**aa UNI OF MALTA LOGO**

**THESIS TITLE**

**Luca Alfino**

**Supervisor: <Supervisor Name>**

**Date – 2026/27?**

**Dissertation submitted to institute for particular degree**

**Authorship Statement**

**This dissertation is based on the results of research carried out by myself, is my own composition, and has not been previously presented for any other certified or uncertified qualification**

**This research was carried out under the supervision of <Supervisor Name>**

**DATE – SIGNATURE**

**Copyright Statement**

**In submitting this dissertation to the…**

**DATE - SIGNATURE**

**ACKNOWLEDGEMENTS**

**I would like to express my sincere gratitude to my mentor…**

**ABTSTRACT**

**Table Of Contents**

**Authorship Statement i**

**Copyright Statement ii**

**Acknowledgements iii**

**Abstract iv**

**List Of Figures v**

**List Of Tables vi**

**List Of Abbreviations vii**

**1 Introduction 1**

**1.1 <Title>**

**1.1.1**

**1.1.2**

**1.1.3**

**1.1.4**

**1.1.5**

**2 Theoretical Background**

**2.1 P vs NP**

**2.1.1 Polynomial Time**

**2.1.2 Non-deterministic Polynomial Time**

**2.1.3 NP-Complete**

**2.1.4 NP-Hard**

**2.1.5 P vs NP Conclusion**

**2.2 Boolean Satisfiability (SAT)**

**2.2.1 Conjunctive Normal Form (CNF)**

**2.2.2 3-SAT**

**2.2.3 SAT Conclusion**

**3 Literature Review**

**3.1 SAT Solvers**

**3.2 DPLL Algorithm**

**3.3 CDCL Algorithm**

**List Of Figures**

**Figure 2.1**: Time Complexity [3] . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . .. [num]

**Figure 2.2**: Complexity Classes [4]

**Figure 3.1**: DPLL-Recursive(F, ρ) [12]

**Figure 3.2**: CDCL Algorithm Pseudocode

**List Of Tables**

**Table 2.1**: Complexity Class Functions . . . . . . . . .. . . . . . . . . . . . . . [Num]

**List Of Abbreviations**

**P** Polynomial Time

**NP** Non-deterministic Polynomial Time

**TSP**  Travelling Salesman Problem

**SAT** Boolean Satisfiability

**UNSAT**  Unsatisfiable

**CNF** Conjunctive Normal Form

**DPLL** Davis-Putnam-Logemann-Loveland

**CDCL** Conflict-driven clause learning

**Theoretical Background**

**2.1 P vs NP**

The P vs NP problem is one of the seven “Millenium Problems”, which are a set of famous unsolved problems in mathematics, founded by Landon T. Clay in 1998 [1]. In principle, the P vs NP problem asks whether every problem whose solution can be quickly verified can also be quickly solved.

**2.1.1 Polynomial Time**

P(Polynomial Time) refers to the class of decision problems that a deterministic algorithm can solve in polynomial time – meaning that the time required to solve the problem grows at most as some fixed power of the input size [2].

A graph of different colored arrows

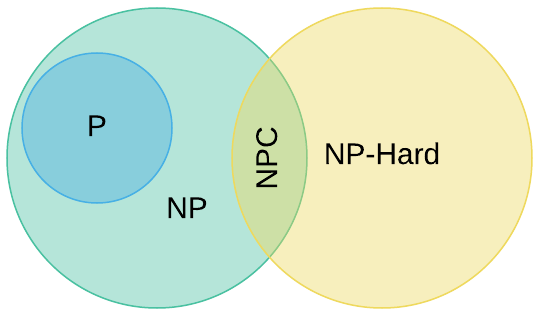
AI-generated content may be incorrect.

**Figure 2.1**: Time Complexity [3]

Figure 1 illustrates common algorithmic time complexities. While quadratic time, O(n²), is relatively inefficient compared to linear or logarithmic time, it still falls within the class of polynomial-time algorithms. In contrast, exponential time, O(2ⁿ), and factorial time, O(n!), grow much faster and lie outside the class P

**2.1.2 Non-deterministic Polynomial Time**

NP(Nondeterministic Polynomial Time) refers to a problem for which a given solution can be verified in polynomial time by a deterministic algorithm – even if finding that solution could prove difficult [3].



