Using Disjunctive Diagrams for Preprocessing of Conjunctive Normal Forms

Victor Kondratiev
ITMO University, St. Petersburg, Russia
JetBrains Research, St. Petersburg, Russia
Matrosov Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia
Email: vikseko@gmail.com

***Abstract*—In this article, in the context of the Boolean satisfiability problem (SAT), the question of speeding up the SAT solvers on specific input formulas is considered. The speedup is achieved using the CNF preprocessing algorithm which is based on the use of decision diagrams of a special kind, called Disjunctive Diagrams. These diagrams allow one to naturally test some sets of partial variable assignments and add more stringent constraints to the original formula. Several families of SAT instances are considered and used to compare the solving time before and after preprocessing, including a selection of tests from the "SAT Competition 2021" and some tests related to the problem of checking the equivalence of Boolean circuits.**

Computational experiments have shown that for hard instances in more than half of the cases the proposed preprocessing algorithm can speed up the solving time of considered CNFs.

***Keywords*—Boolean satisfiability problem; conjunctive normal form; preprocessing; decision diagrams; disjunctive diagrams**

I. Introduction

The Boolean Satisfiability Problem (SAT) is one of the well-known NP hard problems. A large class of combinatorial problems can be reduced to SAT, for which at the moment there are no known algorithms that do not require a search of sets of exponential power. But, in spite of structural hardness of SAT, in practice, thanks to modern fast data structures and heuristic algorithms, it can be solved efficiently quite often. Modern SAT solvers successfully handle formulas containing hundreds of thousands of variables and millions of clauses. Because of this, modern SAT solvers are used in such areas as bioinformatics [1], verification [2], [3], automatic theorem proving [4] and in many others. The issue of improving the efficiency of basic algorithms for SAT solving has been relevant for more than 30 years.

One of the important questions accompanying the development of SAT solving algorithms is the data structures that represent the original formulas. Although one of the most common is the Conjunctive Normal Form (CNF) format, other structures can also be used to represent the Boolean formulas. This paper investigates the possibility of representing Boolean formulas in the form of graphs of a special kind, specifically decision diagrams, and using these representations in combination with modern SAT solvers. In this approach, the information provided by the diagram can be used to speed up the SAT solver. Conversely, the use of the SAT solver as an oracle may allow to optimize the diagram representing the formula under consideration.

When decision diagrams are mentioned, the first thing that comes to mind are Binary Decision Diagrams (BDD) and their optimized form, Reduced Ordered Binary Decision Diagrams (ROBDD). These two types of diagrams have a wide popularity. One of the most cited works related to informatics has long been the Randal Bryant's study of ROBDD processing algorithms [5]. In addition, for many years ROBDD has been used as the main algorithmic tool in symbolic verification [2]. The attractive feature of ROBDDs is that they provide the most compact representation of a complete Boolean function over a class of graphs defined using the well-known Shannon decomposition [6]. However, in the context of this study it is worth noting that the problem of constructing a ROBDD representation for an arbitrary Boolean function defined by a formula is extremely complicated. It can be shown that if there exists an algorithm for solving this problem in polynomial time of the formula length, then $P = \#P$, where $\#P$ is an enumerative analog of class NP. Consequently, it can be argued that in the general case the construction of ROBDD representations of functions given by formulas of large dimension in reasonable time is not possible.

In [7] a class of decision diagrams was presented which was specially designed to represent arbitrary disjunctive normal forms (DNFs) in the form of graphs of special kind, called Disjunctive Diagram (DJD). Such diagrams, like the well-known Zero-suppressed Decision Diagram (ZDD) [8], are constructed for an arbitrary DNF in a polynomial time of the length of the original formula. However, disjunctive diagrams are not binary and do not use additional variables to represent DNFs. In this study, disjunctive diagrams combined with the SAT oracle are used to preprocess arbitrary conjunctive normal forms. Using the information obtained from the disjunctive diagram, it is possible to construct a CNF equivalent to the original one, but containing a much larger number of constraints - clauses. Modern SAT solvers based on the CDCL algorithm [9], [10] (conflict-driven clause learning) are supposed to show better SAT solving speed for CNFs obtained by preprocessing using disjunctive diagrams than for the original CNFs.

