Einführung in das Boolean Satisfiability Problem (SAT)

Steve Jordi Nyami Gochui
Fachbereich Elektrotechnik und Informationstechnik
Technische Universität Darmstadt
Darmstadt, Deutschland
steve.nyami@stud.tu-darmstadt.de

***Zusammenfassung*—In einer Welt, in der Logik dominiert und rechentechnische Herausforderungen allgegenwärtig sind, stellt das Boolean Satisfiability Problem (SAT) eine zentrale Herausforderung in der Informatik dar, insbesondere wegen seiner Klassifizierung als NP-vollständiges Problem. SAT ist eines der grundlegendsten Probleme der theoretischen Informatik und zugleich ein zentrales Werkzeug für viele praxisrelevante Anwendungen wie Softwareverifikation, Planung und künstliche Intelligenz. Diese Seminararbeit liefert eine umfassende Einführung in das SAT-Problem, stellt seine theoretische Grundlage vor und analysiert aktuelle Lösungsansätze. Neben klassischen Verfahren wie dem Davis-Putnam-Logemann-Loveland (DPLL) Algorithmus werden auch moderne, lernbasierte Solver wie Conflict-Driven Clause Learning (CDCL) detailliert betrachtet. Ein besonderer Fokus liegt auf der praktischen Umsetzung durch ein durchgängiges Beispiel, welches schrittweise in die konjunktive Normalform (CNF) überführt und gelöst wird. Zusätzlich werden parallele SAT-Solver als leistungsfähige Antwort auf die wachsende Komplexität realer Problemstellungen diskutiert. Die Arbeit schließt mit einem Ausblick auf offene Forschungsfragen und potenzielle Entwicklungen in der SAT-Landschaft.**

I. Einleitung

Das Boolean Satisfiability Problem (SAT) ist eine fundamentale Herausforderung der theoretischen Informatik. Es stellt sich dabei die Frage, ob eine gegebene Bool'sche Formel durch eine geeignete Belegung ihrer Variablen erfüllbar ist. Trotz seiner scheinbaren Einfachheit ist SAT das erste Problem, für das NP-Vollständigkeit nachgewiesen wurde [1]. SAT bildet die Grundlage zahlreicher realer Anwendungen, wie z. B. Hardware- und Softwareverifikation [2], Modellprüfung [3] oder auch Anwendungen im Bereich der künstlichen Intelligenz hängen maßgeblich von leistungsfähigen SAT-Solvern ab [4].

Die hohe Relevanz von SAT zeigt sich nicht zuletzt an der aktiven Forschungsgemeinschaft. Jährlich werden internationale Wettbewerbe wie die SAT Competition organisiert [5], um neuartige Solver zu evaluieren und Benchmarks zu vergleichen. Das eigentliche Problem entsteht aus der hohen Komplexität vieler realer SAT-Instanzen, die klassische Lösungsansätze wie den Davis-Putnam-Logemann-Loveland (DPLL) Algorithmus schnell überfordern. Um diese Herausforderungen zu meistern, sind moderne Verfahren wie das Conflict-Driven Clause Learning (CDCL) entwickelt worden. Doch selbst diese Verfahren stoßen bei großen industriellen Problemen regelmäßig an Grenzen, insbesondere wenn sie nur auf sequentieller Ausführung basieren. Neben der systematischen Einführung und der gründlichen Erklärung etablierter SAT-Verfahren wie DPLL und CDCL, liegt der Fokus auf einer vergleichenden Analyse und insbesondere auf der Untersuchung paralleler Lösungsstrategien (Parallel SAT-Solving). Dabei steht im Mittelpunkt, ein umfassendes Verständnis dafür zu schaffen, wie parallele Verfahren helfen können, komplexe und praktisch relevante SAT-Probleme besser und schneller zu lösen.

Die verbleibende Ausarbeitung ist wie folgt strukturiert: Kapitel 2 erläutert die theoretischen Grundlagen zum SAT-Problem. Im Kapitel 3 werden sowohl die wichtigsten SAT-Algorithmen (insbesondere DPLL und CDCL) ausführlich vorgestellt, als auch eine Auswertung auf Benchmarks und ein ausführliches, durchgängiges Beispiel dargestellt. Das Kapitel 4 ergänzt die Ausarbeitung um eine detaillierte Betrachtung paralleler SAT-Solving Methoden und diskutiert deren Herausforderungen und Chancen. Danach gibt das Kapitel 5 einen Überblick über verwandte Forschungsarbeiten und die industrielle Relevanz der SAT-Solver. Abschließend führt das Kapitel 6 eine Zusammenfassung und gibt einen Ausblick auf zukünftige Entwicklungen.

II. Grundlagen

In den 1960er Jahren entwickelten Informatiker die Theorie der Berechnungskomplexität, die sich mit den quantitativen Aspekten der Berechnung und insbesondere der Berechnungszeit befasst [6]. In den 1970er Jahren zeigten S.A. Cook und R.M. Karp, dass viele wichtige Probleme zwar in NP gelöst werden können, aber keine bekannten Algorithmen in P existiert. Diese Probleme sind NP-vollständig, was bedeutet, dass jedes Problem in NP auf sie reduziert werden kann und sie somit zu den schwierigsten Problemen in NP gehören [1]. Diese Erkenntnisse sind grundlegend für das Verständnis der Komplexität und Berechenbarkeit in der Informatik und bilden die Basis für moderne Ansätze in der algorithmischen Theorie und Praxis.

A. Komplexitätstheorie und NP-Vollständigkeit

Die Komplexitätstheorie ist ein zentraler Bereich der Informatik, der sich mit der Klassifizierung von Problemen basierend auf ihrem Ressourcenbedarf befasst. Ein fundamentales Konzept in diesem Zusammenhang ist die NP-

Vollständigkeit [7]. Um dies zu verstehen, ist es wichtig, die Unterschiede zwischen den Klassen P, NP, NP-Schwer und NP-Vollständig zu kennen.

- Probleme in der Klassen P können in Polynomialzeit gelöst werden. Das bedeutet, dass die Lösung eines Problems in einer Zeit erfolgen kann, die durch ein Polynom der Eingabelänge beschrieben wird, z. B. n^2 oder n^3. Solche Probleme gelten als effizient lösbar.
- Probleme in der Klasse NP können zwar in polynomieller Zeit verifiziert werden, wenn eine Lösung vorgegeben ist, aber es ist nicht bekannt, ob sie auch in polynomieller Zeit gelöst werden können.
- Ein Problem wird als NP-schwer klassifiziert, wenn ein Algorithmus zur Lösung des Problems übersetzt werden kann, um jedes beliebiges NP-Problem zu lösen. Anders gesagt, jedes Problem in NP in Polynomialzeit kann auf ein NP-schweres Problem reduziert werden. Somit sind NP-schwere Probleme mindestens so schwer wie die schwierigsten Probleme in NP.
- NP-vollständige Probleme sind die schwierigsten Probleme in NP, da jedes Problem in NP effizient auf sie reduziert werden kann. Ein Problem ist NP-vollständig, wenn es sowohl in NP liegt als auch NP-schwer ist. NP-vollständige Probleme sind somit die härtesten Probleme in NP, und ihre Lösbarkeit in polynomieller Zeit würde implizieren, dass P gleich NP ist.

