the weak model was Black-and-White, which is true by Lemma 4. We show that it remains Black-and-White also after the drop. To show this, we show that each clause in the model after the drop is a subset of some clause from the mode before the drop. So the new model may not have more solutions then the old one. But the new model has to have at least two solutions by Lemma 1. So the new model is Black-and-White. Since our graph can be constructed by dropping edges from a complete graph, which has the same set of vertices, its weak model is also Black-and-White.

Theorem 2. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Then $\mathcal{WM}$ is a Black-and-White SAT problem if and only if $\mathcal{D}$ is strongly connected.*

Proof. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. We show that $\mathcal{WM}$ is a Black-and-White SAT problem if and only if $\mathcal{D}$ is strongly connected.

(From left to right:) We assume $\mathcal{WM}$ is a Black-and-White SAT problem, we show by construction that $\mathcal{D}$ is strongly connected.
(R0) Let a_1 be a vertex from $\mathcal{D}$. Let $j = 1$.
(R1) Now we have a sequence of vertices: $a_1, \ldots, a_j$. We work with its last vertex: a_j. From Lemma 3 we know that literal a_j is not pure in $\mathcal{WM}$, so $\neg a_j$ is present in $\mathcal{WM}$. So there must be a clause in $\mathcal{WM}$ generated be $NodeRep$ for vertex a_j. Since $NodeRep$ is a subset of $CycleRep$, there-

fore, by Lemma 2, there is a clause $\{\neg a_j, b_1, \ldots, b_k\}$ in $\mathcal{WM}$. Let a_{j+1} be a vertex from $b_1, \ldots, b_k$ such that a_{j+1} is not in $\{a_1, \ldots, a_j\}$. If this is possible, then let $j = j + 1$ and goto (R1). Note that this is not always possible because $\mathcal{D}$ is a final graph, so eventually each vertex from $b_1, \ldots, b_k$ must be in the set $\{a_1, \ldots, a_j\}$. In this case let a_{j+1} be a vertex from $b_1, \ldots, b_k$ such that $a_{j+1} = a_i$, where $i < j$ and i is the smallest possible. This means that we have i such that $i < j$ and $a_i = a_{j+1}$, i.e., $(a_i, \ldots, a_j)$ is a cycle. Goto (R2).

(R2) Now we have a sequence of vertices: $a_1, \ldots, a_j$, and the end of this sequence from a_i to a_j is a cycle. If $i = 1$ and the cycle contains all the vertices from $\mathcal{D}$, then goto (R3). Otherwise by creation of $CycleRep$ and by Lemma 2 we know that $\mathcal{WM}$ contains a clause $\{\neg a_i, \ldots, \neg a_j, b_1, \ldots, b_m\}$, where $b_1, \ldots, b_m$ are exit points of the cycle $(a_i, \ldots, a_j)$. Let a_{j+1} be a vertex from $b_1, \ldots, b_m$ such that a_{j+1} is not present in $a_1, \ldots, a_j$. If this is possible, then let $j = j + 1$, and goto (R1). Note that this is not always possible because $\mathcal{D}$ is a final graph. so eventually each vertex from $b_1, \ldots, b_k$ must be in the set $\{a_1, \ldots, a_j\}$. In this case let a_{j+1} be a vertex from $b_1, \ldots, b_m$, such that $a_{j+1} = a_{i'}$ and i' is the smallest possible. Since $b_1, \ldots, b_m$ are exit points of the cycle $(a_i, \ldots, a_j)$, we know that i' is smaller than i because each vertex from $b_1, \ldots, b_m$ is different from $a_i, \ldots, a_j$. This means that we have i' such that $i' < j$, and $i' < i$ and $a_{i'} = a_{j+1}$, i.e., $(a'_i, \ldots, a_j)$ is a cycle. Let $i = i'$, and goto (R2).
(R3) Eventually (R2) exists to (R3) because, by Lemma 2, each cycle has at least one exit point, which result in a bigger cycle, or in a longer $a_1, \ldots, a_j$ sequence of vertices. So eventually the sequence $a_1, \ldots, a_j$ will contain all vertices, and eventually i will be 1, so eventually we will find a cycle which contains all vertices. Hence, $\mathcal{D}$ is strongly connected.

(From right to left:) We assume $\mathcal{D}$ is strongly connected, we show by induction that $\mathcal{WM}$ is a Black-and-White SAT problem. Let H be the complete communication graph which has the same set of vertices as $\mathcal{D}$. We know from Lemma 4 that the weak model of H is a Black-and-White SAT problem. This is our induction base. It is well known that complete graphs are also strongly connected. We create G from H by deleting edges from H such that G remains strongly connected. Let Z be the weak model of G. Our induction hypothesis is that Z is a Black-and-White SAT problem. Our induction step is that we delete an edge from G such that it remains strongly connected. Let G' be this graph. Let Z' be the weak model of G'. We show that Z' a Black-and-White SAT problem. To show this, we show that $Z' \geq Z$, i.e., Z' subsumes all clauses from Z.

Without loss of generality let us assume that we deleted the edge from vertex a to vertex b. Each clause from Z is the same as the clauses of Z' excepted the ones generated by $CycleRep$ for those cycles which contain a. From those clauses let $D = \{\neg a_1, \ldots, \neg a_k, b_1, \ldots, b_m\}$ be an arbitrary but fixed one, where $(a_1, \ldots, a_k)$ is a cycle in Z and $b_1, \ldots, b_m$ are its exit points. We know that $k >= 1$ and $a = a_i$ for some $i \in \{1, \ldots, k\}$, and b is one of the other vertices from this clause. There are the following cases:

- If b is one of the exit points and some other vertex from the cycle has an edge to b, then Z' contains the same clause: D.
- If b is one of the exit points, say $b = b_p$, where $p \in \{1, \ldots, m\}$, and no other vertex from the cycle has an edge to b, then Z' contains the clause $\{\neg a_1, \ldots, \neg a_k, b_1, \ldots, b_{p-1}, b_{p+1}, \ldots b_m\}$, which is a subset of D. Note, that the vertex sequence $b_1, \ldots, b_{p-1}, b_{p+1}, \ldots b_m$ contains some vertices, because G' is a strongly connected graph, i.e., each cycle has an exit point, unless it contains all vertices.
- If b is one of the other vertices of the cycle, but b is not the next vertex after a in this cycle, then Z' contains the same clause, D.
- If b is one of the other vertices of the cycle, and b is the next vertex after a in this cycle, then $a_1, \ldots, a_k$ is not a cycle in Z'. If there is no other child vertex of a in $\{a_1, \ldots, a_k\}$, then the clause generated by $NodeRep$ for a in G' is a subset of D. If there are other child vertices of a in $\{a_1, \ldots, a_k\}$, then there must be a subset of $\{a_1, \ldots, a_k\}$ which contains all other child vertices of a except b, and which is a cycle in G'. The clause generated by $CycleRep$ for this cycle in G' is a subset of D.

We showed that Z' subsumes all clauses from Z. This means that $Z' \geq Z$, i.e., Z' may have no other solutions than Z. By our induction hypothesis Z has only two solutions BB and WW. We know from Lemma 1 that BB and WW are solutions for Z', because Z' is a weak model. In addition, since Z' may have no other solution, Z' is a Black-and-White SAT problem. This means that the weak model of any strongly connected communication graph is a Black-and-White SAT problem. Hence, $\mathcal{WM}$ is a blacked-and-white SAT problem, because $\mathcal{D}$ is strongly connected and $\mathcal{WM}$ is its weak model. □

The following lemma states that $\mathcal{SM}$ entails $\mathcal{WM}$ for any communication graph. This means that any solution of $\mathcal{SM}$ is also a solution of $\mathcal{WM}$. To show this, it is enough to show that $\mathcal{SM}$ subsumes all clauses from $\mathcal{WM}$.

Lemma 5. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Then $\mathcal{SM} \geq \mathcal{WM}$.*

Proof. Assume $\mathcal{D}$ is a communication graph, $\mathcal{SM}$ is the strong model of $\mathcal{D}$, and $\mathcal{WM}$ is the weak model of $\mathcal{D}$. We show that $\mathcal{SM} \geq \mathcal{WM}$. To show this it is enough to show that $\mathcal{SM}$ subsumes all clauses from $\mathcal{WM}$. Let C is an arbitrary but fixed element of $\mathcal{WM}$. If C is an element of $NodeRep$, then, without loss of generality, we may assume that $C = \{\neg a, b_1, b_2, \ldots, b_k\}$, where $k > 0$. From the construction of $NodeRep$ we know that in $\mathcal{D}$ there are the following edges: $(a, b_1), (a, b_2), \ldots, (a, b_k)$. Since $\mathcal{SM}$ is generated from the same graph, by its definition we know that $\mathcal{SM}$ contains the clauses $\{\neg a, b_1\}, \{\neg a, b_2\}, \ldots, \{\neg a, b_k\}$ and all of these ones subsume C.

If C is an element of $Cycles$, then, without loss of generality, we may assume that $C = \{\neg d_1, \neg d_2, \ldots, \neg d_m, e_1, e_2, \ldots, e_n\}$, where $m > 0$, and $n > 0$. From the construction of $Cycles$ we know that in $\mathcal{D}$ there is an edge: (d, e) such that d is one of the vertices from $d_1, d_2, \ldots, d_m$, and e is one of the vertices from $e_1, e_2, \ldots, e_m$. Since $\mathcal{SM}$ is generated from the same graph, by its definition we know that $\mathcal{SM}$ contains the clauses $\{\neg d, e\}$, which subsumes C.

Hence, $\mathcal{SM} \geq \mathcal{WM}$. □

The following lemma states that $\mathcal{SM}$ and $\mathcal{WM}$ are equivalent if the represented communication graph is strongly connected or there are no branches in the graph. If the communication graph is strongly connected, then both $\mathcal{SM}$ and $\mathcal{WM}$ are Black-and-White SAT problems, so they are equivalent.

Lemma 6. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$.*
If
(I) $\mathcal{D}$ is strongly connected, or
(II) each vertex in $\mathcal{D}$ has only one child,
then $\mathcal{SM}$ and $\mathcal{WM}$ are equivalent.

Proof. In case of (I) we know that $\mathcal{D}$ is strongly connected, so by Theorem 1, and by Theorem 2 we know that $\mathcal{SM}$ and $\mathcal{WM}$ are Black-and-White SAT problems, so by the definition of clause set equivalent-ness they are equivalent. In case of (II) we have that $NodeRep = \mathcal{SM}$ and $NodeRep \subseteq \mathcal{WM}$. From this we obtain that $\mathcal{WM} \geq \mathcal{SM}$. From Lemma 5 we know that $\mathcal{SM} \geq \mathcal{WM}$. Hence, $\mathcal{SM}$ and $\mathcal{WM}$ are equivalent. □

The following lemma gives a criterion for the communication graph which makes the strong model stronger than the weak model. This lemma is not needed to prove the following theorems but it is still interesting.

Lemma 7. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Then $\mathcal{SM} > \mathcal{WM}$ iff $\mathcal{D}$ is not strongly connected, and there is at least one vertex in it which has more than one child vertex.*

Proof. (From left to right:) We assume $\mathcal{SM} > \mathcal{WM}$, so $\mathcal{SM}$ and $\mathcal{WM}$ are not equivalent. Since the right side of Lemma 6 is negated, the negation of the left side of Lemma 6 is implied. Hence, $\mathcal{D}$ is not strongly connected, and there is at least one vertex in it which has more than one child vertex.

(From right to left:) Assume $\mathcal{D}$ is not strongly connected, and there is at least one vertex in it, say a, which has more than one child vertex, say $b_1, b_2, \ldots, b_k$, where $k > 1$. In this case $WW \setminus \{b_1\} \cup \{\neg b_1\}$ and $BB \setminus \neg a, \neg b_k \cup \{a, b_k\}$ are solutions for $\mathcal{WM}$ but they are not solutions of $\mathcal{SM}$, since they do not satisfies the clause $\{\neg a, b_1\}$ from $\mathcal{SM}$. So $\mathcal{SM}$ and $\mathcal{WM}$ are not equivalent. From this and from Lemma 5 we obtain, that Then $\mathcal{SM} > \mathcal{WM}$. □

The following theorem, the so called Transitions Theorem states, that any "transition" between $\mathcal{SM}$ and $\mathcal{WM}$ is a Black-and-White SAT problem iff the communication graph is strongly connected. This is one of the main results of this work.

Theorem 3 (Transitions Theorem). *If for all communication graphs $\mathcal{D}$ we have $\mathcal{SM} \geq \mathcal{MM} \geq \mathcal{WM}$, where $\mathcal{SM}$ is the strong model of $\mathcal{D}$, and $\mathcal{WM}$ is the weak model of $\mathcal{D}$, and $\mathcal{MM}$ is an arbitrary but fixed model of $\mathcal{D}$, then $\mathcal{MM}$ is a Black-and-White SAT problem iff $\mathcal{D}$ is strongly connected.*

Proof. Assume that $\mathcal{D}$ is a communication graph. Assume that $\mathcal{SM}$ is the strong model of $\mathcal{D}$, and $\mathcal{WM}$ is the weak model of $\mathcal{D}$, and $\mathcal{MM}$ is an arbitrary but fixed model of $\mathcal{D}$. We show that $\mathcal{MM}$ is a Black-and-White SAT problem iff $\mathcal{D}$ is strongly connected.

(From left to right:) Assume $\mathcal{MM}$ is a Black-and-White SAT problem. From this and from the assumption that $\mathcal{SM} \geq \mathcal{MM}$, we have that $\mathcal{SM}$ is a Black-and-White SAT problem, since any strong model has always at least two solutions, the white assignment and the black one, see [2]. From this and from Theorem 1 we have that $\mathcal{D}$ is strongly connected.

(From right to left:) Assume $\mathcal{D}$ is strongly connected. Then by Lemma 6 we know that $\mathcal{SM}$ and $\mathcal{WM}$ are equivalent. From this and from the assumption that $\mathcal{SM} \geq \mathcal{MM} \geq \mathcal{WM}$ it follows that $\mathcal{SM}$ and $\mathcal{MM}$ and $\mathcal{WM}$ are equivalent. Since $\mathcal{D}$ is strongly connected by Theorem 2 we know that $\mathcal{WM}$ is a Black-and-White SAT problem. We know that $\mathcal{MM}$ and $\mathcal{WM}$ are equivalent, hence, $\mathcal{MM}$ is a Black-and-White SAT problem. □

The following theorem shows that the Balatonboglár model of a communication graph is a transition between the strong and the weak model. Actually, Balatonboglár is a small city in Hungary next to lake Balaton. We spent quite a few time next to the lake thinking on a model which is 3-SAT and Black-and-White if the directed graph is strongly connected. As the next theorem shows we was successful. As the respect of that great time we gave the name Balatonboglár to that model.

Theorem 4. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{BM}$ be a Balatonboglár model of $\mathcal{D}$. Then $\mathcal{BM}$ is a Black-and-White 3-SAT problem if and only if $\mathcal{D}$ is strongly connected.*

Proof. Assume $\mathcal{D}$ is a communication graph. Assume $\mathcal{BM}$ is a Balatonboglár model of $\mathcal{D}$. We show $\mathcal{BM}$ is a Black-and-White 3-SAT problem if and only if $\mathcal{D}$ is strongly connected. Because of the construction of $\mathcal{BM}$ it is a 3-SAT problem for any communication graph. Let $\mathcal{WM}$ be the weak model of $\mathcal{D}$. Let $\mathcal{SM}$ be the strong model of $\mathcal{D}$. Now we show that $\mathcal{BM}$ is a Black-and-White 3-SAT problem if and only if $\mathcal{D}$ is strongly connected. From the Transitions Theorem (Theorem 3) we know that to show this, it is enough to show that $\mathcal{SM} \geq \mathcal{BM} \geq \mathcal{WM}$.

(I) As a first step, we show that $\mathcal{SM} \geq \mathcal{BM}$. To show this it is enough to show that $\mathcal{SM}$ subsumes all clauses from $\mathcal{BM}$. Let C be an arbitrary but fixed element of $\mathcal{BM}$. This means that C is either an element of $NodeRep2$, or $NodeRep3$, or $TagForward$. If C is an element of $NodeRep2$, then C is also element of $\mathcal{SM}$, so this case is trivial. If C is an element of $NodeRep3$, then, without loss of generality, we may assume that $C = \{\neg a, b, c\}$, where a is a vertex in $\mathcal{D}$ and $b \in OutE(a)$ and $c \in OutE(a)$ and $b \neq c$. From the construction of $NodeRep3$ we know that in $\mathcal{D}$ there are the following edges: $(a, b), (a, c)$. Since $\mathcal{SM}$ is generated from the same graph, by its definition we know that $\mathcal{SM}$ contains the clauses $\{\neg a, b\}, \{\neg a, c\}$ and both of them subsume C. If C is an element of $TagForward$, then, without loss of generality, we may assume that $C = \{\neg d, \neg e, f\}$, where d is a vertex in $\mathcal{D}$, and $e \in OutE(d)$ and

$f \in OutE(e)$. From the construction of $TagForward$ we know that in $\mathcal{D}$ there are the following edges: $(d, e), (e, f)$. Since $\mathcal{SM}$ is generated from the same graph, by its definition we know that $\mathcal{SM}$ contains the clause $\{\neg e, f\}$ which subsumes C.

(II) As a second step, we show $\mathcal{BM} \geq \mathcal{WM}$. To show this it is enough to show that $\mathcal{BM}$ subsumes all clauses from $\mathcal{WM}$. Let C be an arbitrary but fixed element of $\mathcal{WM}$. This means that C is either an element of $NodeRep$ or $Cycles$. If C is an element of $NodeRep$, then, without loss of generality, we may assume that $C = \{\neg a, b_1, b_2, \ldots, b_k\}$, where $k > 0$. From the construction of $NodeRep$ we know that in $\mathcal{D}$ there are the following edges: $(a, b_1), (a, b_2), \ldots, (a, b_k)$. Since $\mathcal{BM}$ is generated from the same graph, by its definition we know that $\mathcal{BM}$ contains the clause $\{\neg a, b_1\}$, if $k = 1$, or $\{\neg a, b, c\}$, where b and c are different vertices from $b_1, b_2, \ldots, b_k$, and that clause subsumes C. If C is an element of $Cycles$, then, without loss of generality, we may assume that $C = \{\neg d_1, \neg d_2, \ldots, \neg d_m, e_1, e_2, \ldots, e_n\}$, where $m > 0$, and $n > 0$. If $m = 1$ then C is an element of $NodeRep$, which case is already considered, so we may assume that $m > 1$. From the construction of $Cycles$ and from $m > 1$ we know that in $\mathcal{D}$ there are these edges: (d_i, d_{i+1}) and (d_{i+1}, e) such that d_i and d_{i+1} are one of the vertices from $d_1, d_2, \ldots, d_m$, and e is one of the vertices from $e_1, e_2, \ldots, e_m$. Since $\mathcal{BM}$ is generated from the same graph, by its definition we know that $\mathcal{BM}$ contains the clauses $\{\neg d_i, \neg d_{i+1}, e\}$ which subsumes C.

Hence, $\mathcal{BM}$ is a Black-and-White 3-SAT problem if and only if $\mathcal{D}$ is strongly connected. □

8 Future work

This work is purely theoretical, there is no usage described here. This does not mean that this is not possible. In this section we list some possible usage.

The BaW 1.0 SAT solver solves the SAT problems generated by our strong model in linear time [3]. The proof of Theorem 2 suggests that we might find a similar algorithm for SAT problems generated by our weak model.

Most probably this is not possible in case of Balatonboglár model, since in that model there is not enough information to reconstruct the input directed graph. It is an interesting question, in which cases the reconstruction is still possible. Most probably not in all cases, because

then we could translate a 3-SAT problem into a directed graph, and then that directed graph into a 2-SAT problem. Although this direction seems to be unrealistic, it is still interesting and very challenging for us.

One of the problems is what to do with those clauses in a 3-SAT problem, which subsumes the white clause, or the black clause. To solve this problem we need a technique which creates a Black-and-White SAT, if the input SAT problem is unsatisfiable.

Another interesting research idea is the following. We do not know whether the weak model is the weakest model for communication graphs which result in a black-and-what SAT problem in case of a strongly connected graphs, or is there a weaker model with this property.

References

1. B. Aspvall, M. F. Plass, R. E. Tarjan, *A Linear-Time Algorithm For Testing The Truth Of Certain Quantified Boolean Formulas*, Imformation Processing Letters, pp. 121–123, 1979.
2. Cs. Biró, G. Kusper *Equivalence of Strongly Connected Graphs and Black-and-White 2-SAT Problems*, Miskolc Mathematical Notes, Vol. 19, No. 2, pp. 755–768, 2018.
3. Cs. Biró, G. Kusper *BaW 1.0 - A Problem Specific SAT Solver for Effective Strong Connectivity Testing in Sparse Directed Graphs*, IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI 2018), pp. 160–165, 2018.
4. R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Trans. Comput., pp. 677–691, 1986.
5. R. A. Hearn, *Games, Puzzles, and Computation* PhD thesis, MIT, June 2006.
6. L. Hellerman, *A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits*, IEEE Transactions on Electronic Computers, pp. 198–223, 1963.
7. R. P. Langlands, *Problems in the theory of automorphic forms to Salomon Bochner in gratitude*, Lectures in Modern Analysis and Applications III, pp. 18–61, 1970.
8. S. Minato, *Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems*, Proceedings of the 30th International Design Automation Conference, pp. 272–277, 1993.
9. B. Nagy, *Truth-teller-liar puzzles and their graphs*, Central European Journal of Operations Research - CEJOR 11, pp. 57–72, 2003.
10. A. Weil, *Uber die Bestimmung Dirichletscher Reihen durch Funktionalgleichungen*, Mathematische Annalen, 149–156, 1967.
11. A. J. Wiles, *Modular elliptic curves and Fermats Last Theorem*, ANNALS OF MATH, pp. 443–551, 1995.

9 Annex 2. - Investigation of the Efficiency of Conversion of Directed Graphs to 3-SAT Problems

This is the submitted version of G. Kusper, Cs. Biró, T. Balla, *Investigation of the Efficiency of Conversion of Directed Graphs to 3-SAT Problems*, Proceedings of SACI-2020, DOI: 10.1109/SACI49304.2020.9118786, pp. 227–234, 2020. See also [93].

Investigation of the Efficiency of Conversion of Directed Graphs to 3-SAT Problems

1st Gábor Kusper
Faculty of Informatics
Eszterházy Károly University
Eger, Hungary
kusper.gabor@uni-eszterhazy.hu

2nd Csaba Biró
Faculty of Informatics
Eszterházy Károly University
Eger, Hungary
biro.csaba@uni-eszterhazy.hu

3rd Tamás Balla
Faculty of Informatics
Eszterházy Károly University
Eger, Hungary
balla.tamas@uni-eszterhazy.hu

***Abstract*—In our previous works we introduced several 2-SAT *(Strong Model)* and 3-SAT *(Weak Model, Balatonboglár Model and Simplified Balatonboglár Model)* models of directed graphs. We showed that Balatonboglár Model is a `Black-and-White 3-SAT` problem if and only if the represented directed graph is strongly connected. Balatonboglár Model generates a lot of so called Negative-Negative-Positive shaped clauses to represent cycles of the directed graph without detecting cycles, i.e., it can be generated fast but it is bigger than necessary. To overcome this problem, we have introduced the Simplified Balatonboglár Model. In this article, first, we give an example to illustrates the various conversion methods (models) and we briefly explain the theoretical background. After, we present the results of the latest version of CSFLOC on benchmarks generated by different conversion models. Finally, we show how the file sizes of benchmarks and the number of unused clauses changes as a function of the density of the represented directed graph.**

***Index Terms*—2-SAT, 3-SAT, directed graph, Strong Model, Balatonboglár Model, Simplified Balatonboglár Model**

I. Introduction

In logic the most natural representation of an edge of a directed graph, say $a \to b$, is to use implication, i.e., $a \implies b$, i.e., the edge $a \to b$ can be represented by the binary clause: $(\neg a \vee b)$. If a graph contains two edges: $a \to b$, and $a \to c$, then those can be represented by the formula: $(a \implies b) \wedge (a \implies c)$, which is equivalent to two 2-clauses: $(\neg a \vee b) \wedge (\neg a \vee c)$. We call this as the Strong Model ($\mathcal{SM}$) of directed graphs [4], [5], [10].

Our second model is the Weak Model ($\mathcal{WM}$) [10]. The idea is the following: If a graph contains two edges: $a \to b$, and $a \to c$, then those can be represented by the formula: $(a \implies b) \vee (a \implies c)$, which is equivalent to a 3-clause $(\neg a \vee b \vee c)$. We need to represent cycles of the graph, too. If $a_1 \to a_2 \to \cdots \to a_n \to a_1$ is a cycle with exit points $b_1, b_2, \ldots, b_m$, then this cycle can be represented by the clause: $(\neg a_1 \vee \neg a_2 \vee \cdots \vee \neg a_n \vee b_1 \vee b_2 \vee \cdots \vee b_m)$.

Our third model, the Balatonboglár Model ($\mathcal{BB}$) uses the trick that instead of detecting each cycle, it generates from each path $a \to b \to c$ the following 3-clause: $(\neg a \vee \neg b \vee c)$, which is a Negative-Negative-Positive ($\mathcal{NNP}$) shaped clause, even if there is no cycle which contains the vertices a and b. This simplification allows very fast 3-SAT problem generation from a directed graph, and the SAT instance will be a Black-and-White 3-SAT if and only if the input directed graph is strongly connected. On the other hand this trick generates a lot of superfluous clauses.

To overcome this problem, we introduced the Simplified Balatonboglár Model ($\mathcal{SBB}$) [11]. In this model we consider the the strongly connected components (`SCC`) of the directed graph [14], [15]. For each component we generate a cycle which contains all the nodes of the component. Then for each such big-cycle we generate $\mathcal{NNP}$ clauses along it. For example, for the cycle $(n_1, n_2, \ldots n_k)$ we generate the clauses $\{\{\neg n_1, \neg n_2, n_3\}, \ldots, \{\neg n_k, \neg n_1, n_2\}\}$.

We should also generate $\mathcal{NNP}$ clauses which link the components. After representing the big-cycles and the links between them, we can delete their edges and the opposite of those edges. The rest of the edges are "inner-edges" which may form cycles, so called "inner-cycles". We represent them one by one by an $\mathcal{NNP}$ clause, such that the positive literal should be a neighbour node on the big-cycle of one of the negative ones. Then we delete that cycle till there is no more inner-cycles. Other parts of the model are just the same as in case of $\mathcal{BB}$.

II. Related work

There are several links between directed graphs and propositional logic. There are several graph representations, the most prominent ones are: Implication graph [1] , And-Inverter Graph [7], Reduced Ordered Binary Decision Diagram [6], Zero-Suppressed Binary Decision Diagram [12]. Graph representation is used for many purposes, e.g. formula trees can represent logic or fuzzy logic formulae. The performance of the evaluation can be enhanced if pruning can be made using different techniques according to the specific domain [2], [3], [13].

This research was supported by the grant EFOP-3.6.1-16-2016-00001 "Complex improvement of research capacities and services at Eszterházy Károly University"

III. Definitions

A communication graph [4] is a directed graph with two restrictions, for all node x, the edge $x \to x$ is not allowed, and the set of nodes may contain only symbols which can be used as Boolean variables.

We give the formal definition of Simplified Balatonboglár Model ($\mathcal{SBB}$), because in our previous work [11], which was a poster, we had not enough space for that.

We define the Simplified Balatonboglár Model of communication graphs as follows. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{V}$ be the set of vertices of $\mathcal{D}$ and $\mathcal{E}$ the set of edges of $\mathcal{D}$. Since $\mathcal{D}$ is a communication graph, we know that elements of $\mathcal{V}$ can be used as positive literals.

A path from a_1 to a_j in directed graph $\mathcal{D}$ is a sequence of vertices $a_1, a_2, \ldots, a_j$ such that for each $i \in \{1, \ldots, j-1\}$ we have that (a_i, a_{i+1}) is an edge of $\mathcal{D}$. A path from a_1 to a_j in directed graph $\mathcal{D}$ is a *cycle* iff (a_j, a_1) is an edge of $\mathcal{D}$. The cycle $a_1, a_2, \ldots, a_j, a_1$ is represented by the following tuple: $(a_1, a_2, \ldots, a_j)$ and usually called Cyc in the definitions. This tuple can be used as a set of its elements. Note, that in the representation of a cycle the first and the last element must not be the same vertex.
Then we define the following notions:

Let $OutE(a)$ be defined as the set of child nodes of a, i.e., $OutE(a) := \{b \mid (a,b) \in \mathcal{E}\}$.

Let $ExitPonts$ of a cycle be defined as the child nodes of its vertices which are not in the cycle, i.e., $ExitPonts(\ (a_1, a_2, \ldots a_k)\ ) := \{b \mid \exists_{i=1\ldots k}(b \in OutE(a_i)) \wedge \neg\exists_{j=1\ldots k}(b = a_j)\}$.

Let $NodeRep2$ be defined as the set of $\{\neg a, b\}$ clauses, where a has no other child node, only b, i.e., $NodeRep2 := \{\{\neg a, b\} \mid a \in \mathcal{V} \wedge \{b\} = OutE(a)\}$..

Let $NodeRep3(a)$ be defined as $\{\neg a, b, c\}$, where b, c are child nodes of a, i.e., $NodeRep3(a) := \{\neg a, b, c\}$, where $a \in \mathcal{V} \wedge b \in OutE(a) \wedge c \in OutE(a) \wedge b \neq c$.

Let $NodeRep3$ be defined as the union of $\{\ NodeRep3(a)\}$ for all node a, i.e., $NodeRep3 := \{\ NodeRep3(a) |\ a \in \mathcal{V}\}$.

Let $BigCycle(\ (V,E)\ )$ be defined as a cycle which contains all vertices from V, or UnDef if no such cycle exists. Note that it is not UnDef if (V,E) is a strongly connected component. Note that if a, b are consecutive vertices in $BigCycle(\ (V,E)\ )$ then $(a,b) \in E$.

Let $BigCycles$ be defined as the set of $BigCycle(Comp)$ where $Comp$ is a strongly connected component.

Let $CycEdges(\ (a_1, a_2, \ldots a_k)\ )$ be defined as the pairs of consecutive vertices, i.e., $CycEdges(\ (a_1, a_2, \ldots a_k)\ ) := \{(a_i, a_{(i \bmod k)+1}\}$.

Let $BigRep(Cyc)$ be defined as the set of the following NPP clauses: $\{\neg a, \neg b, c\}$, where a, b, c are conscutive vertices in Cyc, i.e., $BigRep(Cyc) := \{\{\neg a, \neg b, c\} \mid (a,b) \in CycEdges(Cyc) \wedge (b,c) \in CycEdges(Cyc)\}$.

Let $BigRep$ be defined as the union of the representation of all strongly connected components which have more than 2 vertices, i.e., $BigRep := \bigcup_{Cyc \in BigCycles \wedge |Cyc|>2} BigRep(Cyc)$.

Let $LinkRep(Cyc)$ be defined as the following NNP clause: $\{\neg a, \neg b, c\}$, where c is an exit point of Cyc and (b,c) is an edge and a,b are consecutive vertices in Cyc, i.e., $LinkRep(Cyc) := \{\neg a, \neg b, c\}$, where $(a,b) \in CycEdges(Cyc) \wedge c \in ExitPoints(Cyc) \wedge c \in OutE(b)$.

Let $LinkRep$ be defined as the union of the representation of links between strongly connected components which have at least 2 vertices, i.e.,
$LinkRep := \bigcup_{Cyc \in BigCycles \wedge |Cyc|>1} LinkRep(Cyc)$.

