LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

LEHR- UND FORSCHUNGSEINHEIT FÜR THEORETISCHE INFORMATIK UND THEOREMBEWEISEN

SAT-Solving mit Hilfe von BDD-Methoden: Leistungsvisualisierung und Analyse

Danail Raykov

Bachelor's Thesis
im Studiengang Informatik

Betreuer: Prof. Jasmin Blanchette

Mentor: M.Sc. Lydia Kondylidou

Ablieferungstermin: 5. Februar 2024

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Munich, 5. Februar 2024 Danail Raykov

Danksagungen
Ein großes Dankeschön möchte ich an Lydia Kondylidou richten, denn durch ihre Vorarbeit und wertvolles Wissen in diesem Gebiet, konnte ich mich schnell im Thema zurecht finden. Bei Fragen wusste Sie immer eine Antwort und konnte mir in jeder Situation lehrreiche Denkanstöße geben. Einen Dank möchte ich auch an Prof. Dr. Jasmin Blanchette richten, indem mir die Möglichkeit geboten wurde, meine Arbeit an Ihrem Lehrstuhl schreiben zu dürfen. Außerdem möchte ich mich noch bei Familie und Freunde bedanken, die mich bei dieser Reise unterstützt haben.

Abstract

A core problem in theoretical computer science and mathematics is the verification of propositional logic formulas for satisfiability. It cannot be directly determined whether a propositional logic formula is satisfiable, posing a fundamental problem to this day. The main goal of the SAT problem is to find an assignment of truth values to the variables of a formula, making it satisfiable. Many modern SAT solvers use algorithms such as the Conflict-Driven-Clause-Learning algorithm (CDCL). To make the process of deciding, whether a formula is satisfiable or not, more efficient, the idea of parallel sat solving has arisen. In particular, sat solvers are running in parallel in certain environments and exchange information. This work is based on a project where a CDCL SAT Solver and a BDD solver were chosen to run in parallel. While BDDs were used as standalone SAT solvers in earlier years, they have been replaced by the CDCL algorithm. Goal of this bachelors project is to collect data and grapically demonstrate the outcome of the two solvers running in parallel, as well as to improve the implementation of parallelization. The analysis of the collected data revealed positive outcomes in several aspects of the parallel computation. An interesting observation from the accumulated tests is the difference in computation times between SAT and UNSAT formulas. Glucose computes satisfiable formulas faster, while parallel computation of BDD + Glucose resolves unsatisfiable formulas more quickly. In summary, the parallel computation of Glucose + BDD is superior to Glucose in certain aspects. Particularly, in the computation of unsatisfiable formulas, the parallel approach outperforms Glucose. However, Glucose has an advantage in satisfiable formulas. The project has shown that parallel computation does not always lead to performance improvement, but a difference is certainly noticeable.

Kurzfassung

Ein grundlegendes Problem in der theoretischen Informatik und Mathematik ist die Überprüfung von aussagenlogischen Formel auf Erfüllbarkeit (SAT Problem). Auf den ersten Blick kann nicht direkt festgestellt werden, ob eine aussagenlogische Formel erfüllbar ist, was bis heute ein grundlegendes Problem darstellt. Das Hauptziel des SAT Problems besteht darin, eine Zuordnung von Wahrheitswerten für Variablen einer Formel zu finden, um sie erfüllbar zu machen. Moderne SAT-Solver verwenden Algorithmen wie den Conflict-Driven-Clause-Learning-Algorithmus (CDCL) um eine Formel zu überprüfen. Um den Prozess der Entscheidung, ob eine Formel erfüllbar ist oder nicht, effizienter zu gestalten, ist die Idee der parallelen Berechnung für das SAT-Problem entstanden. Diese Arbeit basiert auf einem Projekt, bei dem ein CDCL SAT-Solver und ein BDD-Solver ausgewählt wurden, um parallel ausgeführt zu werden. Während BDDs in früheren Jahren als eigenständige SAT-Solver verwendet wurden, sind diese durch den CDCL-Algorithmus ersetzt worden. Das Ziel dieser Bachelorarbeit ist es, Daten zu sammeln und das Ergebnis der beiden parallel laufenden Solver grafisch darzustellen sowie die Implementierung der Parallelisierung zu verbessern. Die Analyse der gesammelten Daten ergab positive Ergebnisse in mehreren Aspekten der parallelen Berechnung. Eine interessante Beobachtung aus den gesammelten Tests ist der Unterschied in den Berechnungszeiten zwischen erfüllbaren und unerfüllbaren Formeln. Glucose berechnet erfüllbare Formeln schneller, während die parallele Berechnung von BDD + Glucose unerfüllbare Formeln schneller auflöst. Zusammenfassend ist die parallele Berechnung von Glucose + BDD in bestimmten Aspekten Glucose überlegen. Insbesondere in der Berechnung unerfüllbarer Formeln übertrifft der parallele Ansatz Glucose. Das Projekt hat gezeigt, dass die parallele Berechnung nicht immer zu einer Leistungsverbesserung geführt hat, aber ein Unterschied bemerkbar ist.