In der Abbildung 1 wird die Korrelation zwischen den oben genannten Komplexitätsklassen veranschaulicht.

Diese Theorie hat weitreichende Implikationen für die Praxis, da sie hilft, die Grenzen der Berechenbarkeit und die Effizienz von Algorithmen zu verstehen.

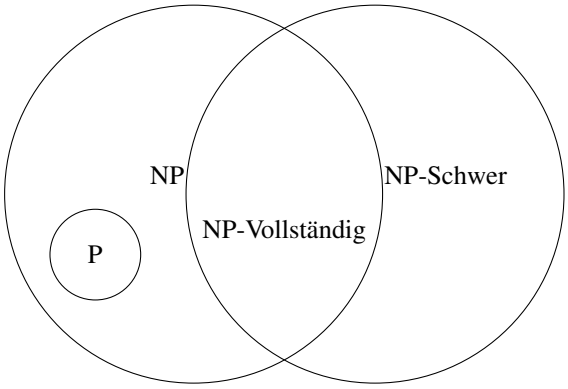

Abbildung 1. Venn-Diagramm der Komplexitätsklassen P, NP, NP-Vollständig und NP-Schwer

B. Einführung in das SAT-Problem

SAT ist ein klassisches Problem der Informatik und Mathematik. Es ist ein Entscheidungsproblem, das untersucht, ob eine gegebene Bool'sche Formel durch eine geeignete Belegung ihrer Variablen wahr gemacht werden kann. Es ist das erste Problem, das als NP-vollständig klassifiziert wurde [1], und stellt sich somit als ein zentrales Element der Komplexitätstheorie dar, da es die Grundlage für das Verständnis vieler anderer schwierigen Probleme bildet.

C. Syntax und formale Definition

SAT-Formel bestehen aus Bool'schen Variablen, den logischen Operatoren $\wedge(UND), \vee(ODER), \neg(NICHT)$ sowie Klammern zur Gruppierung. Jede Formel ϕ über einer Menge von n Variablen $X = \{x_1, x_2, ..., x_n\}$ kann durch Bool'sche Terme beschrieben werden. Eine Formel ist erfüllbar (satisfiable) genau dann, wenn es mindestens eine Belegung $\beta : X \to \{0, 1\}$ der Variablen gibt, bei der die gesamte Formel ϕ wahr wird:

$$\exists \beta \text{ sodass } \phi(\beta) = 1 \tag{1}$$

Ansonsten ist die Formel nicht erfüllbar (not satisfiable).

In der Praxis werden SAT-Probleme standardisiert in einer speziellen syntaktischen Struktur dargestellt: die konjunktive Normalform (CNF). Eine Formel ϕ in CNF besteht aus einer Konjunktion (UND-Verknüpfung "$\wedge$") mehrerer Klauseln, wobei jede Klausel eine Disjunktion (ODER-Verknüpfung "$\vee$") von Literalen ist. Ein Literal ist entweder eine Variable x oder ihre Negation $\neg x$.

$$\phi = \bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{k_i} l_{ij} \right) \tag{2}$$

mit Literalen $l_{ij} \epsilon \{0, 1\}$. Das Ziel ist es, alle Klauseln durch die Belegung der Variablen wahr auszuwerten. Die CNF ist nicht nur ein theoretisches Hilfsmittel, sondern auch das bevorzugte Eingabeformat für nahezu alle modernen SAT-Solver. Ein Beispiel für eine einfache CNF-Formel lautet:

$$\phi_1 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \tag{3}$$

Hier besteht die Formel aus zwei Klauseln über drei Variablen. SAT-Solver durchsuchen den Raum möglicher Wahrheitsbelegungen effizient, um eine Lösung zu finden oder die Unerfüllbarkeit zu beweisen. In der Abbildung 2 wird illustriert, wie eine CNF-Formel als Belegungsbaum dargestellt werden kann.

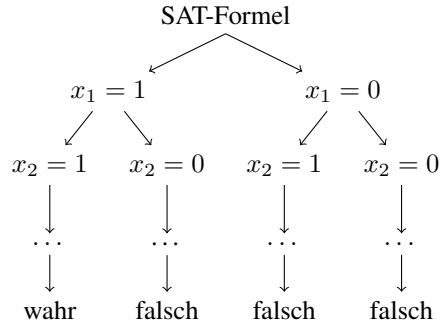

Abbildung 2. Beispielhafte Darstellung einer CNF-Formel als Belegungsbaum

Baumstruktur

- Der Baum zeigt, wie eine SAT-Formel durch systematische Belegung evaluiert wird.
- Jeder Zweig wird durch die Belegungen der Variablen x_1 und x_2 dargestellt.
- Die Blätter des Baums (Pfad gültig und Formel falsch) repräsentieren das Ergebnis der Auswertung der Formel.

D. Varianten des SAT-Problems

SAT existiert in verschiedenen Varianten, die sich in ihrer Struktur und Komplexität unterscheiden. In der Tabelle I wird einen Überblick über die SAT-Varianten und deren Komplexitäten dargestellt.

E. Beispielinstanz (3-SAT-Instanz) zur Veranschaulichung

Betrachten wir folgende 3-SAT-Formel:

$$\phi_1 = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \tag{4}$$

Um zu prüfen, ob diese erfüllbar ist, müsste ein SAT-Solver alle möglichen Belegungen systematisch oder heuristisch prüfen. Für kleine Formeln ist das noch machbar, aber die Komplexität steigt exponentiell mit der Anzahl der Variablen. Die Anzahl der möglichen Belegungen β einer SAT-Formel ϕ mit n Variablen ist 2^n (für das Beispiel haben wir dementsprechend $\beta = 2^3 = 8$ möglichen Belegungen). Die Belegungen werden nacheinander darauf überprüft, ob sie die Formel erfüllen. Trifft dies zu, gibt der Algorithmus erfüllbar aus. Wurden alle möglichen Belegungen eingesetzt, ohne dass eine erfüllende darunter war, ist die Ausgabe unerfüllbar. In der Tabelle II werden alle möglichen Belegungen der Variablen x_1, x_2, und x_3, sowie die resultierende Auswertung (erfüllbar oder unerfüllbar) der Formel ϕ_1 für jede mögliche Belegung dargestellt.