Let $ReverseEdges(Cyc)$ be defined as the set of reverse edges of Cyc, i.e.,
$ReverseEdges(Cyc) := \{(b,a) \mid (a,b) \in CycEdges(Cyc)\}$.

Let $ToDel$ be defined as the union of edges of all strongly connected components and their reverse edges, i.e.,
$ToDel := \bigcup_{Cyc \in BigCycles \wedge |Cyc|>1} (CycEdges(Cyc) \cup ReverseEdges(Cyc))$.

Let $Rest_0$ be defined as the edges of the represented communication graph setminus $ToDel$, i.e., $Rest_0 := \mathcal{E} \setminus ToDel$.

Let ACs_i be the set of cycles in $Rest_i$, i.e.,
$ACs_i := \{(a_1, a_2, \ldots, a_k) \mid \forall_{j=1\ldots k}(a_j, a_{(j \bmod k)+1}) \in Rest_i\}$.

Let AC_i be a cycle from ACs_i with at least 2 vertices, if there is no such cycle, then it is UnDef.

Let $RestRep_0$ be defined as the empty set, i.e., $RestRep_0 := \{\}$.

Let $RestRep_i$ be the union of $RestRep_{i-1}$ and the singleton set of the following NPP clause: $\{\neg a, \neg b, c\}$, where a,b are consecutive vertices in AC_i and b, c are consecutive vertices in the representation of one of the strongly connected components, i.e., $RestRep_i := RestRep_{i-1} \cup \{\{\neg a, \neg b, c\}\}$, where $(a,b) \in CycEdges(AC_i) \wedge \exists_{C \in BigCycles}(b \in C \wedge (b,c) \in CycEdges(C))$, and let $A_i := a$, and let $B_i := b$, because in the next step we have to delete them.

Let $Rest_i$ be defined as $Rest_{i-1}$ minus the edge (A_i, B_i) and its reverse, i.e., $Rest_i := Rest_{i-1} \setminus \{(A_i, B_i), (B_i, A_i)\}$.

Let $RestRep$ be defined as the last one of its sequence, i.e., $RestRep := RestRep_{i-1}$, weher AC_i is UnDef.

Let $\mathcal{SBB}$ be defined as the union of the representation of the nodes, the representation of the strongly connected components (SCCs), the representation of the links between SCCs and the representation of the rest, i.e., $\mathcal{SBB} := NodeRep2 \cup NodeRep3 \cup BigRep \cup LinkRep \cup RestRep$.

We say that $\mathcal{SBB}$ is a Simplified Balatonboglár Model of $\mathcal{D}$. We recall the definition of our Weak Model ($\mathcal{WM}$) [10].

$$Cycles := \{(a_1, a_2, \ldots, a_k) \mid k = 1 \vee \forall_{i=1\ldots k}(a_{(i \bmod k)+1} \in OutE(a_i)\},$$

and for any two elements of $Cycles$ they cannot be equal as a set.

$$ExitPoints(\ (a_1, a_2, \ldots a_k)\ ) := \{b \mid \exists_{i=1\ldots k}(b \in OutE(a_i)) \wedge \neg\exists_{j=1\ldots k}(b = a_j)\}.$$

$$CycleRep := \{\neg C \cup ExitPoints(C) \mid C \in Cycles \wedge ExitPoints(C) \neq \emptyset\}. \mathcal{WM} := CycleRep.$$

We say that clause sets A and B are equivalent if and only if they have the same set of solutions. We say that clause set A entails clause set B if and only if the set of solutions of A is a subset of the set of solutions of B, i.e., A may have no other solutions than B. This notion is denoted by $A \geq B$. Note, that if A subsumes all clauses of B, then $A \geq B$.

We say that A is stronger than B if and only if $A \geq B$ and A and B are not equivalent. This notion is denoted by $A > B$.

IV. Theoretical Results

We recall our results from the field of modelling communication graphs as SAT problems [4], [10]. We also give a more detailed proof of the theorem, that $\mathcal{SM} \geq \mathcal{BB} \geq \mathcal{SBB} \geq \mathcal{WM}$.

Let $\mathcal{D}$ be a communication. Let $\mathcal{SM}$ be its strong model. Let $\mathcal{WM}$ be its weak model. Let $\mathcal{BB}$ be its Balatonboglár model. Let $\mathcal{SBB}$ be its simplified Balatonboglár model. Then:

- $\mathcal{SM}$ is a Black-and-White 2-SAT problem if and only if $\mathcal{D}$ is strongly connected.
- $\mathcal{WM}$ is a Black-and-White SAT problem if and only if $\mathcal{D}$ is strongly connected.
- $\mathcal{SM} \geq \mathcal{WM}$, i.e., the set of solutions of $\mathcal{SM}$ is a subset of the set of solutions of $\mathcal{WM}$.
- $\mathcal{SM} > \mathcal{WM}$ if and only if $\mathcal{D}$ is not strongly connected, and it has at least one node which has more than one child node.

- Theorem I.: If we have $\mathcal{SM} \geq \mathcal{MM} \geq \mathcal{WM}$, where $\mathcal{MM}$ is an arbitrary but fixed model of $\mathcal{D}$, then $\mathcal{MM}$ is a Black-and-White SAT problem if and only if $\mathcal{D}$ is strongly connected.
- Theorem II: $\mathcal{SM} \geq \mathcal{BB} \geq \mathcal{WM}$, i.e., $\mathcal{BB}$ is a Black-and-White 3-SAT problem if and only if $\mathcal{D}$ is strongly connected.
- Theorem III: $\mathcal{SM} \geq \mathcal{BB} \geq \mathcal{SBB} \geq \mathcal{WM}$.

Proof: It is easy to see that $\mathcal{BB} \geq \mathcal{SBB}$, because by the construction of $\mathcal{SBB}$ it is a subset of $\mathcal{BB}$ if we assume that $NodeRep3$ is a deterministic method. From Theorem II we know that $\mathcal{SM} \geq \mathcal{BB}$. We show that $\mathcal{SBB} \geq \mathcal{WM}$

To show this it is enough to show that $\mathcal{SBB}$ subsumes all clauses from $\mathcal{WM}$. Let C be an arbitrary but fixed element of $\mathcal{WM}$. This means that C is an element of either $NodeRep$ or $Cycles$. If C is an element of $NodeRep$, then, this case is already covered in the proof of Theorem II because the simplified Balatonboglár model and the Balatonboglár model represent nodes in the same way, see $NodeRep2$ and $NodeRep3$.

If C is an element of $Cycles$, then, without loss of generality, we may assume that $C = \{\neg d_1, \neg d_2, \ldots, \neg d_m, e_1, e_2, \ldots, e_n\}$, where $m > 1$, and $n > 0$. From the definition of strongly connected components there is a $DC \in BigCycles$ such that for all $i \in 1, \ldots, m$ we have that $d_i \in DC$. There are two cases: (a) either there is $j \in 1, \ldots, n$ such that $e_j \in DC$ or (b) there is no such j. In case (a) let a, b such that $a, b \in \{d_1, d_2, \ldots, d_m\}$ and $b \in OutE(a)$ and $e_j \in OutE(b)$. There are two cases: (aa) either a and b are consecutive nodes in DC or (bb) not.

In case (aa) there is a clause $D \in BigRep$ such that D is a subset of C. In case (bb) there is a clause $D \in RestRep$ such that D is a subset of C. Case (b) is only possible if the cycle represented by C corresponds to a strong connected component, i.e., if $\{d_1, d_2, \ldots, d_m\} = DC$. In this case there is a clause $D \in LinkRep$ such that D is a subset of C.

Hence, $\mathcal{SM} \geq \mathcal{BB} \geq \mathcal{SBB} \geq \mathcal{WM}$.

The next theorem states that the simplified Balatonboglár model is a Black-and-White 3-SAT problem if and only if the represented directed graphs is strongly connected. This result is an immediate consequence of the above result, see Theorem I-III.

V. An example

The $\mathcal{SM}$ of the directed graph on Figure 1 is:

$$\mathcal{SM} = \{\{\neg a, b\}, \{\neg a, c\}, \{\neg b, a\}, \{\neg b, c\}, \{\neg b, d\}, \{\neg c, a\}, \{\neg c, d\}, \{\neg d, a\}\}. \quad (1)$$

Since the graph on Figure 1 is strongly connected, its $\mathcal{SM}$ is a `Black-and-White SAT` problem, i.e., the `SAT` problem in (1) has only these two solutions: $\{a, b, c, d\}$, and $\{\neg a, \neg b, \neg c, \neg d\}$.

Its other 3 models are:

$$\mathcal{WM} = \{\{\neg a, b, c\}, \{\neg b, a, c, d\}, \{\neg c, a, d\}, \{\neg a, \neg b, c, d\}, \{\neg a, \neg b, \neg c, d\}, \{\neg a, \neg c, b, d\}, \{\neg d, a\}, \{\neg a, \neg b, \neg d, c\}, \{\neg a, \neg c, \neg d, b\}\}. \quad (2)$$

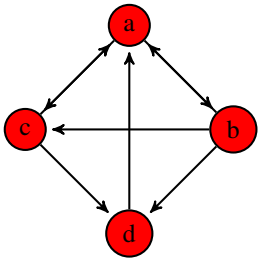

Figure 1. A directed graph with 4 vertices and 8 edges

$$\begin{aligned}\mathcal{BB} = \{&\{\neg a, b, c\}, \{\neg b, c, d\}, \{\neg c, a, d\},\\ &\{\neg d, a\}, \{\neg a, \neg b, c\}, \{\neg a, \neg b, d\},\\ &\{\neg a, \neg c, b\}, \{\neg a, \neg c, d\}, \{\neg b, \neg c, a\},\\ &\{\neg b, \neg c, d\}, \{\neg b, \neg d, a\}, \{\neg c, \neg d, a\},\\ &\{\neg d, \neg a, b\}, \{\neg d, \neg a, c\}\}.\end{aligned} \tag{3}$$

$$\begin{aligned}\mathcal{SBB} = \{&\{\neg a, b, c\}, \{\neg b, c, d\}, \{\neg c, a, d\},\\ &\{\neg d, a\}, \{\neg a, \neg b, c\}, \{\neg b, \neg c, d\},\\ &\{\neg c, \neg d, a\}, \{\neg d, \neg a, b\}, \{\neg a, \neg c, d\}\}.\end{aligned} \tag{4}$$

Note, that all the 4 models are a Black-and-White SAT problem.

VI. Test Results

The tests were done on Intel quadcore machine (CPU: i7-6700HQ CPU 2.60GHz, 2601 Mhz, Memory: 32GB 3000MHz DDR4). For testing, we used Glucose 4.2.1 [1] *(Glucose)*, MapleLCMDistChrono BT-DL v3 [2] *(Maple)* and the latest version of CSLFOC [8], [9] (CSFLOC 18) *(CSFLOC)*.

A. Performance Results

Number of V	CSFLOC	Maple	Glucose
1000	0,1754s	**0,01216s**	0,01552s
1500	0,2013s	**0,0184s**	0,0191s
2000	0,2927s	**0,02422s**	0,0252s
2500	0,3986s	**0,02709s**	0,0275s
3000	0,4853s	**0,0450s**	0,0492s
5000	0,6238s	**0,0652s**	0,0723s
10000	0,7815s	**0,0825s**	0,0906s

Table I
Average runtimes on Strong Model-based problems

We ran the solvers in three ($\mathcal{BB}$-, $\mathcal{SBB}$- and $\mathcal{SM}$-based) categories of benchmarks. For each instance we used a timeout of 900 seconds.

We examined the growth of runtime as a function of the number of $V + E$, where V is the number of vertices in the graph, and E is the number of edges.

For this performance measurement the benchmarks were generated from directed graphs of different densities, with 10-10000 vertices. In all $\mathcal{BB}$-based benchmarks the Glucose won.

[1] https://www.labri.fr/perso/lsimon/glucose/

[2] https://sites.google.com/a/gsd.uwaterloo.ca/maplesat/

In the case of $\mathcal{SBB}$, the individual solvers won alternately. These results we could not translate into a table. These results will be the subject of deeper analysis in the future. In the case of $\mathcal{SM}$-based benchmarks for smaller graphs ($V < 750$), Glucose and Maple won alternately. Results for larger problems ($V > 1000$) are shown in Table I and in Table II. In Table I, all benchmark were generated from rare graphs (the density of graph was smaller than 10%). You can see that Maple won in all cases.

Number of V	CSFLOC18	Maple	Glucose
1000	**0,5235s**	1,0457s	1,0903s
1500	**1,5541s**	5,7417s	6,9023s
2000	**2,4236s**	9,6167s	9,8691s
2500	**2,787s**	10,987s	12,4136s
3000	**3,102s**	12,94s	13,23s
5000	**4,4807s**	17,6777s	18,6290s
10000	**6,1337s**	25,2252s	27,3703s

Table II
Average runtimes on Strong Model-based problems

However, CSFLOC wins each case where the graph density was greater than 10%. These results are shown in Table II.

In our previous work [9], we have shown a connection between the fields of Internet of Things and the Boolean satisfiability. In this sense, this result implies that CSFLOC provides a faster answer to the question whether the communication network is strongly connected or not than the most relevant SAT solvers, if the network consists of a large number of sensors and many direct connections.

In Figures 2-4 we show the runtimes of CSFLOC on various benchmarks. Each graph consists of 100 vertices and the densities are between 0.03 and 1. If the density is 1 then the graph is complete, i.e, if we have 100 vertices, then we have $100 * 99 = 9900$ edges, i.e., $V + E = 10^4$.

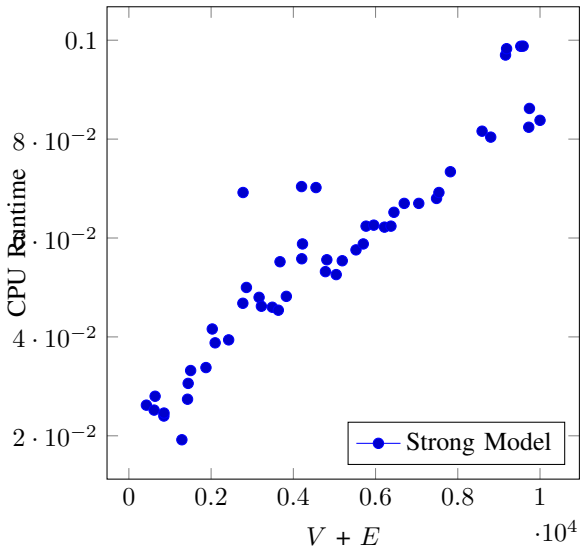

Figure 2. CPU runtimes of CSFLOC as a function of $V + E$ on Strong Model-based benchmarks

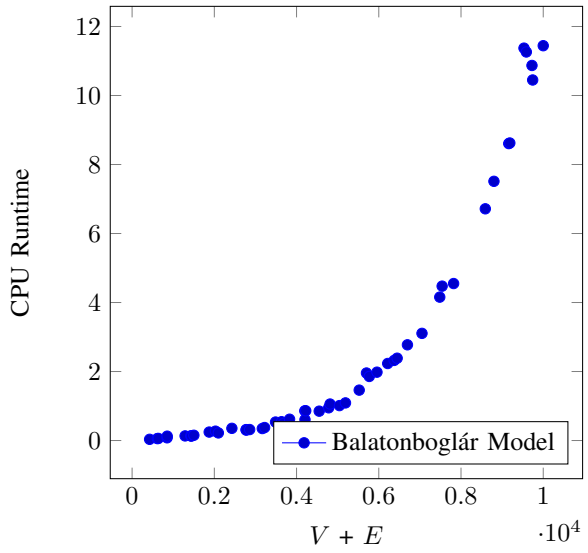

Figure 3. CPU runtimes of CSFLOC as a function of $V+E$ on Balatonboglár Model-based benchmarks

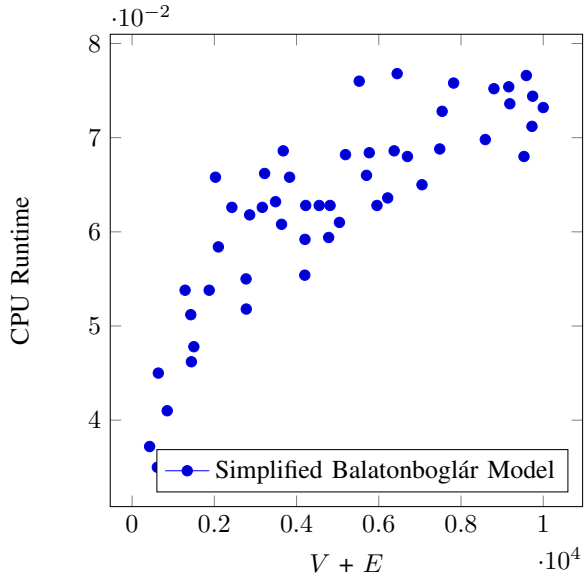

Figure 4. CPU runtimes of CSFLOC as a function of $V + E$ on Simplified Balatonboglár Model-based benchmarks

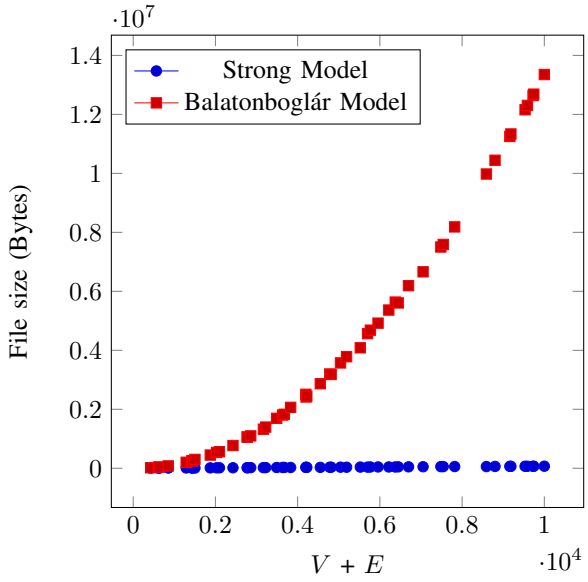

Figure 5. File sizes as a function of the number of $V + E$

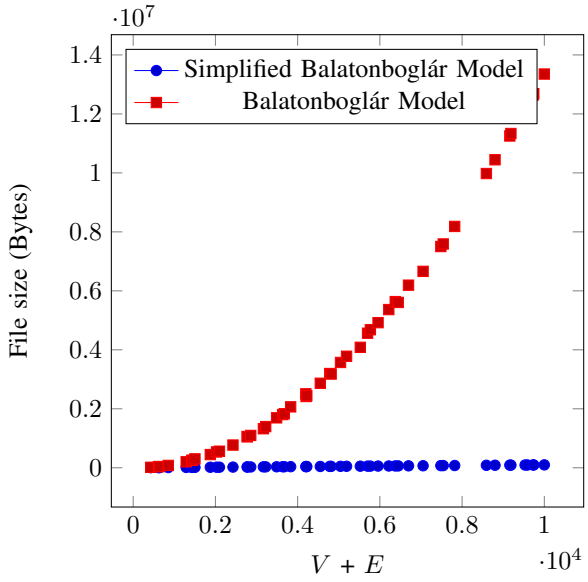

Figure 6. File sizes as a function of the number of $V + E$

Analyzing the runtimes of the three solvers, it can be generally stated that their time complexities are linear on $\mathcal{SM}$-based problems, polynomial on $\mathcal{BB}$-based problems and logarithmic on $\mathcal{SBB}$-based problems.

B. File sizes

We examined the growth of file sizes as a function of the number of $V + E$, where V is the number of vertices in the graph, and E is the number of edges. For this examination, we used graphs with density from 0.03 to 1. As before, each instance consists of 100 vertices. Figure 5-6 show that the file size in the case of Balatonboglár Model grows polynomial, like the previously mentioned time complexity.

We examined the file size ratio, too. Figure 7 shows that above 20% graph density, the ratio of $\mathcal{BB}$ to $\mathcal{SM}$ and $\mathcal{BB}$ to $\mathcal{SBB}$ is below 1%, and $\mathcal{SM}$ is bit smaller than $\mathcal{SBB}$.

The $\mathcal{BB}$ model is very redundant. The $\mathcal{SM}$ may be redundant if the directed graph is redundant. The $\mathcal{SBB}$ model is between them.

C. Unused Clauses

The latest version of CSFLOC includes an option that it monitors the unused clauses (actually the redundancy in the model). In Figures 7-8 we show the ratio of number of unused clauses to the number of all clauses in the various models. In

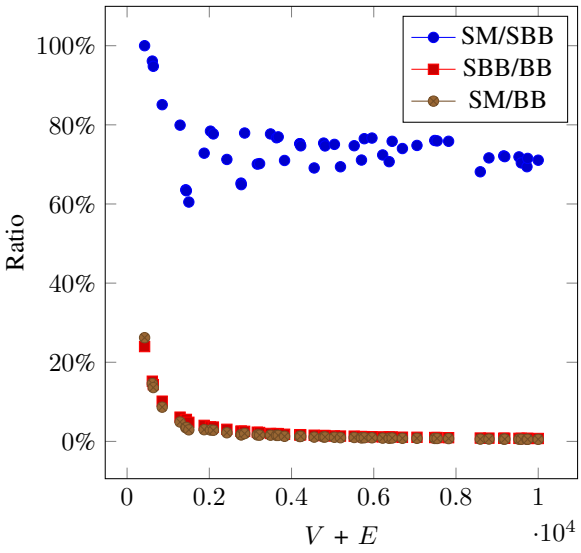

Figure 7. Ratio of file sizes as a function of the number of $V + E$

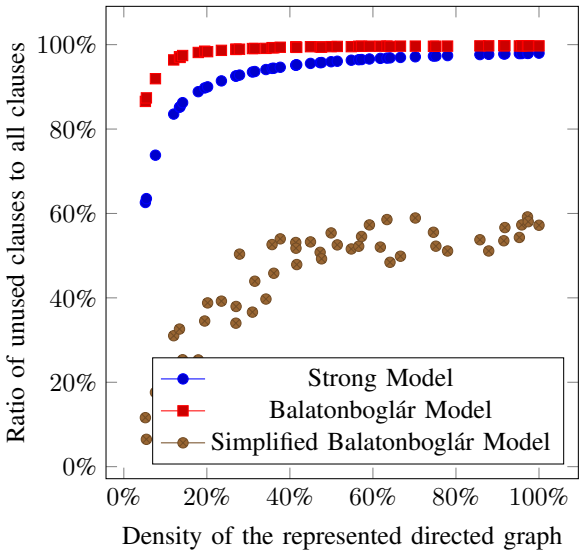

Figure 9. Ratio of unused clauses to all clauses

the case of Figure 7, the graph size ($V + E$) is from 20 to 60000.

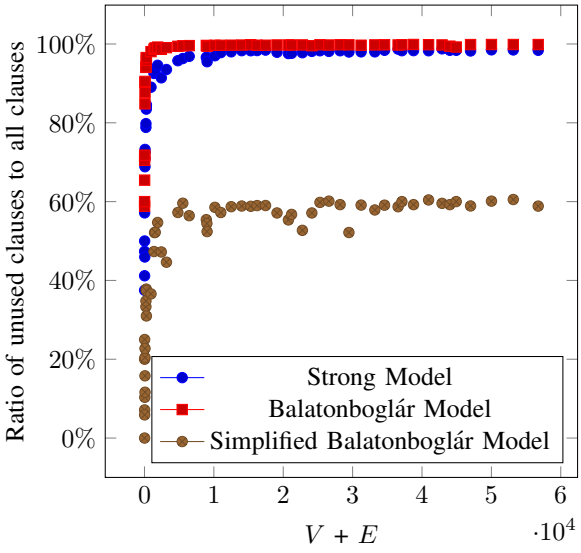

Figure 8. Ratio of unused clauses to all clauses

Figure 9 we show ratio of unused clauses to all clause as the function of the density of the represented directed graph.

Figures 8-9 show that $\mathcal{SBB}$ gives the most concise description. $\mathcal{SBB}$ may provide the theoretical minimum for some very small graphs, but not for bigger ones.

VII. Conclusion

In this paper, we presented three directed graph 3-SAT representations. We introduced the basic theoretical background needed to understand the models. We compared the CSFLOC solver to the currently best SAT solvers (Maple, Glucose). We did some analysis on the problems generated by the models (file sizes, ratio of unused clauses to all clauses). The most important result was that the $\mathcal{SBB}$ size is around 1% of the size of $\mathcal{BB}$, but it is still a Black-and-White 3-SAT problem if and only if the represented directed graph is strongly connected. We checked this property also empirically by using the CSFLOC solver. Our empirical results show that $\mathcal{SBB}$ is near to the smallest possible model, i.e., the number of unused clauses reported by CSFLOC is very small, i.e., $\mathcal{SBB}$ extended by the white and the black clause can be `MIN-UNSAT`, if the density of the graph is small.

As a future goal our aim is to analyze when do we get a `MIN-UNSAT` problem, and create an even better model which generates a `MIN-3-UNSAT` problems out of strongly connected directed graphs.

References

[1] B. Aspvall, M. F. Plass, R. E. Tarjan, *A Linear-Time Algorithm For Testing The Truth Of Certain Quantified Boolean Formulas*, Information Processing Letters, pp. 121–123, 1979.

[2] R. Basbous, B. Nagy, T. Tajti, *Short Circuit Evaluations in Gödel Type Logic*, V. Ravi et al. (eds.), Proceedings of the Fifth International Conference on Fuzzy and Neuro Computing (FANCCO - 2015, India), Advances in Intelligent Systems and Computing - AISC 415 119-138. 2015.

[3] R. Basbous, T. Tajti, B. Nagy, *Fast Evaluations in Product Logic: Various Pruning Techniques*, IEEE WCCI 2016 - IEEE World Congress on Computational Intelligence, FUZZ-IEEE 2016 - the 2016 IEEE International Conference on Fuzzy Systems, Vancouver, Canada, 140-147. 2016.

[4] Cs. Biró, G. Kusper, *Equivalence of Strongly Connected Graphs and Black-and-White 2-SA Problems*, Miskolc Mathematical Notes, Vol. 19, No. 2, pp. 755-768, 2018.

[5] Cs. Biró and G. Kusper, *BaW 1.0 - A Problem Specific SAT Solver for Effective Strong Connectivity Testing in Sparse Directed Graphs*, 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, pp. 137-142, 2018.

[6] R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Trans. Comput., pp. 677–691, 1986.

[7] L. Hellerman, *A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits*, IEEE Transactions on Electronic Computers, pp. 198–223, 1963.

[8] G. KUSPER AND CS. BIRÓ, Solving SAT by an Iterative Version of the Inclusion-Exclusion Principle, *SYNASC 2015*, IEEE Computer Society Press pp. 189–190, 2015.

[9] G. KUSPER, CS. BIRÓ, GY. B. ISZÁLY, *SAT solving by CSFLOC, the next generation of full-length clause counting algorithms*, Future IoT Technologies (Future IoT), 2018 IEEE International Conference, pp.1–9, 2018.

[10] G. KUSPER, CS. BIRÓ, *Convert a Strongly Connected Directed Graph to a Black-and-White 3-SAT Problem by the Balatonboglár Model*, submitted to Theory of Computing, 17 pages, arrived on 24.08.2019, status: under review.

[11] G. KUSPER CS. BIRÓ, T.BALLA *Representing Directed Graphs as 3-SAT Problems using the Simplified Balatonboglár Model*, The 11th International Conference on Applied Informatics to be held in Eger, Hungary, January 29–31, 2020.

[12] S. MINATO, *Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems*, Proceedings of the 30th International Design Automation Conference, pp. 272–277, 1993.

[13] , B. NAGY, R. BASBOUS, T. TAJTI, *Lazy evaluations in Lukasiewicz type fuzzy logic*, Fuzzy Sets and Systems (Elsevier) 376 (2019), 127-151. 2019.

[14] R. TARJAN, *Depth-First Search and Linear Graph Algorithms*, SIAM Journal on Computing, Vol. 1, No. 2, pp. 146-160, 1972.

[15] M. SHARIR, *A strong-connectivity algorithm and its applications in data flow analysis*, Computers And Mathematics with Applications, pp. 67–72, 1981.

10 Annex 3. - Equivalence of Strongly Connected Graphs and Black-and-White 2-SAT Problems

This is the submitted version of Cs. Biró, G. Kusper, *Equivalence of Strongly Connected Graphs and Black-and-White 2-SAT Problems*, Miskolc Mathematical Notes, Vol. 19, No. 2, pp. 755-768, 2018. See also [88].

Miskolc Mathematical Notes
Vol. 19 (2018), No. 2, pp. 755–768

HU e-ISSN 1787-2413
DOI: 10.18514/MMN.2018.2140

EQUIVALENCE OF STRONGLY CONNECTED GRAPHS AND BLACK-AND-WHITE 2-SAT PROBLEMS

CSABA BIRÓ AND GÁBOR KUSPER

Received 09 November, 2016

Abstract. Our goal is to create a propositional logic formula to model a directed graph and use a SAT solver to analyse it. This model is similar to the well-know one of Aspvall et al., but they create a directed graph from a 2-SAT problem, we generate a 2-SAT problem from a directed graph. In their paper if the 2-SAT problem is unsatisfiable, then the generated directed graph is strongly connected, in our case, if the directed graph is strongly connected, then the generated 2-SAT problem is a black-and-white 2-SAT problem, which has two solutions: where each variable is true (the white assignment), and where each variable is false (the black one). If we see a directed graph as a communication model of a network, then we can ask in our model whether a node can send a message to another one through the network. More specifically we can ask whether all nodes can send messages to all other ones, i.e., the graph is strongly connected or not.

2010 *Mathematics Subject Classification:* 94C15; 68R10; 03B05; 03B70

Keywords: black-and-white SAT problem, SAT representation, communication graph, connectivity test, strongly connected graph

1. INTRODUCTION

Propositional satisfiability is the problem of determining, for a formula of the propositional logic, if there is an assignment of truth values to its variables for which that formula evaluates to true. By SAT we mean the problem of propositional satisfiability for formulas in conjunctive normal form (CNF). SAT is one of the most-researched NP-complete problems [14] in several fields of computer science, including theoretical computer science, artificial intelligence, hardware design, and formal verification [1]. Modern sequential SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) [11] algorithm. This algorithm performs Boolean constraint propagation (BCP) and variable branching, i.e., at each node of the search tree it selects a decision variable, assigns a truth value to it, and steps back when a conflict occurs.

The research was supported by the grant EFOP-3.6.1-16-2016-00001 ”Complex improvement of research capacities and services at Eszterhazy Karoly University”.

By k-SAT we mean the problem of propositional satisfiability for formulas in CNF, where each clause consists of at most k literals. While 2-SAT is solvable in linear time [2], 3-SAT is NP-complete [9]. These days one of the most promising branch of mathematics is the idea, that we try to unify different mathematical theories, like in case of Langlands program [10], which relates algebraic number theory to automorphic forms and representation theory. Another nice example is the modularity theorem [16] (formerly called the Taniyama–Shimura–Weil conjecture), which states that elliptic curves over the field of rational numbers are related to modular forms. Without the modularity theorem Andrew Wiles could not prove Fermat's Last Theorem [17].

In this paper we show a link between directed graphs and propositional logic formulas. We prove a theorem which allows to use an algorithm from the field of propositional logic to check a graph property. Namely, we transform a graph into a SAT problem to check whether the graph is strongly connected or not.