The following is a summary of the main results of the article. The next section will contain the basic definitions and results necessary for understanding the rest of the paper.

$$D = (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4) \vee (\neg x_1 \wedge x_2 \wedge x_4) \vee (\neg x_1 \wedge \neg x_4) \vee (x_2 \wedge \neg x_3) \vee (x_3 \wedge x_4)$$

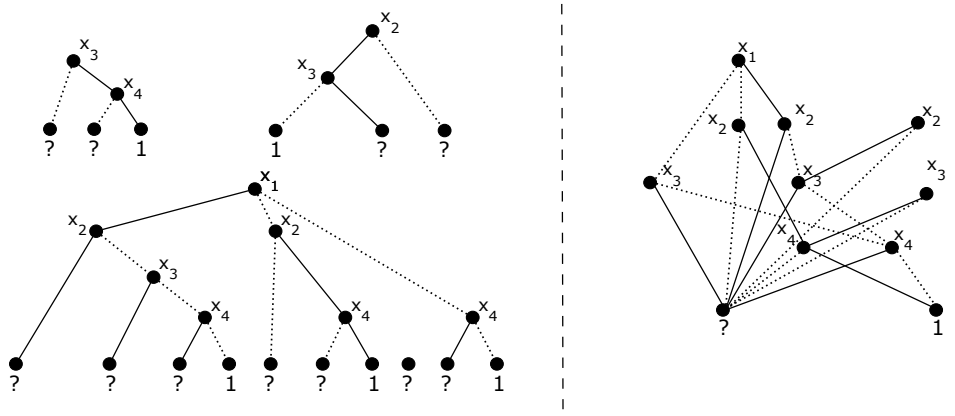

Fig. 1. DNF D with Disjunctive forest $F(D)$ (left) and DJD $R(D)$ (right) for it.

Section 3 gives a brief description of the class of diagrams presented in [7]. These diagrams provide a compact representation of arbitrary DNFs. Section 4 describes algorithms that allow using a combination consisting of disjunctive diagrams and a SAT oracle to preprocess arbitrary CNFs. These algorithms are implemented as the PyDJD program, which takes a Boolean formula in CNF or DNF format as an input and constructs a disjunctive diagram from it. Then, if necessary, the PyDJD program is able to use a SAT oracle to analyze and optimize the constructed diagram and obtain a new CNF. The resulting preprocessed CNF will be equivalent to the original one, but will contain more constraints useful for SAT solvers. The paper then conducts an extensive computational experiment aimed at comparing the solving times of SAT problems for CNFs before and after preprocessing. As the test instances the benchmarks from SAT Competition 2021 as well as some tests related to the task of checking the equivalence of Boolean schemes were used. It turned out that, if we consider the whole range of test instances, preprocessing does not show significant efficiency and speeds up the solution time about as often as it slows it down. However, for the most complex tasks (which are solved in more than 1000 seconds), preprocessing yields substantially more impressive results, significantly speeding up the CNF solving time in more than half of the cases. The last section briefly summarizes the results of the performed research and experiments, and outlines some directions for further work.

II. Preliminaries

Boolean variables are variables that take values from the set $\{0, 1\}$. An arbitrary variable x or its negation $\neg x$ are called literals. A disjunction of literals is called a clause, and a conjunction of clauses is a Boolean formula in conjunctive normal form. Let F be an arbitrary Boolean formula represent a Boolean function f over a set of Boolean variables $X = \{x_1, \ldots, x_n\}$. If there exists an assignment $\alpha \in \{0, 1\}^n$ of variables from X such that $f_F(\alpha) = 1$, then the formula is called satisfiable, and the assignment α is called the satisfying assignment. If there is no such assignment, then the formula is called unsatisfiable. Proceeding from the above, the Boolean satisfiability problem is as follows: given a Boolean formula (usually in CNF format), we need to determine whether the formula is satisfiable or not.