F. Anwendungsbeispiele und Relevanz

Trotz seiner theoretischen Komplexität ist SAT in der Praxis sehr relevant. SAT-Solver wie MiniSAT [9] lösen industrielle Probleme mit Millionen Klauseln effizient.
Beispiele aus der Praxis:

- Hardware- und Software-Verifikation: Das Problem der Korrektheit von Schaltkreisen wird auf SAT reduziert [10].
- Kryptoanalyse: SAT-Solver knacken kryptografische Instanzen durch Modellierung von Schlüsselsuchräumen. Schwachstellen werden auch dadurch identifizieren [11].
- Künstliche Intelligenz: Klassische Planer (z. B. STRIPS) verwenden SAT zur Problemlösung (im Prinzip wandeln sie eine Suche nach einem Plan, der bestimmte Ziele erreicht, in eine Bool'sche Erfüllbarkeitsaufgabe um) [12].

III. Eigener Beitrag

A. Motivation und Zielsetzung

SAT steht in Zentrum zahlreicher Probleme der theoretischen Informatik und praxisnaher Anwendungen wie formale Verifikation, Hardwaredesign, Künstliche Intelligenz oder automatisches Planen. SAT bildet somit nicht nur ein grundlegendes NP-vollständiges Problem [10], sondern fundiert auch als gemeinsamer Nenner zur Lösung verschiedenster real weltlicher Herausforderungen [1]. Die fortschreitende Entwicklung effizienter SAT-Solver hat es ermöglicht, Instanzen mit Millionen von Variablen und Klauseln innerhalb kurzer Zeit zu lösen. Ziel dieses Kapitel ist es, einen praxisnahen Überblick über gängige SAT-Solver Ansätze zu geben, eine kleine Analyse typischer Benchmarks durchzuführen und anhand eines konkreten Beispiels zu demonstrieren, wie sich ein reales Problem in ein SAT-Problem überführen und lösen lässt.

B. Analyse typischer SAT-Solver

Formal gesehen ist ein Solver ein Verfahren zur Lösung des SAT-Problems: Bei gegebener Bool'scher Formel soll ein Solver die Erfüllbarkeit beurteilen und bei Erfüllbarkeit eine gültige Zuordnung liefern. Ein vollständiger Solver kann im Gegensatz zu unvollständigen Algorithmen die Unerfüllbarkeit einer SAT-Instanz anhand eines Beweises ableiten [13].

Zwei zentrale Verfahren prägen das moderne SAT-Solving: das klassische DPLL Verfahren und sein moderner Nachfolger CDCL

Davis-Putnam-Logemann-Loveland (DPLL)-Algorithmus

Der erste Algorithmus zur Lösung von SAT basierte auf Auflösung und wurde von Davis und Putnam in 1960 [14] vorgeschlagen. Der ursprüngliche Algorithmus leidet unter dem Problem der Speicherexplosion. Daher schlugen Martin D. Davis, George Logemann und Donald W. Loveland in 1962 [15] eine modifizierte Version vor, die die Backtrack-Suche verwendet, um den für den Löser erforderlichen Speicherplatz zu begrenzen. Der DPLL-Algorithmus ist ein vollständiger, auf Backtracking basierender Suchalgorithmus zur Bestimmung der Erfüllbarkeit von Formeln der Aussagenlogik in konjunktiver Normalform, d. h. zur Lösung des CNF SAT-Problems.

Der grundlegende Backtracking-Algorithmus funktioniert wie folgt: Ein Literal wird ausgewählt und ihm wird ein Wahrheitswert zugewiesen. Anschließend wird die Formel vereinfacht. Dann wird rekursiv überprüft, ob die vereinfachte Formel erfüllbar ist. Wenn dies der Fall ist, dann ist auch die ursprüngliche Formel erfüllbar. Falls nicht, wird dieselbe rekursive Prüfung mit dem entgegengesetzten Wahrheitswert durchgeführt. Dieser Prozess wird als Splitting-Regel bezeichnet, da er das Problem in zwei einfachere Teilprobleme aufteilt. Durch den Vereinfachungsschritt werden alle Klauseln, die durch die Zuweisung wahr werden, aus der Formel entfernt, und alle Literale, die falsch werden, werden aus den verbleibenden Klauseln gestrichen. Der Algorithmus kombiniert rekursive Tiefensuche mit zwei fundamentalen Operationen:

- $UnitPropagation$: Wenn eine Klausel nur ein nicht erfülltes Literal enthält, wird dieses Literal automatisch auf wahr gesetzt, um die Klausel zu erfüllen.
- $PureLiteralElimination$: Literale, die ausschließlich positiv oder ausschließlich negativ auftreten, werden ge-

Tabelle I
ÜBERSICHT ÜBER DIE SAT-VARIANTEN UND IHRE KOMPLEXITÄTEN

Variante	**Beschreibung**	**Komplexität**
SAT	Allgemeine SAT-Problem	NP-vollständig [1]
2-SAT	Nur Klauseln mit 2 Literalen	P [8]
3-SAT	Klauseln mit genau 3 Literalen	NP-vollständig [1]
K-SAT	Klauseln mit K Literalen ($K \geq 3$)	NP-vollständig
MAX-SAT	Maximale Anzahl erfüllter Klauseln	NP-Schwer [6]
UNSAT	Entscheidung, ob keine erfüllende Belegung existiert (für Verifikation oft relevant)	Co-NP-vollständig

Tabelle II
MÖGLICHEN BELEGUNGEN DER VARIABLEN UND AUSWERTUNG VON ϕ

$\beta(x_1)$	$\beta(x_2)$	$\beta(x_3)$	**Formel** ϕ_1
0	0	0	wahr (Erfüllbar)
0	0	1	wahr (Erfüllbar)
0	1	0	**falsch (Unerfüllbar)**
0	1	1	wahr (Erfüllbar)
1	0	0	**falsch (Unerfüllbar)**
1	0	1	wahr (Erfüllbar)
1	1	0	wahr (Erfüllbar)
1	1	1	wahr (Erfüllbar)

setzt, um alle Klauseln zu erfüllen, in denen sie enthalten sind.