Inhaltsverzeichnis

1 Einleitung

Ein Kernproblem in der theoretischen Informatik und Mathematik ist das Prüfen von aussagenlogischen Formeln auf ihre Erfüllbarkeit. Auf den ersten Blick kann nicht direkt entschieden werden, ob eine aussagenlogische Formel erfüllbar ist. Dies stellt bis heute noch ein grundlegendes Problem dar. Stephen Arthur Cook und L.A. Levin konnten schon im Jahr 1973 unabhängig voneinander beweisen, dass das Erfüllbarkeitsproblem (SAT-Problem) ein NP-Vollständiges Problem ist. Da NP-Vollständige Probleme gleichzeitig in NP und in der NP-Schweren Klasse liegen, lässt sich jedes Problem in NP mithilfe einer deterministischen Turing-Maschine auf das SAT-Problem reduzieren.

Eine aussagenlogische Formel besteht aus Variablen und Klauseln. Eine Klausel enthält eine bestimmte Menge von Literalen, wobei die Literale die Negation oder die positive Affirmation einer Variable symbolisieren. Literale innerhalb von Klauseln sind ausschließlich mit dem ODER Operanden miteinander verknüpft. Das Finden einer Zuordnung von Wahrheitswerten für die Variablen einer Formel, sodass diese erfüllbar wird, stellt das Hauptziel des SAT Problems dar.

SAT ist fundamental für die Berechnung von vielen Problemen, wie die automatische Beweisführung, Robotik, maschinelles Sehen, Datenbanken, Computerarchitektur, Design und viele mehr. Deshalb ist es wichtig, SAT-Formeln effizient lösen zu können, um effiziente Computersysteme zu bauen. In den vergangenen Jahren wurden neue Systeme und Methoden entwickelt, um die Zeitkomplexität zu verbessern. Moderne SAT-Solver verwenden hierfür Algorithmen wie den Conflict-Driven-Clause-Learning (CDCL) Algorithmus . Dieser hat sich in vergangenen Jahren stetig bewiesen und gilt bis heute noch als einer der effizientesten Algorithmen für das SAT-Problem. Neue Methoden verwenden die parallelen Eigenschaften von Prozessoren, um SAT-Solver parallel starten zu lassen. Die Solver tauschen untereinander Daten aus, um die Fortschritte aufzuteilen.

Diese Arbeit baut auf den Forschungsergebnissen von M.Sc. Lydia Kondylidou [Kondylidou (2023)] aus dem Jahr 2023 auf, indem Aspekte der parallelen Berechnung eines CDCL-SAT Solvers mit binären Entscheidungsdiagrammen (BDD) untersucht werden. BDDs wurden schon in früheren Jahren als eigenständige SAT-Solver verwendet, sind aber durch den CDCL-Algorithmus abgelöst worden. Da BDDs sehr fragil auf die Variablenreihenfolge reagieren und das Finden einer optimalen Variablenreihenfolge ein NP-Vollständiges Problem ist, wurden BDDs als Hilfsmittel für den Kernsolver benutzt. Hierfür werden die Formeln in kleinere Teilformeln gespalten, um daraus Konfliktklauseln zu bilden, die dem Kernsolver helfen sollen, eine Lösung effizienter zu finden. Konfliktklauseln enthalten Informationen über gefundene Konflikte und sorgen dafür, dass gleiche Konflikte nicht nochmal durchlaufen werden. Durch eine Umstrukturierung des Klauselaustausches und der Reihenfolge, in welche die beiden Solver starten, soll das bereits existierende Projekt verbessert werden. Die Leistungsunterschiede zwischen Glucose und der parallelen Berechnung von Glucose mit den BDDs wurde visuell dargestellt und untersucht. Als Hilfsmittel wurde eine Python Bibliothek entworfen, welche die gewonnenen Daten verarbeitet. Es soll gezeigt werden, ob die Herangehensweise, dass ein CDCL-SAT Solver parallel mit einem BDD SAT-Solver sowie eine veränderte Klauselübertragung das SAT-Problem effizienter lösen kann. Die aus dieser Arbeit erhaltenen Ergebnisse geben einen Hinweis darüber, ob die Verbindung eines CDCL Solvers mit einem BDD Solver effizienter auf Erfüllbarkeit prüfen kann.

