## Introduction

This part deems to introduce the topic in itself and mentions what the papers describe.

The Boolean Satisfiability Problem, also known as Satisfiability or SAT, is the problem of deciding whether there exists an assignment that satisfies a given CNF formula. The main goal is to systematically assign variables values, either TRUE or FALSE, such that the formula evaluates to TRUE.

TODO: Language of SAT introduce here

In order to understand what the tractable cases concretely refer to; some preliminary information is required.

## Preliminaries

### Boolean Variables and Propositional Logic

Given a Boolean Formula composed of variables and logical connectors, satisfiability in its core asks the question whether there exists an assignment of truth values to the variables such that the formula evaluates to true. To build from the bottom towards the top: we let be a set of variables. Propositional formulas over are defined inductively, where constants and , each variable are formulas. If is a formula, the so is , the negation of . If and are formulas, then so are their conjunction and their disjunction . denotes the set of variables occurring in . represents the size of . Such that:



An assignment for is a finite partial map written as where and . The value of is computed by replacing every by and the simplify according to these rewrite rules:

Figure 1: An exemplary formula in CNF.

Assignment is defined to be total, if , otherwise partial. is said to satisfy , written as , if under this assignment evaluates to true, . is defined to be satisfiable, if for some , and as a tautology, if for all total .

### CNF format

When speaking of satisfiability in general, a formula refers to a formula in Conjunctive Normal Form (CNF). A literal under CNF is a variable or its negation. A clause is a disjunction of literals, for example and a formula is a conjunction of clauses .

The width of a clause is to the number of elements within, therefore the width of clause is . A formula is in if every clause in has width of at most , . With this information in mind, a clause is identified by the set and a CNF formula is identified by the set Throughout this work we will be using to refer to the number of variables in , the number of clauses in , and for the width of .

To further specify the requirement for a CNF formula to be satisfied, if there is a variable for every clause with . In other words, every clause in has at least one variable that evaluates to true.

A unit clause and a pure literal are two special properties that can occur in CNF formulas. Unit clauses are clauses where . Therefore, the literal within the clause needs to evaluate to true for the formula to be satisfiable. A literal is pure in a formula , if the variable occurs only in one polarity. In other words,   does not occur in the formula.

### 2.3 DIMACS Format

Aiming to uniformly represent CNF Formulas, the Center of Discrete Mathematics and Computer Science (DIMACS) at Rutgers University created the DIMACS format for representation of CNF formulas at the 1993 DIMACS challenge. (Johnson & Trick, 1996) The format consisting of a preamble and a body became the norm for the field ever since. (Prestwich, 2009)

The syntax of DIMACS format is as follows. Firstly, a preamble is available containing information regarding the formula. The option to write comments is also available through the usage of the letter ‘c’ at the beginning of a line. Then a line starting with ‘p’ portrays the number of variables and clauses within the formula. Both the variables and the clauses are positive integers. The remainder of the body contains the clauses. Every clause contains a list of non-zero integers, as zero is reserved as a terminator for the clause. The integers representing the variables are numbered from 1 to . Furthermore, the integers must be separated by spaces, tabs, or newlines. In order to represent negative variables a before the variable itself is satisfactory. Finally, the order of the variables within the clauses, and the clauses themselves are not of importance. A clause may even stretch over multiple lines. An example can be found in Figure 2 and 3.

c Pigeonhole principle formula for 3 pigeons and 2 holes

c Generated with `cnfgen` (C) Massimo Lauria <lauria@kth.se>

c https://github.com/MassimoLauria/cnfgen.git

c

p cnf 6 9

1 2 0

3 4 0

5 6 0

-1 -3 0

-1 -5 0

-3 -5 0

-2 -4 0

-2 -6 0

-4 -6 0

Figure 2: An unsatisfiable CNF formula for the pigeonhole principle with 3 pigeons and 2 holes in DIMACS format by cnfgen. The formula consists of 6 variables and 9 clauses.

Figure 3: The same CNF formula found in Figure 2 represented with propositional logic.