Im Algorithmus 1 wird der Pseudo-Code des DPLL-Verfahrens dargestellt, wobei ϕ die CNF-Formel ist:

Algorithm 1 DPLL Algorithm

```
function DPLL(φ)
    φ ← unit_propagate(φ)
    if φ is empty then
        return true
    end if
    if φ contains an empty clause then
        return false
    end if
    l ← choose_literal(φ)
    return DPLL(φ ∧ l) or DPLL(φ ∧ ¬l)
end function
```

Der DPLL-Algorithmus entspricht einer depth-first Back-Track Suche, bei der in jedem Verzweigungsschritt eine Variable ausgewählt und mit einem Wahrheitswert Wert zugewiesen wird. Anschließend werden die logischen Konsequenzen jedes Verzweigungsschritts bewertet. Jedes Mal, wenn eine unerfüllbare Klausel (d.h. ein Konflikt) festgestellt wird, wird ein Backtracking durchgeführt. Backtracking entspricht dem Rückgängigmachen von Verzweigungsschritten, bis ein Branch gefunden wird, bei dem einer der Werte noch nicht ausprobiert wurde. Wenn bei dem ersten Verzweigungsschritt beide Werte berücksichtigt wurden und das Backtracking diesen ersten Verzweigungsschritt rückgängig macht, kann die CNF-Formel als unerfüllbar erklärt werden. Diese Art des Backtrackings wird als chronologisches Backtracking bezeichnet. Die Effizienz von DPLL hängt sehr stark von der Auswahl des Literals l (Branching Literal) ab. Die Probleme von DPLL können in drei Punkten zusammengefasst werden:

- Die Entscheidungen für Branching Literals werden naiv getroffen.
- Aus Konflikten wird nichts gelernt, außer dass die aktuelle (partielle) Variablenbelegung zu einem Konflikt führt. Dabei lassen sich mehr Informationen über die Ursache des Konfliktes extrahieren und so große Teile des Suchraumes ausschließen.
- Backtracking springt nur jeweils eine Ebene im Suchbaum nach oben, was zu einem sehr großen Suchraum führt.

Im Laufe der Jahre wurden viele bedeutende Verbesserungen des grundlegenden DPLL-Algorithmus vorgeschlagen. Insbesondere eine Technik namens Conflict-Driven Learning und nicht-chronologisches Backtracking [16] hat die Leistungsfähigkeit von DPLL SAT-Solver bei Problemfällen aus realen Anwendungen erheblich verbessert und ist zu einem Schlüsselelement moderner SAT-Löser geworden.

Conflict-Driven Clause Learning Algorithmus (CDCL)

Der CDCL-Algorithmus, der erstmals im Solver GRASP [4] vorgeschlagen wurde, ist ein repräsentativer kompletter SAT-Algorithmus. Als Verbesserung des DPLL Algorithmus [15] erweitert CDCL-Verfahren diesen um Backjumping und das Lernen neuer Klauseln. Statt chronologisch zurückzuspringen, analysieren CDCL-Solver Konflikte (diese treten auf, wenn ein Literal gleichzeitig den Wert wahr und den Wert falsch annehmen müsste) und erzeugen aus ihnen Learned Clauses, die helfen, ähnliche Konflikte in Zukunft zu vermeiden [4]. Die primäre Heuristik des CDCL-Algorithmus besteht darin, dass er neue Klauseln aus Konflikten lernt und diese der ursprünglichen Klausel hinzufügt. Im Algorithmus 2 wird die standardmäßige Organisation des Pseudo-Codes eines CDCL SAT-Solvers dargestellt.

Der CDCL-Algoritmus liegt besonderes Augenmerk auf die folgenden vier Hilfsmethoden:

- $UnitPropagation$: besteht aus der wiederholten Anwendung der Regel für Einheitsklauseln. Gegeben einer Einheitsklausel kann die Regel für Einheitsklauseln [14], [15] angewendet werden: Das nicht zugewiesene Literal muss den Wert 1 zugewiesen bekommen, damit die Klausel erfüllt wird. Wenn eine unerfüllte Klausel identifiziert wird, wird ein Konflikt zurückgegeben.

Algorithm 2 CDCL SAT-Solver (ϕ)

```
1: dl ← 0; conflicts ← 0; answer ← UNKNOWN
2: if UnitPropagation(φ) = CONFLICT then
3:     answer ← UNSATISFIABLE
4: end if
5: while answer = UNKNOWN do
6:     if AllVariablesAssigned(φ) then
7:         answer ← SATISFIABLE
8:     else
9:         dl ← dl + 1
10:        AssignBranchingVariable(φ)
11:        while UnitPropagation(φ) = CONFLICT ∧ answer = UNKNOWN do
12:            β ← ConflictAnalysis(φ)
13:            if β < 0 then
14:                answer ← UNSATISFIABLE
15:            else
16:                Backtrack(φ, β)
17:                dl ← β
18:                conflicts ← conflicts + 1
19:            end if
20:        end while
21:        Restart(φ, limit)
22:    end if
23: end while
24: return answer
```

- $AssignBranchingVariable$: besteht darin, eine Variable auszuwählen, die zugewiesen werden soll, und ihren entsprechenden Wert. In der Vergangenheit wurden verschiedene Heuristiken untersucht, einschließlich der VSIDS (Variable State Independent Decaying Sum) Heuristik [17], die als eine der effektivsten gilt. Diese Heuristik verknüpft einen Aktivitätszähler mit jedem Literal. Immer wenn eine gelernte Klausel nach einem Konflikt erstellt wird, wird die Aktivität jedes Literals, das zu dieser Klausel gehört, um einen vordefinierten Wert erhöht. Periodisch werden alle Aktivitätszähler durch eine abgestimmte Zahl geteilt. Wenn AssignBranchingVariable aufgerufen wird, wird das Literal, das noch nicht zugewiesen wurde, mit dem höchsten Wert ausgewählt.
- $ConflictAnalysis$: Die Konfliktanalyse besteht darin, den aktuellsten Konflikt zu analysieren und eine neue Klausel aus dem Konflikt zu lernen. Wenn der Konflikt von keiner Variablenzuweisung abhängt, kann kein Rückverfolgen durchgeführt werden und es wird -1 zurückgegeben. Andernfalls gibt es die Entscheidungsebene zurück, zu der der Solver sicher zurückverfolgen kann.
- $Backtrack$: Verfolgt die Suche bis zur Entscheidungsstufe, die durch die $ConflictAnalysis$ berechnet wurde. Backtracking kann nicht-chronologisch sein. Während chronologisches Backtracking immer zur letzten Entscheidungsstufe zurückkehrt, bei der einer der Werte nicht ausprobiert wurde, kann nicht-chronologisches Backtracking größere Rückverfolgungen zu kleineren Entscheidungsstufen durchführen. Eine Entscheidungsstufe einer Variablen bezeichnet die Tiefe des Entscheidungsbaums, an der der Variablen ein Wert zugewiesen wird. Beim Zurückverfolgen werden alle Variablen auf Entscheidungsstufen, die höher sind als die von der $ConflictAnalysis$ berechnete, nicht mehr zugewiesen.
- $Restart$: Der Neustart besteht darin, die Suche neu zu starten, indem alle Variablen außer denen, die auf Entscheidungsebene 0 zugewiesen wurden, zurückgenommen werden. Variablen, die auf Entscheidungsebene 0 zugewiesen werden, entsprechen Einheitsklauseln und durch Einheitsklauseln erzwungene Implikationen.