The most prominent graph representations are:

- Implication graph [2] is a is a skew-symmetric directed graph, where vertices are literals (boolean variables, and their negation), edges represents implication. Note, that the binary clause $x \vee y$ is represented by two implications in the implication graph: $\neg x \supset y$, and $\neg y \supset x$, and so the implication graph is skew-symmetric, i.e., it is isomorphic to its own transpose graph.
- AIG, And-Inverter Graph [7] is directed acyclic graph where vertices are logical conjunction with two input edges, a marked edge means logical negation, the boolean variables are the input, the formula itself is the output.
- BDD, Reduced Ordered Binary Decision Diagram [5], which is a rooted, directed, acyclic graph consisting of vertices, which are boolean variables and terminal vertices, called 0-terminal, which terminates pathes where the formula evaluates to false; and 1-terminal, which terminates pathes, where the formula evaluates to true. Each non-terminal vertex has two child vertices called low child, corresponding edge is called 0-edge; and high child, corresponding edge is called 1-edge; which are possible values of the parent vertex. One has to merge any isomorphic subgraph and eliminate any vertex whose two children are isomorphic.
- ZDD (called also ZBDD in the literature), Zero-Suppressed Binary Decision Diagram [12], is a kind of binary decision diagram, where instead of the rule "eliminate any vertex whose two children are isomorphic" we use the rule "eliminate those vertices whose 1-edge points directly to 0-terminal". If a SAT problem has only a few solutions then ZDD is a better representation than BDD.

As we can see a great effort has been done in the direction from formulas to graphs. In this paper we study the other way, the direction from graphs to formulas. In our model vertices are boolean variables, and edges are implications. This means that our

model is similar to an implication graph, but in case of implication graphs vertices are literals. The intuition behind our model comes from the field of wireless sensor networks (WSN), where it is a relevant problem whether each sensor can communicate with all other ones through the network. If the network is represented by a directed graph where vertices are the sensor, and an edge represents that a sensor can send data to an other one, then this problem boils down to check whether the graph is strongly connected.

We wanted to solve this problem by a SAT solver, so we had to convert the above directed graph into a SAT problem. Since in our model each edge represents a logical implication we can generate a 2-SAT problem where each clause contains exactly one positive and one negative literal.

We have found that the graph is strongly connected if this 2-SAT problem has exactly two solutions: the first is the one where each boolean variable is true (which is called the white assignment), the second is the one where each boolean variable is false (which is called the black assignment). We call such a SAT problem to be a "black-and-white" SAT problem.

This means that if we add the negation of these two solutions (the black one and the white one) to the generated SAT problem, then it will be unsatisfiable and will be not 2-SAT any more.

In other words, a SAT problem is a black-and-white SAT problem if and only if it is satisfiable and has only two solutions, the white assignment and the black one. So it becomes unsatisfiable if we add the negation of these assignments, which are two full length clauses, the clause which contains only negative literals (which is called the black clause), and the clause which contains only negative literals (which is called the white clause).

In the field of directed graphs the problem to check strongly connectedness is a linear time problem [15]. The black-and-white 2-SAT problem is also a linear time problem as we are going to show that later in this paper. The question arises, while should we transform a graph into a SAT problem to check a property which can be checked in linear time in both fields? We think that our model is a new link between the two fields: We learned that black-and-white 2-SAT problems and strongly connected graphs are equivalent. The black-and-white SAT problem appears also as a special case of weakly nondecisive SAT problems, see Lemma 6. in [3] (in that paper we did not use the term "black-and-white SAT problem", this term is introduced in this paper). This suggests two things: the black-and-white SAT problem could be an interesting problem in general, and since there are weakly nondecisive 3-SAT problems, which are also black-and-white, there might be a 3-SAT representation of directed graphs.

2. Logical model of a directed graph

Let $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ be a directed graph, where $\mathcal{V}$ is the set of vertices, and E is the set of edges. We say that $\mathcal{D}$ is a communication graph if and only if the elements of $\mathcal{V}$ are atomic formulas or each vertex is labeled by a different atomic formula. Note, that any directed graph is a communication graph because we can label each vertex by a different atomic formula. In this paper we assume that the elements of $\mathcal{V}$ are atomic formulas. In other words, if x is an element of $\mathcal{V}$, then for example $\neg x$ must not be an element of $\mathcal{V}$. From a communication graph we create the following model: We represent vertices by boolean variables, and edges by logical implication. The conjunction of these formulas gives the logical model of the directed graph. For example, if the vertex x_1 has edges to vertices x_2 and x_3, then the logical model is:

$$(x_1 \supset x_2) \wedge (x_1 \supset x_3). \tag{2.1}$$

This formula can be easily transformed to a 2-SAT problem by rewriting implications by the rule $x \supset y = \neg x \vee y$. Note, that although each edge in $\mathcal{D}$ is interpreted as logical implication, it is not an implication graph, because it does not contain negative literals. In other words, our model is not the same used by Aspvall et al. in [2]. They create a graph from a 2-SAT problem, we generate a 2-SAT problem from a communication graph. In their directed graph each variable is represented by a positive and a negative literal. In our case we have only positive literals in the directed graph.

We give the definition of our model in a more formal way. The 2-SAT representation $\mathcal{M}$ of a communication graph $\mathcal{D}$ is defined as follows:

$$\mathcal{M} := \bigwedge_{i=1,j=1,i\neq j}^{n} \mathcal{P}(x_i, x_j) \tag{2.2}$$

$$\mathcal{P}(x, y) = \begin{cases} \neg x \vee y, & \text{if } x \text{ has an edge to } y \\ \text{True}, & \text{otherwise} \end{cases} \tag{2.3}$$

Note, that $\mathcal{M}$ is a 2-SAT problem and has a nice property, each clause in it has exactly one positive and one negative literal, therefore, $\mathcal{M}$ is satisfiable. It has at least two solutions: one where each variable is true which is called the white assignment, and one where each variable is false which is called the black assignment.

If $\mathcal{M}$ has only these two solutions, and no other one, then $\mathcal{D}$ has an interesting property, namely it strongly connected. To check this we need the white-or-black constraint, which is the disjunction of the white assignment and the black assignment, that states that all vertices can be reached from all other ones. We denote the white-or-black constraint by $\mathcal{C}$ in this paper, and we define it as follows:

$$\mathcal{C} := \bigwedge_{i=1,j=1,i\neq j}^{n} x_i \supset x_j = (x_1 \wedge x_2 \wedge \cdots \wedge x_n) \vee (\neg x_1 \wedge \neg x_2 \wedge \cdots \wedge \neg x_n) \tag{2.4}$$

Its negation is called the black-and-white constraint, which is the conjunction of the black clause and the white clause:

$$\neg\mathcal{C} = (\neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_n) \wedge (x_1 \vee x_2 \vee \cdots \vee x_n) \tag{2.5}$$

By the formula $\mathcal{C}$ we state that there is a path from any vertex to any other one. To show that $\mathcal{D}$ is strongly connected we have to show that the logical representation of $\mathcal{D}$, which is $\mathcal{M}$, implies $\mathcal{C}$. Now we can use the fact that SAT solving is the dual of theorem proving. So instead of proving that "$\mathcal{D}$ is strongly connected if $\mathcal{M}$ implies $\mathcal{C}$", we prove that "$\mathcal{D}$ is strongly connected if $\mathcal{M} \wedge \neg\mathcal{C}$ is unsatisfiable". This means that we can use a SAT solver to check a property of a graph.

Motivated by this we define the following notion: F is a black-and-white SAT problem if and only if F is satisfiable and has exactly two solutions, the white and the black assignments, which implies that $F \wedge \neg\mathcal{C}$ is unsatisfiable. F is a black-and-white 2-SAT problem if and only if F is a 2-SAT and a black-and-white SAT problem. Lemma 6 in [3] suggests an alternative, more general definition: F is a black-and-white SAT problem if and only if F is satisfiable and has exactly two solutions, A and B such that $A = \neg B$. But we do not use this alternative definition in this paper.

First, we need an auxiliary lemma, which states that: there is a path from vertex x_i to x_j if and only if $x_i \supset x_j$ is subsumed by the logical model of the graph.

Lemma 1. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{M}$ be the 2-SAT representation of $\mathcal{D}$. Then $\mathcal{M}$ implies the formula $x_i \supset x_j$ if and only if there is path from vertex x_i to x_j in graph $\mathcal{D}$.*

Proof. We know that $\mathcal{M}$ is a special SAT instance where each clause contains exactly one positive and one negative literal, hence, each resolvent of clauses from $\mathcal{M}$ is a binary clause with one positive and one negative literal. The main idea of the proof is that if we have two clauses $\neg v_1 \vee v_2$ and $\neg v_2 \vee v_3$, i.e., there is a path in $\mathcal{D}$ from v_1 to v_3, then by resolution we can generate $\neg v_1 \vee v_3$. □

Based on this lemma we can prove the main theorem of this paper which states that the notion of strongly connected graphs and the notion of black-and-white 2-SAT problems are equivalent. This means that the 2-SAT representation of a strongly connected graph is a black-and-white 2-SAT problem, and the other way around, the directed graph representation of a black-and-white 2-SAT problem is a strongly connected graph.

Theorem 1. *Let $\mathcal{D}$ be a communication graph. Let $\mathcal{M}$ be the 2-SAT representation of $\mathcal{D}$. Then $\mathcal{M}$ is a black-and-white 2-SAT problem if and only if the graph $\mathcal{D}$ is strongly connected.*

Proof. Let $\mathcal{D}$ be a communication graph. Let $\mathcal{M}$ be the 2-SAT representation of $\mathcal{D}$. We show that both directions hold.

($\Rightarrow$): The main idea of the proof if the following, if a formula is satisfied by all solutions of a SAT problem, then it is implied by this SAT problem. We know, that $\mathcal{M}$

has only two solutions, the white and the black assignments, both of them satisfy all $x_i \supset x_j$ shaped formulas, i.e., they are implied by $\mathcal{M}$. From this and from Lemma 1. we obtain that in $\mathcal{D}$ there is a path from any vertex to any other one, i.e., $\mathcal{D}$ is strongly connected.

($\Leftarrow$): The main idea of the proof is the following, since $\mathcal{D}$ is strongly connected we know that there is a path $v_1 \supset v_2, v_2 \supset v_3 \ldots, v_{z-1} \supset v_z, v_z \supset v_1$ which is cyclic and contains all vertices from $\mathcal{D}$, the corresponding clause set is $\{(\neg v_1 \vee v_2), (\neg v_2 \vee v_3), \ldots, (\neg v_{z-1} \vee v_z), (\neg v_z \vee v_1)\}$, which is a subset of $\mathcal{M}$, and which can be satisfied only by the white and the black assignments. From this and since each clause in $\mathcal{M}$ is a binary clause, we obtain, that $\mathcal{M}$ is a black-and-white 2-SAT problem. □

To check whether a directed graph is strongly connected or not, we need linear time [15]. We are going to show that its 2-SAT representation can be solved also in linear time.

Theorem 2. *Let F be a black-and-white 2-SAT problem. Then we need linear time to show that $F \wedge \neg C$ is unsatisfiable.*

Proof. The main idea of the proof is that any SAT solver which uses variable branching and BCP, can show that $F \wedge \neg C$ is unsatisfiable by using 1 variable branching and 2 BCP steps as follows: Variable branching will result in a unit. Without lost of generality let us assume that it is a positive one. Then BCP will generate other positive units, because the binary clauses in F contain exactly one positive and one negative literal. BCP will finally result in a conflict, because $\neg C$ contains the negation of the white assignment. Then the other branch will result in a negative unit. This enables a BCP step which will generate negative units, and which will terminate in a conflict because $\neg C$ contains also the negation of the black assignment. Since BCP is a linear time method [19] we need linear time to show that $F \wedge \neg C$ is unsatisfiable. □

Note that DPLL algorithm is a suitable choice to solve the above problem because it uses variable branching and BCP. The question arises, why should we transform a graph into a SAT problem to check a property which can be checked in linear time in both fields? We think that our model is a new link between the two fields which might help to visualize SAT problems. This is not a problem in case of a 2-SAT problem, but in case of 3-SAT it is a problem. This paper and Lemma 6. in [3] together suggest that there might be a 3-SAT representation of directed graphs which is a black-and-white 3-SAT problem. If one finds that representation, then this could help to visualize 3-SAT problems as a directed graph.

3. Our model and the Aspvall model

In this section we show that the constraint used by our model is more general than the one used by the model of Aspvall et al. Fist we show some examples. In Figure 1.

we see a directed graph with 5 vertices. This graph is also a communication graph, since there is no negation sign in the vertices. A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component of a directed graph is a maximal strongly connected subgraph. There are 2 strongly connected components $([1,2,3,5],[4])$ in the following graph.

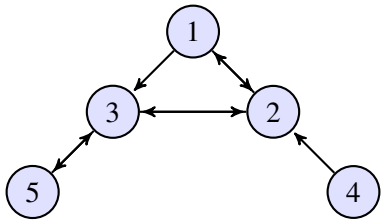

FIGURE 1. A directed graph with 5 vertices

The model of this graph is:

$$\begin{aligned}\mathcal{M} = (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_2 \vee x_3) \wedge \\ (\neg x_3 \vee x_2) \wedge (\neg x_3 \vee x_5) \wedge (\neg x_4 \vee x_2) \wedge (\neg x_5 \vee x_3)\end{aligned} \tag{3.1}$$

From $\mathcal{M}$ we can create $D(\mathcal{M})$ by following the construction defined in [2] by Aspvall et al. Construction steps are the following:

i. For each variable x_i, we add two vertices named x_i and $\neg x_i$ to $D(\mathcal{F}_{\mathcal{M}})$.
ii. For each clause $(x_i \vee x_j)$ of $\mathcal{M}$, we add edges $\neg x_i \to x_j$ and $\neg x_j \to x_i$ to $D(\mathcal{M})$.

In Figure 2. we see the transformed $D(\mathcal{M})$ graph. Aspvall et al. [2] *(Theorem I.)* shows that $\mathcal{M}$ is satisfiable if and only if in $D(\mathcal{M})$ there is no vertex x_i such that it is in the same strong component as its complement $\neg x_i$. We can be sure that this property holds for our model $\mathcal{M}$, because it is always satisfiable.

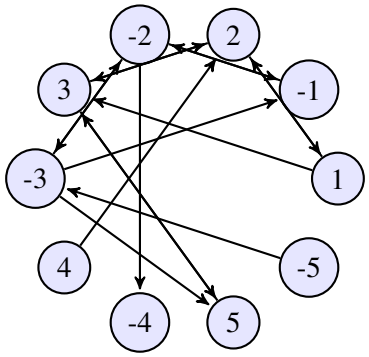

FIGURE 2. $D(\mathcal{F})$, where $\mathcal{F} = \mathcal{M}$

We define the so-called Aspvall kind constraints.

Let $\mathcal{C}_{ASP}(i,j) = (x_i \supset x_j) \wedge (x_i \supset x_j)$, where $1 \leq i,j \leq n$ and $i \neq j$. From this we obtain after simplification that $\neg\mathcal{C}_{ASP}(i,j) = (x_i \vee x_j) \wedge (\neg x_j \vee \neg x_i)$. We can add these clauses to the model to check whether there is a directed path from x_i vertex to x_j and vice versa: $D(\mathcal{M} \wedge \neg\mathcal{C}_{ASP}(i,j))$.

We add two constraints to the example shown in Figure 1. The first one is $\mathcal{C}_{ASP}(1,4) = (x_1 \supset x_4) \wedge (x_4 \supset x_1)$. Let $\mathcal{F} = \mathcal{M} \wedge \neg\mathcal{C}_{ASP}(1,4)$. So we add these two clauses $(x_1 \vee x_4) \wedge (\neg x_1 \vee \neg x_4)$ and these 4 edges, see the red arrows in Figure 3. We see on Figure 3. that in graph $\mathcal{D}(\mathcal{F})$ there exists no vertex x_i which is in the same strong component as its complement $\neg x_i$, hence, $\mathcal{F}$ is still satisfiable.

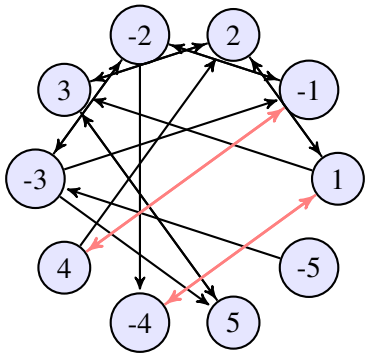

FIGURE 3. $D(\mathcal{F})$, where $\mathcal{F} = \mathcal{M} \wedge \neg\mathcal{C}_{ASP}(1,4)$

Now we add $\mathcal{C}_{ASP}(1,5) = (x_1 \supset x_5) \wedge (x_5 \supset x_1)$. Let $\mathcal{F}' = \mathcal{M} \wedge \neg\mathcal{C}_{ASP}(1,5)$. So we add these two clauses $(x_1 \vee x_5) \wedge (\neg x_1 \vee \neg x_5)$ and these 4 edges, see the red arrows in Figure 4. We can see on Figure 4. that in the graph $\mathcal{D}(\mathcal{F}')$ there exists a vertex x_i which is in the same strong component as its complement $\neg x_i$, hence, $\mathcal{F}'$ is unsatisfiable.

In our approach C is

$$\begin{aligned}(x_1 \supset x_2) \wedge (x_1 \supset x_3) \wedge (x_1 \supset x_4) \wedge \\ (x_2 \supset x_1) \wedge (x_2 \supset x_3) \wedge (x_2 \supset x_4) \wedge \\ (x_3 \supset x_1) \wedge (x_3 \supset x_2) \wedge (x_3 \supset x_4) \wedge \\ (x_4 \supset x_1) \wedge (x_4 \supset x_2) \wedge (x_4 \supset x_3)\end{aligned} \tag{3.2}$$

$$\begin{aligned}(\neg x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (\neg x_1 \vee x_5) \wedge \\ (\neg x_2 \vee x_1) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee x_4) \wedge (\neg x_2 \vee x_5) \wedge \\ (\neg x_3 \vee x_1) \wedge (\neg x_3 \vee x_2) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_3 \vee x_5) \wedge \\ (\neg x_4 \vee x_1) \wedge (\neg x_4 \vee x_2) \wedge (\neg x_4 \vee x_3) \wedge (\neg x_4 \vee x_5) \wedge \\ (\neg x_5 \vee x_1) \wedge (\neg x_5 \vee x_2) \wedge (\neg x_5 \vee x_3) \wedge (\neg x_5 \vee x_4)\end{aligned} \tag{3.3}$$

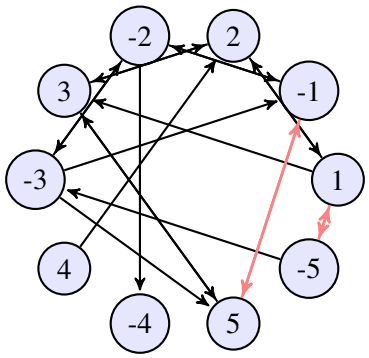

FIGURE 4. $D(\mathcal{F}')$, where $\mathcal{F}' = \mathcal{M} \wedge \neg\mathcal{C}_{ASP}(1,5)$

After simplification $\neg C$ is the following:

$$(x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4 \vee \neg x_5) \tag{3.4}$$

For any i and j we have that $\neg\mathcal{C}_{ASP}(i,j)$ implies $\neg\mathcal{C}$, thus $\neg\mathcal{C}_{ASP}(i,j)$ subsumes $\neg\mathcal{C}$. This means that $\neg\mathcal{C}_{ASP}(i,j)$ defines a stronger constraint to the model, thus $\neg\mathcal{C}$ defines a more general constraint than $\neg\mathcal{C}_{ASP}(i,j)$ to check whether a graph is strongly connected or not.

4. AN APPLICATION

4.1. *Wireless sensor networks*

Ad hoc wireless sensor networks (WSN) are used widely (for example, in military to observe environment). They have the advantage that they consist of sensors with low energy consumption, which can be deployed easily in a cheap way on areas which are out-of-the-way. These sensors are the nodes of WSN. They are capable of processing some limited information and using wireless communication. A big effort has been to do research on how to deploy them in an optimal way to keep efficient energy consumption and communication. Although there are many WSN solutions, the deployment of a WSN is still an active research field [8].

One of the important property of an ad hoc wireless network is node density. The dense layout makes the following properties available: high fault tolerance, high-coverage characteristics, but also cause some problems. The interference is high near dense node areas, and there are a lot of collisions in the case of messaging, which requires complicated operations of a MAC protocol. Because of too many possible routes, routing needs lots of resources [4].

The aim of monitoring techniques is reducing the cost of the distributed algorithms interpreted on the network. The graph, which represents the network, has to be

thinned because of cost-reduction by techniques like disconnecting of nodes, removing links, changing scopes, etc., but the network-quality characteristics (like scalability, coverage, fault tolerance, etc.) must not fall below a required level. The overall aim is to create a scalable, fault-tolerant rare topology, where the degree of the nodes are low, the maximum load is low, energy consumption is low and the paths are short. The following techniques are used to create an optimal topology: reducing the scope of nodes, removing some nodes, introducing a dominating set of nodes, clustering, and adding some new nodes to gain all-all communication [13, 18].

4.2. *Connectivity test by SAT representation*

If we have a WSN, then we can redefine communication graph as follows, we say that $\mathcal{D}$ is a communication graph if and only if the elements of $\mathcal{V}$, the vertices, represent the sensor nodes of the WSN, and elements of $\mathcal{E}$, the edges, represent a one way communication between two nodes.

We intend to examine which sensors can communicate with which ones, thus we can create the communication graph and its 2-SAT representation. In this approach it can be checked fairly quickly in a given state of a randomly distributed sensor network (assuming all sensors are awaken) whether it can be ensured that each sensor can communicate with every other one [6].

For example let us model a randomly distributed heterogeneous sensor network which has 10 nodes. For representing the input clause set for a SAT solver, the

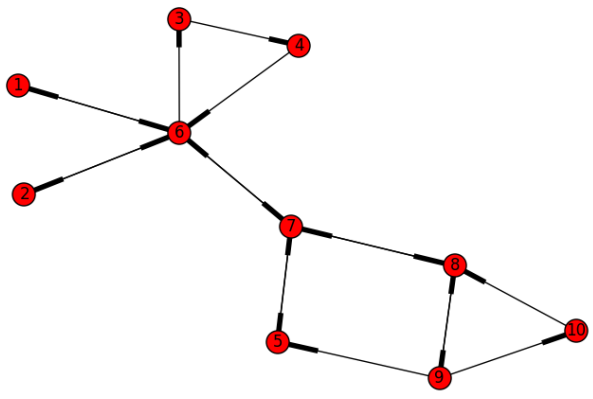

FIGURE 5. Heterogeneous sensor nodes with their communication graph

DIMACS CNF[1] format is commonly used, which references a Boolean variable by its (1-based) index. A negative literal is referenced by the negated reference to its variable. A clause is represented by a sequence of the references to its literals terminated by a ”0”.
DIMACS CNF format is the following: `p cnf 10 20`

```
c model
-1 6 0
-6 1 0
-2 6 0
-6 2 0
-6 3 0
-3 4 0
-4 6 0
-5 6 0
-6 7 0
-7 6 0
-5 7 0
-7 5 0
-7 8 0
-8 9 0
-9 8 0
-9 10 0
-10 8 0
-9 5 0
c constraint
1 2 3 4 5 6 7 8 9 10 0
-1 -2 -3 -4 -5 -6 -7 -8 -9 -10 0
```

Our main goal is to examine the produced DIMACS CNF file with `MiniSat 2.2.0`[2], which is a complete SAT solver, which returns unsatisfiable (UNSAT). Thus the model fulfills the requirements, namely communication is ensured between any two nodes. In this example the result is UNSAT, thus the represented communication graph is strongly connected.

This model can also give a more detailed analysis. For example, if we take the sensors out of the model one by one, and with the remaining we test the communication problem again between any two nodes. If the solver returns satisfiable (SAT) for the reduced model, then the currently removed node is extremely important to the sensor network, as its removal breaks the requirement of unhindered communication between any two sensors. The significance of this in graph theory is that the removal

[1] www.satlib.org/Benchmarks/SAT/satformat.ps

[2] http://minisat.se

of this node makes the graph not connected anymore. In our example the removal of nodes 3, 4, 5, 6, 7, 8 and 9 would return with SAT, thus the malfunction of those sensors means the communication in that sensor network is inadequate.

It is generally true that if the graph is strongly connected, then a DPLL-based SAT solver returns the following values.

- number of solutions : 0
- number of conflicts : 2
- number of decisions : 1
- number of unit propagations : $u = 2n$, where n the number of literals (number of vertices)

These results also show that the black-and-white 2-SAT problem with the black-and-white constraint can be solved in linear time since the number of decisions is 1, and the number of unit propagations is $2n$.

5. Conclusions and future work

In this paper we have introduced a SAT representation that can be used for modeling directed graphs. The only restriction is that the names of the vertices should be boolean variables. This model, which is a 2-SAT problem, makes it possible to check whether a graph is strongly connected by adding two clauses, the black and the white ones, to the model and asking a SAT solver whether this formula is unsatisfiable. The two clauses are a constraint which state that it is not true that there is a path from any vertex to any other one. This constraint is more general than the one used by Aspvall et al. We have shown that the representation of a strongly connected graph is a black-and-white 2-SAT problem. The black-and-white SAT problem appears also as a special case of weakly nondecisive SAT problems, see Lemma 6. in [3]. This suggests two things: the black-and-white SAT problem could be an interesting problem in general, and since there are weakly nondecisive 3-SAT problems, which are also black-and-white, there might be a 3-SAT representation of directed graphs. Our model can be applied also in a natural way to networks of sensors, such as wireless sensor networks. If we use a SAT solver to solve our model, then it gives back not only the solution but also lots of numbers. An interesting question is that how one can use these metrics, like *number of solutions, conflicts, decisions, unit propagations*, to say something about the topology of the graph.

It seems the the following generalized notion is also interesting: F is a generalized black-and-white SAT problem if and only if F is satisfiable and for any assignment A: if A is a solution of F, then F has another solution B such that $A = \neg B$. For example, the empty clause set is a generalized black-and-white SAT problem. Another interesting example is the set of black and the white clauses. We are going to investigate this notion in a future study.

REFERENCES

[1] A. Biere, M. Heule, H. van Maaren, T. Walsh, *Handbook of Satisfiability*. Amsterdam: IOS Press, 2009.

[2] Bengt Aspvall, Michael F. Plass and Robert Endre Tarjan, "A Linear-Time Algorithm For Testing The Truth Of Certain Quantified Boolean Formulas," *Information Pprocessing Letters*, vol. 8, no. 3, pp. 121–123, 1979, doi: 10.1016/0020-0190(79)90002-4.

[3] C. Biró, G. Kusper, and T. Tajti, "How to generate weakly nondecisive sat instances," in *2013 IEEE 11th International Symposium on Intelligent Systems and Informatics (SISY)*," doi: 10.1109/SISY.2013.6662583, Sept 2013, pp. 265–269.

[4] Bolic, Miodrag, Simplot-Ryl, David Stojmenovic, Ivan (eds.), *RFID Systems - Research Trends and Challenges*. New York: John Wiley & Sons, 2010.

[5] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986, doi: 10.1109/TC.1986.1676819. [Online]. Available: http://dx.doi.org/10.1109/TC.1986.1676819

[6] Cs. Biro, G. Kusper, T. Radvanyi, S. Kiraly, P. Szigetvary, P. Takacs, "SAT Representation of Randomly Deployed Wireless Sensor Networks," *Proc. of ICAI09*, vol. 2, pp. 101–111, 2014, doi: 10.14794/ICAI.9.2014.2.101.

[7] L. Hellerman, "A catalog of three-variable or-invert and and-invert logical circuits," *IEEE Transactions on Electronic Computers*, vol. EC-12, no. 3, pp. 198–223, June 1963, doi: 10.1109/PGEC.1963.263531.

[8] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, "Wireless sensor networks: a survey ," *Computer Networks*, vol. 38, no. 4, pp. 393–424, 2002, doi: doi:10.1016/S1389-1286(01)00302-4.

[9] R. M. Karp, *Reducibility among Combinatorial Problems*. Boston, MA: Springer US, 1972, pp. 85–103, doi: 10.1007/978-1-4684-2001-29.

[10] R. P. Langlands, *Problems in the theory of automorphic forms to Salomon Bochner in gratitude*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1970, pp. 18–61, doi: 10.1007/BFb0079065.

[11] M. Davis, G. Logemann, D. Loveland, "A Machine Program for Theorem Proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962, doi: 10.1145/368273.368557.

[12] S.-i. Minato, "Zero-suppressed bdds for set manipulation in combinatorial problems," in *Proceedings of the 30th International Design Automation Conference*, ser. DAC '93, doi: 10.1145/157485.164890. New York, NY, USA: ACM, 1993, pp. 272–277. [Online]. Available: http://doi.acm.org/10.1145/157485.164890

[13] Paolo Santi, "Topology Control in Wireless Ad Hoc and Sensor Networks ," *ACM Computing Surveys* , vol. 37, no. 2, pp. 164–194, 2005, doi: 10.1145/1089733.1089736.

[14] S. A. Cook, "The Complexity of Theorem-Proving Procedures," *Proc. of STOC'71*, pp. 151–158, 1971, doi: 10.1145/800157.805047.

[15] M. Sharir, "A strong-connectivity algorithm and its applications in data flow analysis," *Computers And Mathematics with Applications*, vol. 7, no. 1, pp. 67 – 72, 1981, doi: http://dx.doi.org/10.1016/0898-1221(81)90008-0.

[16] A. Weil, "Über die bestimmung dirichletscher reihen durch funktionalgleichungen," *Mathematische Annalen*, vol. 168, no. 1, pp. 149–156, 1967, doi: 10.1007/BF01361551.

[17] A. J. Wiles, "Modular elliptic curves and fermat's last theorem," *ANNALS OF MATH*, vol. 141, pp. 443–551, 1995, doi: 10.2307/2118559.

[18] Yu Wang, "Topology Control for Wireless Sensor Networks," *Springer - Wireless Sensor Networks and Applications*, vol. 148, no. 2, pp. 113–147, 2008, doi: 10.1007/978-0-387-49592-7.

[19] H. Zhang and M. E. Stickel, "An efficient algorithm for unit propagation," in *In Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, 1996, pp. 166–169.

Authors' addresses

Csaba Biró
Eszterházy Károly University, Hungary, Institute of Mathematics and Informatics
E-mail address: biro.csaba@uni-eszterhazy.hu

Gábor Kusper
Eszterházy Károly University, Hungary, Institute of Mathematics and Informatics
E-mail address: kusper.gabor@uni-eszterhazy.hu

11 Annex 4. - How to generate weakly nondecisive SAT instances

This is the submitted version of Cs. Biró, G. Kusper, T. Tajti, *How to generate weakly nondecisive SAT instances*, Proceedings of SISY 2013, DOI: 10.1109/SISY.2013.6662583, pp. 265–269, 2013. See also [99].

How to Generate Weakly Nondecisive SAT Instances

Csaba Biró, Gábor Kusper and Tibor Tajti
Eszterházy Károly College, Eger
Institute of Mathematics and Informatics
Email: see http://fmv.ektf.hu

***Abstract*—In this paper we solve a problem which was unsolved until now. The problem is: How to generate weakly nondecisive SAT instances? As a solution we introduce a very simple algorithm, called WnDGen, which generates weakly nondecisive clause sets. Its input is a clause and a number, its output is a clause set. WnDGen(C, k) works as follows: It generates all k-length subsets of C. For each subset it adds k clauses to the output, negating every time another literal in the subset. Then it does the same with the negation of C. We show that the resulting clause set is always weakly nondecisive and satisfiable. Actually, C and negation of C are solutions of the SAT instance generated by WnDGen(C, k). We also show that there is a threshold: Let n be the length of C; let S be the union of WnDGen(C, k) and the set of C and negation of C; if $n \geq 2k-3$, then S is unsatisfiable, if $n < 2k-3$, the S is satisfiable. We show that around this threshold there are SAT instances, which are difficult for state-of-the-art SAT solvers, i.e., they are good for testing SAT solvers.**

***Index Terms*—SAT problem generator, nondecisive clause, blocked clause**

I. Introduction

In this paper we study the notion of weakly nondecisive literal. It is a generalization of blocked literals but a specialization of nondecisive literals. The notion of blocked literal and clause was introduced by O. Kullman in [2], [3]. A. V. Gelder generalized this idea and introduced the notion of nondecisive literal and clause [1]. The definitions of these notions are given in the next section. Here we give a general overview.