Disjunctive diagrams were first introduced in [7] as a way to compactly represent arbitrary disjunctive normal forms of large dimensions [11]. As in many other kinds of decision diagrams, each nonterminal vertex V in a disjunctive diagram is associated with variable x_V. In the same way, from each nonterminal vertex two kinds of edges can go down: solid (1-edges) and dashed (0-edges). The edges from vertex V_x are interpreted so that when considering the path through solid edge, the variable x takes value 1 and through dashed edge the variable x takes value 0. However, as discussed above, DJD also has significant differences from the more familiar types of decision diagrams. First, DJDs are not representations of a Boolean function, but of a Boolean formula, so they can be constructed efficiently in the general case. And second, DJDs are not binary: any vertex, except the two terminal ones, can have up to $2n - 2$ descendants, where n is the number of variables in the DNF under consideration.

Let D be a DNF which is a negation of some CNF C. The construction of DJD $R(D)$ representing D is divided into two stages. At the first stage a disjunctive forest is constructed, in which each elementary conjunction from D will be represented by a path from one of the roots to the terminal vertex marked with symbol "1". This symbol means that assigning values to variables corresponding to the path to a given vertex satisfies the DNF D and, consequently, makes CNF C unsatisfiable. In case a vertex has no descendants by any type of edges (solid or dashed), it is added a terminal vertex marked by "?" symbol as its descendant by such edge. This symbol, in its turn, means that the result of assigning values to variables corresponding to the path to this vertex is unknown. After constructing the disjunctive forest, all vertices with identical descendants are glued. The result is a disjunctive diagram.

Figure 1 shows DNF and the corresponding disjunctive forest (left) and the disjunctive diagram (right) constructed with the order $x_1 < x_2 < x_3 < x_4$.

III. Preprocessing CNFs using Disjunctive Diagrams

In this section we describe a new algorithm for preprocessing CNFs based on disjunctive diagrams.

The goal of the preprocessing is to make a CNF better in some way, for example to reduce its size or to improve the runtime of modern SAT solvers on it. One of the most important works on the preprocessing of CNFs is the well-known article [12], which describes the SatELite preprocessor. In it, preprocessing is aimed specifically at reducing the size of the original formula through various algorithms. This study proposes another direction: improving the CNF by adding new constraints. This approach is justified by the fact that the only deduction rule in modern CDCL SAT solvers is Unit Propagation [13], whose efficiency can be significantly increased by adding new clauses to the formula. The following will describe the algorithm for preprocessing of CNFs using disjunctive diagrams.

Let there be an arbitrary CNF C. First of all, we construct its negation, thereby obtaining the DNF D. By D we can construct a disjunctive diagram $R(D)$. In the diagram $R(D)$ any path from one of the roots to terminal "1" gives a partial assignment of variables which turns C into 0, and for any path to terminal "?" the value of the function represented by C is not known. For paths to terminal "?", however, the satisfiability of C on the corresponding set can be checked using the SAT oracle. To check a path using the SAT oracle, do the following:

1) Output the path from the diagram as a set of literals $\{x_i, x_j, ..., x_k\}$;
2) Run the SAT oracle on the formula

$$C \wedge x_i \wedge x_j \wedge ... \wedge x_k \tag{1}$$

It is worth noting that formulas of the form (1) are often very complex, so the SAT-oracle is run with a time limit. If the SAT oracle proves that a formula of the form (1) is unsatisfiable, we can redirect this path in the diagram from the terminal vertex "?", to the terminal vertex "1", since now we know the function value at the given assignment, without breaking the structure of the diagram. There is also a chance that the SAT oracle will find a satisfying assignment for formula (1) . Such an satisfying assignment will also be the satisfying assignment for the initial CNF, i.e. the problem will be solved at the preprocessing stage. However, in practice, such situations are possible only for simple formulas for which the preprocessing is not required.

If for a number of paths to "?" the SAT oracle did not give an answer in an acceptable (usually very short) time, we can make a reverse transition from the existing diagram to CNF C*. This requires to take all paths in the diagram from the root vertices to the terminal vertex "1" as sets of literals, thereby obtaining DNF D*. The CNF C*, which is the negation of the DNF D*, will be the result of preprocessing. If the SAT oracle did not find an assignment that satisfies C at the previous step, then C is satisfiable if and only if C* is satisfiable. This follows from the fact that all new clauses that distinguish C* from C are obtained from paths redirected during preprocessing from "?" to "1". Each such path is a set of literals $\{x_i, x_j, ..., x_k\}$ for which $C \wedge x_i \wedge x_j \wedge ... \wedge x_k = 0$. It implies that CNF C is equivalent to CNF $C \wedge (x_i \vee x_j \vee ... \vee x_k)$.