In Abschnitt D wird anhand eines Beispiels dargestellt, wie das DPLL-Verfahren genau auf ein konkretes Problem angewendet werden kann. Außerdem existieren zahlreiche fortschrittliche SAT-Solver, die sich in der Praxis bewährt haben und regelmäßig in Wettbewerben wie dem SAT Competition [18] evaluiert werden:

- **MiniSAT**: Ein kompakter und effizienter CDCL-basierter Solver, der sich durch modulare Architektur und einfache Erweiterbarkeit auszeichnet. MiniSAT wurde in C++ implementiert und bildet die Grundlage vieler moderner Werkzeuge [9].
- **Glucose**: Eine Weiterentwicklung von MiniSAT mit besonderem Fokus auf die Analyse von Konfliktklauseln (glue clauses). Er nutzt heuristische Verbesserungen, um die Effizienz in realitätsnahen Instanzen deutlich zu steigern [2].
- **Kissat**: Ein hochoptimierter CDCL-Solver, der speziell für moderne Hardwarearchitekturen konzipiert wurde. Er ist derzeit einer der schnellsten Solver auf vielen Benchmarks und verwendet aggressive Restart-Strategien [18].

C. Auswertung auf Benchmarks

Zur Bewertung der Leistungsfähigkeit typischer SAT-Solver wurde eine kleine experimentelle Auswertung durchgeführt. Dabei wurden drei etablierte CDCL-basierte Solvers – MiniSAT, Glucose und Kissat – hinsichtlich ihrer durchschnittlichen Laufzeit auf zwei Benchmark-Datensätzen untersucht: dem LEC-Datensatz und dem bekannten SAT-Competition-Datensatz [5]. Die Ergebnisse in der Tabelle III stammen aus dem Paper "IB-Net Initial Branch Network for Variable Decision in Boolean Satisfiability"[19].

Tabelle III
DURCHSCHNITTLICHE LAUFZEIT VERSCHIEDENER CDCL-SOLVER FÜR ZUFÄLLIGE 100-PROBEN-UNSAT-PROBLEME AUS DEM LEC-DATENSATZ UND DEM SAT-COMP-DATENSATZ

Solver	LEC (s)	SAT Comp (s)
MiniSAT [9]	810 s	450 s
Glucose [2]	529 s	308 s
Kissat [18]	335 s	195 s

Diese Ergebnisse zeigen, dass:

- **Kissat** im Vergleich zu den anderen Solver eine deutlich geringere durchschnittliche Laufzeit aufweist. Dies lässt sich unter anderem auf eine Reihe moderner Heuristiken und optimierter Implementierungstechniken zurückführen.
- **Glucose** zeigt im Vergleich zu MiniSAT ebenfalls eine signifikante Verbesserung, was auf die effektive Verwendung der Literal Block Distance (LBD)-basierten Klauselbewertung zurückzuführen ist [2].

Besonders in der Industrie kann die Wahl des richtigen Solvers entscheidend sein, um komplexe Probleme praktisch zu lösen.

D. Minimales Beispiel aus dem Alltag

Um die praktische Funktionsweise eines SAT-Solvers anhand eines minimalen Beispiels zu demonstrieren, konstruieren wir ein Problem, das sich auf eine einfache Entscheidungsregel aus dem Alltag bezieht. Anschließend wird die daraus resultierende ϕ Formel mit einem DPLL-Algorithmus gelöst.

Szenario: Seminarmeeting-Planung

In einer kleinen Uni-Gruppe stehen drei Studenten – Anna (A), Ben (B) und Carla (C) – vor der Entscheidung, wer am anstehenden Seminarmeeting teilnimmt. Um die Planung zu erleichtern, haben sie sich auf einige klare Regeln geeinigt:

- **(R1)** Damit das Team vertreten ist, soll mindestens eine Person am Meeting teilnehmen
- **(R2)** Wenn Anna teilnimmt, dann soll auch Ben teilnehmen, da sie gemeinsame Themen besprechen möchten.
- **(R3)** Entweder Ben oder Carla nimmt teil – aber nicht beide gleichzeitig, um Doppelarbeit zu vermeiden.
- **(R4)** Sollte Carla nicht teilnehmen, wird Anna das Meeting übernehmen.
- **(R5)** In jedem Fall hat Carla ihre Teilnahme zugesagt.

Formale Variablenbelegung und Übersetzung in CNF

Wir definieren zuerst die Variablen a, b, c, wobei $a =$ Anna nimmt teil, $b =$ Ben nimmt teil und $c =$ Carla nimmt teil. Danach formalisieren wir die Regeln und wir überführen die CNF.

- **(R1):** $(a \vee b \vee c)$
- **(R2):** $(a \to b) \Leftrightarrow (\neg a \vee b)$
- **(R3):** $(b \vee c) \wedge (\neg b \vee \neg c)$
- **(R4):** $(\neg c \to a) \Leftrightarrow (c \vee a)$
- **(R5):** (c)

So ergibt sich die folgende resultierende Formel ϕ, da alle Regeln gleichzeitig erfüllt werden müssen und daher durch ein boolesches UND verknüpft sind:

$$\phi = (a \vee b \vee c) \wedge (\neg a \vee b) \wedge (b \vee c) \wedge (\neg b \vee \neg c) \wedge (c \vee a) \wedge (c) \tag{5}$$

Anwendung des DPLL-Algorithmus

Unit-Propagation(ϕ)*:* Wir starten mit der Einheitsklausel (c) und setzen somit $(c) = wahr$. Wir führen die Unitprogation mit $(c) = wahr$ und wir vereinfachen alle Klauseln durch diese Belegung:

- $(a \vee b \vee c) \;\to\; wahr\,(wegen\,(c) = wahr) \;\Rightarrow$ Klausel erfüllt.
- $(\neg a \vee b) \to$ bleibt bestehen.
- $(b \vee c) \to wahr \Rightarrow$ Klausel erfüllt.
- $(\neg b \vee \neg c) \to (\neg b \vee \neg wahr) = (\neg b \vee falsch) = (\neg b)$
- $(c \vee a) \to wahr \Rightarrow$ Klausel erfüllt.
- Übrig bleiben $(\neg a \vee b)$ und $(\neg b)$.