O. Kullmann showed that a blocked clause can be removed or added without changing the satisfiability of the clause set. He studied the blocked clauses, because he wanted to have a better worst-case upper bound for solving the SAT problem [3].

A. V. Gelder generalized further this idea and introduced the notion of nondecisive clause. He showed that a nondecisive clause can be removed or added without changing the satisfiability of the clause set [1] .

G. Kusper showed in [5] that blocked clause sets (a clause set is blocked if all its clauses are blocked) are satisfiable. The same is note true for nondecisive clause sets. He wanted to find between these two notions a new one which still ensures satisfiablity. Therefore, he introduced the notion of weakly nondecisive clause set in his thesis [4]. He showed that not all weakly nondecisive clause sets are satisfiable. But he left open the question of how to generate weakly nondecisive clause sets. We show two generator functions: WnDGen1 and WnDGen, which answer this question.

WnDGen1(C, k) works as follows (C is a clause, k is a number): It generates all k-length subsets of C. For each subset it adds k clauses to the output, negating every time another literal in the subset. WnDGen is the union of WnDGen1 on C and WnDGen1 on the negation of C.

We see that these functions are very simple ones, but it was very difficult to find them. First we created a function which tests whether a clause set is weakly nondecisive or not. Then we generated lots of random clause sets and we were lucky to find some weakly nondecisive ones. We did not do this blindly, because we already had a weakly nondecisive clause set from [4]. Then we started to analyze the structure of the new ones. Finally we have found these two functions after several months of work. There should exists some more way to generate weakly nondecisive clause set, because these two functions is unable to generate the very same example from [4].

Our main lemma states the following: Let n be the length of C; let S be the union of WnDGen(C, k) and the set of C and negation of C; if $n \geq 2k-3$, then S is unsatisfiable, if $n < 2k-3$, the S is satisfiable. This means that there is a threshold, $2k-3$, such that below that S is satisfiable, otherwise unsatisfiable. It is a common experience that around a threshold there are interesting things. In this case it turned out that on the threshold, i.e., if $|C| = 2k-3$, the generated SAT instances are hard to solve for state-of-the-art SAT solvers.

One can download the WnDGen tool from the website: *http://fmv.ektf.hu*

II. Definitions

In this section we give the definitions. What we define is written in italics. In the formal definitions we use the meta symbol colon (:) to identify what we define. The word "iff" is the abbreviation of the phrase "if and only if".

Let V be a finite *set of Boolean variables*. The *negation of a variable* v is denoted by $\overline{v}$. *Negation of negation* is defined as $\overline{\overline{v}} := v$. Given a set U, we denote $\overline{U} := \{\overline{u} \mid u \in U\}$ and call the *negation of the set* U. *Variables of a set* C is denoted by $\mathbf{V}ar(C)$ and defined as $\mathbf{V}ar(C) := (C \cup \overline{C}) \cap V$.

Literals are the members of the set $W := V \cup \overline{V}$. *Positive literals* are the members of the set V. *Negative literals* are their

negations. If w denotes a negative literal $\overline{v}$, then $\overline{w}$ denotes the positive literal v.

Clauses and *assignments* are finite sets of literals that do not contain simultaneously any literal together with its negation. A *clause set* is a finite set of clauses. In this section B, C, D, and E are clauses; S is a clause set.

The *length* of a set U is its cardinality, denoted by $|U|$. Let $n := |V|$.

If $|C| = k$, then we say that C is a k*-clause*. Special cases are *unit clauses* or *units* which are 1-clauses, and *clear* or *total clauses* which are n-clauses.

C is *subsumed by* S, denoted by $C \supseteq\in S$, iff

$$C \supseteq\in S :\Longleftrightarrow \exists[B \in S]B \subseteq C.$$

An assignment M is a *model* or *solution* of S iff for all $C \in S$ we have $M \cap C \neq \emptyset$. S is *satisfiable* iff it has a model, otherwise it is *unsatisfiable*.

If a clause set subsumes all total clauses, then it is unsatisfiable. In this paper we do not prove this observation, we use it as an axiom.

The *clause difference* of C and D, denoted by $diff(C, D)$, is defined as $diff(C, D) := C \cap \overline{D}$. If $diff(C, D) \neq \emptyset$ then we say that C *differs from* D.

Resolution can be performed on two clauses iff they differ only in one variable. If resolution can be performed then the *resolvent*, denoted by $Res(C, D)$, is defined as $Res(C,D) := (C \cup D) \setminus (diff(C,D) \cup diff(D,C))$.

We define the notion of *blocked, nondecisive* and *weakly nondecisive* literal, denoted by $Blck(c, C, S)$, $NonD(c, C, S)$, $WnD(c, C, S)$, respectively.

$$\begin{aligned} Blck(c, C, S) :\Longleftrightarrow\ & \forall[B \in S][\overline{c} \in B] \\ & \exists[b \in B][b \neq \overline{c}]\overline{b} \in C \end{aligned} \tag{1}$$

$$\begin{aligned} NonD(c, C, S) :\Longleftrightarrow\ & \forall[B \in S] \\ & [\overline{c} \in B](\exists[b \in B][b \neq \overline{c}]\overline{b} \in C \vee \\ & Res(C, B) \cup \{c\} \supseteq\in S \setminus \{C\}) \end{aligned} \tag{2}$$

$$\begin{aligned} WnD(c, C, S) :\Longleftrightarrow\ & \forall[B \in S][\overline{c} \in B] \\ & (\exists[b \in B][b \neq \overline{c}]\overline{b} \in C \vee Res(C, B) \supseteq\in S) \end{aligned} \tag{3}$$

In other words, a literal is weakly nondecisive in C and S iff it is blocked or the resolvent of C and a non-blocking B is subsumed by S.

C is a *weakly nondecisive clause* in S iff it has a weakly nondecisive literal. S is a *weakly nondecisive clause set* iff its clauses are weakly nondecisive.

In this paper we generate weakly nondecisive clause sets. The first generator function is *WnDGen1*. If C is a clause, k is a natural number, and $2 \leq k \leq |C|$, then

$$\begin{aligned} WnDGen1(C,k) := \{D \mid & \mathbf{V}ar(D) \\ & \subseteq \mathbf{V}ar(C) \wedge |D| = k \\ & \wedge |diff(D,C)| = 1\}. \end{aligned} \tag{4}$$

This means that WnDGen1(C, k) generates all k-clauses which differ only in one literal from C. C is called the generator clause.

We give two examples for WnDGen1:

$$\begin{aligned} WnDGen1(\{a, b, c\}, 2) = \{&\{a,\overline{b}\}, \\ &\{\overline{a},b\},\{a,\overline{c}\},\{\overline{a},c\},\{b,\overline{c}\},\{\overline{b},c\}\}. \end{aligned} \tag{5}$$

$$\begin{aligned} WnDGen1(\{a, b, c\}, 3) = \{&\{a,b,\overline{c}\}, \\ &\{a,\overline{b},c\},\{\overline{a},b,c\}\}. \end{aligned} \tag{6}$$

Alternatively, we can introduce an auxiliary function, *WnDGen0*, and redefine WnDGen1 with the help of it. We will use WnDGen0 in Section IV.

$$\begin{aligned} WnDGen0(D) := \{E \mid & \mathbf{V}ar(E) = \mathbf{V}ar(D) \\ & \wedge |diff(E,D)| = 1\}. \end{aligned} \tag{7}$$

$$\begin{aligned} WnDGen1(C,k) := \bigcup\{&WnDGen0(D) \\ & \mid \mathbf{V}ar(D) \subseteq \mathbf{V}ar(C) \wedge |D| = k\}. \end{aligned} \tag{8}$$

The second generator function is *WnDGen*. If C is a clause, k is a natural number, and $2 \leq k \leq |C|$, then

$$\begin{aligned} WnDGen(C, k) := & WnDGen1(C,k) \\ & \cup WnDGen1(\overline{C}, k). \end{aligned} \tag{9}$$

III. Properties of WnDGen1 and WnDGen

In this section we prove that WnDGen1 and WnDGen generate weakly nondecisive clause sets. We give a criterion on the shape of clauses which are subsumed by WnDGen1. Based on this result we show how to use WnDGen to generate unsatisfiable clause sets. First we give the shape criterion.

Lemma 1: Assume C and D are clauses and $\mathbf{V}ar(D) \subseteq \mathbf{V}ar(C)$. Assume k is a natural number and $2 \leq k \leq |C|$. If $|diff(D, C)| \geq 1$ and $|D| \geq |diff(D, C)| + k - 1$, then D is subsumed by WnDGen1(C, k).

Proof: Without loss of generality we may assume that C contains only positive literals. Then WnDGen1(C, k) contains all clauses with $k-1$ positive literals and 1 negative one. Since $|diff(D, C)|$ is the number of negative literals in D and $|D| - |diff(D, C)|$ is the number of positive literals in D we have that D is subsumed by WnDGen1(C, k) if $|diff(D, C)| \geq 1$ and $|D| - |diff(D,C)| \geq k - 1$. ■

Note, that there is a special case. If $|diff(D, C)| = 1$ then it is enough to show that $|D| \geq k$ to prove that D is subsumed by WnDGen1(C, k). We will use this case in Lemma 2. We use another case in Lemma 3 when $|diff(D, C)| = k - 1$. In this case it is enough to show that $|D| \geq 2k - 2$.

Lemma 2: Assume C is a clause, k is a natural number, and $2 \leq k \leq |C|$. Let S = WnDGen1(C, k). Then S is a weakly nondecisive clause set.

Proof: Let S = WnDGen1(C, k). The main idea of the proof is that for any A in S (this means that $|A| = k$ and $|diff(A,C)| = 1$) there is a literal a, which occurs positively (negatively) in A but negatively (positively) in C and this literal is a nondecisive literal in A and S. To show this, we show that for any B in S in which a occurs negatively (positively) we have that either $Res(A,B)$ is not a clause, i.e., B blocks A; or $Res(A,B)$ is a clause (this means that $|diff(A,B)| = 1$), but in this case, by Lemma 1, $Res(A,B)$ is subsumed in S, because $|Res(A,B)| \geq k$ (in general we have $|Res(A,B)| \geq k-1$, but $|Res(A,B)| = k-1$ would contradict the assumption

$|diff(A,C)| = 1$) and $|diff(Res(A,B),C)| = 1$ (which comes from that $diff(A,C) = \{a\}$, $|diff(B,C)| = 1$, and $a \notin Res(A,B)$). Hence, S is weakly nondecisive. ■

Actually, we could state a more powerful statement, that in WnDGen1(C, k) all literals are weakly nondecisive, see Lemma 4. But this weaker lemma helps us to prove that WnDGen(C,k) is also weakly nondecisive.

Lemma 3: Assume C is a clause, k is a natural number, and $2 \leq k \leq |C|$. Let S = WnDGen(C, k). Then S is a weakly nondecisive clause set.

Proof: Let S = WnDGen(C, k). We show that S is weakly nondecisive. To show this we show that any A in S is weakly nondecisive. Without loss of generality we may assume that A is an element of WnDGen1(C, k), this means that $|A| = k$ and $|diff(A,C)| = 1$. The main idea of the proof is that there is a literal a, which occurs positively (negatively) in A, but negatively (positively) in C and this literal is a nondecisive literal in A. To show this, we show that for any B in S we have that either B blocks A or $Res(A,B)$ is subsumed in S. We know from Lemma 2 that this is true if B is an element of WnDGen1(C, k). If B is in WnDGen1($\overline{C}$, k) then we have $|diff(B,\overline{C})| = 1$, i.e., $|diff(B,C)| = k-1$ and we also know that $|diff(A,C)| = 1$ which means that there is only one case when B does not block A, when $|\mathbf{V}ar(A) \cap \mathbf{V}ar(B)| = 1$, but in this case, by Lemma 1 $Res(A,B)$ is subsumed in S, because $|Res(A,B)| = 2k-2$ and $|diff(Res(A,B),C)| = k-1$. Hence, S is weakly nondecisive. ■

Now we prove that in WnDGen1(C, k) all literals are weakly nondecisive.

Lemma 4: Assume C is a clause, k is a natural number, and $2 \leq k \leq |C|$. Let S = WnDGen1(C, k). Then all literals in S are weakly nondecisive.

Proof: Without loss of generality we may assume that C contains only positive literals. From definition of WnDGen1 we know that in each clause in S there is only one negative literal, this is the literal in which each clause differs from C. From the proof of Lemma 2 we know that these literals are weakly nondecisive. This means that all negative literals in S are weakly nondecisive. From this we can obtain that all positive literals are also weakly nondecisive, because if a positive literal is involved in a resolution, there must be involved also a negative one, and the resolvent will be subsumed because the negative literal is weakly nondecisive. Hence, all literals in S are weakly nondecisive. ■

We proof now that $WnDGen(C,k)$ is always satisfiable, which means naturally that $WnDGen1(C,k)$ is always satisfiable.

Lemma 5: Assume C is a clause, k is natural number, and $2 \leq k \leq |C|$. Then $WnDGen(C,k)$ is satisfiable.

Proof: It is easy to see that for all $D \in WnDGen(C,k)$ we have that $D \cap C \neq \emptyset$ and $D \cap \overline{C} \neq \emptyset$. This means that C and $\overline{C}$ are models for $WnDGen(C,k)$. Hence, $WnDGen(C,k)$ is satisfiable. ■

We have seen that WnDGen(C, k) is satisfiable. Now let us see what we should add to it to make it unsatisfiable.

Lemma 6: Assume C is a clause, k is natural number, and $2 \leq k \leq |C|$. Then $WnDGen(C,k) \cup \{C, \overline{C}\}$ is unsatisfiable if and only if $|C| \geq 2k-3$.

Proof: Let $V = \mathbf{V}ar(C)$, i.e., $n = |C|$. Without loss of generality we may assume that C contains only positive literals.

($\Leftarrow$) We assume that $n \geq 2k-3$. We show that $WnDGen(C,k) \cup \{C, \overline{C}\}$ is unsatisfiable. To show this, it is enough to show that $WnDGen(C,k) \cup \{C, \overline{C}\}$ subsumes all total clauses. Any total clause has either $0, 1, \ldots, n-1$, or n negative literals. C subsumes those total clauses which have 0 negative literals (actually there is only one such clause: C itself). Furthermore, $\overline{C}$ subsumes those total clauses which have n negative literals (because it subsumes itself). Therefore, we have to show that $WnDGen(C,k)$ subsumes all total clauses with 1, 2, …, $n-2$, or $n-1$ negative literals. Let D be a total clause such that $D \neq C$. This means that $|diff(D,C)| \geq 1$, and, by definition of total clause, $|D| = n$. From Lemma 1 we know that D is subsumed by WnDGen1(C, k) if $|D| \geq |diff(D,C)| + k - 1$, i.e., $n-k+1 \geq |diff(D,C)|$. Note, that the number $|diff(D,C)|$ is the number of negative literals in D since C contains only positive literals. This means that WnDGen1(C, k) subsumes those total clauses which contain $1, 2, \ldots, n-k$, or $n-k+1$ negative literals. From this, by definition of WnDGen, we can obtain that we have to show that WnDGen1($\overline{C}$, k) subsumes those total clauses which contain $n-k+2, n-k+3, \ldots, n-2$, or $n-1$ negative literals; on the other way around, we have to show that WnDGen1($\overline{C}$, k) subsumes those total clauses which contain $n-(n-k+2), n-(n-k+3), \ldots, n-(n-2)$, or $n-(n-1)$ positive literals, i.e., $k-2, k-3, \ldots, 2$, or 1 ones. So there are $k-2$ such cases. From Lemma 1 we know that WnDGen1($\overline{C}$, k) subsumes those total clauses which contain $1, 2, \ldots, n-k$, or $n-k+1$ positive literals. So there are $n-k+1$ such cases. This means that we have to show that the number of cases which are subsumed by WnDGen1($\overline{C}$, k) is greater or equal than the number of cases which remain un-subsumed by WnDGen1(C, k), i.e., $n-k+1 \geq k-2$. We already know that $n \geq 2k-3$, if we add $-k+1$ to both sides, then we obtain $n-k+1 \geq k-2$. This means that $WnDGen(C,k)$ subsumes all total clauses with 1, 2, …, $n-2$, or n-1 negative literal. Hence, $WnDGen(C,k) \cup \{C, \overline{C}\}$ is unsatisfiable.

($\Rightarrow$) We assume $WnDGen(C,k) \cup \{C, \overline{C}\}$ is unsatisfiable. We show $n \geq 2k-3$. From the assumption we obtain that $WnDGen(C,k) \cup \{C, \overline{C}\}$ subsumes all total clauses. This means that $WnDGen(C,k)$ subsumes all total clauses with 1, 2, …, or $n-1$ negative literals. Note, that for an arbitrary clause D the number $|diff(D,C)|$ is the number of negative literals in D since C contains only positive literals. From Lemma 1 we can obtain that WnDGen1(C, k) subsumes those total clauses which contain $1, 2, \ldots, n-k$, or $n-k+1$ negative literals. We can also obtain that WnDGen1($\overline{C}$, k) subsumes those total clauses which contain $n-1, n-2, \ldots, n-(n-k)$, or $n-(n-k+1)$ negative literals. This means that both WnDGen1(C, k) and WnDGen1($\overline{C}$, k) subsumes $n-k+1$ cases from the sequence $1, 2, \ldots,$ or $n-1$, but start from different ends. From our assumption we know that there is no such case which is not subsumed from this sequence. This means that the number of subsumed cases must be greater or equal than the number of cases in the sequence, i.e.,

$2 * (n - k + 1) \geq n - 1$. After simplification we obtain $n \geq 2k - 3$. Hence, $|C| \geq 2k - 3$. ■

This is our main lemma. This lemma was a kind of surprise for us, because we had a feeling that WnDGen with the generator clause and with the negation of the generator clause should be unsatisfiable. It turned out that sometimes this structure is satisfiable. This lemma gives as a threshold, $2k - 3$, such that below that this structure is satisfiable. It is a common experience that around a threshold there are interesting things. In this case it turned out that on the threshold, i.e., if $|C| = 2k - 3$, the generated SAT instances are hard to solve for state-of-the-art SAT solvers.

IV. Test results

At the first sight we had no idea who could we use WnDGen to solve pratical problems. Afterwards we had the idea that Lemma 6 gives us a threshold, and maybe the generated SAT instances around this threshold are interesting. It turned out that on the threshold, i.e., if $|C| = 2k - 3$, the generated SAT instances are hard to solve for state-of-the-art SAT solvers. We developed a so called WnDGen tool, which can generate WnD SAT instances with the WnDGen algorithm. One can download this tool from *http://fmv.ektf.hu/*. The tool has a switch, $-unsat$, which puts two extra clauses at the end of the generated file, the generator clause and its negation, i.e., it creates the structure used in Lemma 6. This case is called *unsat* in the rest of the paper. By *sat* we mean those SAT instances which where generated without the $-unsat$ switch. In the rest of the paper let $n = |C|$, i.e., the length of the generator clause.

Usage: *java WnDGen switches N K*. Generates a *WnD* clause set with *N* variables. Each clause has *K* literals. If *K* is not given, then each clause has *3* literals.
Switches:

- -sat: Generates a satisfiable clause set. It is the default switch.
- -unsat: Tries to generate an unsatisfiable clause set, which is *WnD* upto the two last clauses.
- -help: Prints how to use *WnDGen*.

Our tests were done on Intel quadcore machine with 6 GB of RAM running at 2.66 Ghz. For testing, we used the MiniSat[1] 2.2.0. . The MiniSat is a modern CDCL solver, has four major features of these: - Conflict-driven clause learning [7], [8], - Random search restarts [9], - Boolean constraint propagation using lazy data structures [10], - Conflict-based adaptive branching [10]. For each instance, we used a timeout of 1 hour. The result are presented in Tab. I and Tab. II. The first column presents the WnDGen instances. In the second column, shows number of restarts. The next four columns gives information for number of conflicts, decisions, propagations and conflict literals. Finally, the last column, shows CPU times.

V. Conclusions and future work

In this paper we have introduced two SAT instance generators, WnDGen1 and WnDGen, which generate highly structured SAT instances, which have very interesting properties.

[1]http://www.minisat.se

WnDGen	restarts	conflicts	decisions	propagations	conflict literals	CPU time (s)
n=5 k=4	1	1	4	7	3	0.00
n=7 k=5	1	9	12	30	30	0.00
n=9 k=6	1	31	34	90	124	0.00
n=11 k=7	2	105	111	304	524	0.01
n=13 k=8	3	283	297	796	1812	0.04
n=15 k=9	6	1502	1533	4320	11028	13.64
n=17 k=10	9	6335	6406	18510	52287	9.61
n=19 k=11	12	24960	25089	72661	229711	157.14
n=21 k=12	15	78298	78472	222964	812344	2779.43

Table I
RUNTIMES AND RESULTS (ALL INSTANCES IS SAT)

WnDGen	restarts	conflicts	decisions	propagations	conflict literals	CPU time (s)
n=5 k=4	1	8	7	18	12	0.00
n=7 k=5	1	22	21	54	49	0.00
n=9 k=6	1	72	71	192	228	0.00
n=11 k=7	3	256	257	705	1056	0.01
n=13 k=8	5	935	945	2651	4805	0.11
n=15 k=9	8	3481	3514	10148	21544	1.92
n=17 k=10	11	12960	13029	37004	92650	30.45
n=19 k=11	14	49168	49296	140618	399670	448.03
n=21 k=12	TIME OUT					

Table II
RUNTIMES AND RESULTS (ALL INSTANCES IS UNSAT)

Firstly, these SAT instances are satisfiable, but have only few solutions, the generator clause and its negation.

Secondly, if we add those clauses, then this structure is unsatisfiable, but not always, there is a threshold. And this threshold allow us to generate difficult SAT instances. We created a tool, so one can play with WnD SAT insatnces. This tool also has a $-unsat$ switch.

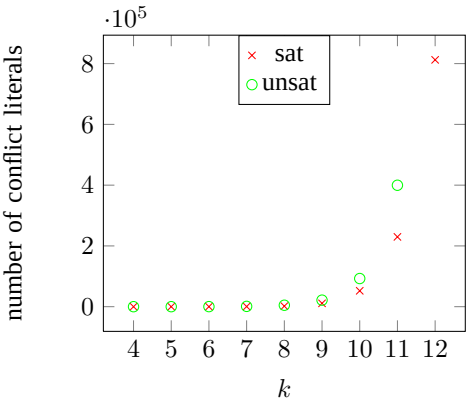

Figure 1. Case: $n = 2k - 3$

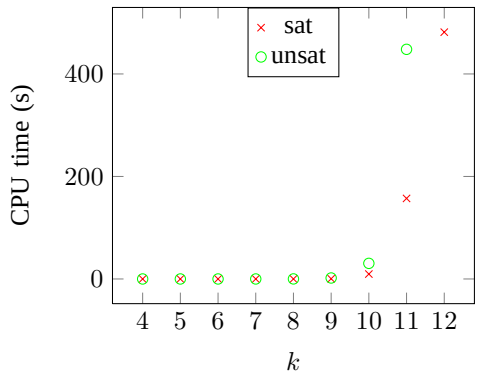

Figure 2. Case: $n = 2k - 3$

As a future work we shall try to delete clauses from SAT instances generated with the $-unsat$ switch and try to keep the unsatisfiable property of them. We think that it is possible, since WnD SAT instances are highly structured, so they contain lot of implied clauses.

REFERENCES

[1] A. V. Gelder, *Propositional Search with k-Clause Introduction Can be Polynomially Simulated by Resolution*, Proceedings of the 5th International Symposium on Artificial Intelligence and Mathematics, 1998.

[2] O. Kullmann, *New methods for 3-SAT decision and worst-case analysis*, Theoretical Computer Science, 223(1-2):1–72, 1999.

[3] O. Kullmann, *On a generalization of extended resolution*, Discrete Applied Mathematics, 96-97(1-3):149–176, 1999.

[4] G. Kusper, *Solving and Simplifying the Propositional Satisfiability Problem by Sub-Model Propagation*, PhD thesis, Johannes Kepler University Linz, RISC Institute, 1–146, 2005.

[5] G. Kusper, *Finding Models for Blocked 3-SAT Problems in Linear Time by Systematical Refinement of a Sub-Model*, Lecture Notes in Computer Science 4314, 128–142, 2007.

[6] G. Tseitin, *On the Complexity of Derivation in Propositional Calculus*, Seminars in Mathematics Volume 8: Studies in Constructive Mathematics and Mathematical Logic, Part II:115-125, 1968.

[7] Marques-Silva, J., Sakallah, K.A., *GRASP: A new search algorithm for satisability*, Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 220227. Springer, Heidelberg (1996)

[8] Marques-Silva, J., Sakallah, K.A., *GRASP-A search algorithm for propositional satisability*, IEEE Transactions on Computers 48(5), 506521 (1999)

[9] Gomes, C.P., Selman, B., Kautz, H.,*Boosting combinatorial search through randomization*, National Conference on Articial Intelligence, pp. 431437 (July 1998)

[10] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.,*Engineering an efficient SAT solver*, Design Automation Conference, pp. 530535 (June 2001)

12 Annex 5. - Simplifying the propositional satisfiability problem by sub-model propagation

This is the submitted version of G. Kusper, L. Csőke, G. Kovásznai, *Simplifying the propositional satisfiability problem by sub-model propagation*, Annales Mathematicae et Informaticae, Vol. 35, ISSN 1787-5021, pp. 75–94, 2008. See also [90].

Annales Mathematicae et Informaticae
Manuscript
http://www.ektf.hu/ami

Simplifying the Propositional Satisfiability Problem by Sub-Model Propagation

Gábor Kusper, Lajos Csőke, Gergely Kovásznai

Eszterházy Károly College

10 December 2008

Abstract

We describes cases when we can simplify a general SAT problem instance by sub-model propagation. Assume that we test our input clause set whether it is blocked or not, because we know that a blocked clause set can be solved in polynomial time. If the input clause set is not blocked, but some clauses are blocked, then what can we do? Can we use the blocked clauses to simplify the clause set? The Blocked Clear Clause Rule and the Independent Blocked Clause Rule describe cases when the answer is yes. The other two independent clause rules, the Independent Nondecisive- and Independent Strongly Nondecisive Clause Rules describe cases when we can use nondecisive and strongly nondecisive clauses to simplify a general SAT problem instance.

Keywords: SAT, blocked clause, nondecisive clause

MSC: 03-04

1. Introduction

Propositional Satisfiability is the problem of determining, for a formula of the propositional calculus, if there is an assignment of truth values to its variables for which that formula evaluates the true. By SAT we mean the problem of propositional satisfiability for formulae in conjunctive normal form (CNF).

SAT is the first, and one of the simplest, of the many problems which have been shown to be NP-complete [Coo71]. It is dual of propositional theorem proving, and many practical NP-hard problems may be transformed efficiently to SAT. Thus, a good SAT algorithm would likely have considerable utility. It seems improbable that a polynomial time algorithm will be found for the general SAT problem but we know that there are restricted SAT problems that are solvable in polynomial time. So a "good" SAT algorithm should check first the input SAT instance whether it is

an instance of such a restricted SAT problem or can be simplified by a preprocess step. In this paper we introduce some possible simplification techniques. We list some polynomial time solvable restricted SAT problems:

1. The restriction of SAT to instances where all clauses have length k is denoted by k-SAT. Of special interest are *2-SAT* and *3-SAT*: 3 is the smallest value of k for which k-SAT is **NP**-complete, while 2-SAT is solvable in linear time [EIS76, APT79].

2. *Horn SAT* is the restriction to instances where each clause has at most one positive literal. Horn SAT is solvable in linear time [DG84, Scu90], as are a number of generalizations such as *renamable Horn SAT* [Asp80], *extended Horn SAT* [CH91] and *q-Horn SAT* [BHS94, BCH+94].

3. The hierarchy of *tractable* satisfiability problems [DE92], which is based on Horn SAT and 2-SAT, is solvable in polynomial time. An instance on the k-th level of the hierarchy is solvable in $O(nk + 1)$ time.

4. *Nested SAT*, in which there is a linear ordering on the variables and no two clauses overlap with respect to the interval defined by the variables they contain [Knu90].

5. SAT in which no variable appears more than twice. All such problems are satisfiable if they contain no unit clauses [Tov84].

6. *r,r-SAT*, where r,s-SAT is the class of problems in which every clause has exactly r literals and every variable has at most s occurrences. All r,r-SAT problems are satisfiable in polynomial time [Tov84].

7. A formula is *SLUR* (Single Lookahead Unit Resolution) *solvable* if, for all possible sequences of selected variables, algorithm SLUR does not give up. Algorithm SLUR is a nondeterministic algorithm based on unit propagation. It eventually gives up the search if it starts with, or creates, an unsatisfiable formula with no unit clauses. The class of SLUR solvable formulae was developed as a generalization including Horn SAT, renamable Horn SAT, extended Horn SAT, and the class of CC-balanced formulae [SAF+95].

8. *Resolution-Free* SAT Problem, where every resolution results in a tautologous clause, is solvable in linear time [Kus05].

8. *Blocked* SAT Problem, where every clause is blocked, is solvable in polynomial time [Kul99a, Kul99b, Kus07].

In this paper we describes cases when we can simplify a general SAT problem instance by sub-model propagation, which means hyper-unit propagating [Kus02, Kus05] a sub-model [Kus07]. Assume that we test our input clause set whether it is blocked or not, because we know [Kus07] that a blocked clause set can be solved in polynomial time. If the input clause set is not blocked, but some clauses are blocked, then what can we do? Can we use the blocked clauses to simplify the clause set? The Blocked Clear Clause Rule and the Independent Blocked Clause Rule describe cases when the answer is yes.

The other two independent clause rules, the Independent Nondecisive- and Independent Strongly Nondecisive Clause Rules describe cases when we can use nondecisive and strongly nondecisive clauses to simplify a general SAT problem instance.

The notion of blocked [Kul99a, Kul99b] and nondecisive clause [Gel98] was introduced by O. Kullmann and A. V. Gelder. They showed that a blocked or nondecisive clause can be added or deleted from a clause set without changing its satisfiability.

Intuitively a blocked clause has a liter on which every resolution in the clause set is tautology. A nondecisive clause has a literal on which every resolution in the clause set is either tautology or subsumed. We also use the notion of strongly nondecisive clause, which has a liter on which every resolution in the clause set is either tautology or entailed. We also use very frequently the notion of clear clause. A clause is clear if every variable which occurs in the clause set occurs also in this clause either positively or negatively. Note, that clear clauses are called also total or full clauses in the literature.

The Blocked Clear Clause Rule describes two cases. The two cases have a common property: the input clause set contains a blocked clear clause. In the first case the input clause set is a subset of CC, in the second case the blocked clear clause is not subsumed. In both cases the sub-model generated from the blocked clear clause and from one of its blocked literals is a model for the input clause set.

In both cases we need in the worst-case $O(n^2m^3)$ time to decide whether the input clause set fulfills the requirements of the Blocked Clear Clause Rule. We need $O(n^2m^3)$ time, because we have to check blocked-ness in both two cases, which is an $O(n^2m^2)$ time method, and not subsumed-ness in the second case, which is an $O(m)$ time method.

The Independent Blocked Clause Rule is a generalization of the Blocked Clear Clause Rule. We can apply it if we have a blocked clause and it subsumes a clear clause that is it not subsumed by any other clause from the clause set, i.e., the blocked clause is independent. In this case the sub-model generated from the independent blocked clause and from one of its blocked literals is a partial model, i.e., we can simplify the input clause set by propagating this sub-model.