### 2.4 Genesis – DPLL

Named after Davis, Putnam, Logemann, and Loveland, the DPLL algorithm is the backbone of many known SAT Solvers, such as but not limited to, zChaff and MiniSat. The algorithm picks branching variables with the aid of backtracking in the case of a conflict. Moreover, DPLL is known to be complete and sound, meaning that it only delivers a solution if and only if the formula is satisfiable. (Prasad, Biere, & Gupta, 2005)

The algorithm works as follows: after simplification steps, an unassigned variable is picked. This variable is assigned either true or false. Here the search tree is branched. If a logical conflict occurs, then the algorithm backtracks and reverts the actions done up until the branching variable. Afterwards the opposite value of the original assignment is done. The algorithm terminates with an assignment if all clauses are satisfied. Otherwise, if both assignments of the initial variable results in a conflict, returns UNSAT. DPLL is a depth-first search where the worst-case runtime is of , which corresponds to the tree size of variables and two choices per node. A pseudo-code for the general DPLL algorithm can be found below.

* General DPLL Algorithm

DPLL(

simplify()

if then return UNSAT

if then return

pick and

DPLL()

if

then return

else return DPLL()

The simplify()function refers to simplification processes that are utilized in order to reduce the number of decisions that need to be taken by the DPLL algorithm and to shorten the formula. Among others, these may include unit propagation (UP), pure literal elimination (PLE), and subsumption.

UP is a branching strategy in which if a variable occurs in a unit clause , then this variable will be picked. If the variable occurs positively, then that clause can be discarded after its assignment, as it is already satisfied. If the variable occurs negatively, then that literal can be discarded after its assignment for that clause, as the satisfiability of the clause is not dependent on it anymore. PLE, similar to UP, is the strategy of picking the variable that is pure in . The aim here is assign the variable in such a way that the clauses it appears in evaluate to true. The pseudo-codes for these strategies can be found below.

* UP Algorithm

UnitProp(

while contains unit clause

return

* PLE Algorithm

PureLit(

while contains pure literal

## Literature Review

What has been said about each paper

Survey about backdoor sets Stefan Szeider – TU Wien Marco something

Backdoor for nested

Handbook of Sat about nestedness

## Tractable Cases of SAT and Their Algorithms

#### 2 Sat

2-SAT is the class of formulas in which the width of the formula is equal to at most two, i.e. for every in is true. 2-SAT is known to be one of the trivial tractable cases. (Schaefer, 1978) There exists multiple ways to solve 2-SAT, such as but not limited to, graph-based, random walk, resolution, and unit propagation-based approaches. (Dantsin & Hirsch, 2009) We will be introducing the graph-based algorithm and our version based on DPLL.

In their 1979 article, Aspvall, Plass, and Tarjan devised a linear-time algorithm for this case. Their approach aims to create a directed graph for the formula . contains vertices, representing the variables in and their negations. The edges of are created by utilizing the Implication Law of De Morgan, where the clause is equivalent to   and . The direction of the edges follows the direction of the implication. If there is a unit clause present, then is added as an edge. The algorithm checks the strongly connected components of for cycles that include a variable and its negation at the same time. If and only if that is the case, then is unsatisfiable. For the worst-case upper bound, the algorithm requires where is the number of variables and is the number of edges. (Aspvall, Plass, & Tarjan, 1979)

We have chosen to utilize a different approach than the graph-based algorithm. Our algorithm bases itself on DPLL and uses unit propagation to work through the formula. Starting with an arbitrary variable choice and the assignment of the value, it looks for new unit clauses and stores them in a queue. The disposal of clauses in the case of a positive occurrence and the disposal of literals in the case of a negative occurrence is implemented. These new unit clauses found through the negative polarity literals disposal, will be propagated in the order they were found. These clauses are also seen as forced, as they were created through the assignment of another variable. In the case that there is a logical conflict, we revert the assignments and restore the clauses and assign the opposite value to the variable. If the opposite value for a variable also leads to a conflict, then we terminate and return UNSAT. If no conflict is found, the algorithm carries on assigning autark[[1]](#footnote-1) assignments. With out algorithm we exploit the width of 2-SAT formulas since each assignment of a variable is certain to create a unit clause. Therefore, in the worst case, one full run containing all the variables, , will be satisfactory to determine whether a formula is satisfiable or not. Furthermore, the algorithm either returns a fulfilling assignment or UNSAT. The flowchart in Figure 4 can be seen as a graphical representation of our algorithm. It should be noted that our algorithm carries out the same procedure without the explicit construction of the graph explained above.