Da wir wieder eine Einheitsklausel $(\neg b)$ haben, fangen wir direkt damit und wir setzen $b = falsch \Rightarrow \neg b = wahr$.

Unit-Propagation mit $b = falsch$*:*

- $(\neg a \vee b) = (\neg a \vee falsch) = (\neg a)$ (wieder eine Einheitsklausel.
- Wir setzen $a = falsch \Rightarrow (\neg a) = wahr$

Wir bekommen als finale Belegung der Variablen: $a = falsch$, $b = falsch$ und $c = wahr$. Angewendet in der Formel ϕ, erhalten wir:

$\phi = (falsch \vee falsch \vee wahr) \wedge (\neg falsch \vee falsch) \wedge (falsch \vee wahr) \wedge (\neg falsch \vee \neg wahr) \wedge (wahr \vee falsch) \wedge (wahr) = (wahr) \wedge (wahr) \wedge (wahr) \wedge (wahr) \wedge (wahr) \wedge (wahr) = wahr$

Das Szenario ist erfüllbar, und der DPLL-Algorithmus führt uns mit simplen Unit Propagation- und Backtracking-freien Schritten direkt zur Lösung: Anna = $nein$, Ben = $nein$, und Carla = ja (Diese Zuweisung erfüllt alle Bedingungen des Meetingszenarios).

IV. Paralleles SAT-Solving

A. Warum paralleles SAT-Solving?

Während sequentielle SAT-Solver wie MiniSAT, Glucose oder Kissat durch fortgeschrittene Methoden wie CDCL und effiziente Heuristiken viele praktische Instanzen lösen können, stoßen sie bei sehr großen oder komplexen Instanzen schnell an Grenzen [10]. Anwendungen wie die Hardwareverifikation, Softwareprüfung oder künstliche Intelligenz generieren regelmäßig SAT-Probleme mit Millionen von Variablen und Klauseln, deren Lösung auf herkömmlichen Ein-Prozessor-Systemen nicht praktikabel ist [4]. Hier setzt paralleles SAT-Solving an, das durch gleichzeitige Nutzung mehrerer Prozessoren oder Rechner die Lösungsgeschwindigkeit deutlich erhöhen und somit realistisch große Probleme bewältigen kann [20].

B. Methoden

Paralleles SAT-Solving lässt sich grundsätzlich in drei Strategien unterteilen:

1) Portfolio-Solver: Beim Portfolio-Ansatz werden mehrere Instanzen des gleichen Solvers mit unterschiedlichen Konfigurationen (etwa variierenden Restart-Strategien, Entscheidungsheuristiken oder Clause-Learning-Parametern) parallel ausgeführt [21]. Sobald eine der Instanzen eine Lösung findet oder die Unerfüllbarkeit beweist, werden alle anderen gestoppt. Bekannte Portfolio-basierte Solver sind beispielsweise ManySAT und Plingeling [21], [22].

2) Search-Space-Partitioning: Diese Methode teilt den Suchraum explizit auf. Jeder Prozessor erhält einen Teilraum, der unabhängig durchsucht wird. Ein klassischer Ansatz ist das sogenannte *Cube-and-Conquer-Verfahren*, bei dem zunächst eine Vorpartitionierung mittels Look-Ahead-Verfahren erfolgt (Cubing) und anschließend jeder Teilraum mit einem CDCL-Solver abgearbeitet wird (Conquering) [23].

3) Hybrid-Ansätze: Hybridverfahren kombinieren Portfolio- und Search-Space-Partitioning-Strategien, um deren Vorteile optimal zu nutzen. Ein Beispiel hierfür ist Treengeling, welches eine hierarchische Aufteilung mit Portfolio-Methoden verbindet und besonders gut auf Systemen mit vielen Prozessorkernen skaliert [3].

C. Minimales Beispiel zur Parallelisierung

Betrachten wir die folgende einfache Bool'sche Formel ϕ in CNF:

$$\phi = (a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c \vee \neg a) \tag{6}$$

Ein sequentieller CDCL-Solver könnte beispielsweise folgende Schritte gehen:

- Entscheide $a = wahr$; Klausel $(a \vee b)$ erfüllt.
- Entscheide $b = wahr$; Klausel $(\neg b \vee \neg c)$ ist noch nicht erfüllt, daher wird $c = falsch$ gesetzt, damit die Klausel erfüllt ist.
- Die Klausel $(\neg a \vee c)$ verursacht Konflikt. Sie kann nicht erfüllt werden, da $\neg a = falsch$ und $c = falsch$.
- Backtracking und Neubewertung nötig.

Ein paralleler Ansatz könnte hingegen den Suchraum von Anfang an aufteilen:

- Solver 1 setzt $a = wahr$ und verfolgt die oberen Implikationen.
- Solver 2 setzt $a = falsch$ und erkundet die alternativen Pfade. In diesem Fall wäre z. B. $a = falsch$, $b = wahr$ und $c = wahr/falsch$ (kann beides sein) eine mögliche Belegung, die die Formel ϕ erfüllt.

Beide Threads erkunden somit disjunkte Pfade im Suchbaum. Sobald ein Thread eine erfüllende Belegung findet, terminiert das System. Durch diese parallele Exploration des Suchbaums können Lösungen oder Konflikte effizienter erkannt werden, was insbesondere bei großen Problemen Vorteile bieten kann.

D. Herausforderungen des parallelen SAT-Solving

Trotz der Vorteile sind parallele Ansätze mit spezifischen Herausforderungen verbunden:

- *Lastverteilung*: Eine gleichmäßige Verteilung des Suchraums auf mehrere Solver-Instanzen ist oft schwierig, da einige Teilbereiche erheblich komplexer als andere sein können. Dynamische Load-Balancing-Techniken wie Work-Stealing werden deshalb eingesetzt [3], [21].
- *Kommunikation und Clause Sharing*: Zu häufiger Clause-Austausch zwischen parallelen Instanzen verursacht hohen Overhead und Speicherverbrauch. Daher sind effiziente Strategien zur Auswahl relevanter Klauseln (etwa basierend auf Literal Block Distance, LBD) notwendig [21], [22].
- *Synchronisations-Overhead*: Die Kommunikation zwischen parallelen Solver-Instanzen, insbesondere bei häufigem Austausch von Klauseln, kann erhebliche Synchronisationskosten verursachen und die Skalierbarkeit begrenzen [22].