Note that if we know the subsumed clear clause which is not subsumed by any other clause from the input clause set then we know the whole model. This applies for the other independent clause rules.

We need in the worst-case $O(2^n n^2m^3)$ time to decide whether the input clause set fulfills the requirements of the Independent Blocked Clause Rule. We need $O(2^n n^2m^3)$ time, because we have to check blocked-ness, which is an $O(n^2m^2)$ time method, and independent-ness, which is an $O(2^n m)$ time method.

The Independent Nondecisive Clause Rule is a generalization of the Independent Blocked Clause Rule. We can apply it if we have a independent nondecisive clause. In this case the sub-model generated from it and from one of its nondecisive literals is a partial model, i.e., we can simplify the input clause set by propagating this sub-model.

We need in the worst-case $O(2^n nm^4)$ time to decide whether the input clause set fulfills the requirements of the Independent Nondecisive Clause Rule. We need $O(2^n nm^4)$ time, because we have to check nondecisive-ness, which is an $Max\{O(n^2m^2), O(nm^3)\}$ time method, and independent-ness, which is an $O(2^n m)$

time method. We assume that $nm^3 > n^2m^2$.

The Independent Strongly Nondecisive Clause Rule is a generalization of the Independent Nondecisive Clause Rule. We can apply it if we have a independent strongly nondecisive clause. In this case the sub-model generated from it and from one of its strongly nondecisive literals is a partial model, i.e., we can simplify the input clause set by propagating this sub-model.

We need in the worst-case $O(2^{n+1}m)$ time to decide whether the input clause set fulfills the requirements of the Independent Strongly Nondecisive Clause Rule. We need $O(2^{n+1}m)$ time, because we have to check strongly nondecisive-ness, which is an $O(n^2)$ time method, and independent-ness, which is an $O(2^n m)$ time method.

Since the independent clause test is too expensive (it is exponential) we introduce some heuristics which can guess which clause might be independent. Furthermore, we introduce an algorithm which might find strongly nondecisive clauses in $O(n^3m^2)$ time.

2. Definitions

Set of Variables, Literals

Let V be a finite *set of Boolean variables*. The *negation of a variable* v is denoted by $\overline{v}$. Given a set U, we denote $\overline{U} := \{\overline{u} \mid u \in U\}$ and call the *negation of the set* U.

Literals are the members of the set $W := V \cup \overline{V}$. *Positive literals* are the members of the set V. *Negative literals* are their negations. If w denotes a negative literal $\overline{v}$, then $\overline{w}$ denotes the positive literal v.

Clause, Clause Set, Assignment, Assignment Set

Clauses and *assignments* are finite sets of literals that do not contain simultaneously any literal together with its negation.

A clause is interpreted as disjunction of its literals. An assignment is interpreted as conjunction of its literals. Informally speaking, if an assignment A contains a literal v, it means that v has the value $True \in A$. A *clause set* or *formula* (formula in CNF form) is a finite set of clauses. A clause set is interpreted as conjunction of its clauses. If C is a clause, then $\overline{C}$ is an assignment. If A is an assignment, then $\overline{A}$ is a clause. The empty clause is interpreted as False. The empty assignment is interpreted as True. The empty clause set is interpreted as True.

The empty set is denoted by $\emptyset$. The *length* of a set U is its cardinality, denoted by $|U|$. The natural number n is the *number of variables*, i.e., $n := |V|$.

Cardinality, k-Clause, Clear Clause, CC

If C is a clause and $|C| = k$, then we say that C is a *k-clause*. Special cases are *unit clauses* or *units* which are 1-clauses, and *clear* or *total clauses* which are

n-clauses. Note that any unit clause is at the same time a clause and an assignment.

In this paper we prefer the name clear clause instead of total or full clause. Although, total clause is used in the literature, in our point of view the name clear clause is more intuitive.

The clause set CC is the set of all clear clauses.

Subsumption, Entailed-ness, Independent-ness

The clause C *subsumes* the clause B iff C is a subset of B. The interpretation of the notion of subsumption is logical consequence, i.e., B is a logical consequence of C.

We say that a clause C is *subsumed by the clause set* S, denoted by $C \supseteq\in S$, iff there is a clause in S which subsumes it. We say that a clause C is *entailed by the clause set* S, denoted by $C \supseteq\in_{CC} S$, iff for any clear clause, which is subsumed by C, there is a clause in S which subsumes that clear clause.

The interpretation of the notion of subsumed and entailed is the same, logical consequence, i.e., C is a logical consequence of S. Note that if a clause is subsumed by a clause set then it is entailed, but not the other way around. Furthermore, if a clear clause is subsumed by a clause set then it is entailed and the other way around.

$$C \supseteq\in S :\Longleftrightarrow Clause(C) \wedge ClauseSet(S) \wedge \exists[B \in S]B \subseteq C.$$

$$C \supseteq\in_{CC} S :\Longleftrightarrow Clause(C) \wedge ClauseSet(S) \wedge \forall[D \in CC][C \subseteq D]\exists[B \in S]B \subseteq D.$$

We shall explain the intuition behind the notation $\supseteq\in$. If we rewrite its definition and leave out the "not interesting" parts (written in brackets) then we obtain this notation:

$$\exists[B \in S]B \subseteq C \iff (\exists[B])C \supseteq (B \wedge B) \in S \iff C \supseteq\in S.$$

We say that a clause C is *independent in clause set* S iff it is not entailed by S.

Clause Difference, Resolution

We introduce the notion of *clause difference*. We say that two clauses *differ in* some variables iff these variables occur in both clauses but as different literals. If A and B are clauses then the clause difference of them, denoted by $\mathrm{diff}(A, B)$, is

$$\mathrm{diff}(A, B) := A \cap \overline{B}.$$

If $\mathrm{diff}(A, B) \neq \emptyset$ then we say that A *differs from* B. Note that $\mathrm{diff}(A, B) = \overline{\mathrm{diff}(B, A)}$.

We say that *resolution can be performed* on two clauses iff they differ only in one variable. Note that this is not the usual notion of resolution, because we allow resolution only if it results in a non-tautologous resolvent. For example resolution cannot be performed on $\{v, w\}$ and $\{\overline{v}, \overline{w}\}$ but can be performed on $\{v, w\}$ and

$\{\overline{v}, z\}$. If resolution can be performed on two clauses, say A and B, then the *resolvent*, denoted by $\mathrm{Res}(A, B)$, is their union excluding the variable they differ in:

$$\mathrm{Res}(A, B) := (A \cup B) \setminus (\mathrm{diff}(A, B) \cup \mathrm{diff}(B, A)).$$

Note that if we interpret $\mathrm{Res}(A, B)$ as a logical formula then it is a logical consequence of the clauses A and B.

Pure Literal, Blocked- Literal, Clause, Clause Set

We say that a literal $c \in C$ is *blocked in* the clause C and in the clause set S iff for each clause B inS which contains $\overline{c}$ we have that there is a literal $b \in B$ such that $b \neq \overline{c}$ and $\overline{b} \in C$. A *clause is blocked in a clause set* iff it contains a blocked literal. A *clause set is blocked* iff all clauses are blocked in it. We denote these notions by $Blck(c, C, S)$, $Blck(C, S)$ and $Blck(S)$, respectively.

Note that if literal $c \in C$ is blocked in C, S then for all $B \in S, \overline{c} \in B$ we have that resolution cannot be performed on C and B. This means that this clause is "blocked" against resolution.

We say that a literal is *pure* in a clause set if its negation does not occur in the clause set. Note that pure literals are blocked.

(Weakly / Strongly) Nondecisive- Literal, Clause, Clause Set

We define formally the notion of *weakly nondecisive* literal, clause and clause set. We denote these notions by $WnD(c, C, S)$, $WnD(C, S)$ and $WnD(S)$, respectively.

$$WnD(c, C, S) : \Longleftrightarrow \forall[B \in S][\overline{c} \in B](\exists[b \in B][b \neq \overline{c}]\overline{b} \in C \vee \mathrm{Res}(C, B) \supseteq\in S).$$

$$WnD(C, S) : \Longleftrightarrow \exists[c \in C]WnD(c, C, S).$$

$$WnD(S) : \Longleftrightarrow \forall[C \in S]WnD(C, S).$$

We define formally the notion of*nondecisive* literal, clause and clause set. We denote these notions by $NonD(c, C, S)$, $NonD(C, S)$ and $NonD(S)$, respectively.

$$NonD(c, C, S) : \Longleftrightarrow$$

$$\forall[B \in S][\overline{c} \in B](\exists[b \in B][b \neq \overline{c}]\overline{b} \in C \vee \mathrm{Res}(C, B) \cup \{c\} \supseteq\in S \setminus \{C\}).$$

$$NonD(C, S) : \Longleftrightarrow \exists[c \in C]NonD(c, C, S).$$

$$NonD(S) : \Longleftrightarrow \forall[C \in S]NonD(C, S).$$

We define formally the notion of *strongly nondecisive* literal, clause and clause set. We denote these notions by $SND(c, C, S)$, $SND(C, S)$ and $SND(S)$, respectively.

$$SND(c, C, S) : \Longleftrightarrow$$

$$\forall[B \in S][\overline{c} \in B](\exists[b \in B][b \neq \overline{c}]\overline{b} \in C \vee \mathrm{Res}(C, B) \cup \{c\} \supseteq\in_{CC} S \setminus \{C\}).$$

$$SND(C, S) : \Longleftrightarrow \exists[c \in C]SND(c, C, S).$$

$$SND(S) : \Longleftrightarrow \forall[C \in S]SND(C, S).$$

Resolution-Mate, Sub-Model

If C is a clause and c is a literal in C then the *resolution-mate* of clause C by literal c, denoted by $\mathrm{rm}(C, c)$, is

$$\mathrm{rm}(C, c) := (C \cup \{\overline{c}\}) \setminus \{c\}.$$

Note that resolution can be always performed on C and $\mathrm{rm}(C, c)$, and

$$\mathrm{Res}(C, \mathrm{rm}(C, c)) = C \setminus \{c\}.$$

This means that we obtain a shorter clause.

The *sub-model* generated from the clause C and from the literal c, denoted by $\mathrm{sm}(C, c)$, is

$$\mathrm{sm}(C, c) := \overline{\mathrm{rm}(C, c)}.$$

We say that C and c are the *generator* of $\mathrm{sm}(C, c)$. The name "sub-model" comes from the observation that in a resolution-free clause set an assignment created from one of the shortest clauses in this way is a part of a model [Kus05], i.e., a sub-model.

Note that $\mathrm{rm}(C, c)$ is a clause but $\mathrm{sm}(C, c)$ is an assignment.

The sub-model $\mathrm{sm}(C, c)$ is a special assignment which always satisfies clause C, since it sets literal c to be True.

Model, (Un)Satisfiable

An assignment M is a *model for a clause set* S iff for all $C \in S$ we have $M \cap C \neq \emptyset$.

A clause set is *satisfiable* iff there is a model for it. A clause set is *unsatisfiable* iff it is not satisfiable. A clause set is *trivially satisfiable* iff it is empty and it is *trivially unsatisfiable* if it contains the empty clause.

3. The Blocked Clear Clause Rule

In this section we introduce the Blocked Clear Clause Rule, a generalization of the Clear Clause Rule. This rule is introduced by the author.

Assume we test our input clause set whether it is blocked or not, because we know [Kus07] that a blocked clause set can be solved in polynomial time. If the input clause set is not blocked, but some clauses are blocked, then what can we do? Can we use the blocked clauses to simplify the clause set? If it contains a not subsumed blocked clear clause, we can. This is what the Blocked Clear Clause Rule states.

It has two variants. The first one states that if a clause set contains only clear clauses and one of them is blocked then the sub-model generated from this blocked clause and from one of its blocked literal is a model. This is a very rare case, but since we can construct for each clause set the equivalent clear clause set, this rule plays an important role.

The second one states that if a clause set contains a not subsumed blocked clear clause then the sub-model generated from it and from one of its blocked literals is a model. This case is still a very rare one, but might occur more frequently as the first variant.

Lemma 3.1 (Blocked Clear Clause Rule). *Let S be a clause set. Let $C \in S$ be a blocked and clear clause. Let $a \in C$ be a blocked literal C, S.*

(a) If S is a subset of CC, then $sm(C, a)$ is a model for S.

(b) If C is not subsumed by $S \setminus \{C\}$, then $sm(C, a)$ is a model for S.

Proof:

(a) To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap \mathrm{sm}(C, a)$ is not empty. Since S is a subset of CC we know that B is a clear clause. Hence, there are two cases, either $a \in B$ or $\overline{a} \in B$.

In case $a \in B$ we have, by definition of sub-model, that $a \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

In case $\overline{a} \in B$, since $a \in C$ is blocked in C, S we know, by definition of blocked literal, that for some $b \in B$ we have $b \neq \overline{a}$ and $\overline{b} \in C$. From this, by definition of sub-model, we know that $b \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

Hence, if S is a subset of CC, then $\mathrm{sm}(C, a)$ is a model for S.

(b) To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap \mathrm{sm}(C, a)$ is not empty. Since C is not subsumed by $S \setminus \{C\}$ we know, by definition of subsumption, that $B \not\subseteq C$. From this, since C is a clear clause we know that for some $b \in B$ we have $\overline{b} \in C$. There are two cases, either $\overline{b} = a$ or $\overline{b} \neq a$.

In the first case we have $\overline{b} = a$, i.e., $\overline{a} \in B$. From this since $a \in C$ is blocked in C, S we know, by definition of blocked literal, that for some $d \in B$ we have that $d \neq \overline{a}$ and $\overline{d} \in C$. From this, by definition of sub-model, we know that $d \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

In the second case we have $\overline{b} \neq a$. From this and from $\overline{b} \in C$ we know, by definition of sub-model, that $b \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

Hence, If C is not subsumed by $S \setminus \{C\}$, then $\mathrm{sm}(C, a)$ is a model for S.

An alternative proof idea is that we say that it suffices to show that the resolution-mate of C ($\mathrm{rm}(C, a)$) is not subsumed by S. Then we know, by Clear Clause Rule, that its negation ($\mathrm{sm}(C, a)$) is a model.

This alternative proof idea shows in which sense say we that the Blocked Clear Clause Rule is a generalization of the Clear Clause Rule.

This rule is the base of the independent clause rules. Therefore, it is very important for us.

4. The Independent Blocked Clause Rule

In this section we introduce the Independent Blocked Clause Rule, a generalization of the Blocked Clear Clause Rule. This rule is introduced by the author.

The Independent Blocked Clause Rule states that if a clause set contains an independent blocked clause, then it is satisfiable and a sub-model generated from this clause and from one of its blocked literals is a partial model, i.e., we can simplify the clause set by propagating this sub-model. These requirements are fulfilled quite often by real or benchmark problems, but checking independent-ness is expensive.

We know that a clause $A \in S$ is independent in the clause set $S \setminus \{A\}$ if it is not entailed by $S \setminus \{A\}$. The formal definition is the following:

$$A \textit{ independent in } S : \Longleftrightarrow \exists[C \in CC][A \subseteq C]\forall[B \in S][B \neq A]B \nsubseteq C.$$

The following algorithm checks whether the input clause is independent or not in the input clause. If it is independent, then it returns a clear clause subsumed by the input clause but not subsumed by any other clause from the input clause set. Otherwise, it returns the empty clause. In the worst-case it uses $O(2^n m)$ time, because it follows the definition of independent, and there we have two quantifiers, one on CC which has 2^n elements, the other on the input clause set, which has m elements.

Independent Clause Test.

```
1  function IsIndependent(S : clause set, A : clause) : clause
2  begin
3    for each C ∈ CC, A ⊆ C do
4      B_notsubsumes_C := True;
5      for each B ∈ S, B ≠ A while B_notsubsumes_C is True do
6        if (B ⊆ C) then B_notsubsumes_C := False;
7      od
8      if (B_notsubsumes_C) then return C;
9        // In this case we found a suitable C, we return it.
10   od
11   return ∅;
12     // In this case we found no suitable clause.
13     // Therefore, we return the empty clause.
14 end
```

One can see that the independent clause test is very expensive (exponential). We will discuss later how can we get around this problem by suitable heuristics.

Lemma 4.2 (Independent Blocked Clause Rule). *Let S be a clause set. Let $A \in S$ be blocked in S and independent in $S \setminus \{A\}$. Let $a \in A$ be a blocked literal in A, S. Then there is a model M for S such that $sm(A, a) \subseteq M$.*

> Proof: We know that A is independent in $S \setminus \{A\}$. Hence, by definition of independent, we know that there is a clear clause C that is subsumed by A and not subsumed by any other clause in S. Since $A \subseteq C$ we know that $\mathrm{sm}(A, a) \subseteq \mathrm{sm}(C, a)$. Hence, it suffices to show that $\mathrm{sm}(C, a)$ is a model for S. To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap \mathrm{sm}(C, a)$ is not empty. The remaining part of the proof is the same as the proof of the (b) variant of the Blocked Clear Clause Rule.
>
> Hence, $B \cap \mathrm{sm}(C, a)$ is not empty. Hence, there is a model M for S such that $\mathrm{sm}(A, a) \subseteq M$.

This proof is traced back to the proof of Blocked Clear Clause Rule. We can do this because we know that there is a clear clause which is blocked and not entailed by $S \setminus \{A\}$. We know that for clear clauses the notion of subsumed and entailed are the same.

The proof of this lemma shows that if we perform an independent clause check and we find a clear clause which is subsumed by only one clause, then we know the whole model ($\mathrm{sm}(C, a)$) and not only a part of the model ($\mathrm{sm}(A, a)$). But usually we do not want to perform expensive independent-ness checks. How can we get around this problem? The solution is a heuristic which tells us which blocked clause could be independent.

Such a heuristic could be for instance the selection of the shortest blocked clause. The shortest clause subsumes the largest number of clear clauses. Therefore, it has a good chance to be independent, but there is no guarantee for it. We give more details about heuristics after the discussion of the simplifying rules.

5. The Independent Nondecisive Clause Rule

In this section we introduce the Independent Nondecisive Clause Rule, a generalization of the Independent Blocked Clause Rule. This rule is introduced by the author.

The Independent Nondecisive Clause Rule states that if a clause set contains an independent nondecisive clause, then it is satisfiable and a sub-model generated from this clause and from one of its nondecisive literals is a partial model, i.e., we can simplify the clause set by propagating this sub-model. These requirements are fulfilled quite often by real or benchmark problems, but checking independent-ness is expensive.

Lemma 5.1 (Independent Nondecisive Clause Rule). *Let S be a clause set. Let $A \in S$ be nondecisive in S and independent in $S \setminus \{A\}$. Let $a \in A$ be a nondecisive literal in A, S. Then there is a model M for S such that $sm(A, a) \subseteq M$.*

Proof: We know that A is independent in $S \setminus \{A\}$. Hence, by definition of independent, we know that there is a clear clause C that is subsumed by A and not subsumed by any other clause in S. Since $A \subseteq C$ we know that $\mathrm{sm}(A, a) \subseteq \mathrm{sm}(C, a)$. Hence, it suffices to show that $\mathrm{sm}(C, a)$ is a model for S. To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap \mathrm{sm}(C, a)$ is not empty. There are three cases: either (a) $a \in B$ or (b) $\overline{a} \in B$ or (c) $a \notin B$ and $\overline{a} \notin B$.

In case (a) we have $a \in B$. From this and from the definition of sub-model we know that $a \in B \cap \mathrm{sm}(C, a)$.

In case (b) we have $\overline{a} \in B$. From this and from $a \in A$ is nondecisive in A, S, by definition of nondecisive literal, we know that either there is a literal $b \in B$ which has $b \neq \overline{a}$ and $\overline{b} \in A$ or there is a clause $D \in S, D \neq A$ which has $D \subseteq A \cup B\{\overline{a}\}$.

In the first case we know, by definition of sub-model, that $b \in \mathrm{sm}(A, a)$.

In the second case since C is independent in $S \setminus \{A\}$, by definition of independent, we know that D does not subsume C, i.e., for some $d \in D$ we have $d \notin C$. From this and from $A \subseteq C$ and from $D \subseteq A \cup B\{\overline{a}\}$ we can show that $d \notin A$, $d \in B$ and $d \neq \overline{a}$. From $d \notin C$ we know, by definition of clear clause, that $\overline{d} \in C$. Hence, by definition of sub-model, $d \in B \cap \mathrm{sm}(C, a)$.

In case (c) we have $a \notin B$ and $\overline{a} \notin B$. Since C is not subsumed by $S \setminus \{A\}$ we know, by definition of subsumption, that $B \not\subseteq C$. From this, since C is a clear clause we know that for some $b \in B$ we have $\overline{b} \in C$. There are two cases, either $\overline{b} = a$ or $\overline{b} \neq a$.

In the first case we have $\overline{b} = a$, i.e., $\overline{a} \in B$. But we already know that $\overline{a} \notin B$. Hence, this is a contradiction.

In the second case we have $\overline{b} \neq a$. From this and from $\overline{b} \in C$ we know, by definition of sub-model, that $b \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

Hence, there is a model M for S such that $\mathrm{sm}(A, a) \subseteq M$.

This lemma is more powerful then the Independent Blocked Clause Rule, because each blocked clause is nondecisive but not the other way around.

6. The Independent Strongly Nondecisive Clause Rule

In this section we introduce the Independent Strongly Nondecisive Clause Rule, a generalization of the Independent Nondecisive Clause Rule. This rule is introduced by the author.

The Independent Strongly Nondecisive Clause Rule states that if a clause set contains an independent strongly nondecisive clause, then it is satisfiable and a sub-model generated from this clause and from one of its strongly nondecisive literals is a partial model, i.e., we can simplify the clause set by propagating this sub-model. These requirements are fulfilled very often by 3-SAT benchmark problems, but checking independent-ness and strongly nondecisive-ness is expensive.

We will see from our test result that the Independent Blocked Clause Rule can be applied only on few 3-SAT instances. The Independent Nondecisive Rule is better, but still can be applied only on every tenth benchmark problem. Therefore, we tried to find an even more powerful simplification rule. Finally, we found the Independent Strongly Nondecisive Clause Rule.

The idea is the following: We know that a nondecisive clause is either blocked or a special construction $(\mathrm{Res}(A, B) \cup \{a\})$ is subsumed. This rings a bell. If we would use the notion of entailed instead of subsumed then the rule would be more powerful. Let us check whether this idea works or not.

Lemma 6.1 (Independent Strongly Nondecisive Clause Rule). *Let S be a clause set. Let $A \in S$ be strongly nondecisive in S and independent in $S \setminus \{A\}$. Let $a \in A$ be a strongly nondecisive literal in A, S. Then there is a model M for S such that $sm(A, a) \subseteq M$.*

> Proof: We know that A is independent in $S \setminus \{A\}$. Hence, by definition of independent, we know that there is a clear clause C that is subsumed by A and not subsumed by any other clause in S. Since $A \subseteq C$ we know that $\mathrm{sm}(A, a) \subseteq \mathrm{sm}(C, a)$. Hence, it suffices to show that $\mathrm{sm}(C, a)$ is a model for S. To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap \mathrm{sm}(C, a)$ is not empty. There are three cases, either (a) $a \in B$ or (b) $\overline{a} \in B$ or (c) $a \notin B$ and $\overline{a} \notin B$.
>
> In case (a) we have $a \in B$. From this and from the definition of sub-model we know that $a \in B \cap \mathrm{sm}(C, a)$.
>
> In case (b) we have $\overline{a} \in B$. From this and from $a \in A$ is nondecisive in A, S, by definition of nondecisive literal, we know that either there is a literal $b \in B$ which has $b \neq \overline{a}$ and $\overline{b} \in A$ or $\mathrm{Res}(A, B) \cup \{a\})$ is entailed in $S \setminus \{A\}$.
>
> In the first case we know, by definition of sub-model, that $b \in \mathrm{sm}(A, a)$.
>
> In the second case we know that $\mathrm{Res}(A, B) \cup \{a\})$ is entailed in $S \setminus \{A\}$.

From this we know, by definition of entailed, that

$$\forall[D \in CC][A \cup B \setminus \{\overline{a}\} \subseteq D]\exists[E \in S][E \neq A]E \subseteq D.$$

From this we know that there is a literal $b \in B, b \neq a$ such that $b \notin C$ because otherwise we would have that $A \cup B \setminus \{\overline{a}\} \subseteq C$, which would mean that C is subsumed in $S \setminus \{A\}$, which would be a contradiction. From $b \notin C$ we know, by definition of clear clause, that $\overline{b} \in C$. From $b \neq a$ we know, by definition of sub-model, that $b \in \mathrm{sm}(C, a)$. Hence, $b \in B \cap \mathrm{sm}(C, a)$.

In case (c) we have $a \notin B$ and $\overline{a} \notin B$. Since C is not subsumed by $S \setminus \{A\}$ we know, by definition of subsumption, that $B \not\subseteq C$. From this, since C is a clear clause we know that for some $b \in B$ we have $\overline{b} \in C$. There are two cases, either $\overline{b} = a$ or $\overline{b} \neq a$.

In the first case we have $\overline{b} = a$, i.e., $\overline{a} \in B$. But we already know that $\overline{a} \notin B$. Hence, this is a contradiction.

In the second case we have $\overline{b} \neq a$. From this and from $\overline{b} \in C$ we know, by definition of sub-model, that $b \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

Hence, there is a model M for S such that $\mathrm{sm}(A, a) \subseteq M$.

Note that $\mathrm{Res}(A, B) \cup \{a\} = A \cup B \setminus \{\overline{a}\}$.

We see that this proof is almost the same as the proof of the Independent Nondecisive Clause Rule except for the second part of case (b). Here we use the following idea: C is subsumed by A but not by $A \cup B \setminus \{\overline{a}\}$, hence there is a literal $b \in B$ which has $b \neq a$ and $b \notin C$.

So the Independent Strongly Nondecisive Clause Rule works. But to decide whether we can apply it or not we have to perform an entailed-ness check, which is an exponential time method.

What can we do? There are some special cases when it is easy to check entailed-ness. For example the clause E is entailed in the clause set S if we have $E \in S$ or there is a clause $B \in S$ which simply subsumes E. This cases are very rare. The case we are going to describe occurs very often in 3-SAT problem instances.

Assume that we want to check whether the clause E is entailed in the clause set S. Assume we found a clause $D \in S$ which has the following two properties: (a) $\mathrm{diff}(E, D) = \emptyset$ and (b) $D \setminus E$ is a singleton. The first property is needed otherwise D could not subsume any clear clause subsumed by E. The second property says that D subsumes the "half" of E.

Assume that $D \setminus E = \{d\}$. Then D subsumes all clear clauses which are the superset of $E \cup \{d\}$. If E subsumes $2k$ clear clauses and $d \notin E$ then $E \cup \{d\}$ subsumes k clear clauses and $E \cup \{\overline{d}\}$ subsumes the remaining k clear clauses. Hence, we can say that D subsumes the "half" of E. So we can reduce the problem to whether $E \cup \{\overline{d}\}$, the remaining "half", is entailed in S or not. We call this step to cut E in half.

This situation occurs very often in 3-SAT problem instances, because our $E = A \cup B \setminus \{\overline{a}\}$ has a length of 5, clauses in the input clause set have a length of 3, and usually we have $n >> 5$, where n is the number of variables. This means that it is very likely that we can use this step at least once.

The following algorithm uses this step to find strongly nondecisive clauses. In the worst-case it is a $O(n^3m^2)$ time method, but there is no guarantee that it finds any strongly nondecisive clauses.

GetSNDClauses.

```
1  function GetSNDClauses(S : clauseset) : array of ⟨clause, literal⟩
2  begin
3    i := 0;
4    //  We need i to index the array SND.
5    for each A ∈ S do
6      a_is_snd := False;
7      for each a ∈ A while a_is_snd is False do
8        B_snds_a := True;
9        for each B ∈ S, ā ∈ B while B_snds_a is True do
10         b_blocks_a := False;
11         D_subsumes_E := False;
12         B := B \ {ā};
13         if (diff(B, A) ≠ ∅) then b_blocks_a := True;
14         else
15           E := A ∪ B;
16           for each D ∈ S, D ≠ A while D_subsumes_E is False do
17             if (D ⊆ E) then D_subsumes_E := True;
18             if (diff(D, E) = ∅ ∧ |D \ E| = 1) then
19               E := E ∪ (D \ E);
20               Restart the last loop ;
21               //  We have to restart the loop on clauses D,
22               //  because the remaining half could be subsumed
23               //  by a clause, which was already considered.
24             fi
25           od
26         fi
27         if (¬b_blocks_a ∧ ¬D_subsumes_E) then B_snds_a := False;
28       od
```

```
29          if (B_snds_a) then a_is_snd := True;
30        od
31        if (a_is_nond) then (SND[i], i) := (⟨A, a⟩, i + 1);
32      od
33      return SND;
34  end
```

The new rows are the ones from 14 till 26. We use in the 20th row a very interesting solution, we restart the innermost loop. We discuss this issue a bit later.

One can see that this algorithm returns an array of ordered pairs. An ordered pair contains a strongly nondecisive clause C and a strongly nondecisive literal $c \in C$.

Note that this algorithm might not find all strongly nondecisive clauses, because it does not use entailed-ness check, but the "cut E in half" step, described above.

This algorithm is an $O(n^3m^2)$ time method in the worst-case, where n is the number of variables and m is the number of clauses of the input clause set. It is an $O(n^3m^2)$ time method, because we have two loops on clauses and two loops on literals, but the innermost loop might be restarted n times in the worst-case.

One might ask, why do we need to restart the innermost loop? Assume we have the situation that we can cut E in half, i.e., we have found a clause $D \in S, D \neq A$ which has $\text{diff}(D, E) = \emptyset$ and $D \setminus E$ is a singleton. Then there is no D' clause among the ones we already considered such that D' subsumes $E \cup (D \setminus E)$, because D' fulfills the same requirements as D, i.e., it would be already used to cut E in half. Then why should we restart?

That is true, but there might be clauses among the ones we already considered which can cut the new E in half and in the rest of the clause set there is no suitable clause which subsumes E or can cut it in half. Therefore, we have to restart the innermost loop.

7. Heuristics

In this subsection we introduce three heuristics. All of them are suitable more or less to guess whether a clause is independent or not.

All three heuristics are based on the following idea. A clause A is independent in the clause set $S \setminus \{A\}$ if $\overline{A}$ is a subset of a model of S, i.e., after propagating $\overline{A}$ on S, let us call the resulting clause set S', S' is satisfiable. Of course we do not want to perform expensive satisfiability checks, but we want to guess whether it is satisfiable or not. The idea is the following: the less clauses are contained in S', the more likely is that it is satisfiable.

This means that we have to count the clauses in S'. But propagation of an assignment is still to expensive for us. Therefore, we count the clauses in the

following set:

$$\{B \mid B \in S \wedge \text{diff}(A, B) = \emptyset\}.$$

Note that if a clause C is in this set then the clause $C' = C \setminus A$ is element of S'.

In the first version, called $IBCR$-1111, we just count each blocked clause A the clauses B that have $\text{diff}(A, B) = \emptyset$ and we choose the one for which this number is the smallest.

Our test results on 3-SAT problem instances shows that this heuristic provides an independent blocked clause in 68% of the cases if there is an independent blocked clause.

In the other two versions we use weights.

In the second version, called $IBCR$-1234, we count each blocked clause A the clauses B which has $\text{diff}(A, B) = \emptyset$ and we choose the one for which this number is the smallest. But we count clauses B with different weights. The weight W_B is

$$W_B := 1 + |A \cap B|.$$

For example if A is a 3-clause and $|A \cap B| = 2$ then $W_B = 3$.

Our test results on 3-SAT problem instances shows that this heuristic provides an independent blocked clause in 71% of the cases if there is an independent blocked clause.