**Figure 2.2**: Complexity Classes [4]

Figure 2 shows the complexity classes working together, it clearly depicts P ⊆ NP always. The table below describes the key relationships between the class and its function (NP-Complete and NP-Hard are followed through in sections 2.3 and 2.4 respectively):

| **Class** | **Description** |
| --- | --- |
| **P** | Problems solvable quickly (polynomial time) by deterministic algorithms. |
| **NP** | Problems whose solutions can be verified quickly, includes all of P. |
| **NP-Complete** | The hardest problems within NP. Efficiently solving one implies efficient solutions for all NP problems. |
| **NP-Hard** | Problems at least as hard as NP-complete, but not necessarily in NP-or even decidable. |

**Table 2.1**: Complexity Class Functions

**2.1.3 NP-Complete**

NP-complete problems are a subset of NP with two defining properties:

* **In NP** - Their solutions can be verified in polynomial time by a deterministic algorithm.
* **NP-hardness** - they are at least as hard as any problem in NP, meaning every NP problem can be transformed (reduced) to them in polynomial time [5].

Such example of an NP-Complete problem is the Traveling Salesman Problem (TSP), given a list of cities and the distances between them, the task is to determine the shortest possible route that visits each city exactly once and returns to the starting point. Verifying a proposed route’s total distance is fast (polynomial time), but finding the optimal route may require checking exponentially many possibilities - making TSP a classic NP-complete problem [5]

**2.1.4 NP-Hard**

NP-hard problems are those to which every problem in NP can be reduced in polynomial time, making them at least as challenging as NP-complete problems. Importantly, NP-hard problems are not required to be in NP - they may lack efficient verifiers or even be undecidable. In contrast, NP-complete problems must be both in NP (verifiable in polynomial time) and NP-hard, placing them among the most difficult problems within NP [6].

Such example of an NP-Hard problem is the Halting Problem, which asks whether a given computer program will eventually stop running (halt) or continue executing forever, for a specified input. Alan Turing proved in 1936 that there is no general algorithm capable of solving this problem for all possible program–input pairs, making it undecidable [7].

**2.1.5 P vs NP Conclusion**

Hence, P = NP is equivalent to stating “Is every problem whose solution is easy to check, also easy to solve?”. This question is one of the most important open problems in theoretical computer science, with far-reaching implications. If P = NP were proven true, a wide range of problems currently thought to be intractable could be solved efficiently, revolutionizing fields such as cryptography, optimization, logistics, artificial intelligence and bioinformatics amongst others. Conversely, if P != NP, it would confirm that certain problems are incapable of efficient solutions.

**2.2 Boolean Satisfiability (SAT)**

Boolean Satisfiability is the task of determining whether a Boolean formula can be made true by assigning truth values to its variables. Given a propositional formula: ϕ(x₁, x₂, …, xₙ), SAT checks whether there exists an assignment for the given literals such that ϕ evaluates to True [8]. Example:

(x ∨ y) ∧ (¬x V z) is satisfiable with the assignment:

X = False, y = True, z = True

**2.2.1 Conjunctive Normal Form (CNF)**

To apply SAT solvers, which will be discussed on later throughout the paper – Boolean formulas are expressed in Conjunctive Normal Form:

* Literal: A variable (x) or its negation (¬x)
* Clause: A disjunction (OR) of literals, e.g., (x V ¬y V z)
* CNF Formula: A conjunction (AND) of Clauses, e.g.:

(x V y) ∧ (¬x V z) ∧ (¬y V ¬z)

This representation is central due to most SAT algorithms operating on CNF [9].

**2.2.2 3-SAT**

3-SAT is a restricted version of SAT where each clause contains at most three literals [10] . Example:

(x V ¬y V z) ∧ (¬x V y V w)

3-Sat is highly significant as it was one of the first problems shown to be NP-Complete similarly to SAT, in contrast to 2-SAT, which is solvable in polynomial time [10]. Due to its central role in complexity theory and its use as a benchmark for satisfiability algorithms, 3-SAT has become a primary focus of research - and it will likewise be the focus of this thesis.

**2.2.3 SAT Conclusion**

As the first problem proven to be NP-Complete, SAT lies at the heart of the P vs NP question. Its significance reaches beyond theory since it is universal – due to a wide range of problems such as graph coloring, scheduling and circuit verification can be efficiently reduced to SAT.

Moreover, SAT’s practical relevance in modern SAT solvers are capable of handling instances with millions of variables – which will be discussed later in the thesis. As a result, SAT-based approaches have become indispensable tools in diverse applications including software and hardware verification, security analysis, artificial intelligence and cryptography.

Together, these properties make SAT not only a pivotal theoretical construct but also a practical framework for solving real-world problems. Thus, SAT and the P vs NP problem are inseparably connected which explains why SAT continues to attract significant research interest and why it serves as a natural focus for this thesis.