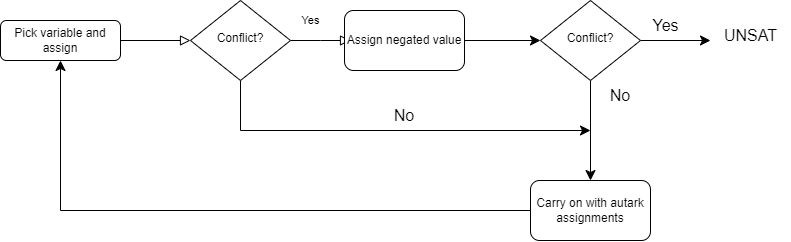


Figure 4: Graphical Flowchart Representation of our 2 SAT algorithm

It should be noted that during parsing if we find a unit clause, we store that value in a queue. The chosen branching strategy is incremental; however, it only comes into effect if there are no other elements remaining in the queue. Therefore, our algorithm assigns more importance to original unit clauses and the unit clauses created by extension.

#### Horn Formulas

One of the classes of the tractable cases of SAT is Horn Satisfiability, also called HORNSAT. The name stems from Alfred Horn, who in a 1951 article pointed out their importance. Unlike general SAT, HORNSAT defines restrictions regarding the polarity of variables in clauses and how the general structure of the formula is built up. A Horn formula is a conjunction of Horn clauses, which contain at most one positive literal. (Downing & Gallier, 1984)

Figure 5: An exemplary Horn formula

NOTE if space left: Would a paragraph about connection to logic programming etc. be beneficial.

The satisfiability of positive Horn formulas can be tested as follows. If contains positive unit clauses, then apply unit clause elimination until no unit clauses are left. Therefore, assigning all positive unit literals the value true and all negative unit literals the value false. If the resulting formula does contain the empty clause, then is unsatisfiable. Otherwise, the clauses within all must contain at least two literals and at least one of the clauses has to be negative. Consequently, is satisfiable by assigning false to all the remaining literals. By extension, is satisfiable as well. One can trivially see that this algorithm is polynomial, to be exact, where represents the number of variables and the number of clauses. (Dantsin & Hirsch, 2009)

We have tweaked the algorithm for the sake of simplicity and readability. In our version the algorithm works as follows. Starting with the empty assignment , iteratively assign all positive unit clauses to true, and then assign the remaining variables to false. If this assignment satisfies then return . Otherwise return UNSAT. Such that, the pseudo-code for our algorithm is as follows:

* HORN ALGORITHM

while positive unit clause in

if

then return

else return UNSAT

Our algorithm, alike its counterpart explained earlier, has a worse case upper bound of . However, instead of doing complete unit propagation, we have chosen to only propagate the positive unit literals. This, combined other optimization approaches, such as marking unit clauses during parsing, can lead to improved performance.

### Nested Satisfiability

Knuth in his 1990 paper titled “Nested Satisfiability” explores a special case of the satisfiability problem where if a formula has a specific hierarchical structure, that formula gets transformed into Dynamic 2 SAT and therefore is solvable in linear time. He acknowledges the work of Lichtenstein where it was proved, that the joint satisfiability problem of two sets of nested clauses is NP-complete. (Lichtenstein, 1982). However, his exploration shows that the mirrored question for nested clauses is efficiently decidable. (Knuth, 1990).

In order to achieve this, a linear order through is defined, where is a finite alphabet of Boolean variables. Here literals over are elements of the form or where . Literals belonging to are called positive and negative otherwise.

To refine this linear order, linear preordering is introduced where all the literals are arranged in a “natural way”, disregarding the signs. As an example, if , then . If and are literals, then the relational operation can only be true, if or is true and the relation is false.

A clause over is defined as a set of literals on distinct variables, such that the clause can be written in increasing order . The set of clauses over is satisfiable, if there exists a clause over that has a nonempty intersection with every clause in . For example: the clauses in :

are satisfiable by the clause , which has a nonempty intersection with each clause.

A clause straddles , if there are literals and in and in such that Two clauses overlap if they straddle each other. For example, for and :

straddles , since: there exists and and , such that .

straddles , since: there exists and and , such that .

Therefore, and overlap. Clauses that have only 2 elements each, in other words are E2-CNF, such as and can be overlapping. A set of clauses in which no two overlap is defined to be *nested*.

Regarding the structure of these clauses and the way they are represented further refinements are made. A clause over an ordered alphabet has a least literal and a greatest literal . (Knuth, 1990) Any other variable that lies strictly between these literals is defined to be interior to that clause. Since the definition of nestedness sets out the condition for the clauses to not be overlapping, a literal can occur as an interior literal on at most one of the clauses in the set of clauses, otherwise those clauses would be overlapping. This property forces the number of total elements among nested clauses on variables to be . In detail: 1 least and 1 greatest literal per clause and then all variables occur once as an interior literal, therefore .