1.1 Überblick

Die Arbeit ist wie folgt strukturiert: Im zweiten Kapitel wird zunächst eine umfassende Einführung in die Aussagenlogik gegeben, wobei erläutert wird, was sie genau ausmacht und wie das Erfüllbarkeitsproblem definiert wird. Des Weiteren wird im darauf folgenden Kapitel die Konstruktion eines Binary Decision Diagrams behandelt. Im vierten Kapitel werden die wesentlichen Bestandteile des Conflict Driven Clause Learning Algorithmus erläutert. Des Weiteren erfolgt eine detaillierte Beschreibung der Funktionsweise des Glucose SAT Solvers sowie der Methoden zur Integration der Binary Decision Diagrams (BDDs) mit dem Glucose SAT Solver. In den Kapiteln fünf und sechs liegen die Schwerpunkte auf den durchgeführten Tests und der visuellen Darstellung der Leistung und Effektivität aus der Kombination von BDD und Glucose. In Kapitel sieben folgt die Schlussfolgerung aus den Ergebnissen.

2 Aussagenlogik und das Erfüllbarkeitsproblem

Logik ist die Wissenschaft des logischen Schlussfolgerns und der Untersuchung von Aussagen auf ihre Gültigkeit. Aussagen aus der realen Welt können in die Welt der Logik übertragen werden. Die daraus folgenden Schlüsse lassen sich anschließend auf ihre Wahrheitswerte überprüfen. Eine vorteilhafte Eigenschaft der Aussagenlogik ist, dass sich die Wahrheitswerte solcher Aussagen leicht mit dem Computer berechnen lassen können. Das Finden einer Belegung für eine Formel, in der jede Teilaussage aus dieser Formel zu wahr auswertet, ist die Grundidee des Erfüllbarkeitsproblems. Jedoch stellt das Erfüllbarkeitsproblem bis heute noch ein ungelöstes Problem in der Informatik und Mathematik dar, obwohl die Aussagenlogik an sich intuitiv ist.

2.1 Grundlagen der Aussagenlogik

Die klassische Aussagenlogik bezieht sich auf Aussagen, denen einen Wahrheitswert, wahr oder falsch, bzw. 1 oder 0 zugewiesen werden kann. Das Alphabet der Aussagenlogik besteht aus (und), also Klammern, den nullstelligen Junktoren Verum ($\top$) und Falsum ($\bot$), der vier binären logischen Verknüpfungen (Konjunktion, Disjunktion, Implikation und Biimplikation) ($\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) und der unären Verknüpfung der Negation ($\neg$). Außerdem verfügt die Aussagenlogik über eine abzählbare Menge an Aussagesymbolen, die mit Indizes identifiziert werden [Biere et al. (2009)].

Die Idee einer Formel leitet sich von folgenden Regeln ab, die es ermöglichen neue Formeln zu bilden:

1. Jede aussagenlogische Variable ist eine Formel.
2. Wenn X eine Formel ist, dann ist $\neg X$ auch eine Formel.
3. Wenn X und Y Formeln sind, dann ist jede binäre Verknüpfung ($\wedge$, $\vee$, $\rightarrow$) auch eine Formel ($X \wedge Y, X \vee Y, X \rightarrow Y$).