In the third version, called $IBCR$-1248 the weight W_B is

$$W_B := 2^{|A \cap B|}.$$

For example if A is a 3-clause and $|A \cap B| = 2$ then $W_B = 4$.

Our test results on 3-SAT problem instances shows that this heuristic provides an independent blocked clause in 73% of the cases if there is an independent blocked clause.

After this short overview we give more details. First we have to explain the names of the three heuristics: $IBCR$-1111, $IBCR$-1234, and $IBCR$-1248. The word "IBCR" is just the abbreviation of Independent Blocked Clause Rule.

We have tested these heuristics on 3-SAT problem instances, where $|A \cap B|$ can be 0, 1, 2, or 3. The remaining part of the names comes from the values of weights. In the first heuristic the weight is the constant 1. Therefore, its name is $IBCR$-1111. In the second one the weight is defined by $1+|A \cap B|$, i.e., the weights are 1, 2, 3, or 4, respectively. Therefore, its name is $IBCR$-1234. In the third one the weights are 1, 2, 4, 8, respectively. Therefore, its name is $IBCR$-1248.

We present the pseudo-code of the third variant. This algorithm is an $O(n^2m^2)$ time method in the worst-case, where n is the number of variables and m is the number of clauses in the input clause set. It is an $O(n^2m^2)$ time method, because we have two loops on clauses and other two on literals.

IBCR-1248.

```
1 function IBCR-1248(S : clause set) : ⟨clause, literal⟩
```

```
2  begin
3    min_Counter := Infinite;
4      // The variable min_Counter stores the minimum value of Counter.
5      // First time should be big enough.
6    for each A ∈ S do
7      a_is_blocked := False;
8      for each a ∈ A while a_is_blocked is False do
9          // Here begins the code which is relevant for the heuristic
10       Counter := 0;
11       B_blockes_a := True;
12       for each B ∈ S while B_blocks_a is True do
13         if (diff(A, B) = ∅) then Counter := Counter + 1 * (2^|A∩B|);
14           // The weight is 2^|A∩B|.
15         if (ā ∉ B) then continue ;
16           // Remember, we have to visit all B ∈ S which has ā ∈ B
17           // to decide whether a ∈ A is blocked or not.
18         b_blocks_a := False;
19         for each b ∈ B, b ≠ ā while b_blocks_a is False do
20           if (b̄ ∈ A) then b_blocks_a := True;
21         od
22         if (¬b_blocks_a) then B_blocks_a := False;
23       od
24       if (B_blocks_a) then a_is_blocked := True;
25       if (a_is_blocked and (Counter < min_Counter)) then
26         (min_Counter, min_A, min_a) := (Counter, A, a);
27       fi
28     od
29   od
30   return ⟨min_A, min_a⟩;
31 end
```

From this algorithm one can easily construct the other two or even other heuristics.

We can see that this heuristic returns a clause, say C, and a literal, say c. The clause C is a blocked clause and the literal c is a blocked literal in it. The heuristic state that C is independent. But this might be false.

If it is true, then it is fine because we can simplify our input clause set by a sub-model propagation using $\mathrm{sm}(C, c)$.

If it is false, then we still can gain something. We can add a shorter clause than C, because, by the Lucky Failing Property of Sub-Models, we know that $C \setminus \{c\}$ is entailed by the input clause set.

We do not know which case will be applied but we hope that the first one occurs more frequently.

These heuristics do not use the fact that the clause is blocked or not. Therefore, we can generalize them very easily for guessing independent-ness of (strongly) nondecisive clauses.

In the names of these heuristics we use the following acronyms: INCR for Independent Nondecisive Clause Rule; ISNCR for Independent Strongly Nondecisive Clause Rule.

8. Test Results

In this section we describe shortly our java implementation of the simplification rules and we present the test results we have got on problems from the SATLIB problem library.

Our java implementation has three classes, Clause, ClauseSet and Satisfiable. The class Clause contains two BitSet objects, *positive* and *negative*. If we represent a clause where the first variable occurs positively then the first bit of the BitSet *positive* is set (1) and the first bit of BitSet *negative* is clear (0). This means that our implementation is close to the Literal Matrix View.

This implementation is not competitive with the newest SAT solvers because it does not use enhanced data structures or techniques like back jumping but it is good enough to test whether the simplification rules can be applied on benchmark problems or not.

We have tested the heuristics on Uniform Random-3-SAT problems [CKT91] from the SATLIB – Benchmark Problems homepage:
http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html.

We used the smallest problem set, uf20-91.tar.gz, which contains 1000 problems, each has 91 clauses and 20 variables and is satisfiable.

We used a Pentium 4, 2400 MHz PC machine with 1024 MB memory to perform the tests.

Here we present our test results for the problems of uf20-91.tar.gz as a table ($IBCR$: Independent Blocked Clause Rule, $INCR$: Independent Nondecisive Clause Rule, $ISNCR$: Independent Strongly Nondecisive Clause Rule):

	$IBCR$	$INCR$	$ISNCR$	from
SND clauses:	601	1128	61122	91000
Problems with SND:	256	465	1000	1000
Independent SND:	77	125	4011	91000
Prob.s with indep. SND:	60	102	951	1000
X-1111:	41 / 60	61 / 102	89 / 951	
X-1234:	43 / 60	72 / 102	142 / 951	
X-1248:	44 / 60	76 / 102	166 / 951	

By "SND clauses" we mean in the column of Independent Blocked Clause Rule blocked clauses, in the next column nondecisive clauses, and in the next column strongly nondecisive clauses. The column "from" shows how many clauses and clause sets, respectively, do we have in total.

The line X-1111: 41 / 60 61 / 102 89 / 951 means that: $IBCR$-1111 successfully guesses 41 times an independent blocked clause from the 60 cases where we checked whether we have independent blocked clauses; $INCR$-1111 is successful 61 times from 102; and $ISNCR$-1111 is successful 89 times from 951.

Now we give the same table but the results are given in percentages.

	$IBCR$	$INCR$	$ISNCR$	from
SND clauses:	0.66%	1.23%	67.16%	91000
Problems with SND:	25.6%	46.5%	100%	1000
Independent SND:	0.08%	0.13%	4.4%	91000
Prob.s with indep. SND:	6%	10.2%	95.1%	1000
X-1111:	68.334%	59.8%	9.35%	
X-1234:	71.667%	70.58%	14.93%	
X-1248:	73.334%	74.5%	17.45%	

We can see that the X-1248 is the best heuristic, but still it could guess an independent strongly nondecisive clause only in 17% of the cases where we know that there are some.

It is so because it is very hard to guess independent clauses. We have better results in the other two cases because there are a lot of instances where we have only one or two independent blocked or nondecisive clauses. One can see that only the 0.66% of clauses are blocked while 67% are strongly nondecisive.

We believe that these simplifications are very useful, because if it turns out that the selected blocked clause is not independent, after propagating a sub-model generated from it, then we can still, by the Lucky Failing Property of Sub-Models, add a shorter clause to our clause set.

Acknowledgements. Partially supported by TéT 2006/A-16.

References

[APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. Information Processing Letters, 8(3):121–132, 1979.

[Asp80] B. Aspvall. Recognizing Disguised NR(1) Instances of the Satisfiability Problem. J. of Algorithms, 1:97–103, 1980.

[BHS94] E. Boros, P. L. Hammer, and X. Sun. Recognition of q-Horn Formulae in Linear Time. Discrete Applied Mathematics, 55:1–13, 1994.

[BCH+94] E. Boros, Y. Crama, P. L. Hammer, and M. Saks. A Complexity Index for Satisfiability Problems. SIAM J. on Computing, 23:45–49, 1994.

[CH91] V. Chandru and J. Hooker. Extended Horn Sets in Propositional Logic. J. of the ACM, 38(1):205–221, 1991.

[CKT91] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the Really Hard Problems Are. Proceedings of the IJCAI-91, 331–337, 1991.

[Coo71] S. A. Cook. The Complexity of Theorem-Proving Procedures. Proceedings of the 3rd ACM Symposium on Theory of Computing, 151–158, 1971.

[DE92] M. Dalal and D. W. Etherington. A Hierarchy of Tractable Satisfiability Problems. Information Processing Letters, 44:173–180, 1992.

[DG84] W. F. Dowling and J. H. Gallier. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. J. of Logic Programming, 1(3):267–284, 1984.

[EIS76] S. Even, A. Itai, and A. Shamir. On the Complexity of Timetable and Multi-Commodity Flow Problems. SIAM J. on Computing, 5(4):691–703, 1976.

[Gel98] A. V. Gelder. Propositional Search with k-Clause Introduction Can be Polynomially Simulated by Resolution. Proceedings of the 5th International Symposium on Artificial Intelligence and Mathematics, 1998.

[Knu90] D. E. Knuth. Nested Satisfiability. Acta Informatica, 28:1–6, 1990.

[Kul99a] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. Theoretical Computer Science, 223(1-2):1–72, 1999.

[Kul99b] O. Kullmann. On a Generalization of Extended Resolution. Discrete Applied Mathematics, 96-97(1-3):149–176, 1999.

[Kus02] G. Kusper. Solving the SAT Problem by Hyper-Unit Propagation. RISC Technical Report 02-02, 1–18, University Linz, Austria, 2002.

[Kus05] G. Kusper. Solving the Resolution-Free SAT Problem by Hyper-Unit Propagation in Linear Time. Annals of Mathematics and Artificial Intelligence, 43(1-4):129–136, 2005.

[Kus07] G. Kusper. Finding Models for Blocked 3-SAT Problems in Linear Time by Systematical Refinement of a Sub-Model. Lecture Notes in Artificial Intelligence 4314, KI 2006: Advances in Artificial Intelligence, 128-142, 2007.

[SAF+95] J. S. Schlipf, F. Annexstein, J. Franco, and R. P. Swaminathan. On finding solutions for extended Horn formulas. Information Processing Letters, 54:133–137, 1995.

[Scu90] M. G. Scutella. A Note on Dowling and Gallier's Top-Down Algorithm for Propositional Horn Satisfiability. J. of Logic Programming, 8(3):265–273, 1990.

[Tov84] C. A. Tovey. A Simplified NP-complete Satisfiability Problem. Discrete Applied Mathematics, 8:85–89, 1984.

Gábor Kusper
Eszterházy Károly College, Eszterházy tér 1, Eger, H-3300
e-mail:gkusper@aries.ektf.hu

Lajos Csőke
Eszterházy Károly College, Eszterházy tér 1, Eger, H-3300
e-mail:csoke@aries.ektf.hu

Gergely Kovásznai
Eszterházy Károly College, Eszterházy tér 1, Eger, H-3300
e-mail:kovasz@aries.ektf.hu

13 Annex 6. - SAT solving by CSFLOC, the next generation of full-length clause counting algorithms

This is the submitted version of G. Kusper, Cs. Biró, Gy. B. Iszály, *SAT solving by CSFLOC, the next generation of full-length clause counting algorithms*, Proceedings of IEEE International Conference on Future IoT Technologies 2018, DOI: 10.1109/FIOT.2018.8325589, 2018. See also [97].

SAT Solving by CSFLOC, the Next Generation of Full Length Clause Counting Algorithms

Gábor Kusper, Csaba Biró, György Barna Iszály

Eszterházy Károly University, Uinversity of Nyíregyháza,
Email: see http://fmv.ektf.hu

***Abstract*—In this paper we introduce the `CSFLOC` algorithm which counts full length clauses. It is the successor of the `Optimized CCC` algorithm. By studying `Optimized CCC` we observed that its full length clause counter can be increased on its last 1 bit in the best case. As a main contribution we prove the lemma that this observation is generally true for `Optimized CCC`. The new algorithm, `CSFLOC`, uses this lemma. It uses also a data structure in which the clauses are ordered by the index of their last literal. These two improvements result in a faster algorithm which can compete with a state-of-the-art SAT solver on problems with lots of clauses, like black-and-white 2-SAT problems and weakly nondecisive SAT problems.**

***Keywords*-SAT, #SAT, SAT Solver.**

I. Introduction

Propositional satisfiability is the problem of determining, for a formula of the propositional logic, if there is an assignment of truth values to its variables for which that formula evaluates to be true. SAT is one of the most-researched NP-complete [7] problems in several fields of computer science, including theoretical computer science, artificial intelligence, hardware design, and formal verification [3]. By SAT we mean the problem of propositional satisfiability for formulas in conjunctive normal form (CNF). Modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm1 [8].

An interesting question is how many models does a SAT problem instance have. This problem is known as the #SAT problem. In other words, by #SAT we mean the problem of counting the models of a SAT instance [10]. The most popular way to solve this problem is to use a variant of the DPLL method which does not stop if it finds a model but keep exploring the search space. One of the first examples is CDP [6].

Another way to count the models is to use the inclusion-exclusion principle. It is well known that this principle can solve the #SAT problem [11], [15], [2].

Let $A_1, A_2, \ldots, A_m$ be sets, where $m > 0$. The inclusion-exclusion principle states that $|\bigcup_{i=1}^{m} A_i| = \sum_{i=1}^{m} |A_i| - \sum_{1 \le i < j \le m} |A_i \cap A_j| + \ldots (-1)^{m+1} |A_1 \cap A_2 \cap \cdots \cap A_m|$.

We recall that clause C subsumes clause D if and only if C is a subset of D. To be able to use the inclusion-exclusion principle on SAT we need the so called Clear Clause View [13]. In this view we have to see the clauses as sets of the subsumed full length clauses. For example if we have 4 variables, a, b, c, d, then the clause $\{a, b\}$ should be seen as the set of subsumed full length clauses: $\{\{a, b, \overline{c}, \overline{d}\}, \{a, b, \overline{c}, d\}, \{a, b, c, \overline{d}\}, \{a, b, c, d\}\}$. Now, let $C_1, \ldots, C_m$ be the clauses of a SAT problem instance. Let A_i be the set of full length clauses which are subsumed by C_i, $i = 1 \ldots m$. Then $|A_i|$ means the number of full length clauses subsumed by C_i, and $|A_i \cap A_j|$ means the number of full length clauses subsumed by both C_i and C_j. Note, that $|A_i| = 2^{n-k}$, where n is the number of variables in the clause set, and k is the number of literals in C_i. Also note that the number of solution of a clause set is 2^n minus the number of subsumed full length clauses, see [2].

Instead of using the inclusion-exclusion principle one can use our new algorithm, the `C`ounting `S`ubsumed `F`ull `L`ength `O`rdered `C`lauses algorithm, for short `CSFLOC`. It counts the subsumed full length clauses, as the inclusion-exclusion principle, but in an iterative way. It is easy to see that there are 2^n different full length clauses. A SAT problem is unsatisfiable if it subsumes all full length clauses. If a SAT problem subsumes fewer, then it is satisfiable. This means that by counting the subsumed full length clauses we can decide satisfiability.

In a previous paper a method was introduced to translate a directed graph into a SAT problem [5]. The motivation was to check whether the communication graph of a wireless sensor network (WSN) is strongly connected or not. We showed that if the graph is strongly connected, then the generated SAT instance is a black-and-white SAT problem, which is a special weakly nondecisive SAT (WnD) problem [13], [4]. In this paper we introduce a SAT solver algorithm, called `CSFLOC`, which is very good at solving WnD problems generated by the `WnDGen` SAT problem generator.

The `CSFLOC` algorithm is the successor of the `Optimized CCC` algorithm, which counts those full length clauses which are subsumed by the input clause set. To do this, it uses a counter. By studying `Optimized CCC` we have observed that in the best case it increases the counter on its last 1 bit. For example, if the counter is 24, which is 16+8, then the biggest possible increment is 8.

As a main contribution we prove the lemma that this observation is generally true for `Optimized CCC`. The new algorithm, `CSFLOC`, uses this lemma. It uses also a data structure in which the clauses are ordered by the index of their last literal.

These two improvements result in a faster algorithm which

is faster than a state-of-the-art SAT solver on problems with lots of clauses but few variables, like WnD problems. Otherwise it is slower, but it uses no fine-tuned heuristics, and it is built on different ideas then a state-of-the-art SAT solver, so it is still interesting.

Our experience shows that overconstrained problems with less than 100 variables, like WnD problems, are easier for `CSFLOC`. Our experience shows also that if we cluster the variables of a SAT problems then `CSFLOC` runs faster. Since `CSFLOC` is good at solving WnD problems, it might be a useful tool to check whether the communication graph of a WSN is strongly connected or not.

This work is based on a poster presented at SYNASC 2015, and which was documented by an extended abstract, see [14]. That work contains the main idea of `CCC` and `Optimized CCC`. In this work we recall those two algorithms, and present their successor, the `CSFLOC` algorithm.

In the introduction we describe on a high level `CCC`, then `Optimized CCC`, finally we describe the new idea in `CSFLOC`.

`CCC` works as follows: It sets its counter to be 0. It converts 0 to a full length clause, called C, which is the one with only negative literals. In general, C is created from the counter in the following way: Each bit of the counter is represented by a literal, the literal is positive if the corresponding bit is 1, negative otherwise. It checks whether this full length clause is subsumed by the input problem. This means that it looks for a clause D from the input SAT problem that is a subset of C. If such a clause D is found, then it adds 1 to the counter and repeats the process until it finds a full length clause which is not subsumed, which means that the input is satisfiable, or all possible full length clauses are visited, i.e., the input is unsatisfiable.

It is easy to see that this algorithm stops for any SAT problem and can decide whether the SAT instance is satisfiable or not. This means that this simple algorithm is sound and complete, but in case of an unsatisfiable SAT instance it visits all the 2^n full length clauses.

The next version is called `Optimized CCC`, which uses the observation that a clause subsumes 2^{n-i} consecutive ordered full length clauses, where i is the index of its last literal. This means that if we find a clause D which subsumes C, then we can increase the counter by 2^{n-i} instead of 1. So we do not need to visit all full length clause to decide satisfiability. `Optimized CCC` always selects that D in the loop which grants the biggest step, i.e., where the number 2^{n-i} is the greatest. This optimization does not affect the soundness and the completeness of the algorithm.

The next version of this algorithm family is `CSFLOC`. Here we use the observation that if we always use the best D in the loop, as in `Optimized CCC`, then the biggest possible step is 2^{n-j}, where j is the index of the last 1 bit in the counter. Note, that this bit always comes from the previous run of the loop. It is so, because if there were an even better D in the input clause set, then the algorithm would select that D even in the previous loop and setting a bit to 1 which has smaller index than j. This means that we do not have to scan the whole input clause set to find the best D, we have to consider only those clauses where the index of the last literal is greater than or equal to j. This optimization does not affect the soundness and the completeness of the algorithm.

Counting full length clauses is not a new idea. Already Iwama used this idea in 1989 [11]. Later Lozinskii introduced an approximation algorithm for counting models [15]. They count full length clauses following the inclusion-exclusion principle more closely. This means that they also decrement the counter, see Algorithm 3 in [2]. In case of the `CCC` algorithm family we always increment the counter. So we can use the above observation about the biggest possible step, while the classical algorithms cannot do so.

Full length clauses are called also maximal clauses in the literature. A good overview of these papers can be found in [1]. In this paper Andrei introduce an inverse resolution to generate maximal clauses. He also overviews the field of maximal clause counting. We could not access the old papers, like the one written by Tanaka [16] and by Dubois [9]. From Andrei's overview it seems that Tanaka's algorithm is similar to `Optimized CCC`. We could not find a similar one to `CSFLOC`.

The test results show that `CSFLOC` is always faster than `Optimized CCC` and it can compete with a state-of-the-art SAT solver on problems with lots of clauses, like black-and-white 2-SAT problems [5] and weakly nondecisive SAT problems [13], [4].

II. Definitions

A *literal* is a boolean variable, called positive literal, or the negation of a boolean variable, called negative literal. A *clause* is a set of literals. A *clause set* is a set of clauses. An *assignment* is a set of literals. Clauses are interpreted as disjunction of their literals. Assignments are interpreted as conjunction of their literals. If a clause or an assignment contains exactly k literals, then we say it is a *k-clause* or a *k-assignment*, respectively. A *full length clause* is an n-clause, where n is the number of variables. A *full length assignment* is an n-assignment.

We denote negation by overbar, i.e., $\overline{a}$ means the negation of a. Note, that $\overline{\overline{a}} = a$. If H is a set, then $\overline{H}$ means that all elements in H are negated.

We say that clause C *subsumes* clause D if and only if (iff) C is a subset of D. Its interpretation is logical consequence, i.e., D is a logical consequence of C.

We say that clause set S *subsumes* clause C iff there is a clause in S which subsumes C. Formally: $S \; subsumes \; C \iff \exists D(D \in S \wedge D \subseteq C)$.

We say that assignment M is a *model*, or a *solution* for clause set S iff for all $C \in S$ we have $M \cap C \neq \emptyset$. We say that a model of a clause set is a *full length model* if it is a full length assignment.

We say that the injective function I from variables to the natural numbers $1 \ldots n$ is a *variable index function*, where n is the number of variables. We can also define I on literals

such that $I(lit) = I(\overline{lit})$, where lit is a literal. If otherwise is not stated I is the lexicographical ordering of variables.

If we use the same variable index function on the clauses of a clause set, then we call them *ordered clauses*. In an ordered clause we can select the *first literal*, which is the one with the smallest index, and the *last literal*, which is the one with the greatest index, in the ordered clause.

We also define two functions: IndexOfLastLiteral(C) := $\max(\{I(lit)|lit \in C\})$, and IndexOfLastPositiveLiteral(C) := $\max(\{I(lit)|lit \in C \wedge lit\ is\ a\ positive\ literal\} \cup \{1\})$.

We also define how to convert a bit array into a clause: FullLengthClauseRepresentation($bits$) := $\{I^{-1}(i)|i \in \{1 \dots n\} \wedge bits_i = 1\} \cup \{\overline{I^{-1}(i)}|i \in \{1 \dots n\} \wedge bits_i = 0\}$, where $bits_i$ is the i-th bit in the integer $bits$, and I^{-1} is the inverse of I.

III. Theory

The CCC algorithm is based on the following lemmas.

Lemma 1. *[Clear Clause Rule, see Lemma 4.3.1. in [13]] Let S be a clause set and C be a full length clause. Then*

$$(a)\ \ S\ subsumes\ C \Leftrightarrow \neg(\overline{C}\ is\ a\ model\ for\ S);$$

$$(b)\ \ \neg(S\ subsumes\ C) \Leftrightarrow \overline{C}\ is\ a\ model\ for\ S.$$

The proof of this lemma can be found in [13]. This lemma states that if S subsumes a full length clause, say C, then its negation is not a model. But if S does not subsume C, then its negation, $\overline{C}$, is a model for S. Hence, it is enough to find a not subsumed full length clause to find a model for S. We use this lemma in all three variants of our algorithm.

The next lemma is a trivial consequence of the previous one.

Lemma 2. *Let S be a clause set with n variables. Then S has m different full length models iff S subsumes $2^n - m$ different full length clauses.*

This lemma has two important corollaries.

Corollary 1. *(a) Let S be a clause set with n variables. Then S is satisfiable iff S subsumes less than 2^n full length clauses; and S is unsatisfiable iff S subsumes 2^n full length clauses. (b) Let C be a k-clause. Then C subsumes 2^{n-k} full length clauses.*

We use the following representation: In a full length clause all variables are present either as a positive literal, or as a negative one. This means that we can represent them by an n bit wide unsigned integer. If it is an ordered full length clause then we can use the following representation: bit value 0 corresponds to negative literal, 1 to a positive one on the same index. We give an example:

Example 1. *The clause $\{a, \overline{b}, c, d, \overline{e}\}$ is represented by `10110`.*

With this representation we can order full length clauses.

Lemma 3. *Let n be the number of boolean variables in use. Let I be a variable index function of these variables. Let D be a non-empty clause. Let i be the index of the last literal of D, i.e., let i =IndexOfLastLiteral(D). Then D subsumes 2^{n-i} consecutive full length clauses.*

Proof: We know that D does not contain any literal with index greater than i. Let **C** be the set of full length clauses which are subsumed by D. Let C_0 be the element of **C** which has the smallest integer representation among elements of **C**. We show that D subsumes C_0 and the next $2^{n-i}-1$ full length clauses. It is easy to see that the last $n-i$ literals of C_0 are negative ones. The next full length clause is the same as C_0 except that its last literal is positive. Let us denote this clause, i.e., the next one according to its integer representation, by C_1. D subsumes C_1 unless $i = n$, i.e., unless the case that the last variable is present in D. We can create C_2 by adding 1 to the integer representation of C_1 and translate it back to a full length clause. In this way we can construct the consecutive full length clauses. We can see that D subsumes each full length clause in this sequence until the i-th literal is not affected, i.e., it subsumes $C_0, C_1, \dots, C_{2^{n-i}-1}$, but not $C_{2^{n-i}}$. Hence, D subsumes 2^{n-i} consecutive full length clauses. ■

To visualize the above lemma, we give an example.

Example 2. *Let us assume that we have 6 variables, $a, b, \dots, f$. This means that $n = 6$. Let $I(a) = 1, I(b) = 2, \dots, I(f) = 6$. Let $D = \{\overline{b}, d\}$, i.e., $i = 4$. Then D subsumes the following 2^{n-i} consecutive full length clauses (we give in brackets also the bit representation): $C_0 = \{\overline{a}, \overline{b}, \overline{c}, d, \overline{e}, \overline{f}\}$ `(000100)`, $C_1 = \{\overline{a}, \overline{b}, \overline{c}, d, \overline{e}, f\}$ `(000101)`, $C_2 = \{\overline{a}, \overline{b}, \overline{c}, d, e, \overline{f}\}$ `(000110)`, $C_3 = \{\overline{a}, \overline{b}, \overline{c}, d, e, f\}$ `(000111)`, but does not subsume the next one, which is: $C_4 = \{\overline{a}, \overline{b}, c, \overline{d}, \overline{e}, \overline{f}\}$ `(001000)`.*

IV. The CCC Algorithm

We present two version of CCC. The first counts one by one. The second is more optimized.

Algorithm 1 CCC(S)

Require: S is a non-empty clause set.
Ensure: If S is satisfiable it returns a model for S, otherwise returns the empty set.

```
1:  n := number of variables in S;
2:  count := 0;
3:  while count < 2^n do
4:     C :=FullLengthClauseRepresentationOf(count);
5:     if ∃D ∈ S(D subsumes C) then
6:        count := count + 1;
7:     else
8:        return C̄;
9:     end if
10: end while
11: return {};
```

Algorithm 1 works as follow: It sets $count$ to zero, then starts a loop. In the loop, in Line 4, it converts $count$ to a

full length clause. In Line 5 it checks whether this full length clause is subsumed or not. If yes, it increases $count$ by one in Line 6. If not, by point (b) of Lemma 1, a model is found, which is returned in Line 8. Eventually $count$ either grows beyond $2^n - 1$ or a model is found. In the first case we know from point (a) of Corollary 1 that S is unsatisfiable.

Lemma 4. *[Soundness of CCC] Algorithm 1 is sound.*

Proof: There are two cases, the input clause set is either satisfiable (I), or unsatisfiable (II).
(I:) Let us assume that the input clause set S is satisfiable. From point (a) of Corollary 1 we know that S subsumes less than 2^n full length clauses, i.e., there is a full length clause which is not subsumed by S. Since Algorithm 1 iterates over all full length clauses, eventually it encounters a full length clause, called C in Line 5, which is not subsumed by S and returns its negation in Line 8. We know from point (b) of Lemma 1, that $\overline{C}$ is a model S, which means that the input clause set is satisfiable, which is the correct answer.
(II:) Let us assume that the input clause set S is unsatisfiable. From point (a) of Corollary 1 we know that S subsumes all 2^n full length clauses. Therefore, the **if** expression in Line 5 will be always true, so $count$ will be increased by 1 in Line 6. Eventually $count$ will be 2^n and, hence, the empty set will be returned in Line 11, which means that the input clause set is unsatisfiable, which is the correct answer. ■

Lemma 5. *[Completeness of CCC] Algorithm 1 is complete.*

Proof: We already know from Lemma 4 that Algorithm 1 is sound. So it is enough to show that it stops for each valid input. The algorithm has one **while** loop, which terminates either if $count = 2^n$ or if we found a full length clause which is not subsumed by S. In the **while** loop there is one **if** statement. In the **else** branch we have a **return** statement, i.e., the **while** loop terminates in that branch. If it does not terminate in the **else** branch, then in the **then** branch, in Line 6, we increase $count$ by one. This means that if it does not terminate in the **else** branch, then $count$ will be eventually equal to 2^n, i.e., the **while** loop eventually terminates. Hence, Algorithm 1 is complete. ■

Lemma 6. *[Worst case complexity of CCC] The worst case complexity of CCC is $O(2^n \cdot Check \cdot Conversion)$, where Check is the complexity of a subsumption check, and Conversion is the complexity of converting a number to a full length clause.*

Proof: In the worst case the algorithm has to visit all the 2^n full length clauses. ■

V. The Optimized CCC Algorithm

Now we give a more optimized version of CCC. First we present it in a very intuitive way as follows:
Let x be the first assignment in lexicographical order.
Repeat until last assignment is reached:
If x satisfies the input formula, return x and stop.
Otherwise:
Let i be the smallest number such that x_i appears in an unsatisfied clause as its last literal. Let x be the assignment that comes 2^{n-i} places after current x in lexicographical order.

Now we give it more formally, see 2.

Algorithm 2 Optimized CCC(S)

Require: S is a non-empty clause set.
Ensure: If S is satisfiable it returns a model for S, otherwise returns the empty set.

```
1:  n := number of variables in S;
2:  count := 0;
3:  while count < 2^n do
4:    C :=FullLengthClauseRepresentationOf(count);
5:    i := n + 1; increment := 0;
6:    for all D ∈ S do
7:      if D subsumes C then
8:        lastI :=IndexOfLastLiteral(D)
9:        if lastI < i then
10:         i := lastI; increment := 2^(n−i);
11:       end if
12:     end if
13:   end for
14:   if increment = 0 then
15:     return C̄
16:   else
17:     count := count + increment;
18:   end if
19: end while
20: return {};
```

Algorithm 2 works as follow: It counts full length clauses in the local variable $count$. In Line 2 it sets $count$ to zero, then starts a loop. In the loop, in Line 4, it converts $count$ to a full length clause, called C. In the second loop, see Line 6-13, it computes i, which is the smallest of the last literal indices of clauses which subsumes C. If no such i exists, i.e. there exists no clause in S, which subsumes C, then a model, $\overline{C}$ is found, and returned in Line 15. If we cannot find a not subsumed full length clause, then $count$ will be eventually greater or equal than 2^n. In this case it returns the empty set in Line 20.

Since the structure of the optimized algorithm is the same as CCC we do not need to prove that it stops for every valid input. Its soundness is also clear from soundness of Algorithm CCC, see Lemma 4, and from the fact that $count$ is incremented by 2^{n-i}, which is the number of consecutive full length clauses which are subsumed by D, see Lemma 3.

The algorithm suggests that in case of an unsatisfiable input clause set $count$ will be eventually greater or equal than 2^n, but actually in this case $count$ will be eventually equal to 2^n. We give only an intuitive reason to support this observation: If we would have more than n variables (we might run the algorithm on a subproblem), then the integer 2^n would represent the next possible clause to be check.

Line 8 implies that we have to index the variables. Assume that S contains a clause $\{a, b, c\}$ with indices i_a, i_b, and

i_c, where $i_a < i_b < i_c$, then CCC selects this clause in the worst case, i.e., if *count* is increased always by one, in $2^{i_a-1} \cdot 2^{i_b-i_a-1} \cdot 2^{i_c-i_b-1} = 2^{i_c-3}$ times in Line 7. This means that different indexing results in different running time.

While we investigated this algorithm, we made the observation that if we always select the best D in the for loop, then the biggest possible step is 2^{n-j}, where j is the index of the last 1 bit in the counter. Note, that this bit always comes from the previous run of the loop. It is so, because if there were an even better D in the input clause set, then the algorithm would select that D even in the previous run of the while loop and setting a bit to 1 which has smaller index than j. We use this observation in the next version of the algorithm, called CSFLOC.