Furthermore, a transitive relational operation is introduced where represents that straddles but does not straddle . The transitive property of this relation makes it possible to topologically sort any set of nested clauses into a linear arrangement where each clause appears after the clause it straddles. With such an arrangement and the elements presented in order, Knuth defines an algorithm that decides in steps where is the number of clauses and is the number of variables.

#### Nested SAT Algorithm

The presented algorithm determines whether a given nested formula is satisfiable or not using a table to keep track of the satisfiability conditions across intervals of literals. It partitions the literals into intervals, which represent clauses, and updates the sat table to reflect whether the processed clauses can be satisfied based on the literal’s polarity.

The alphabet is assumed to represent the positive integers up to , where each variable is deemed as equivalent to its negation, . The clauses are then specified in two arrays: one where all the literals are present called , and one called signifying the starting positions of a given clause within such that:

and

where the literals of clause are between

in increasing order as increases. The clauses are ordered in such a way that the clause does not straddle clause when . It is assumed that the clauses contain at least two literals.

As the algorithm progresses through each clause, it conceptually divides the set of all encountered clauses so far into intervals . This ensures that all literals of any previously processed clause are in one of these defined intervals. The current intervals are kept in an array, ,where for .

Essentially the heart of the algorithm, the array is interpreted as follows: If is an interval of the current partition, then [] will either be 0 or 1 for each pair . It is 1 if and only if the clauses already processed, belonging to interval , are satisfiable by clauses in which the least and greatest literals are respectively and , where:

As an example, assume we have only seen the clause . Then the sat table looks as follows

The main task of the algorithm is to maintain the array as a new clause is examined. The variables of , belong to the current partition of the variables. Assume . The algorithm runs a variable through the values and keeps the information to update where the interior variables within are eliminated within the partition. We further let be the literals element of that are strictly less than and let be the clauses that are preceding whose literals are limited by . Then an auxiliary table named , with the help of a function, updates the values as follows:

Concretely, means that there is a clause with and that has a nonempty intersection with each clause of the given set of clauses.

For example, suppose and . Furthermore, suppose that clauses have led to the following values:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  | 0 | 0 | 0 |
|  |  | 1 | 1 | 0 |
|  |  | 1 | 1 | 0 |
|  |  | 1 | 0 | 1 |

Table 1: table for the example

Then the table looks as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  | 1 | 0 | 2 | 0 |
|  |  | 0 | 2 | 0 | 0 |
|  |  | 0 | 1 | 2 | 0 |
|  |  | 1 | 1 | 1 | 1 |

Table 3: newsat table for the example

Table 3: Updated sat table after newsat calculation.

In this case, a 1 is the current , the values for will be updated with those in Table 3 and will be assigned 4 as interior variables will be discarded.

If the clause were instead, the computation of the table would have been the same, however the values of would all be 0 and the clauses would be unsatisfiable since is equal to 1 and not 2.

The algorithm itself starts with the setup of the array and initializes all values to 1. Then the array is initialized as explained and partitions the formula into its clauses. For each clause, the literals that are in the partition are determined and the table is computed. Depending on whether the new clauses allow a satisfiable combination of the literals in the current partition, table is used to update the array. Here only the least and greatest literals are considered as interior literals are removed from the partition. After iterating through all clauses, the algorithm checks the final status of the table and determines whether the entire set of clauses is satisfiable and outputs SAT or UNSAT accordingly.

The runtime of the algorithm, as explained above, is since each value of is either the first or the last in the current clause or will be permanently removed from the partition.

Although the algorithm provides a viable solution to the problem, its assumptions and the lack of variable assignments in the output present limitations. It is possible to deduce the variable assignments through an analysis of the table after the algorithm delivers a result, however, this would involve a similar approach to DPLL as one must check four values for each clause. During this process logical conflicts can occur, which requires backtracking to be implemented alongside the usual assignment structures. Therefore, either external help from other solvers or an effort to implement a DPLL-like algorithm is required.

Furthermore, as it is also noted by the authors that whether the formula is nested in some ordering of its variables and clauses is not considered. Firstly, that process requires that the variables in each clause to be ordered by the hierarchy. Depending on what kind of ordering algorithm is used, this will require some overhead. Moreover, the clauses themselves are assumed to be ordered in such a way that the clause does not straddle if , again leading to an overhead. Finally, the nestedness condition, i.e. no overlapping clauses, requires that each clause is checked against all the other clauses. Such a comparison requires pairwise checking, which has a time complexity of .