Die Sprache der Aussagenlogik ist die Menge aller aussagenlogischen Formeln. Formal kann ein boolischer Ausdruck mit folgender abstrakten Grammatik dargestellt werden:

$\langle \textit{logical-expression} \rangle ::= \langle \textit{variable} \rangle \mid 0 \mid 1 \mid \neg \langle \textit{logical-expression} \rangle \mid \langle \textit{logical-expression} \rangle \wedge \langle \textit{logical-expression} \rangle \mid \langle \textit{logical-expression} \rangle \vee \langle \textit{logical-expression} \rangle \mid \langle \textit{logical-expression} \rangle \Rightarrow \langle \textit{logical-expression} \rangle \mid \langle \textit{logical-expression} \rangle \leftrightarrow \langle \textit{logical-expression} \rangle$

Beispiel eines Aussagenlogischen Ausdrucks:

$$((a \wedge b) \vee c) \Rightarrow d$$

In der Sprache der Aussagenlogik können die Aussagesymbole durch Wahrheitswerte interpretiert und zugewiesen werden. Die Zuweisung passiert durch die Analyse der aussagenlogischen Variablen, indem jeder Variable ein Wahrheitswert zugeteilt wird. Die resultierenden Wahrheitswerte können aus Wahrheitstabellen erschlossen werden. In der Aussagenlogik ist es wichtig zu zeigen, ob eine Formel aus aussagenlogischen Variablen erfüllbar ist. Hierfür hat jede Formel eine dieser drei Eigenschaften:

1. Eine Formel ist allgemeingültig, bzw. eine Tautologie, wenn die Belegung der Variablen, egal für welche Wahrheitswerte, die Formel wahr werden lässt.
2. Eine Formel A heißt erfüllbar, wenn eine Belegung der Variablen existiert, welche die ganze Formel wahr werden lässt.
3. Eine Formel heißt unerfüllbar, wenn keine Belegung der Variablen gefunden werden kann, damit die Formel wahr wird.

Existiert eine Interpretation M für die Wahrheitswerte der Variablen in einer Formel A, so heißt M das Modell von A. Aussagenlogische Formeln können aus der oben genannten Grammatik definiert und erstellt werden und besitzen eine bestimmte Struktur. Diese lassen sich in andere Normalformen umwandeln, wie der Klauselnormalform oder der Disjunktiven Normalform.

2.2 Normalformen

Damit Formeln leichter auf Erfüllbarkeit geprüft werden können, sollten diese in eine bestimmte Form gebracht werden. Einige Beweismethoden die auf Erfüllbarkeit testen, sind nur für Formeln in einer konkreten, eingeschränkten syntaktischen Form gewährleistet. Eine Normalform ist eine Vorgehensweise, eine Formel auf eine bestimmte Weise darzustellen. Das hat den Hintergrund, dass Konflikte bei den Zuweisungen besser erkannt und gelöst werden können.

2.2.1 Konjunktive Normalform

Die konjunktive Normalform oder Klauselnormalform (KNF) ist eine Methode, die eine Formel als eine Konjunktion von Disjunktionen gestaltet. Dabei werden die Disjunktionen als Klauseln von Literalen dargestellt. Ein Literal kann ein aussagenlogisches Symbol oder dessen Negation sein, dem ein Wahrheitswert zugewiesen werden kann. Eine Klausel ist eine Menge von Literalen, die innerhalb einer Klausel mit logischen ”ODER ” aneinandergebunden sind. Die einzelnen Klauseln werden anschließend mit dem logischen ”UND” verknüpft. Außerdem ist es möglich, eine Formel, die in einer anderen Form geschrieben ist, zum Beispiel der disjunktiven Normalform, in die KNF umzuwandeln.

Beispiel einer Formel in KNF:

$$(a \vee b) \wedge (b \vee c) \wedge (\neg a \vee b)$$

2.2.2 Disjunktive Normalform

Die disjunktive Normalform (DNF) ist eine Darstellung, die als Disjunktion von Konjunktionen aufgebaut ist. In einer DNF-Formel finden sich ausschließlich ”ODER ”-Verknüpfungen zwischen den einzelnen Klauseln, wobei die Literale innerhalb der Klauseln durch logische ”UND” miteinander verbunden sind.