The following is a pseudocode of the described algorithm:

for each $path \in GetAllQuestionPaths(diagram)$ **do**
    $literals \leftarrow GetLiteralsFromPath(path)$
    $result, model \leftarrow SATsolver(initial\_formula, literals)$
    if $result == False$ **then**
        $RedirectPathFromQuestionToOne(path)$
    else if $result == True$ **then**
        return model
    else if $result == None$ **then**
        continue
    end if
end for

Thus, the described transition procedure $C \rightarrow D \rightarrow R(D) \rightarrow D^* \rightarrow C^*$ can be considered as a preprocessing based on the properties of the disjunctive diagram.

IV. Computational experiments

This paper examines the SAT instances used in "SAT Competition 2021", and the SAT instances encoding Boolean circuit equivalence checking problems. Each CNF was given as an input to the PyDJD [14] program, which implements an algorithm for constructing disjunctive diagrams by an arbitrary CNF and an algorithm for preprocessing CNFs using a SAT oracle. The PySAT program [15] was used as the SAT oracle, which allows us to incremental check certain values of variables in CNFs. The solver which was used in PySAT was the MapleLCMDistChronoBT SAT solver [16], with a timeout of 0.01 seconds to check one path to "?". The PyDJD program results in CNFs equivalent to the original ones, but containing additional information that can help to solve the corresponding SAT problems.

As tests, we selected problems for which the MapleLCMDistChronoBT SAT solver takes at least 1000 seconds. This is due to the fact that for most of the considered problems the preprocessing takes at least 200 seconds, due to the size of the initial CNFs (the number of paths in "?" checked during preprocessing depends on the size of the initial formula). As a consequence, the application of this method of CNF preprocessing to low-complexity problems does not seem expedient.

A. Tests related to the problem of checking the equivalence of Boolean circuits

The first group of computational experiments used CNFs corresponding to the topic of "Equivalence Checking" [17] [18], specifically encoding problems of equivalence of sorting algorithms.

The idea of such problems can be described as follows. We consider two different sorting algorithms that sort d arbitrary

TABLE I
COMPARISON OF SOLVING TIME FOR CNF BEFORE AND AFTER PREPROCESSING. EQUIVALENCE CHECKING.

Test name	Solve time before preprocessing (s.)	Solve time after preprocessing (s.)
PancakeVsBubble		
7_6	**1316,98**	1477,607662
7_7	**2458,48**	2917,035374
7_8	**3097,39**	3758,226035
8_4	2828,6	**2609,92778**
8_5	25325,6	**18335,82424**
9_3	**1408,31**	1780,599204
PancakeVsSelection		
7_6	1401,16	**1383,325554**
7_7	**2246,82**	3233,119807
7_8	3771,18	**3557,880461**
8_4	2922,97	**2517,607242**
8_5	**16781,5**	24091,96743
9_3	1553,69	**1525,458104**
PancakeVsInsert		
7_6	**1838,97**	2671,226513
8_4	**22081**	22720,89599
9_3	19184,4	**16522,83834**

l-bit numbers. These algorithms are given by Boolean circuits, and it is required to prove that the obtained circuits are equivalent. The problem is reduced to SAT using the software tool Transalg [19].

The following pairs of sorting algorithms were considered in this study:

1) Pancake sorting [20] and Bubble sorting (PancakeVsBubble);
2) Pancake sorting and Selection sorting [21] (PancakeVsSelection);
3) Pancake sorting and Insert sorting [22] (PancakeVsInsert).

This class of tests is very well parameterizable and allows to obtain instances with different complexity. The results of the first stage of experiments are shown in Table I.

Below are some comments to Table I. The first column of the table contains the parameters of the CNF under consideration. The first number means the number of sorted bit words, and the second number means their length. The second column of the table specifies the time to solve the original CNF, before preprocessing. The third column of the table indicates the total time of preprocessing and the time of solving the CNF resulting from preprocessing. The table does not include tests that were not solved in 24 hours (86400 seconds) either before or after preprocessing.