**Literature Review**

**3.1 SAT Solvers**

A SAT solver is a tool that takes a CNF Formula as input and outputs either a satisfying Boolean assignment or an UNSAT if it is not [11]. SAT solvers provide combinatorial reasoning with the underlying representational formalism being propositional logic. However, the full potential of SAT solvers becomes apparent in their practical uses that are not viewed as propositional reasoning tasks such as [12]:

* AI Planning
* Hardware Verification
* Cryptanalysis
* Scheduling

Multiple SAT solvers have been created throughout the years, with the first instance being the Davis-Putnam-Logemann-Loveland algorithm created in the 1960s [11]. Afterwards, the Conflict-driven clause learning algorithm was created during the mid-90s as an improvement on the DPLL algorithm [13].

Furthermore, more modern and robust SAT solvers have been created throughout the years

**3.2 DPLL Algorithm**

The Davis-Putnam-Logemann-Loveland algorithm underlies most modern SAT solvers. It was introduced by Martin Davis, Hilary Putnam, George Logemann and Donald Loveland. However, it had improvements performed over the 1960s by Davis-Putnam [14]. The DPLL algorithm works as such:

1. **Unit Propagation**

* If a clause has only one literal, that literal must be true to satisfy the clause.
* Assign it as true, remove all clauses containing it, and delete its negation from other clauses.

1. **Pure Literal Elimination**

* If a variable appears with only one polarity (always positive or always negated), assign it to satisfy all clauses containing it.
* Remove the satisfied clauses from the formula.

1. **Splitting / Decision Step**

* Select an unassigned variable and assign it a truth value (e.g., true).
* Recursively check satisfiability under this assignment.
* If this leads to a contradiction (UNSAT), backtrack and try the opposite assignment (false) as shown below.

1. **Backtracking**

* When a conflict is detected, backtrack to the most recent decision point and try an alternative assignment.
* If no alternatives remain, the formula is unsatisfiable [15].

The below depiction shows pseudocode of the DPLL algorithm

A screenshot of a computer program

AI-generated content may be incorrect.

**Figure 3.1**: DPLL-Recursive(F, ρ) [12]

DPLL is inefficient due to its limitations:

* It does not keep track of what it learns – With each backtrack, it does not track why a conflict occurred.
* It may revisit the same conflict pattern multiple times
* Even though it backtracks, it only undoes the most recent decision even If the real cause of the conflict may of have occurred several levels deeper. [16]

Furthermore, a raw full coding solution of the DPLL algorithm can be found in Appendix A via the help of an LLM, no external packages were used.

**3.3 CDCL Algorithm – REWORD STEPS, UNCLEAR**

The Conflict-Driven Clause Learning Algorithm is an extension over DPLL, retaining many components such as unit propagation and backtracking. However, it adds several powerful add-ons which are particularly effective on large, real-world formulas [16]. The CDCL algorithm works as such:

**1. Unit Propagation (Same as DPLL)**

* If a clause has only one unassigned literal, that literal must be true to satisfy the clause.
* Assign it as true, remove satisfied clauses, and delete its negation from other clauses.
* In CDCL, watched literals are used to efficiently track unit clauses without scanning every literal in every clause.

**2. Decision / Branching**

* Select an unassigned variable and assign it a truth value (e.g., true).
* Push this assignment onto a decision stack and increment the decision level.
* Continue propagation to deduce further assignments.

**3. Conflict Detection**

* If unit propagation produces a clause where all literals are false, a conflict occurs.
* Unlike DPLL, CDCL does not simply backtrack chronologically to the last decision. Instead, it analyses the conflict.

**4. Conflict Analysis & Clause Learning**

* CDCL constructs an implication graph representing assignments and their dependencies.
* Using the First Unique Implication Point (UIP), it learns a new clause that prevents the same conflict in the future.
* This learned clause is added to the formula, pruning the search space.

**5. Non-Chronological Backtracking (Back-jumping)**

* Analyse the learned clause to determine the highest decision level that must be reverted.
* Backtrack (jump) directly to this level instead of the last assignment.
* This avoids exploring irrelevant parts of the search tree, unlike DPLL’s chronological backtracking.

**6. Repeat**

* Continue with unit propagation, decision assignments, and conflict analysis iteratively until:
  + All variables are assigned (SAT), or
  + A conflict occurs at decision level 0 (UNSAT) [17]

The below depiction shows pseudocode of the CDCL algorithm

A screenshot of a computer code

AI-generated content may be incorrect.

**Figure 3.2**: CDCL Algorithm Pseudocode [16]

Furthermore, a raw full coding solution of the DPLL algorithm can be found in Appendix B via the help of an LLM, no external packages were used.