Beispiel einer Formel in DNF:

$$(a \wedge \neg b \wedge C) \vee (\neg a \wedge b \wedge c) \vee (\neg a \wedge b)$$

2.3 Erfüllbarkeitsproblem der Aussagenlogik

In der Aussagenlogik kann auf den ersten Blick nicht sofort entschieden werden, ob eine Formel erfüllbar ist. Diese können aus einer großen Anzahl von Klauseln und Literalen bestehen und somit die Komplexität der ganzen Formel erheblich erhöhen. Um die Erfüllbarkeit zu überprüfen, werden diese in der konjunktiven Normalform verwendet. Eine Klausel in der KNF-Formel gilt erst dann als erfüllt, wenn mindestens ein Literal innerhalb dieser Klausel dem Wert ”wahr” evaluiert. Wenn alle Werte innerhalb einer Klausel als ”falsch” ausgewertet werden, gilt die gesamte Klausel als nicht erfüllbar. Es gibt verschiedene Möglichkeiten, ein Modell für eine KNF-Formel zu erhalten.

Eine gängige Methode ist der EinSATz von SAT-Solvern, die eine KNF-Formel als Eingabe erhalten und durch geschickten EinSATz von effizienten und präzisen Algorithmen lösen. Einer dieser Algorithmen ist der Conflict Driven Clause Learning Algorithmus, der in Kapitel vier näher erläutert wird. Außerdem besteht die Möglichkeit, dass mehrere SAT-Solver parallel laufen und in enger Zusammenarbeit ein Model für eine Formel finden. Ein SAT-Solver, der sich gut für die

Unterstützung des CDCL Algorithmus geeinget hat, benutzt binäre Entscheidungsdiagramme um auf Erfüllbarkeit zu prüfen. Diese haben einige Eigenschaften, die den Mechanismus des CDCL-Solvers verbessern können. Die Solver können sich gegenseitig unterstützen, indem Konfliktklauseln ausgetauscht werden, um so den Suchraum für das SAT-Problem weiter einzuschränken. Mehrüber Konfliktklauseln kann in Kapitel 4.7 gelesen werden.

Die zu überprüfende Behauptung lautet, dass ein CDCL-SAT Solver, der parallel mit einem BDD-SAT Solver gestartet wird, schneller zu einem Ergebnis kommt als ein CDCL-SAT Solver, der ohne die Verwendung von BDDs nach einem Modell sucht. Im folgenden Kapitel werden binäre Entscheidungsdiagramme näher erklärt.

3 Binäre Entscheidungsdiagramme

Eine boolesche Formel lässt sich als Binary Decision Diagram darstellen. Dabei ist ein BDD eine Datenstruktur, die insbesondere in der formalen Verifikation verwendet wird. Hierfür werden boolesche Funktionen effizient und kompakt dargestellt. Die Repräsentation einer booleschen Funktion $f : B^n \to B$, wobei $B = \{0,1\}$ über die Menge aller Variablen $X_n = \{X_1, \ldots, X_n\}$ ist ein gerichteter, azyklischer Graph mit der Menge aller Knotenpunkte V. Die Knoten sind in terminale und nicht-terminale (interne) Knoten unterteilt. Solch ein terminaler Knotenpunkt ist mit einer Eins oder Null gekennzeichnet. Jeder nicht-terminale Knotenpunkt v ist durch eine Variable gekennzeichnet, welche in der Menge der Variablen $x \in X_n$ vorhanden ist und besitzt zwei Nachfolger Knoten, low(v) und high$(v) \in V$ entsprechend dazu, ob die Variable zu 0 oder 1 ausgewertet wird. Für ein gegebenes Modell der Variablen wird der Funktionswert durch das Verfolgen eines Pfades vom Wurzelknoten zu einem terminalen Knoten ausgewertet. Für eine gegebene Eingabe $m = (m_1, \ldots, m_n)$ beginnt die Auswertung am Wurzelknoten. An einem nicht-terminalen Knoten v mit der Bezeichnung x_i wird, wenn $m_i = 0$ ist, die ausgehende Kante entsprechend zu low(v) gewählt [Moeinzadeh et al. (2009)].