This stage of the experiment showed the following results: out of the 15 problems selected for the experiment, only 7 problems were solved faster after preprocessing, the average change in the solving speed was $\approx 15\%$ towards the slowdown. At the same time, if we consider the most complex problems with a solving time > 15000 seconds, the results are not so bad, 2 tests showed an acceleration, 2 tests slowed down, the average change in the solution speed was $\approx 1\%$ in the direction of acceleration.

TABLE II
COMPARISON OF SOLVING TIME FOR CNF BEFORE AND AFTER PREPROCESSING. SAT COMPETITION 2021.

Test name	Solve time before preprocessing (s.)	Solve time after preprocessing (s.)
ASG_72_keystream76_1	3039,31	**787,1908561**
Circuit_multiplier26	3945,06	**2788,741418**
Circuit_multiplier29	33615,4	**834,9904189**
Circuit_multiplier45	**1597,88**	7223,025944
Circuit_multiplier53	35174,2	**2412,75471**
ak016modbtsimpbisc	**3662,89**	3792,163886
ak032modbtmodbtisc	**3992,56**	4582,399526
LABS_n041_goal003	**2347,92**	3420,597748
LED_round_1-32_ faultAt_30_fault _injections_2_seed_ 1579630418	> 86400	**10981,89762**
PRESENT_round_1-32_ faultAt_29_fault _injections_3_seed_ 1579630418	**7725,38**	15250,49428
edit_distance007_85	**1282,6**	1580,576448
erin2_0x1e3216	1924,48	**1648,97992**
hitag2-10-60-0-0xdf7fa6426edec07-17	**1340,17**	1468,829492
hitag2-7-60-0-0xe8fa35372ed37e2-80	3454,71	**2752,672507**
hitag2-7-60-0-0xe97b5f1bee04d70-47	2381,05	**2189,049442**
hitag2-8-60-0-0x880693399044612-25-SAT	13996	**4140,704456**
ktf_TF-4tf_4_004_35	**2416,44**	3738,811934
ktf_TF-4tf_4_006_52	> 86400	**26472,0904**
mp1-Nb6T07	**9401,8**	25494,58524
randomG-Mix-n17-d05	13730,5	**12474,58774**
rphp4_080_shuffled	**2208,38**	2443,670938
satch2ways14u	**1433,56**	1809,270507
sha1r17m149ABCD_p	2479,06	**628,0078286**
sp5-26-19-bin-nons-tree-noid	26131,1	**16873,53247**
sp5-26-19-una-nons-tree-noid	**2945,19**	4688,296983
sted12a_0x1e1-67	47562,4	**5281,334628**

B. Tests from SAT Competition 2021

After the unsatisfactory results of the first stage of experiments, it was decided to choose more diverse problems. For this purpose, instances from the annual SAT solvers competition "SAT Competition 2021" were selected. These problems involve a variety of areas, such as verification, combinatorics, cryptanalysis, and others. Among the 400 problems from the competition, 26 were chosen for the following reasons:

1) The problem should be solved in less than 86400 seconds (before or after preprocessing), to filter out too complicated problems;
2) The time of solving a problem before preprocessing by MapleLCMDistChronoBT should be more than 1000 seconds, to filter out too simple problems for the reasons mentioned above;
3) The size of the formula must be less than 200000 clauses. This condition has been motivated by the fact that the greater the size of the formula, the longer preprocessing takes, and when the number of clauses in the formula is greater than 200000, preprocessing may well take several thousand seconds, which often leads to the loss of benefits from the preprocessing itself.

The results of the second stage of experiments are shown in Table II, which has the same structure as Table I.

The results of the second stage of experiments are much more interesting. In 14 out of 26 tests it was possible to achieve acceleration of CNF solving after using preprocessing. It is worth noting that despite the close to 50% result in the context of the number of accelerated problems, if we consider the degree of acceleration, the result is much more impressive. The average change in the solving speed was $\approx 280\%$ towards acceleration, which is a very good achievement.

V. Conclusion and future work

This paper presents an algorithm for preprocessing arbitrary CNFs using disjunctive diagrams which were introduced in [7] for the first time. By constructing a disjunctive diagram by the CNF and analyzing the information contained in it, it is possible to obtain new clauses, which can significantly speed up the solution of the considered CNF. According to the results of the experiments, it was found that on a sample of medium and high complexity problems from SAT Competition 2021, preprocessing allowed to achieve acceleration of the solving of the considered CNFs in more than half of the cases, and in those cases where the solving was accelerated, it was much more significant than in those where it was slowed down.