4.AND THEN FIND OUT BY MYSELF WHAT ML SOLVERS EXIST AND DECIDE WHAT TO DO THESIS ON.

**Expressions Explainability in Appendix?**

P ⊆ NP – Every problem in P is automatically in NP.

**REFERENCES**

[1] Clay Mathematics Institute, "The Clay Mathematics Institute — Overview and History." Accessed Aug. 15, 2025. [Online]. Available: https://www.claymath.org/about/the-clay-mathematics-institute-overview-and-history/

[2] L. Hardesty, "Explained: P vs. NP," MIT News, Oct. 29, 2009. [Online]. Available: https://news.mit.edu/2009/explainer-pnp. [Accessed: Aug. 15, 2025].

[3] Launch School, "Exploring time complexities," Introduction to Data Structures and Algorithms. [Online]. Available: https://launchschool.com/books/dsa/read/exploring\_time\_complexities. [Accessed: Aug. 15, 2025].

[4] “P vs. NP – Intractability,” Hideous Humpback Freak. [Online]. Available: https://hideoushumpbackfreak.com/algorithms/np-hard-problems.html. [Accessed: Aug. 15, 2025].

[5] A. Gohil, M. Tayal, T. Sahu, and V. Sawalpurkar, "Travelling Salesman Problem: Parallel Implementations & Analysis," arXiv, May 28, 2022. doi: 10.48550/arXiv.2205.14352.

[6] “Computational Complexity Theory (P, NP, NP-Complete and NP-Hard Problems),” ResearchGate. [Online]. Available: https://www.researchgate.net/publication/342335335\_Computational\_Complexity\_TheoryPNPNP-Complete\_and\_NP-Hard\_Problems. [Accessed: Aug. 15, 2025].

[7] A. M. Turing, “On Computable Numbers, with an Application to the Entscheidungsproblem,” Proc. London Math. Soc., vol. s2-42, no. 1, pp. 230–265, 1936. doi: 10.1112/plms/s2-42.1.230.

[8] Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman.

[9] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[10] Cook, S. A. (1971). *The complexity of theorem-proving procedures*. Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC), 151–158.

[11] E. Erdem, “Satisfiability and SAT solvers,” CSE 331 support, University at Buffalo, Buffalo, NY, USA. [Online]. Available: <https://cse.buffalo.edu/~erdem/cse331/support/sat-solver/index.html>

[12] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, “Satisfiability Solvers,” in *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz, and B. Porter, Eds., Elsevier B.V., 2008, ch. 2. [Online]. Available: [https://www.cs.cornell.edu/gomes/pdf/2008\_gomes\_knowledge\_satisfiability.pdf](https://www.cs.cornell.edu/gomes/pdf/2008_gomes_knowledge_satisfiability.pdf?utm_source=chatgpt.com)

[13] J. Marques-Silva, I. Lynce, and S. Malik, “Conflict-Driven Clause Learning,” in *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., Amsterdam, The Netherlands: IOS Press, 2008, ch. 4. [Online]. Available: [https://www.cs.princeton.edu/~zkincaid/courses/fall18/readings/SATHandbook-CDCL.pdf](https://www.cs.princeton.edu/~zkincaid/courses/fall18/readings/SATHandbook-CDCL.pdf?utm_source=chatgpt.com)

[14] J. Worrell, *The DPLL Algorithm*, Hilary Term 2024 lecture notes, Department of Computer Science, University of Oxford. [Online]. Available: [https://www.cs.ox.ac.uk/people/james.worrell/lecture06.pdf](https://www.cs.ox.ac.uk/people/james.worrell/lecture06.pdf?utm_source=chatgpt.com)

[15] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem-proving,” *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962.

[16] Marques-Silva, J., & Sakallah, K. A. (1996). GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers, 48*(5), 506–521. https://ieeexplore.ieee.org/document/769433

[17] T. Junttila, “Conflict-driven clause learning (CDCL) SAT solvers,” *Aalto University*, [Online]. Available: https://users.aalto.fi/~tjunttil/2022-DP-AUT/notes-sat/cdcl.html

**Appendix A – DPLL Algorithm Implementation**

A screen shot of a computer program

AI-generated content may be incorrect.

**Figure [Placeholder]**: DPLL Algorithm Implementation, part 1

A computer screen shot of a program

AI-generated content may be incorrect.

**Figure [Placeholder]**: DPLL Algorithm Implementation, part 2

A screen shot of a computer program

AI-generated content may be incorrect.

**Figure [Placeholder]**: DPLL Algorithm Implementation, part 3

**Appendix B – CDCL Algorithm Implementation**