Ein wichtiges Problem, welches sich in allen Varianten der BDDs zeigt, ist die Empfindlichkeit gegenüber der Variablenreihenfolge. Je nach gewählter Reihenfolge kann dabei die Größe der BDDs entweder linear oder exponentiell stark anwachsen. Das optimale Finden einer Variablenreihenfolge ist wiederum auch ein NP-Vollständiges Problem, welches die Implementierung und Evaluation erschweren kann [Bollig & Wegener (1996)].

Im Folgenden wird erklärt, wie aus einer booleschen Funktion ein BDD errichtet werden kann.

3.1 Shannon Erweiterung

Die Shannon-Erweiterung oder auch Shannon-Expansion genannt, bezieht sich auf ein mathematisches Konzept im Zusammenhang mit der Booleschen Algebra und mit den BDDs. Die Shannon-Erweiterung bietet die Möglichkeit, eine boolesche Funktion in Bezug auf ihre Variablen und deren Werte in kleinere Teilfunktionen zu unterteilen. Formal ausgedrückt kann die Shannon-Erweiterung folgendermaßen dargestellt werden:

$$f = x \vee (f \wedge x) \vee (f \wedge \neg x)$$

Dabei ist f eine boolesche Funktion x eine Variable und $\neg x$ der komplementäre Wert von x. Die Ausdrücke F_x und $F_{\neg x}$ sind die Funktion f mit dem Argument, dass x entweder auf 1 oder auf 0

gesetzt wird. Die Terme f_x und $f_{\neg x}$ können auch als der positive oder negative Shannon-Kofaktor bezeichnet werden [Kondylidou (2023)].

Außerdem gilt: $t = x \rightarrow t[1/x], t[0/x]$. Hier repräsentiert $t[1/x]$ die Funktion t, wenn $x = 1$ gesetzt ist und $t[0/x]$ repräsentiert die Funktion für $x = 0$
Schaut man sich nun folgenden booleschen Ausdruck an: $t = (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$. Wenn die Shannon-Erweiterung basierend auf einer festgelegten Variablenreihenfolge durchgeführt wird, in diesem Fall x_1, y_1, x_2, y_2, erhalten wir folgende Ausdrücke:

$$
\begin{aligned}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{11}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{11} &= x_2 \rightarrow t_{111}, t_{110} \\
t_{000} &= y_2 \rightarrow 0, 1 \\
t_{001} &= y_2 \rightarrow 1, 0 \\
t_{110} &= y_2 \rightarrow 0, 1 \\
t_{111} &= y_2 \rightarrow 1, 0
\end{aligned}
$$

Eine solche Darstellung kann auch als Entscheidungsbaum aufgestellt werden. Gestrichelte Linien im Baum symbolisieren die low-Kanten und die soliden Linien, die high-Kanten, siehe Abbildung 1. Einige dieser Ausdrücke sind identisch miteinander und können somit ersetzt werden. In unserem Beispiel kann t_{110} mit t_{000}, sowie t_{111} mit t_{001} ausgetauscht werden. Wenn im rechten Teilbaum t_{000} für t_{110} und t_{001} mit t_{111} substituiert wird, kann man erkennen, dass t_{00} und t_{11} gleichartig sind und können somit in t_1 t_{11} mit t_{00} ersetzt werden. Wenn auf dieselbe Weise alle identischen Ausdrücke gefunden werden, erhält man ein binäres Entscheidungsdiagramm. Dieser ist dann ein gerichteter azyklischer Graph (DAG) [Andersen (1997)]. Durch Anwendung der Substitutionen und Reduzierung der nicht verwendeten Ausdrücke kann t jetzt wie folgt betrachtet werden:

$$
\begin{aligned}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{00}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{000} &= y_2 \rightarrow 0, 1 \\
t_{001} &= y_2 \rightarrow 1, 0
\end{aligned}
$$

Jede Teilaussage kann als der Knoten eines Graphen angesehen werden. Solch ein Knoten kann entweder ein terminaler oder ein nicht-terminaler Knotenpunkt sein. Ein nicht-terminaler Knoten hat eine niedrige Kante, die dem Ausdruck entspricht, der an das Komplement der Variable gebunden ist und eine hohe Kante, die dem Ausdruck der daran gebundenen Variable entspricht. Ein terminaler Knoten besitzt den Wert 0 oder 1, wenn die aktuelle Variablenbelegung zu falsch bzw. zu wahr auswertet. Da Variablen in den rekursiven Aufrufen bei der Ausdehnung des Ausdrucks immer in derselben Reihenfolge ausgewählt werden, treten die Variablen in denselben Anordnungen auf

allen Pfaden vom Wurzelknoten des binären Entscheidungsdiagramms auf. So ein Diagramm wird auch als geordnetes binäreres Entscheidungsdiagramm bezeichnet [Andersen (1997)].