In the future it is planned to specify more specifically the classes of tests in which preprocessing gives good results. In addition, it is planned to continue developing algorithms that analyze and optimize disjunctive diagrams, as well as their interaction with modern SAT solvers.

Acknowledgments

The research was supported by the Russian Science Foundation, project № 22-21-00583.

References

[1] I. Lynce and J. Marques-Silva, "SAT in Bioinformatics: Making the Case with Haplotype Inference," in *Theory and Applications of Satisfiability Testing - SAT 2006*, A. Biere and C. P. Gomes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 136–141.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT Press, 1999.

[3] M. N. Velev and R. E. Bryant, "Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors," *Journal of Symbolic Computation*, vol. 35, no. 2, pp. 73–106, 2003.

[4] C. Chin-Liang, C. Chang, J. Zhang, R. Lee, and C. coaut, *Symbolic Logic and Mechanical Theorem Proving*, ser. Computer science and applied mathematics : a series of monographs and textbooks. Elsevier Science, 1973.

[5] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, p. 677–691, Aug. 1986.

[6] C. E. Shannon, "The Synthesis of Two-Terminal Switching Circuits," *Bell System Technical Journal*, vol. 28, no. 1, pp. 59–98, 1949.

[7] A. A. Semenov and I. V. Otpuschennikov, "On one class of decision diagrams," *Automation and Remote Control*, vol. 77, no. 4, pp. 617–628, 2016.

[8] S.-I. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," in *Proceedings of DAC '93*. ACM, 1993, p. 272–277.

[9] J. P. Marques-Silva and K. A. Sakallah, "GRASP: a New Search Algorithm for Satisfiability," in *Proceedings of ICCAD '96*, 1996, pp. 220–227.

[10] J. P. Marques-Silva and K. A. Sakallah, "GRASP: a search algorithm for propositional satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[11] V. Kondratiev, I. Otpuschennikov, and A. Semenov, "Using Decision Diagrams of Special Kind for Compactification of Conflict Data Bases Generated by CDCL SAT Solvers," in *43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, 2020, pp. 1046–1051.

[12] N. Eén and A. Biere, "Effective Preprocessing in SAT Through Variable and Clause Elimination," in *Theory and Applications of Satisfiability Testing*, F. Bacchus and T. Walsh, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 61–75.

[13] W. F. Dowling and J. H. Gallier, "Linear-time algorithms for testing the satisfiability of propositional horn formulae," *The Journal of Logic Programming*, vol. 1, no. 3, pp. 267–284, 1984. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0743106684900141

[14] "PyDJD. A program for building and analyzing disjunctive diagrams." https://github.com/Vikseko/PyDJD/, accessed: 2022-02-07.

[15] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *SAT*, 2018, pp. 428–437.

[16] A. Nadel and V. Ryvchin, "Chronological Backtracking," in *SAT*, ser. LNCS, vol. 10929, 2018, pp. 111–121.

[17] A. Kuehlmann and F. Krohm, "Equivalence Checking Using Cuts and Heaps," in *Proceedings of the 34th Annual Design Automation Conference*, ser. DAC '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 263–268. [Online]. Available: https://doi.org/10.1145/266021.266090

[18] P. Molitor and J. Mohnke, *Equivalence Checking of Digital Circuits: Fundamentals, Principles, Methods*. Springer, 2007.

[19] I. Otpuschennikov, A. Semenov, I. Gribanova, O. Zaikin, and S. Kochemazov, "Encoding Cryptographic Functions to SAT Using TRANSALG System," in *ECAI 2016, Frontiers of Arificial Intelligence an Applications*, vol. 285, 2016, pp. 1594–1595.

[20] W. H. Gates and C. H. Papadimitriou, "Bounds for sorting by prefix reversal," *Discrete Mathematics*, vol. 27, no. 1, pp. 47–57, 1979. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0012365X79900682

[21] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, ser. The MIT electrical engineering and computer science series. MIT Press, 1990.

[22] D. E. Knuth, "5.2.1: Sorting by Insertion," in *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998, p. 80–105.