In summary, this algorithm answers the question, whether a given nested formula is satisfiable or not with a worse case upper bound of Although it must be noted that the assumptions made and the prerequisite that a formula to be nested does require further checks and structures to be built.

### Co-nested Formulas

In Kratochvil and Krivanek’s work titled “Satisfiability of co-nested formulas” they introduce a graph-based approach to define Knuth’s nestedness term and define new types of nestedness, specifically co-nestedness, double nestedness and double co-nestedness. Their work assumes the usual prerequisites for CNF formulas adapted for SAT, such that is a formula with a set of clauses over a set of variables . (Kratochvil & Krivanek, 1993)

The main extension for Knuth’s work begins with the definition of a so-called clause linked graph of where . The redefinition of Knuth’s nested formula is as follows: is nested if the *variables* can be ordered in a way where for the graph allows a noncrossing drawing in the plane so that the circle of variables bounds the outer face. The definition of co-nestedness is made in a similar way where the clause linked graph allows a noncrossing drawing in the plane such that the clauses bounds the outer face. (Kratochvil & Krivanek, 1993) The authors define a recursive algorithm which computes the maximum number of satisfiable clauses in a given co-nested formula. The runtime of this algorithm is set to be linear in the number of clauses , added with the number of variables . This algorithm will be analyzed in later chapters.

Furthermore, the notion of double co-nestedness is introduced. is double co-nested if is planar, i.e. the double co-nested formula can be splitted into two co-nested formulas and such that for . (Kratochvil & Krivanek, 1993) The authors acknowledge that the satisfiability for double co-nested formulas is NP-complete, since Lichtenstein proved that even if every clause contains at most three variables, every variable occurs in exactly three clauses and the variables only occur once negatively and twice positively (Lichtenstein, 1982).

# Bibliography

Aspvall, B., Plass, M. F., & Tarjan, R. E. (1979). A LINEAR-TIME ALGORITHM FOR TESTING THE TRUTH OF CERTAIN QUANTIFIED BOOLEAN FORMULAS. *Information Processing Letters, Volume 8*, 121-123.

Biere, A., Heule, M., Van Maaren, H., & Walsh, T. (2021). *Handbook of Satisfiability.* Amsterdam: IOS Press.

Dantsin, E., & Hirsch, E. A. (2009). Worst-Case Upper Bounds. In A. Biere, M. Heule, H. van Maaren, & T. Walsh, *Handbook of Satisfiability* (pp. 403-424). Amsterdam : IOS Press.

Downing, W. F., & Gallier, J. H. (1984). Linear-time algorithms for testing the satisfiability of propositional horn formulae,. *The Journal of Logic Programming, Volume 1, Issue 3,*, 267-284.

Johnson, D., & Trick, M. (1996). Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge.

Kimura, K., & Makino, K. (2018). Autark assignments of Horn CNFs. *Japan Journal of Industrial and Applied Mathematics, Volume 35*, 297-309.

Knuth, D. E. (1990). Nested Satisfiability. *Acta Inform. 28 (1990), no. 1, 1--6*, 1-6.

Kratochvil, J., & Krivanek, M. (1993). Satisfiability of co-nested formulas. *Acta Informatica*, 397-403.

Lichtenstein, D. (1982). Planar Formulae and Their Uses. *SIAM journal on computing, 1982-05, Vol.11 (2)*, 329-343.

Prasad, M. R., Biere, A., & Gupta, A. (2005). A survey of recent advances in SAT-based formal verification. *International journal on software tools for technology transfer*, 156-172.

Prestwich, S. (2009). CNF Encodings. In A. Biere, M. Heule, H. van Maaren, & T. Walsh, *Handbook of Satisfiability* (pp. 75-97). Amsterdam: IOS Press.

Schaefer, T. (1978). The complexity of satisfiability problems. *In Proceedings of the 10th Annual ACM Symposium on Theory of Computing, STOC*, 216-226.

## Literature Review

* About 2 SAT
  + There is a graph based algorithm I do not use
* HORN
  + Other tractable horn cases -> dual horn, q-horn etc.
  + Minimum modal as a result of the algorithm
* Handbook of satisfiability talks about nested satisfiability and discusses limitations
* But not about non-interlaced and co-nested
* Horn Formulas
  + Conjecture of Horn Clauses

40k – 80k characters

1. An autark assignment is a partial assignment that satisfies a clause with a variable fixed by it. (Kimura & Makino, 2018) [↑](#footnote-ref-1)