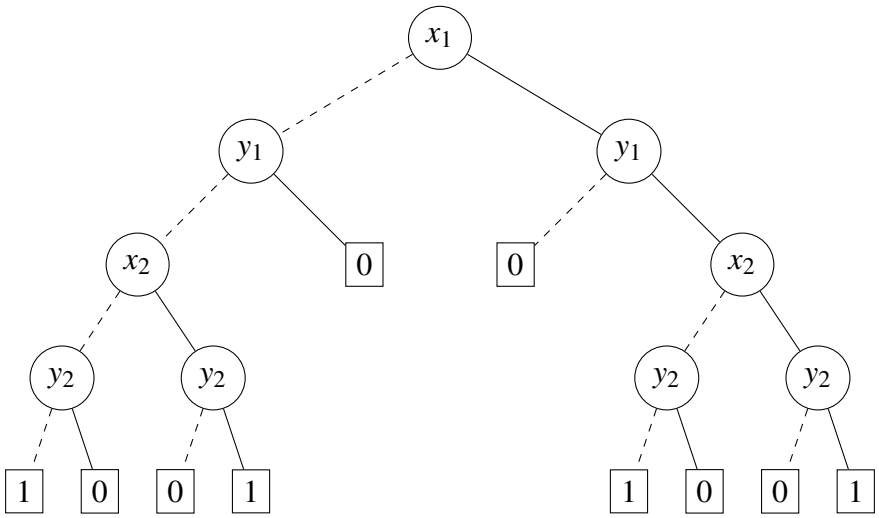


Abbildung 1: Entscheidungsbaum für die Anordnung: x_1, y_1, x_2, y_2

3.2 Reduzierte geordnete binäre Entscheidungsdiagrammme

Wenn zwei unterschiedliche Knoten u und v nicht denselben Variablennamen tragen und dieselben niedrigen und hohen Nachfolger haben sowie kein Knoten identische niedrige und hohe Nachfolger hat, dann spricht man von einem reduzierten geordneten BDD.

Ein Merkmal der ROBDD ist, dass boolesche Ausdrücke kompakt dargestellt werden können und ermöglichen es, alle Arten von effizienten Algorithmen darauf zu verwenden. Außerdem gilt, dass für jede Funktion $f : B^n \to B$ genau ein ROBBD existiert, der diese Funktion repräsentiert [Andersen (1997)].

Die Implementierung der BDDs, die in dieser Arbeit benutzt wurden, sind von Msc. Lydia Kondylidou implementiert worden und verwenden ausschließlich reduzierte geordnete binäre Entscheidungsdiagramme. Diese können als alleinstehende Architektur als SAT-Solver benutzt werden. Da aber ihre Größe exponentiell stark mit der Eingabe der Variablen anwachsen kann und die Variablen Reihenfolge eine kritische Rolle spielt, eignen sich reduzierte geordnete binäre Entscheidungsdiagramm auch als unterstützende Mittel zur Lösung des Erfüllbarkeitproblems. Das bedeutet, dass für die Erfüllbarkeitsberechnung von booleschen Formeln andere Algorithmen verwendet werden müssen, um die Erfüllbarkeit so effizient wie möglich zu berechnen.

4 CDCL SAT Solver

Um den CDCL Algorithmus besser zu verstehen, wird im folgenden Unterkapitel der Davis-Putnam-Logemann-Loveland (DPLL) Algorithmus näher gebracht.