Paralleles SAT-Solving stellt trotz bestehender Herausforderungen eine leistungsstarke Methode dar, um die Grenzen der sequentiellen Lösungsverfahren zu überwinden und komplexe industrielle SAT-Probleme realistisch zu lösen.

V. Relevante Arbeiten

Die Erforschung von SAT-Solvern hat in den letzten Jahrzehnten bemerkenswerte Fortschritte gemacht. Dabei hat sich die Forschung zunächst auf grundlegende Verfahren wie DPLL konzentriert, bevor moderne Ansätze wie CDCL oder hybride Verfahren entwickelt wurden. Dieses Kapitel gibt einen Überblick über relevante Beiträge aus der Literatur, die die Entwicklung, Effizienz und Anwendung von SAT-Solvern geprägt haben.

A. Historische Fortschritte

Der Ursprung moderner SAT-Solver liegt im Davis-Putnam-Verfahren (DP) [14] und dessen Nachfolger DPLL [15], die bereits in den 1960er Jahren formuliert wurden. DPLL führte das Backtracking und die Literalauswahl mit Heuristiken ein (ein Prinzip, das bis heute eine wichtige Grundlage ist). Ein wesentlicher Fortschritt war die Einführung von Clause Learning, das in GRASP [4] erstmals systematisch beschrieben wurde. Dies ermöglichte es, aus Konflikten während der Suche zu lernen und die Effizienz durch sogenannte Learned Clauses drastisch zu erhöhen.

B. Fortschritt der CDCL-Solver

CDCL-basierte Solver wie Chaff [17], MiniSAT [9], Glucose [2] oder CaDiCaL [3] bauen auf den Grundlagen von DPLL auf, erweitern diese aber durch eine starke Konfliktanalyse, Non-Chronological Backtracking (Backjumping) und moderne Heuristiken wie Variable State Independent Decaying Sum (VSIDS) [17].

Chaff [17] war einer der ersten Solver, der in den $2000er$ Jahren ein effizientes Watched Literal Scheme einführte, um das Unit Propagation drastisch zu beschleunigen. MiniSAT wiederum machte den Solver-Code modular und leicht erweiterbar, was ihn zu einem beliebten akademischen Standard machte. Viele moderne SAT-Solver basieren auf den Techniken und dem Design von MiniSAT [9]. Es hat die Entwicklung von SAT-Solvern stark beeinflusst und ist ein wichtiger Referenzpunkt in der Forschung. In den $2010er$ Jahren wurde die Forschung durch Verbesserungen in Glucose [2] weiter vorangetrieben, insbesondere durch LBD-basierte Klauselbewertung [2]. Schließlich führte CaDiCaL [3] in den späten $2010er$ Jahren mit vollständig neu geschriebenen, hocheffizienten und schnellen Ansätzen zu einem modernen CDCL-Framework. In der Tabelle IV wird einen Überblick auf die Evolution der SAT-Solver zwischen 1996 und 2017 dargestellt.

Tabelle IV
ENTWICKLUNG VON SAT-SOLVERN

SAT-Solver	Jahr	Hauptmerkmale	Bedeutung
GRASP [4]	1996	Conflict Learning	Erster systematischer CDCL-Ansatz
Chaff [17]	2001	Watched Literals, VSIDS [17]	Hohe Effizient im Praxiseinsatz
MiniSAT [9]	2003	Modularer Aufbau	Basis für viele Forschungsarbeiten
Glucose [2]	2009	LBD-basierte Klauselbewertung [2]	Verbesserung von MiniSAT
CaDiCaL [3]	2017	Vollständig neu geschrieben, hohe Effizient und Geschwindigkeit	Modernes CDCL-Framework

C. Industrielle Relevanz

SAT-Solver finden heute Anwendung in verschiedensten Bereichen – von Hardware-Verifikation [24], über Software Testing bis hin zu KI-Planung und Kryptografie [11]. Kommerzielle Tools wie Lingeling [3] oder Kissat [18] haben sich in der Industrie etabliert, insbesondere im Kontext von Bounded Model Checking (BMC).

In jüngster Zeit gibt es zudem Entwicklungen in Richtung Portfolio-Solver (z. B. SATzilla [25]), die mehrere Strategien parallel ausführen und je nach Problemcharakteristik automatisch den besten Solver wählen. Ebenso werden SAT-Solver zunehmend mit Machine Learning [26] kombiniert, um Prädiktoren für Heuristiken oder Restart-Strategien zu trainieren. Diese Entwicklungen zeigen, dass SAT-Solver nicht nur in der Theorie, sondern auch in der Praxis und der Industrie eine zentrale Rolle spielen und sich kontinuierlich weiterentwickeln.

VI. ZUSAMMENFASSUNG UND AUSBLICK

A. Zusammenfassung

In dieser Seminararbeit wurde umfassend das SAT Problem behandelt, eines der zentralen und zugleich anspruchsvollsten Probleme in der theoretischen und angewandten Informatik. Ausgangspunkt der Betrachtung war die Erkenntnis, dass SAT nicht nur theoretische Bedeutung besitzt, sondern vor allem auch zahlreiche praktische Anwendungen in Bereichen wie Verifikation, KI-Planung und Optimierung ermöglicht.

Nach einer formalen Einführung in die theoretischen Grundlagen von SAT und der Darstellung seiner NP-Vollständigkeit wurden in dieser Arbeit zentrale Lösungsverfahren ausführlich erläutert. Zunächst wurde der klassische DPLL-Algorithmus vorgestellt, dessen grundlegendes Backtracking-Konzept die Basis moderner Solver bildet. Anschließend erfolgte eine detaillierte Analyse moderner CDCL-basierter Solver, die mittels Clause-Learning und Heuristiken heutige industrielle Anwendungen effektiv bewältigen können. Ein eigens entwickeltes Beispiel (Seminarmeeting-Planung) demonstrierte anschaulich, wie reale Entscheidungsszenarien mittels SAT modelliert und schrittweise gelöst werden können.

Um praktische Grenzen zu analysieren, wurden ausgewählte Solver wie MiniSAT, Glucose und Kissat betrachtet. Dabei zeigte sich, dass die Effizienz moderner Solver zwar beeindruckend ist, jedoch bei sehr großen Instanzen sequentielle Lösungsverfahren schnell an ihre Grenzen stoßen. Hieraus ergab sich die Motivation für das Kapitel über paralleles SAT-Solving, in dem aktuelle Strategien wie Portfolio-Methoden, Partitionierung des Suchraums und hybride Ansätze erläutert und kritisch verglichen wurden. Es wurde deutlich, dass parallele Methoden eine äußerst vielversprechende Möglichkeit darstellen, enorm große SAT-Instanzen in einem überschaubaren Zeithorizont zu lösen.