VI. The CSFLOC Algorithm

Now we introduce the Counting Subsumed Full Length Ordered Clauses algorithm, for short CSFLOC. It is based on Optimized CCC, see Algorithm 2. In Optimized CCC we would like to increase the counter by the biggest possible 2^{n-i} step, i.e., we try to find a clause D which subsumes the counter and the index of its last literal is the smallest. While we experienced Optimized CCC we observed that the biggest possible step is 2^{n-j}, where j is the index of the last 1 bit in the counter. So it is useless to search the whole clause set to find the best clause. It is enough to check only those clauses where the index of the last literal is greater than equal to j.

The best solution is to use ordered clauses. Then we can define the following structure: $S[i] := \{C|C \in S \wedge \text{IndexOfLastLiteral}(C) = i\}$, , where $i = 1..n$, and where S is a clause set. In this case $S[k]$ is a set which contains those clauses from S where the last literal has variable index k.

With the help of this data structure we can redefine Optimized CCC. This version does not have to visit all clauses in each run. It runs until it finds the first clause which subsumes the clause representation of the counter searching from $S[1]$ to $S[n]$.

Algorithm 3 works as follow: It counts full length clauses in the local variable *count*. In Line 2 it creates the $S[]$ data structure which is an array of subsets of S where $S[i]$ contains those clauses from S where the index of last literal is i. In Line 3 it sets *count* to zero, then starts a loop. In the loop, in Line 6 it sets *increment* to zero. In Line 7, it converts *count* to a full length clause, called C. In the second loop, see Line 8-14, it computes j, which is the smallest of the last literal indices of clauses which subsumes C. If j is found then it sets the *increment* to be 2^{n-j} in Line 11. Line 12 is a kind of break statement, it stops the loop, because the smallest j value is found. If no such j is found, i.e. there exists no clause in S, which subsumes C, then a model, $\overline{C}$ is found, and returned in Line 16. If we cannot find a not subsumed full length clause, then *count* will be eventually greater or equal than 2^n. In this case it returns the empty set in Line 21.

Algorithm 3 Optimized CCCv2(S)

Require: S is a non-empty list of ordered clauses with variable indexing function I.

Ensure: If S is satisfiable it returns a model for S, otherwise returns the empty set.

```
1:  n := number of variables in S;
2:  S[i] := {C|C ∈ S∧IndexOfLastLiteral(C)= i}, where i = 1..n;
3:  count := 0;
4:  i := 0;
5:  while count < 2^n do
6:     increment := 0;
7:     C :=FullLengthClauseRepresentationOf(count);
8:     for j := 1; j <= n; j := j + 1 do
9:        if ∃D ∈ S[j] such that D subsumes C then
10:          C_i := C; D_i := D; i := i + 1;
11:          increment := 2^(n−j);
12:          j := n + 1; // it stops the for loop
13:       end if
14:    end for
15:    if increment = 0 then
16:       return C̄
17:    else
18:       count := count + increment;
19:    end if
20: end while
21: return {};
```

We show that the last 1 bit of the counter is always set by the previous run of the outer loop.

Lemma 7. *[Observation 1.] In Algorithm 3 we have that if C_x has a value and k =IndexOfLastPositiveLiteral(C_x), then k-th literal is negative in C_{x-1}, where $x >= 1$ and x is a natural number.*

Proof: We prove this by induction. This is true for $i = 1$, because C_0 is the clause where all literal is negative, because of Lines 3-4 and Line 7. The induction hypothesis is that this is true for C_{x-1}. We show that this is true for C_x. Let k =IndexOfLastPositiveLiteral(C_x). We have two cases, either the k-th literal is negative in C_{x-1} or positive. We have to show that this later case is not possible. Let us assume that the k-th literal in C_{x-1} is positive. Let m =IndexOfLastPositiveLiteral(C_{x-1}). Then there are 3 cases: (a) either $m < k$, (b) or $m > k$, (c) or $m = k$. Case (a) contradicts that the k-th literal in C_{x-1} is positive. In case (b) we have that C_{x-1} and C_x are clause representation of $counter_{x-1}$ and $counter_x$, respectively, such that $counter_x = counter_{x-1} + increment$, see Lines 11 and Line 18. We know that the m-th bit is 1 in $counter_{x-1}$ but 0 in $counter_x$, so we cannot have $counter_x = counter_{x-1} + increment$, because *increment* has only one 1 bit all others are 0, see Line 11. This is a contradiction. In case (c) we know that the k-th literal is the last positive literal in C_{x-1}

and also in C_x. From this we know that $j > k$, where j =IndexOfLastPositiveLiteral(D_{x-1}). From this we know that (I) from the k-th to n-th literals are not present in D_{x-1}. We also know from the induction hypothesis that in $count_{x-2}$ the k-th bit is 0, which implies that (II) from the 1-st to the $k-1$-th bits $count_{x-1}$ and $count_{x-2}$ are the same, and (III) the index of the last literal in D_{x-2} is less then k, because $counter_{x-1} = counter_{x-2} + increment$ and $increment$ has only one 1 bit all others are 0, see Lines 3, 9-11, and Line 18. From (I) and (II) it follows that D_{i-1} would be a suitable choice in Line 9 for the case $i = x-2$, i.e., in the $x-2$-th run of the outer loop, but this contradicts (III). ■

Now we show that the biggest possible $increment$ value is 2^{n-k}, where k is the index of the last 1 bit in the counter.

Lemma 8. *[Observation 2.] In Line 9 of Algorithm 3 we have that $2^{n-j} <= 2^{n-k}$, where k =IndexOfLastPositiveLiteral(C)., i.e., $j >= k$.*

Proof: Assume k =IndexOfLastPositiveLiteral(C) in Line 9 of Algorithm 3. Our goal is to show that $j >= k$ in Line 9 of Algorithm 3. We assume that our goal is not true, i.e., $j < k$. We show that this assumption leads to a contradiction, i.e., our goal is true. From the assumption $j < k$ it follows that there is a clause, say D, in $S[j]$, such that D subsumes C. From 7 we know that the k-th bit of the $counter$ is set in the previous run of the outer loop. It means that the previous $counter$ and the recent $counter$ differs only in $z >= 1$ bits on the indices from k to $k+z-1$, such that in the previous $counter$ the k.-th bit was 0 and now it is 1, $k+1$.-th bit was 1 and now it is 0, ... $k+z-1$.-th bit was 1 and now it is 0. In this case clause D is a suitable choice in Line 9 in the previous run of the outer loop, because the k-th ... $k+z$-th literals are not present in D, because the index of its last literal is j and $j < k$. Since the last literal index of D is j and $j < k$ the algorithm had to select this clause in the previous run of the outer loop, which would cause that the index of last 1 in the $counter$, i.e., the value of k should be greater than equal to j which contradicts that $j < k$. ■

The two observation lemmas 7 and 8 mean that if we always use the best D in the inner loop, as in `Optimized CCC`, then the biggest possible step is 2^{n-j}, where j is the index of the last 1 bit in the counter. This also means that Algorithm 3 is the same as Algorithm 2, except that it uses an auxiliary data structure.

Since Algorithm 3 is the same as Algorithm 2, we do not need to prove its sound- and completeness.

Algorithm 3 serves as the basis of `CSFLOC` which is introduced as Algorithm 4 in this paper. To get the new algorithm we need to change only one assignment: from $j := 1$ to $j := IndexOfLastPositiveLiteral(C)$, i.e., we use the observation that in case of `Optimized CCC` the biggest possible step is 2^{n-j}, where j is the index of the last 1 bit in the counter. So the new algorithm is defined as follows:

The observation is that the biggest possible step is 2^{n-j}, where j is the index of the last 1 bit in the counter. Note, that this bit always comes from the previous run of the loop.

Algorithm 4 `CSFLOC`(S)

Require: S is a non-empty list of ordered clauses with variable indexing function I.

Ensure: If S is satisfiable it returns a model for S, otherwise returns the empty set.

```
1:  n := number of variables in S;
2:  S[i] := {C|C ∈ S∧IndexOfLastLiteral(C) = i}, where i = 1..n;
3:  count := 0;
4:  while count < 2^n do
5:    increment := 0;
6:    C :=FullLengthClauseRepresentationOf(count);
7:    for j :=IndexOfLastPositiveLiteral(C); j <= n; j := j + 1 do
8:      if ∃D ∈ S[j] such that D subsumes C then
9:        increment := 2^(n-j);
10:       j := n + 1;
11:     end if
12:   end for
13:   if increment = 0 then
14:     return C̄
15:   else
16:     count := count + increment;
17:   end if
18: end while
19: return {};
```

It is so, because if there were an even better D in the input clause set, then the algorithm would select that D even in the previous loop and setting a bit to 1 which has smaller index than j. This optimization does not affect the soundness and the completeness of the algorithm.

VII. Implementation

We have implemented Algorithm 2 in Java and C#. The implementation can be downloaded from https://github.com/cccsat/cccsat/.

We have implemented Algorithm 3 and 4 also in ANSI C. The implementation can be downloaded from http://aries.ektf.hu/~gkusper/opt_ccc.c and http://aries.ektf.hu/~gkusper/csfloc.c. Actually the two files are the same except that in csfloc.c a flag called 'doIndexing' is set. Our goal was to demonstrate that the `CSFLOC` algorithm works. We did not want to come up with a fine-tuned algorithm, but we wanted to have a nice, readable code. The main idea of the implementation is to use arrays of integers for the bit representation of clauses.

The old Java implementation can handle SAT problems with at most 62 variable, the C# one is good until 64 variables. The new ANSI C implementation can handle any number of variables. This implementation uses an integer array to represent a clause and the counter. Since we use integers we can use fast bit operations to implement the methods of a clause.

We have also an experimental implementation in Java. This can be downloaded from http://aries.ektf.hu/~gkusper/CSFLOC6.java. This version uses a BitSet to represent the counter and ArrayList to represent a clause. This implementation can handle also any number of variables.

VIII. Test results

We tested the ANSI C version of Algorithm 3 and 4 and the experimental Java version of 4.

The tests were done on iMac macOS Sierra (CPU: 2,5GHz Intel Core i5, Memory: 4GB 1333MHz DDR3). We tested our algorithm against Glucose[1] 3.0, which is a widely used, standard SAT solver. We used Glucose 3.0 with simplification mode turned on.

The SAT instances were downloaded from the SATLIB Benchmark Problems [2] page. The rest is generated by the WnDGen SAT problem generator [4].

For each instance we used a timeout of 900 seconds. The result are presented in tables. The first column is the name of the tested SAT instance. In the 2nd to 3rd columns we show the number of variables (n), the number of clauses (m). The last three columns are CPU times in sec for Glucose 3.0, `Optimized CCC` and `CSFLOC`.

One of our goals was to countercheck the completeness of `CSFLOC` on lot of problems. Table I shows that we have run it for all uf20*, uf50*, and uuf50* instances from SATLIB Benchmark Problems. For all the 3000 instances it returned right result. The same is true for all other tests we performed.

From Table I we can see that random SAT instances, like uf50* and uuf50*, are difficult for `CSFLOC`, but very easy for Glucose. We can also see that `CSFLOC` outperforms `Optimized CCC`. We can see on lines hole6-9 that on pigeon hole problems, where we place $n+1$ pigeons in n holes without placing 2 pigeons in the same hole, `CSFLOC` can compete with Glucose if the number of variables is relatively small. It is so, because in pigeon hole problems there are lot of short clauses, which help `CSFLOC` to do big steps.

Columns **n** and **m** show the number of variables and the number of clauses, respectfully. This is true also for the other tables.

	n	**m**	**Glucose**	**CSFLOC**	**OptCCC**
uf20*	20	91	0,0042s	0,0046s	0,0043s
uf50*	50	218	0,0044s	1,0477s	5,1632s
uuf50*	50	218	0,0041s	5,4807s	26,2165s
hole6	42	133	0,0085s	0,0061s	0,0284s
hole7	56	204	0,0920s	0,0758s	0,5499s
hole8	72	297	1,1338s	1,1734s	12,1701s
hole9	90	415	13,1981s	19,7952s	TIME OUT

Table I
Runtimes on problems from SATLIB Benchmark Problems, uf* are SAT, others are $UNSAT$

[1] http://www.labri.fr/perso/lsimon/glucose/
[2] http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

One of our reviewer asked a question whether variable clustering gives any speed-up or not. To test this we used the variable clustering tool written by Prof. Tudor Jebelean [12]. The tool is available at http://aries.ektf.hu/~gkusper/Clustering-8-Jul-2010.tar.gz. Our experience shows, see Table II, that if we cluster the variables of a SAT problems then `CSFLOC` runs faster. Especially it can solve random SAT problems with 75 or 100 variables, which was otherwise very difficult for `CSFLOC`.

Instances	**n**	**m**	**clustering factor**	**CSFLOC**
uf20*	20	91	no clustering	0,0046 s
uf20*	20	91	clusters of 2 vars	0.0033s
uf20*	20	91	clusters of 3 vars	0.0031s
uf50*	50	218	no clustering	1,0477s
uf50*	50	218	clusters of 2 vars	0.2161s
uf50*	50	218	clusters of 3 vars	0.7917s
uf75*	75	325	no clustering	TIME OUT
uf75*	75	325	clusters of 2 vars	147,7269s
uf75*	75	325	clusters of 3 vars	129,1973s
uf100*	100	430	no clustering	TIME OUT
uf100*	100	430	clusters of 2 vars	TIME OUT
uf100*	100	430	clusters of 3 vars	25,0288s

Table II
Runtimes on problems after clustering, all instances are SAT

IX. IoT inspired test results

In a previous paper [5] a method was introduced to translate a directed graph into a SAT problem. The motivation was to check whether the communication graph of a wireless sensor network (WSN) is strongly connected or not. This means, that paper is a bridge between the fields of IoT and SAT. We showed that if the graph is strongly connected, then the generated SAT instance is a black-and-white SAT problem. A black-and-white SAT problem has only two solutions, the white assignment in which each variable is true, and the black assignment in which each variable is false.

We generated WSNs with 100 sensors with different density, and translated them into black-and-white 2-SAT problems using the tool written by Biró [5]. We added also the black and the white assignments to the tested SAT instances to make the strongly connected ones to be unsatisfiable. The less dense WSNs are not strongly connected, i.e., the resulting SAT problems are satisfiable. The more dense WSNs are strongly connected, i.e., the corresponding SAT instances are unsatisfiable.

We tested Glucose and our experimental Java implementation of `CSFLOC` on these instances. Table III shows the test results. The name of the instance indicates whether it is satisfiable (SAT), or unsatisfiable ($UNSAT$). We can see that Glucose is around 10 times faster than `CSFLOC`.

We also generated WSNs with more sensors: 10-10 instances with 200, 500, 1000, 2000 sensors, and with different density. We used the same method described in the previous test case, i.e., we translated the WSNs into black-and-white 2-SAT problems. Table IV shows average test results. Here

	n	m	Glucose	CSFLOC
WSN_100_SAT_1.cnf	100	676	0,0017s	0,0181s
WSN_100_SAT_2.cnf	100	852	0,0021s	0,0202s
WSN_100_SAT_3.cnf	100	837	0,0022s	0,0201s
WSN_100_UNSAT_1.cnf	100	1464	0,0026s	0,0241s
WSN_100_UNSAT_2.cnf	100	1458	0,0030s	0,0230s
WSN_100_UNSAT_3.cnf	100	2263	0,0037s	0,0271s
WSN_100_UNSAT_4.cnf	100	4631	0,0060s	0,0370s
WSN_100_UNSAT_5.cnf	100	6172	0,0093s	0,0461s
WSN_100_UNSAT_6.cnf	100	8134	0,0047s	0,0481s
WSN_100_UNSAT_7.cnf	100	842	0,0021s	0,0240s

Table III
RUNTIMES ON BLACK-AND-WHITE 2-SAT PROBLEMS

the number of clauses, i.e., column **m** is an average number. We can see that from 1000 sensors the CSFLOC algorithm is better to check whether the communication graph of a WSN is strongly connected or not. This result was a big surprise for us, because these SAT problems have the same number of variables as the number of sensors, see column **n**. It shows that a random SAT instance is very difficult for CSFLOC already with 100 variables, but not a black-and-white 2-SAT problem.

	n	m	Glucose	CSFLOC
WSN_100*	100	2732,9	0,0038s	0,0317s
WSN_200*	200	5171,6	0,0108s	0,0494s
WSN_500*	500	50310,0	0,1499s	0,1652s
WSN_1000*	1000	271103,6	2,4919s	0,5336s
WSN_2000*	2000	1006531,6	20,1476s	3,7704s

Table IV
AVERAGE RUNTIMES ON BLACK-AND-WHITE 2-SAT PROBLEMS

The black-and-white SAT problem is a special weakly nondecisive SAT (WnD) problem [13], [4]. A WnD problems can be generated by our tool, called WnDGen tool, available here: http://fmv.ektf.hu/tools.html. These problems have an interesting property, they have only two solutions, and the two solution are the opposite of each other, as in the case of black-and-white SAT problems. This means that black-and-white SAT problems are subsumed by WnD problems.

It seems that CSFLOC is very good at solving WnD problems generated by the WnDGen SAT problem generator, see Table V. To create the WnD problem instances we used the "-unsat" switch of WndGen tool, which adds the negation of the two solutions of the WnD problems to the instance, making it unsatisfiable. To create this table we used the Java implementation. This shows that overconstrained problems with less than 100 variables, like WnD problems, are easier for CSFLOC, than general SAT problems.

The Java implementation could not handle the last WnD problem instance, which is a huge one. We got an OutOfMemoryError in line 476. We are still working on this implementation.

Instances	n	m	Glucose	CSFLOC
WndGen_unsat_10_3.cnf	10	722	0,0017s	0,0281s
WndGen_unsat_10_4.cnf	10	1682	0,0112s	0,0242s
WndGen_unsat_10_5.cnf	10	2522	0,0276s	0,0321s
WndGen_unsat_10_6.cnf	10	2522	0,0316s	0,0532s
WndGen_unsat_15_3.cnf	15	2732	0,0027s	0,0281s
WndGen_unsat_15_4.cnf	15	10922	0,2088s	0,0893s
WndGen_unsat_15_5.cnf	15	30032	1,9055s	0,1282s
WndGen_unsat_15_6.cnf	15	60062	9,3856s	0,2062s
WndGen_unsat_15_7.cnf	15	90092	25,2388s	0,2991s
WndGen_unsat_15_8.cnf	15	102962	38,8714s	0,4343s
WndGen_unsat_15_9.cnf	15	90092	36,3287s	3,1751s
WndGen_unsat_20_3.cnf	20	6842	0,0063s	0,0532s
WndGen_unsat_20_4.cnf	20	38762	1,8473s	0,1271s
WndGen_unsat_20_5.cnf	20	155042	37,5251s	0,3022s
WndGen_unsat_20_6.cnf	20	465122	TIME OUT	0,7083s
WndGen_unsat_20_7.cnf	20	1085282	TIME OUT	3,5827s
WndGen_unsat_20_8.cnf	20	2015522	TIME OUT	8,7173s
WndGen_unsat_20_9.cnf	20	3023282	TIME OUT	ERROR

Table V
RUNTIMES ON WND PROBLEMS (ALL INSTANCES ARE $UNSAT$)

X. HOW TO CREATE A PARALLEL VERSION?

From CSFLOC we can create naturally a parallel algorithm. Assume we have q clients, then the j.-th CSFLOC instance has to count from $j \cdot 2^{n/q}$ till $(j+1) \cdot 2^{n/q-1}$, where $j = 0 \dots q-1$ and n is the number of variables in the input SAT problem.

The algorithm is so simple that there is no communication necessary between the clients.

Each node has to maintain only a counter, they can share the input SAT problem, because it does not change during the execution. This means that the memory usage is minimal.

A preliminary version can be found here: https://github.com/cccsat/cccsat/blob/master/CCC_BigInt_v1Dot2.java. We plan to implement it also for a GPU platform.

XI. FUTURE WORK

One could find a more closer connection between the inclusion-exclusion principle and CSFLOC. Let us assume, we have a following situation:

Example 3. *Let us have* 3 *variables,* $\{a, b, c\}$*, i.e.,* $n = 3$*. Let* $I(a) = 1, I(b) = 2, I(c) = 3$*. Let count be* 0*, so* $D = \{\overline{a}, \overline{b}, \overline{c}\}$*. Let the input clause set be* $\{C_1, C_2\}$*, where* $C_1 = \{\overline{a}, \overline{b}\}$*,* $C_2 = \{\overline{a}, \overline{c}\}$*. We can see that both* C_1 *and* C_2 *subsumes* D*. By the* CSFLOC *algorithm we have to increase count only by* 2^{3-2}*, because the minimum of maximum indices among* C_1 *is* $I(b) = 2$*. Now count* $= 2$ *and* $D = \{\overline{a}, b, \overline{c}\}$*. This is subsumed only by* C_2*. Its last variable is* c *and* $I(c) = 3$*. So we can increase count by* 2^{3-3}*. Now count* $= 3$ *and* $D = \{\overline{a}, b, c\}$*. It is not subsumed, so its negation is a solution.*

In the above example we can see that we find the same clause, C_1, in the first and also in the second loop, but we used it only in the second one. So it seems that in the first loop we could use both C_1 and C_2. We believe that the inclusion-exclusion principle could help us to find a better solution here.

A clever data structure could help to find directly those clauses from the input SAT problem which subsumes the counter. We have no suggestion here.

REFERENCES

[1] S.ANDREI,Countingforsatisfiabilitybyinvertingresolution, *ArtificialIntelligenceReview*,Volume 22, Issue 4,339–366,2004.

[2] H. BENNETT AND S. SANKARANARAYANAN, Model Counting Using the Inclusion-Exclusion Principle, *Theory and Applications of Satisfiability Testing - SAT 2011 Lecture Notes in Computer Science*, Volume 6695, 362–363, 2011.

[3] A. BIERE, M. HEULE, H. VAN MAAREN, T. WALSH, Handbook of Satisfiability, *IOS Press*, Amsterdam, 2009.

[4] CS. BIRO AND G. KUSPER, How to generate weakly nondecisive SAT instances, *Proceedings of 11th International IEEE Symposium on Intelligent Systems and Informatics (SISY)*, 265–269, Subotica, 2013.

[5] CS. BIRO AND G. KUSPER, Equivalence of Strongly Connected Graphs and Black-and-White 2-SAT Problems, *Miskolc Mathematical Notes*, accepted manuscript, MMN-2140.

[6] E. BIRNBAUM AND E. L. LOZINSKII, The good old Davis-Putnam procedure helps counting models, *Journal of Artificial Intelligence Research*, Volume 10, 457–477, 1999.

[7] S. A. COOK, The Complexity of Theorem-Proving Procedures, *Proc. of STOC'71*, 151–158, 1971.

[8] M. DAVIS, G. LOGEMANN, D. LOVELAND, A Machine Program for Theorem Proving, *Communications of the ACM*, Volume 5, 394–397, 1962.

[9] O. DUBOIS, Counting the Number of Solutions for Instances of Satisiability, *Theoretical Computer Science*, Volume 81, 49-64, 1991.

[10] CARLA P. GOMES, ASHISH SABHARWAL, AND BART SELMAN, Model Counting, *Chapter 20 of Handbook of Satisfiability*, IOS Press, Amsterdam, 2009.

[11] K. IWAMA, CNF-satisfiability test by counting and polynomial average time, *SIAM Journal on Computing*, Volume 18, Issue 2, 385–391, 1989.

[12] T. JEBELEAN AND G. KUSPER, MultiDomain Logic and its Applications to SAT, (invited talk), *SYNASC08*, DOI: 10.1109/SYNASC.2008.93, IEEE Computer Society Press, ISBN 978-0-7695-3523-4, 3–8, 2008.

[13] G. KUSPER, Solving and Simplifying the Propositional Satisfiability Problem by Sub-Model Propagation, *PhD thesis, Johannes Kepler University Linz, RISC Institute*, 1–146, 2005.

[14] G. KUSPER AND CS. BIRO, Solving SAT by an Iterative Version of the Inclusion-Exclusion Principle, *SYNASC 2015, DOI: 10.1109/SYNASC.2015.38, IEEE Computer Society Press, ISBN 978-1-5090-0461-4*, 189–190, 2015.

[15] E. L. LOZINSKII, Counting propositional models, *Information Processing Letters*, Volume 41, 327–332, 1992.

[16] Y. TANAKA, A Dual Algorithm for the Satisability Problem, *Information Processing Letters* Volume 37, 85-89, 1991.

14 Annex 7. - Solving SAT by an Iterative Version of the Inclusion-Exclusion Principle

This is the submitted version of G. Kusper, Cs. Biró, *Solving SAT by an Iterative Version of the Inclusion-Exclusion Principle*, Proceedings of SYNASC 2015, DOI: 10.1109/SYNASC.2015.38, pp. 189–190, 2015. See also [98].

Solving SAT by an Iterative Version of the Inclusion-Exclusion Principle

Gábor Kusper, Csaba Biró

Eszterházy Károly College, Eger
Institute of Mathematics and Informatics
Email: see http://fmv.ektf.hu

***Abstract*—Our goal is to present a basic, novel, and correct SAT solver algorithm; show its soundness; compare it with a standard SAT solver, give some ideas in which cases might it be competitive. We do not present a fine-tuned, state-of-the-art SAT solver, only a new basic algorithm. So we introduce CCC, a SAT solver algorithm which is an iterative version of the inclusion-exclusion principle. CCC stands for Counting Clear Clauses. It counts those full length (in our terminology: clear) clauses, which are subsumed by the input SAT problem. Full length clauses are n-clauses, where n is the number of variables in the input problem. A SAT problem is satisfiable if it does not subsume all n-clauses. The idea is that in an n-clause each of n variables is present either as a positive literal or as a negative one. So we can represent them by n bits. CCC is motivated by the inclusion-exclusion principle,it counts full length clauses as the principle does in case of the SAT problem, but in an iterative way. It works in the following way: It sets its counter to be 0. It converts 0 to an n-clause, which is the one with only negative literals. It checks whether this n-clause is subsumed by the input SAT problem. If yes, it increases the counter and repeats the loop. If not, we have a model, which is given by the negation of this n-clause. We show that almost always we can increase the counter by more than one. We show that this algorithm always stops and finds a model if there is one. We present a worst case time complexity analysis and lot of test results. The test results show that this basic algorithm can outperform a standard SAT solver, although its implementation is very simple without any optimization. CCC is competitive if the input problem contains lot of short clauses. Our implementation can be downloaded and the reader is welcome to make a better solver out of it. We believe that this new algorithm could serve as a good basis for parallel algorithms, because its memory usage is constant and no communication is needed between the nodes.**

***Keywords*—AT, #SAT, SAT Solver, Inclusion-Exclusion Principle.AT, #SAT, SAT Solver, Inclusion-Exclusion Principle.S**

I. Introduction

Propositional satisfiability is the problem of determining, for a formula of the propositional logic, if there is an assignment of truth values to its variables for which that formula evaluates to be true. SAT is one of the most-researched NP-complete [1] problems in several fields of computer science, including theoretical computer science, artificial intelligence, hardware design, and formal verification [2]. By SAT we mean the problem of propositional satisfiability for formulas in conjunctive normal form (CNF). Modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [3]. An interesting question is how many models does a SAT problem instance have. This problem is known as the #SAT problem. In other words, by #SAT we mean the problem of counting the models of a SAT instance [4]. The most popular way to solve this problem is to use a variant of the DPLL method which does not stop if it finds a model but keep exploring the search space. One of the first examples is CDP [8].

Another way to count the models is to use the inclusion-exclusion principle. It is well known that this principle can solve the #SAT problem [5], [6], [7].

Let $A_1, A_2, \ldots, A_m$ ***be sets, where*** $m > 0$***. The the inclusion-exclusion principle states that*** $|\bigcup_{i=1}^{m} A_i| = \sum_{i=1}^{m} |A_i| - \sum_{1 \leq i < j \leq m} |A_i \cap A_j| + \ldots (-1)^{m+1} |A_1 \cap A_2 \cap \cdots \cap A_m|$.

We recall that clause C ***subsumes clause*** D ***if and only if*** C ***is a subset of*** D***. In this paper we use clear clause as a synonym of full length clause.***

To be able to use the inclusion-exclusion principle on SAT we need the so called Clear Clause View [9]. We have to see the clauses as sets of the subsumed full length clauses. For example if we have 4 variables, a, b, c, d***, then the clause*** $\{a, b\}$ ***should be seen as the following set:*** $\{\{a, b, \overline{c}, \overline{d}\}, \{a, b, \overline{c}, d\}, \{a, b, c, \overline{d}\}, \{a, b, c, d\}\}$.

Now we can give more formaly how to use inclusion-exclusion principle on SAT. Let $C_1, \ldots, C_m$ ***be the clauses of a SAT problem instance. Let*** A_i ***be the set of full length clauses which are subsumed by*** C_i***,*** $i = 1 \ldots m$***. Then*** $|A_i|$ ***means the number of full length clauses subsumed by*** C_i***, and*** $|A_i \cap A_j|$ ***means the number of full length clauses subsumed by both*** C_i ***and*** C_j***. It is easy to see that*** $|A_i| = 2^{n-k}$***, where*** n ***is the number of variables in the clause set, and*** k ***is the number of literals in*** C_i***.***

In this way one can obtain the following formula from [7]. Let $T = \{C_1, C_2, \ldots, C_m\}$ ***be a clause set. Let*** $lits(S)$ ***be a function, which returns the set of literals in the clause set*** S***. Let*** n ***be the number of variables in*** T***. Then the number of models of*** T ***is*** $2^n - \sum_{S \subseteq \{C_1, C_2, \ldots, C_m\}} t(S)$***, where***

$$t(S) = \begin{cases} 0 & \text{, if } \exists a(\{a, \overline{a}\} \subseteq lits(S)); \\ (-1)^{|S|+1} \cdot 2^{n-|lits(S)|} & \text{, otherwise.} \end{cases}$$

Note that if C ***is a clause,*** A ***is a set of full length clauses which are subsumed by*** C***, then we have that*** $2^{n-|lits(C)|} = |A|$***. So we can also give the number of models of*** T ***as follows: Let*** $T = \{C_1, C_2, \ldots, C_m\}$ ***be a clause set. Let*** n ***be the number of variables in*** T***. Let*** A_i ***be the set of full length clauses which are subsumed by*** C_i***,*** $i = 1 \ldots m$***. Then the number of models of*** T ***is*** $2^n - \sum_{i=1}^{m} |A_i| - \sum_{1 \leq i < j \leq m} |A_i \cap A_j| + \ldots (-1)^{m+1} |A_1 \cap A_2 \cap \cdots \cap A_m|$.

From this expression 2^n ***is the number of possible full length clauses and the rest is the number of full length clauses which are subsumed by*** T***. Note that this part of the formula is just the same as the inclusion-exclusion principle.***

And here is the connection between the old principle and our new algorithm. Our algorithm counts the full length clauses as the inclusion-exclusion principle, but it does this in an iterative way as it is described below.

Our algorithm, CCC, is an iterative version of the inclusion-exclusion principle. This algorithm is a not a #SAT solver but a SAT solver. This means that while other ones use a SAT solver algorithm, namely DPLL, to solve the #SAT problem, we use an idea from the field of #SAT to solve the SAT problem. This gives us the advantage that our algorithm can be easily turned into a parallel solution.

The name CCC stands for Counting Clear Clauses. This algorithm counts those full length (in our terminology: clear) clauses, which are subsumed by the input SAT problem. Full length clauses are n-clauses, where n is the number of variables in the input problem. It is easy to see that there are 2^n different n-clauses. A SAT problem is unsatisfiable if it subsumes all n-clauses. If a SAT problem subsumes fewer, then it is satisfiable. This means that by counting the subsumed n-clauses we can decide satisfiability.