B. Ausblick

Die Forschung rund um SAT und SAT-Solving ist hochdynamisch und eröffnet vielfältige Perspektiven. Drei zentrale Trends zeichnen sich derzeit ab:

- *Hybride Solver-Ansätze*: Zukünftige Systeme werden voraussichtlich unterschiedliche Lösungsstrategien kombinieren, um die jeweiligen Stärken maximal auszunutzen und die allgemeine Robustheit zu erhöhen.
- *Integration von Machine Learning*: Der Einsatz lernbasierter Heuristiken für Variable-Branching, Restart-Entscheidungen oder Clause-Deletion verspricht deutliche Verbesserungen der Performanz, besonders bei schwierigen, bisher kaum lösbaren Instanzen.
- *Skalierbarkeit durch Parallelisierung und Verteilung*: Forschung im Bereich der verteilten und parallelen Solver-Architekturen wird weiter an Bedeutung gewinnen, insbesondere im Zusammenhang mit Cloud-basierten Systemen und großen Rechenclustern.

Abschließend lässt sich festhalten, dass das Verständnis und die Weiterentwicklung von SAT-Solvern entscheidend sind, um anspruchsvolle, reale Probleme der Informatik effektiv anzugehen. Diese Arbeit fasst dafür einige Grundlagen zusammen und regt zugleich zur weiteren Erforschung und Innovation in diesem wichtigen Bereich an.

LITERATUR

[1] S. A. Cook, "The complexity of theorem-proving procedures," *Proceedings of the third annual ACM symposium on Theory of computing*, 1971. doi: 10.1145/800157.805047 .

[2] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *International Joint Conference on Artificial Intelligence*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:2828104

[3] A. Biere, "Lingeling, treengeling and y al sat entering the sat competition 2017," 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:45860975

[4] J. Marques-Silva and K. A. Sakallah, "Grasp: A search algorithm for propositional satisfiability," *IEEE Trans. Computers*, vol. 48, pp. 506–521, 1999. doi: 10.1109/12.769433 .

[5] T. Balyo, N. Froleyks, M. J. Heule, M. Iser, M. Järvisalo, and M. Suda, "Proceedings of sat competition 2020: Solver and benchmark descriptions," 2020. doi: https://doi.org/10.1016/j.artint.2021.103572 .

[6] J. Hartmanis, "Computers and intractability: A guide to the theory of np-completeness (michael r. garey and david s. johnson)," *Siam Review*, vol. 24, pp. 90–91, 1982. [Online]. Available: https://api.semanticscholar.org/CorpusID:122180176

[7] O. Goldreich, "P, np, and np-completeness: the basics of computational complexity by oded goldreich," *ACM SIGSOFT Softw. Eng. Notes*, vol. 36, pp. 37–38, 2010. doi: 10.1145/1921532.1921551 .

[8] 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*, vol. 8, no. 3, pp. 121–123, 1979. doi: 10.1016/0020-0190(79)90002-4 .

[9] N. Eén and N. Sörensson, "An extensible sat-solver," in *International Conference on Theory and Applications of Satisfiability Testing*, 2003. doi: https://doi.org/10.1007/978-3-540-24605-$3_3$7 .

[10] A. Biere, M. Heule, and H. van Maaren, *Handbook of Satisfiability*, ser. Frontiers in artificial intelligence and applications. IOS Press, 2009. doi: 10.5555/1550723 . [Online]. Available: https://books.google.de/books?id=YVSM3sxhBhcC

[11] F. Massacci and L. Marraro, "Logical cryptanalysis as a sat problem," *Journal of Automated Reasoning*, vol. 24, pp. 165–203, 2000. doi: https://doi.org/10.1023/A:1006326723002 .

[12] H. A. Kautz, B. Selman *et al.*, "Planning as satisfiability," in *ECAI*, vol. 92, 1992, pp. 359–363. [Online]. Available: https://api.semanticscholar.org/CorpusID:42462267

[13] M. J. H. Heule, "Proofs for satisfiability problems," 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:6825560

[14] M. D. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, pp. 201–215, 1960. doi: 10.1145/321033.321034 .

[15] M. D. Davis, G. Logemann, and D. W. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, 2011. doi: 10.1145/368273.368557 .

[16] R. J. Bayardo and R. C. Schrag, "Using csp look-back techniques to solve real-world sat instances," in *AAAI/IAAI*, 1997. [Online]. Available: https://api.semanticscholar.org/CorpusID:1302756

[17] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 530–535. doi: 10.1145/378239.379017 .

[18] A. Biere, M. Fleury, and F. Pollitt, "Cadical vivinst, isasat, gimsatul, kissat, and tabularasat entering the sat competition 2023," 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:260459496

[19] T. H. Chan, W. Xiao, J. Huang, H.-L. Zhen, G. Tian, and M. Yuan, "Ib-net: Initial branch network for variable decision in boolean satisfiability," *ArXiv*, vol. abs/2403.03517, 2024. doi: https://doi.org/10.48550/arXiv.2403.03517 .

[20] R. Martins, V. Manquinho, and I. Lynce, "An overview of parallel sat solving," *Constraints*, vol. 17, pp. 304–347, 2012. doi: 10.1007/s10601-012-9121-3 .

[21] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a parallel sat solver," *Journal on Satisfiability, Boolean Modelling and Computation*, vol. 6, no. 4, pp. 245–262, 2010. doi: 10.3233/SAT190070 .

[22] G. Audemard and L. Simon, "Refining restarts strategies for sat and unsat," in *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*. Springer, 2012, pp. 118–126. doi: https://doi.org/10.1007/978-3-642-33558-$7_1$1 .

[23] M. J. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding cdcl sat solvers by lookaheads," in *Haifa Verification Conference*. Springer, 2011, pp. 50–65. doi: https://doi.org/10.1007/978-3-642-34188-5_8 .

[24] K. O. Strichman, "Decision procedures: An algorithmic point of view," 2008. doi: https://doi.org/10.1007/978-3-662-50497-0 .

[25] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: Portfolio-based algorithm selection for sat," *ArXiv*, vol. abs/1111.2249, 2008. doi: https://doi.org/10.48550/arXiv.1111.2249 .

[26] W. Guo, J. Yan, H.-L. Zhen, X. Li, M. jie Yuan, and Y. Jin, "Machine learning methods in solving the boolean satisfiability problem," *Machine Intelligence Research*, vol. 20, pp. 640 – 655, 2022. doi: https://doi.org/10.1007/s11633-022-1396-2 .