The CCC works as follows: It sets its counter to be 0. It converts 0 to an n-clause, which is the one with only negative literals. It checks whether this n-clause is subsumed by the input problem. This means that it looks for a clause D that is a subset of this n-clause. If such a clause D is found then it adds 1 to the counter and repeats the process until we find an n-clause which is not subsumed. It is very likely that the next n-clause will be subsumed by the same clause. Therefore, we add instead 2^{n-i} to the counter, where i is the index of the last literal of D. In this way we do not lose soundness.

The test results show that this basic algorithm can outperform a standard SAT solver, although its implementation is very simple without any optimization. CCC is competitive if the input problem contains lot of short clauses.

We have implemented it in Java and C#. Our implementation can be downloaded from this page: https://github.com/cccsat/cccsat/. The reader is welcome to make a better solver out of it.

II. Definitions

A** literal **is a boolean variable or the negation of a boolean variable. A** clause **is a set of literals. A** clause set **is a set of clauses. An** assignment **is a set of literals. Clauses are interpreted as disjunction of their literals. Assignments are interpreted as conjunction of their literals. A** clear clause **and a** clear assignment **has n literals, where n is the number of variables.

We denote negation by overbar, i.e., $\overline{a}$ means the negation of a. Note, that $\overline{\overline{a}} = a$. If H is a set, then $\overline{H}$ means that all elements in H are negated. If C is a clause, then $\overline{C}$ is an assignment. If A is an assignment, then $\overline{A}$ is a clause.

We say that clause C subsumes clause D if and only if (iff) C is a subset of D. Its interpretation is logical consequence, i.e., D is a logical consequence of C.

We say that clause set S subsumes clause C iff there is a clause in S which subsumes C. Formally: S subsumes $C \iff \exists D(D \in S \wedge D \subseteq C)$.

We say that assignment M is a** model **for clause set S iff for all $C \in S$ we have $M \cap C \neq \emptyset$. We say that a model of a clause set is a** clear model **if it is a clear assignment.

We say that the injective function I from variables to the natural numbers $1..n$ is a variable index function, where n is the number of variables. We can also apply I on literals, in this case $I(lit) = I(\overline{lit})$, where lit is a literal. If otherwise is not stated I is the lexicographical ordering of variables.

III. Theory

The CCC algorithm is based on the following lemmas.

Lemma 1. *[Clear Clause Rule, see Lemma 4.3.1. in [9]]] Let S be a clause set and C be a clear clause. Then*

$$(a)\ \ S \text{ subsumes } C \Leftrightarrow \neg(\overline{C} \text{ is a model for } S);$$

$$(b)\ \ \neg(S \text{ subsumes } C) \Leftrightarrow \overline{C} \text{ is a model for } S.$$

The proof of this lemma can be found in [9]. This lemma states that if S subsumes a clear clause, say C, then its negation is not a model. But if S does not subsume C, then its negation, $\overline{C}$, is a model for S. Hence, it is enough to find a not subsumed clear clause to find a model for S. Our algorithm is based on this fact.

The next lemma is trivial so we skip its proof.

Lemma 2. *Let S be a clause set with n variables. Then S has m different clear models iff S subsumes $2^n - m$ different clear clauses.*

This lemma has two important corollaries.

Corollary 1. *(a) Let S be a clause set with n variables. Then S is satisfiable iff S subsumes less than 2^n clear clauses. (b) Let C be a k-clause. Then C subsumes 2^{n-k} clear clauses, where n is the number of variables.*

IV. The CCC Algorithm

We use the following representation: In an n-clause all variables are present either as a positive literal, or as a negative one. This means that we can represent them by an n bit wide unsigned integer, where 0 corresponds to a negative literal on the same index, 1 to a positive one. We give an example:

Example 1. $\{a, \overline{b}, c, d, \overline{e}\} \iff$ `10110`

Using this observation we can order clear clauses by their n bit integer representation and prove the following lemma.

Lemma 3. *Let n be the number of boolean variables in use. Let I be an index function of these variables. Let D be a non-empty clause. Let i be the largest of the indices of literals of D, i.e., let $i = max\{I(d)|d \in D\}$. Then D subsumes 2^{n-i} consecutive clear clauses.*

Proof: We know that D does not contain any literal with index less than i. Let C be the set of clear clauses which are subsumed by D. Let C_0 be the element of C which has the smallest integer representation among elements of C. We show that D subsumes C_0 and the next $2^{n-i} - 1$ clear clauses. It is easy to see that the last $n - i$ literals of C_0 are negative literals. The next clear clause is the same as C_0 except that its last literal is positive. Let us denote this clause (i.e, the next one according to its integer representation) by C_1. D subsumes C_1 unless $i = n$, i.e., unless the case that the last variable is present in D. We can create C_2 by adding 1 to the integer representation of C_1 and translate it back to a clear clause. In this way we can construct the consecutive clear clauses. We can see that D subsumes each clear clause in this sequence until the i-th literal is not affected, i.e., it subsumes $C_0, C_1, \ldots, C_{2^{n-i}-1}$, but not $C_{2^{n-i}}$. Hence, D subsumes 2^{n-i} consecutive clear clauses. ■

To visualize the above lemma, we give two examples.

Example 2. *Let us assume that we have 6 variables, $a, b, \ldots, f$. This means that $n = 6$. Let $I(a) = 1, I(b) = 2, \ldots, I(f) = 6$. Let $D = \{\overline{b}, d\}$, i.e., $i = 4$. Then D subsumes the following 2^{n-i} consecutive clear clauses (we give in brackets the bit represantation of the clear clauses): $C_0 = \{\overline{a}, \overline{b}, \overline{c}, d, \overline{e}, \overline{f}\}$ `(000100)`, $C_1 =$*

$\{\overline{a}, \overline{b}, \overline{c}, d, \overline{e}, f\}$ (000101), $C_2 = \{\overline{a}, \overline{b}, \overline{c}, d, e, \overline{f}\}$ (000110), $C_3 = \{\overline{a}, \overline{b}, \overline{c}, d, e, f\}$ (000111), but does not subsume the next one, which is: $C_4 = \{\overline{a}, \overline{b}, c, \overline{d}, \overline{e}, \overline{f}\}$ (001000)

In the next example we show the case when the last literal is negative in D.

Example 3. *Let us assume that we have 6 variables, $a, b, \ldots f$. This means that $n = 6$. Let $I(a) = 1, I(b) = 2, \ldots, I(f) = 6$. Let $D = \{\overline{b}, \overline{d}\}$, i.e., $i = 4$. Then D subsumes the following 2^{n-i} consecutive clear clauses (we give in brackets the bit represantation of the clear clauses): $C_0 = \{\overline{a}, \overline{b}, \overline{c}, \overline{d}, \overline{e}, \overline{f}\}$ (000000), $C_1 = \{\overline{a}, \overline{b}, \overline{c}, \overline{d}, \overline{e}, f\}$ (000001), $C_2 = \{\overline{a}, \overline{b}, \overline{c}, \overline{d}, e, \overline{f}\}$ (000010), $C_3 = \{\overline{a}, \overline{b}, \overline{c}, \overline{d}, e, f\}$ (000011), but does not subsume the next one, which is: $C_4 = \{\overline{a}, \overline{b}, \overline{c}, d, \overline{e}, \overline{f}\}$ (000100)*

We present two version of CCC. The first counts one by one. The second is more optimized.

Algorithm 1 works as follow: It sets $count$ to zero, then starts a loop. In the loop, in Line 4., it converts $count$ to a clear clause. In Line 5. it checks whether this clear clause is subsumed or not. If yes, it increases $count$ by one in Line 6. If not, by point (b) of Lemma 1, a model is found, which is returned in Line 8. Eventually $count$ either grows beyond $2^n - 1$ or a model is found. In the first case we know from point (a) of Corollary 1 that S is unsatisfiable.

We also give this algorithm in a very intuitive way as follows: Go trough all possible truth assignments x in the lexicographical order. If x satisfies the input formula, return x and stop. If there is no more assignment, return the empty set and stop.

Lemma 4. *[Soundness of CCC] Algorithm 1 is sound.*

Proof: Algorithm 1 counts clear clauses in the local variable $count$. In Line 2. it sets $count$ to zero, then starts a loop. In the loop, in Line 4., it converts $count$ to a clear clause, called C. In Line 5. it checks whether C is subsumed or not. If C is subsumed in S, then it increases $count$ by one in Line 6. This means that in the next run of the loop it checks whether the next clear clause is subsumed or not. If C is not subsumed in S, by point (b) of Lemma 1, a model, $\overline{C}$ is found, which is returned in Line 8. If we cannot find a not subsumed clear clause, then $count$ will be eventually equal to 2^n. From this we know, by point (a) of Corollary 1, that S is unsatisfiable. In this case it returns the empty set in Line 11. ■

Lemma 5. *[Completeness of CCC] Algorithm 1 is complete.*

Proof: We already know from Lemma that Algorithm 1 is sound. So it is enough to show that it stops for each valid input. Algorithm has one while loop, which terminates either if $count \geq 2^n$ or if we found a clear clause which is not subsumed by S. In the while loop there is one if statement. In the else branch we have a return statement, i.e., the while loop terminates in that branch. If it does not terminate in the else branch, then in the then branch, in Line 6., we increase $count$ by one. This means that if it does not terminate in the else branch, then $count$ will be eventually equal to 2^n, i.e., the while loop eventually terminates. Hence, Algorithm is complete. ■

Lemma 6. *[Worst case complexity of CCC] The worst case complexity of CCC is $O(2^n \cdot Check \cdot Conversion)$, where Check is the complexity of a subsumption check, and Conversion is the complexity of converting a number to a clear clause.*

Proof: In the worst case we have to visit all the 2^n clear clauses. ■

Now we give a more optimized version of CCC. First we present it in a very intuitive way as follows:
Let x be the first assignment in lexicographical order.
Repeat until last assignment is reached:
If x satisfies the input formula, return x and stop.
Otherwise:
Let i be the smallest number such that x_i appears in an unsatisfied clause as its last literal. Let x be the assignment that comes 2^{n-i} places after current x in lexicographical order.

Now we give it more formally.

Algorithm 2 works as follow: It sets $count$ to zero, then starts a loop. In the loop, see Line 4., it converts $count$ to a clear clause, called C. In the second loop, see Line 6., we select that clause which subsumes this clear clause, see Line 7., and the index of its last literal is smallest, see Lines 8-11. If there exists no such clause, then a model is found, we return it in Line 15. If the clear clause is subsumed, then we increases $count$ by 2^{n-i}, see Line 17., where i is the minimum of the last literal indices in those clauses in S which subsumes C. Once more, i is the minimum of the maximum indices from the subsuming clauses. This grants the biggest 2^{n-i} step.

Line 8. implies that we have to index the variables. Assume that S contains a clause $\{a, b, c\}$ with indices i_a, i_b, and i_c, where $i_a < i_b < i_c$, then CCC selects this clause in the worst case, i.e., if $count$ is increased always by one, in $2^{i_a - 1} \cdot 2^{i_b - i_a - 1} \cdot 2^{i_c - i_b - 1} = 2^{i_c - 3}$ times in Line 7. This means that different indexing results in different running time.

In Line 17. we do not increase $count$ by one, but by 2^{n-i}. We can do that, because D subsumes not only C but a range of clear clauses. See the examples.

Example 4. *Let us index the variables from a to e by 1 to 5. Let D be $\{a, c\}$, let count be 10100 in binary format, so C is $\{a, \overline{b}, c, \overline{d}, \overline{e}\}$. One can see that D also subsumes these clear clauses: $\{a, \overline{b}, c, \overline{d}, e\}$, $\{a, \overline{b}, c, d, \overline{e}\}$, and $\{a, \overline{b}, c, d, e\}$. So we can increase count by 4, i.e., by 2^{5-3}, where 5 is n and 3 is the index of the last literal of D.*

Let us see the same example but with a negative literal in D.

Example 5. *Let D be $\{a, \overline{c}\}$, let count be 10000, so C is $\{a, \overline{b}, \overline{c}, \overline{d}, \overline{e}\}$. One can see that D also subsumes these clear clauses: $\{a, \overline{b}, \overline{c}, \overline{d}, e\}$, $\{a, \overline{b}, \overline{c}, d, \overline{e}\}$, and $\{a, \overline{b}, \overline{c}, d, e\}$. So we can increase count by 2^{5-3}.*

Since the structure of the optimized algorithm is the same as CCC, we prove only soundness of it.

Lemma 7. *[Soundness of OptimizedCCC] Algorithm 2 is sound.*

Proof: Algorithm 1 counts clear clauses in the local variable $count$. In Line 2. it sets $count$ to zero, then starts a loop. In the loop, in Line 4., it converts $count$ to a clear clause, called C. In the second loop, see Line 5-13., it computes i, which is the smallest of the last literal indices of clauses which subsumes C. If no such i exists, i.e. there exists no clause in S, which subsumes C, then , by point (b) of Lemma 1, a model, $\overline{C}$ is found, and returned in Line 15. Otherwise, by Lemma 3, we can increase $count$ by 2^{n-i}, see Line 17. Assume that i comes from clause D from Line 7. To be able to use Lemma 3 we also have to show that the last $n - i$ literals of C are negative. Assume that this does not true. Then in one of the previous iteration we had to increase the counter along a clause which differs in no liter from D, but the index of its last literal is bigger then i. But then already in that previous iteration we had to choice D. This is a contradiction so we can use Lemma 3. If we cannot find a not subsumed clear clause, then $count$ will be eventually greater or equal than 2^n. From this we know, by point (a) of Corollary 1, that S is unsatisfiable. In this case it returns the empty set in Line 20. ■

Algorithm 1 CCC(S)

Require: S is a non-empty clause set.
Ensure: If S is satisfiable it returns a model for S, otherwise returns the empty set.

```
1: n := number of variables in S;
2: count := 0;
3: while count < 2^n do
4:    C :=ClearClauseRepresentationOf(count);
5:    if ∃D(D subsumes C) then
6:       count := count + 1;
7:    else
8:       return C̄;
9:    end if
10: end while
11: return ∅;
```

Algorithm 2 OptimizedCCC(S)

Require: S is a non-empty clause set.
Ensure: If S is satisfiable it returns a model for S, otherwise returns the empty set.

```
1: n := number of variables in S;
2: count := 0;
3: while count < 2^n do
4:    C :=ClearClauseRepresentationOf(count);
5:    i := ∞;
6:    for all D ∈ S do
7:       if D subsumes C then
8:          lastI := max{I(d)|d ∈ D}
9:          if lastI < i then
10:            i := lastI;
11:         end if
12:      end if
13:   end for
14:   if i = ∞ then
15:      return C̄
16:   else
17:      count := count + 2^(n−i);
18:   end if
19: end while
20: return ∅;
```

V. Implementation

We have implemented Algorithm 2. in Java and C#. The implementation can be downloaded from https://github.com/cccsat/cccsat/.

Our goal was to demonstrate that the CCC algorithm works. We did not want to come up with a fine-tuned algorithm, but we wanted to have a nice, readable code.

The Java implementation (CCC_v1Dot0.java) is only 175 lines with comments. The C# implementation (SATCounter.cs) contains also some experimental functions.

The main idea of the implementation is to use long integers for the bit representation of clauses. Two long integers represent one clause, see the Clause class in CCC_v1Dot0.java.

The Java implementation can handle SAT problems with at most 62 variable, the C# one is good until 64 variables. Since we use long integers we can use fast bit operations to implement the methods of a clause. We have also an implementation (CCC_BigInt_v1Dot2.java) based on java.math.BigInteger, where there is no limit on the number of variables.

We checked by a profiler how many percent of the running time is spent in each method. We found that CCC_v1Dot0.java spends 60-95% of its time in the SubsumersOf and Subsumes methods, and 5-10% in SolveFrom0.

So it seems that one of the bottle-necks of CCC is the subsumption check. We introduced a cheaper version of subsumption check in CCC_v1Dot1.java, but we got no significant speed-up. So this version is in experimental phase.

The same is true for the bombing style version of CCC (CCCBomberv1Dot0.java). This one starts many counters (in this context: bombs) distributed over the search space. The more bombs are the better chance is that one of the them is lucky and finds a solution quickly.

VI. Test results

We tested only the first Java version (CCC_v1Dot0.java) from the implemented ones.

The tests were done on a dual-core machine with 4 GB of RAM running at 2.26 Ghz. We tested our algorithm against MiniSat[1] 2.2.0, which is a widely used, standard SAT solver.

The SAT instances were downloaded from the SATLIB Benchmark Problems [2] page, from the SAT Challenge 2012 (section Hard Combinatorical) [3] and the rest is generated by the WnDGen SAT problem generator [10].

For each instance we used a timeout of 900 seconds. The result are presented in tables. The first column is the name of the tested SAT instance. In the 2nd to 5th columns we show the number of variables (n), the number of clauses (m), the length of clauses (k), and the ratio of clauses and variables ($r = m/n$). The last two columns are CPU times in sec for CCC and MiniSat.

One of our goals was to countercheck the correctness of CCC on lot of problems. Tables I-II. show that we have run it for all uf20*, uf50*, and uuf50* instances from SATLIB Benchmark Problems. For all the 3000 instances it returned right result. The same is true for all other tests we performed.

From Tables I-II. we can see that random SAT instances, like uf50* and uuf50*, are difficult for CCC, but very easy for MiniSat. Random SAT instances with few variables, like uf20*, are also easy

[1] http://www.minisat.se/
[2] http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html
[3] http://baldur.iti.kit.edu/SAT-Challenge-2012/downloads/sc2012-hard-combinatorial.tar

for CCC. MiniSat outperforms CCC in almost all cases, CCC has chance only in cases where are few variables and the clauses are short, see instance_n2_i.*

Table III. shows a very interesting picture. All the cases are either very simple of very difficult for CCC. We have analyzed these easy instances deeply and found that all these examples would be also very difficult for CCC, but their solution happens to be very close to the initial value of the counter.

This result suggests a bombing style CCC algorithm, which starts a parallel search from many different counters (in this context: bombs) distributed over the search space. One of the bombs might be lucky to find a quick solution.

We have an experimental version (CCCBomberv1Dot0.java) of this bombing style CCC algorithm. The preliminary results show that this version can solve even Q3inK11.cnf. It solves it in 14,674 sec., so it is much faster than MiniSat.

From Tables IV-V. we can see that CCC is very competitive on weakly nondecisive SAT problems [9], [10].

These problems are over-constructed, they contain lot of clauses. It seems that this is good for CCC, because lot of clauses subsume the counter, and there is a good chance that one of them allow to make a big step. As the length of the clauses are bigger and bigger (column k), then this step becomes smaller and smaller scales as the number of variables are increasing. [10] suggest to have $n = 2k-3$ to construct difficult problems. One can see the exponential growth as one expects in case of a SAT solver. On these problems CCC is better than MiniSat if the number of clauses are bigger than few thousands. To summarize, the performance of CCC is good if there are a lot of classes, preferably lot of short ones. It seems that in this case it can handle also SAT instances with relatively big number of variables.

VII. How to create a parallel version?

From CCC we can create naturally a parallel algorithm. Assume we have q clients, then the j.-th CCC instance has to count from $j \cdot 2^{n/q}$ till $(j+1) \cdot 2^{n/q-1}$, where $j = 0 \dots q-1$ and n is the number of variables in the input SAT problem.

The algorithm is so simple that there is no communication necessary between the clients.

Each node has to maintain only a counter, they can share the input SAT problem, because it does not change during the execution. This means that the memory usage is minimal.

VIII. Future work

It seems that the bottle-neck of CCC is the subsumption check. We introduced a cheaper version (CCC_v1Dot1.java), which uses only 2 long integer bit methods. The older version uses 4 ones. But we did not get any speed-up. We have to analyze this situation deeply.

One could find a more closer connection between the inclusion-exclusion principle and CCC. Let us assume, we have a following situation:

Example 6. *Let us have 3 variables, $\{a, b, c\}$, i.e., $n = 3$. Let $I(a) = 1, I(b) = 2, I(c) = 3$. Let count be 0, so $D = \{\overline{a}, \overline{b}, \overline{c}\}$. Let the input clause set be $\{C_1, C_2\}$, where $C_1 = \{\overline{a}, \overline{b}\}$, $C_2 = \{\overline{a}, \overline{c}\}$. We can see that both C_1 and C_2 subsumes D. By the CCC algorithm we have to increase count only by 2^{3-2}, because the minimum of maximum indices among C_1 $I(b) = 2$. Now count $= 2$ and $D = \{\overline{a}, b, \overline{c}\}$. This is subsumed only by C_2. Its last variable is c and $I(c) = 3$. So we can increase count by 2^{3-3}. Now count $= 3$ and $D = \{\overline{a}, b, c\}$. It is not subsumed, so its negation is a solution.*

In the above example we can see that we find the same clause, C_1, in the first and also in the second loop, but we use it only in the second one. So it seems that in the first loop we could use both C_1 and C_2. We believe that the inclusion-exclusion principle could help us to find a better solution here.

A clever data structure could help to find directly those clauses from the input SAT problem which subsumes the counter. We have no suggestion here.

It seems that the bombing style CCC algorithm is very fast on big satisfiable problems. Our implementation is pure, serves only as an experimental version. A better implementation shall use threads and/or GPUs.

IX. References

References

[1] **S. A. Cook**, *The Complexity of Theorem-Proving Procedures,* **Proc. of STOC'71**, *151–158, 1971.*

[2] **A. Biere, M. Heule, H. van Maaren, T. Walsh**, *Handbook of Satisfiability,* **IOS Press**, *Amsterdam, 2009.*

[3] **M. Davis, G. Logemann, D. Loveland**, *A Machine Program for Theorem Proving.* **Communications of the ACM**, *vol. 5, 394–397, 1962.*

[4] **Carla P. Gomes, Ashish Sabharwal, and Bart Selman**, *Model Counting,* **Chapter 20 of Handbook of Satisfiability**, *IOS Press, Amsterdam, 2009.*

[5] **K. Iwama**, *CNF-satisfiability test by counting and polynomial average time,* **SIAM Journal on Computing**, *Vol. 18, No. 2, 385–391, 1989.*

[6] **E. L. Lozinskii**, *Counting propositional models,* **Information Processing Letters**, *Vol. 41, 327–332, 1992.*

[7] **H. Bennett and S. Sankaranarayanan**, *Model Counting Using the Inclusion-Exclusion Principle,* **Theory and Applications of Satisfiability Testing - SAT 2011 Lecture Notes in Computer Science**, *Volume 6695, 362–363, 2011.*

[8] **E. Birnbaum and E. L. Lozinskii**, *The good old Davis-Putnam procedure helps counting models,* **Journal of Artificial Intelligence Research**, *10:457–477, 1999.*

[9] **G. Kusper**, *Solving and Simplifying the Propositional Satisfiability Problem by Sub-Model Propagation,* **PhD thesis, Johannes Kepler University Linz, RISC Institute**, *1–146, 2005.*

[10] **Cs. Biro and G. Kusper**, *How to generate weakly nondecisive SAT instances,* **Proceedings of 11th International IEEE Symposium on Intelligent Systems and Informatics (SISY)**, *265–269, Subotica, 2013.*

Instances	n	m	k	r	CCC_v.1.0	MiniSat
uf20-all instance (average time)	20	91	3	4,55	0,0249 s	0,0024 s
uf50-all instance (average time)	50	218	3	4,36	22,1921 s	0,0032 s
ais6.cnf	61	581	2 ... 6	9,52	0,4262 s	0,0035 s
anomaly.cnf	48	261	2 ... 3	5,43	0,0346 s	0,0375 s

Table I. RUNTIMES AND RESULTS (ALL INSTANCES ARE sat)

Instances	n	m	k	r	CCC_v.1.0	MiniSat
uuf50-instance(average time)	50	218	3	4,36	105,0742 s	0,0038 s
aim-50-1_6-no-4.cnf	50	80	3	1,6	TO	0,0038 s
dubois20.cnf	60	160	3	2,66	TO	0,0038 s
Pret60_60.cnf	60	160	3	2,66	TO	0,0038 s
urq35.cnf	46	470	2...7	10,21	TO	147,0772 s
instance_n2_i3_pp_ci_ce.cnf	39	80	2...3	2,05	0,0032 s	0,0200 s
instance_n2_i2_pp_ci_ce.cnf	27	53	2...3	1,96	0,0036 s	0,0221 s
battleship-5-8-unsat.cnf	40	105	2...7	2,62	3,6677 s	0,0402 s
battleship-6-9-unsat.cnf	54	171	2 ...9	3,16	78,7101 s	0,2560 s
hole6.cnf	42	133	2...6	3,16	0,3519 s	0,0121 s
hole7.cnf	56	204	2...7	3,64	4,5313 s	0,0920 s
s57-100.cnf	57	124	3	2,17	TO	2,7521 s

Table II. RUNTIMES AND RESULTS (ALL INSTANCES ARE $unsat$)

Instances		n	m	k	r	CCC_v.1.0	MiniSat
Q32inK08.cnf	SAT	36	15120	12	420	0,03321 s	0,7202 s
Q32inK09.cnf	UNSAT	36	7938	4...12	220,5	TO	67,4242 s
Q32inK10.cnf	UNSAT	45	38430	4...12	854	TO	108,5270 s
Q3inK09.cnf	SAT	36	15120	12	420	0,03915 s	0,7360 s
Q3inK10.cnf	SAT	45	75600	12	1680	0,06784 s	17,08511 s
Q3inK11.cnf	SAT	55	277200	12	5040	TO	413,4461 s

Table III. RUNTIMES AND RESULTS

Instances	n	m	k	r	CCC_v.1.0	MiniSat
WndGen_sat_10_3.cnf	10	720	3	72	0,0049 s	0,0041 s
WndGen_sat_10_4.cnf	10	1680	4	168	0,0162 s	0,0076 s
WndGen_sat_10_5.cnf	10	2520	5	252	0,0023 s	0,0422 s
WndGen_sat_10_6.cnf	10	2520	6	252	0,0025 s	0,0441 s
WndGen_sat_15_3.cnf	15	2730	3	182	0,0235 s	0,0012 s
WndGen_sat_15_4.cnf	15	10920	4	728	0,0254 s	0,2560 s
WndGen_sat_15_5.cnf	15	30030	5	2002	0,0327 s	2,6401 s
WndGen_sat_15_6.cnf	15	60060	6	4004	0,0884 s	12,7168 s
WndGen_sat_15_7.cnf	15	90090	7	6006	0,9110 s	33,6661 s
WndGen_sat_15_8.cnf	15	102960	8	6864	0,8383 s	56,1115 s
WndGen_sat_15_9.cnf	15	90090	9	6006	12,3781 s	49,4391 s
WndGen_sat_20_3.cnf	20	6840	3	342	0,0295 s	0,0198 s
WndGen_sat_20_4.cnf	20	38760	4	1938	0,0439 s	2,3321 s
WndGen_sat_20_5.cnf	20	155040	5	7752	0,0912 s	48,8191 s
WndGen_sat_20_6.cnf	20	465120	6	23256	1,4017 s	628,2151 s
WndGen_sat_20_7.cnf	20	1085280	7	54264	4,4150 s	TO
WndGen_sat_20_8.cnf	20	2015520	8	100776	17,7954 s	TO
WndGen_sat_20_9.cnf	20	3023280	9	151164	32,9482 s	TO
WndGen_sat_20_10.cnf	20	3695120	10	184756	137,1843 s	TO

Table IV. RUNTIMES AND RESULTS (ALL INSTANCES ARE sat)

Instances	n	m	k	r	CCC_v.1.0	MiniSat
WndGen_unsat_10_3.cnf	10	722	3…10	72,2	0,0162 s	0,0017 s
WndGen_unsat_10_4.cnf	10	1682	4…10	168,2	0,0329 s	0,0181 s
WndGen_unsat_10_5.cnf	10	2522	5…10	252,2	0,0341 s	0,0400 s
WndGen_unsat_10_6.cnf	10	2522	6…10	252,2	0,0379 s	0,0520 s
WndGen_unsat_15_3.cnf	15	2732	3…15	910,66	0,0313 s	0,0008 s
WndGen_unsat_15_4.cnf	15	10922	4…15	728,13	0,0512 s	0,3160 s
WndGen_unsat_15_5.cnf	15	30032	5…15	2002,13	0,0657 s	2,9121 s
WndGen_unsat_15_6.cnf	15	60062	6…15	4004,13	0,3991 s	14,5689 s
WndGen_unsat_15_7.cnf	15	90092	7…15	6006,13	2,3084 s	41,9386 s
WndGen_unsat_15_8.cnf	15	102962	8…15	6864,13	5,6152 s	74,1166 s
WndGen_unsat_15_9.cnf	15	90092	9…15	6006,13	21,3174 s	71,0364 s
WndGen_unsat_20_3.cnf	20	6842	3…20	342,1	0,0481 s	0,0115 s
WndGen_unsat_20_4.cnf	20	38762	4…20	1938,1	0,0819 s	2,8081 s
WndGen_unsat_20_5.cnf	20	155042	5…20	7752,1	0,4262 s	59,9397 s
WndGen_unsat_20_6.cnf	20	465122	6…20	23256,1	3,4489 s	815,1759 s
WndGen_unsat_20_7.cnf	20	1085282	7…20	54264,1	12,9735 s	TO
WndGen_unsat_20_8.cnf	20	2015522	8…20	100776,1	67,5555 s	TO
WndGen_unsat_20_9.cnf	20	3023282	9…20	151164,1	328,1805 s	TO
WndGen_unsat_20_10.cnf	20	3695122	10…20	184756,1	TO	TO

Table V. RUNTIMES AND RESULTS (ALL INSTANCES ARE $unsat$)

Instances	n	m	k	r	CCC_v.1.0	Minisat
WndGen_sat_5_4.cnf	5	40	4	8	0,0039 s	0,0010 s
WndGen_sat_7_5.cnf	7	210	5	30	0,0122 s	0,0020 s
WndGen_sat_9_6.cnf	9	1008	6	112	0,0933 s	0,0122 s
WndGen_sat_11_7.cnf	11	4620	7	420	0,1162 s	0,1240 s
WndGen_sat_13_8.cnf	13	20592	8	1584	0,2995 s	2,20814 s
WndGen_sat_15_9.cnf	15	90090	9	6006	13,2413 s	46,0549 s
WndGen_sat_17_10.cnf	17	388960	10	22880	142,2362 s	TO
WndGen_sat_19_11.cnf	19	1662804	11	87516	546,7632 s	TO
WndGen_sat_21_13.cnf	21	7054320	12	335920	TO	TO

Table VI. RUNTIMES AND RESULTS (ALL INSTANCES ARE sat)

Instances	n	m	k	r	CCC_v.1.0	Minisat
WndGen_unsat_5_4.cnf	5	42	4	8,4	0,0041 s	0,0010 s
WndGen_unsat_7_5.cnf	7	212	5	30,28	0,0182 s	0,0020 s
WndGen_unsat_9_6.cnf	9	1010	6	112,22	0,0927 s	0,0160 s
WndGen_unsat_11_7.cnf	11	4622	7	420,18	0,1719 s	0,1800 s
WndGen_unsat_13_8.cnf	13	20594	8	1584,15	0,9125 s	3,2242 s
WndGen_unsat_15_9.cnf	15	90092	9	6006,13	29,6420 s	76,2888 s
WndGen_unsat_17_10.cnf	17	388962	10	22880,11	262,9622 s	TO
WndGen_unsat_19_11.cnf	19	1662806	11	87516,10	TO	TO
WndGen_unsat_21_13.cnf	21	7054322	12	335920,09	TO	TO

Table VII. RUNTIMES AND RESULTS (ALL INSTANCES ARE $unsat$)