4.1 Ursprung des CDCL Algorithmus

Der DPLL-Algorithmus gilt als Vorreiter des CDCL-Algorithmus. Dieser nimmt eine KNF Formel als Eingabe und erstellt rekursiv ein Modell. Nach jedem Aufruf des Algorithmus wird einer Variable ein Wahrheitswert zugewiesen. In diesem Schritt werden sogenannte Einheitsklauseln bevorzugt, also Klauseln, die nur ein Literal enthalten und denen noch kein Wert zugeteilt wurde. In jedem rekursiven Verzweigungsschritt werden die logischen Auswirkungen mit der dazugehörigen Entscheidung evaluiert. Die ganze Formel wird erst dann erfüllbar, wenn alle Klauseln zu wahr evaluieren. Wenn eine Klausel unerfüllbar ist, also wenn jedem darin enthaltenem Literal ein Wert zugeteilt wurde, sodass die Klausel nicht erfüllt wurde und nur die leere Klausel am Ende bleibt, muss ein sogenanntes Backtracking stattfinden. Backtracking heißt, dass die Entscheidung für das vorherige Literal rückgängig gemacht wird. Diesem Literal wird dann der komplementäre Wahrheitswert von der letzten Entscheidung weitergegeben. Wenn es anschließend wieder zu einem Konflikt kommt, geht man weitere Enscheidungsebene zurück, bis man schlussendlich zu der ersten Entscheidungsebene gelangt. Sollten hier beide Evaluationen für das Literal zu einem Konflikt führen, so ist die komplette KNF Formel unerfüllbar. Solch eine Art von Rückverfolgung nennt sich chronologische Rückverfolgung. Findet man eine Belegung für die Wahrheitswerte für die Literale, sodass alle Klauseln erfüllt sind und alle Variablen einen Wahrheitswert zugeteilt bekommen haben, so ist die ganze Formel erfüllbar [Nieuwenhuis et al. (2006)]. Dieser grundlegenden Idee folgt der CDCL-Algorithmus, welcher neben den genannten Methoden noch einige weitere besitzt.

4.2 Conflict Driven Clause Learning Algorithmus

Einer der erfolgreichsten und effektivsten Algorithmen, der für das Erfüllbarkeitsproblem von booleschen Formeln entwickelt wurde, ist der CDCL-Algorithmus. Dieser ist auf Basis des DPLL-Algorithmus aufgebaut und über die Jahre verbessert worden. Der CDCL-Algorithmus wird als eine Erweiterung des DPLL-Algorithmus angesehen, der mit zusätzlichen Erweiterungen, wie Klauselerlernung und einer effizienten Datenstruktur für die Unit Propagation ausgestattet ist. Resolution spielt eine wichtige Rolle in der Erfüllbarkeit. Zwei Klauseln $c1$ und $c2$ können aufgelöst werden, genau dann, wenn ein Literal l existiert und $l \in c1$ und $\neg l \in c2$.

$c1 - \{l\} \cup c2 - \{\neg l\}$ ist die Resolution in l von $c1$ und $c2$. Dieser Shritt ermöglicht es effizienter Formeln zu durchlaufen, sowie die Klauselgrößen zu verkleinern [Biere et al. (2009)]. Kleinere Klauseln brauchen eine geringere Anzahl an Zuweisungen um auf Einheitsklauseln transformiert zu werden.

4.3 Unit Propagation

Eine wichtige Regel in SAT-Solvern, welche das Finden einer Lösung erleichtert, ist die Regel der Einheitsklausel. Eine solche Klausel enthält nur ein ungelöstes Literal, der zum Wert 1 auswerten muss, damit die Klausel erfüllt wird. Dieser Prozess des automatischen Zuweisens einer Variable wird auch als Unit Propagation bezeichnet. Der Schritt der Unit Propagation wird nach jeder Zuweisung eines Literals und während der Vorverarbeitung eingeleitet. In modernen CDCL-SAT Solvern entstehen die meisten logischen Zusammenhänge in enger Verbindung mit der Unit Propagation. Das Ziel der Unit Propagation ist es, Literale zu finden, denen genau ein Wahrheitswert zugewiesen werden muss, um die Berechnung so effizient wie möglich zu gestalten. Wenn eine unerfüllte

Klausel durch Unit Propagation entdeckt wird, führt dies zu einer Konfliktbedingung, und der Algorithmus versucht durch Zurückverfolgung des Konflikts diesen Konfliktzustand zu lösen.

4.4 Entscheidungsebene

In CDCL SAT Solvern werden jeder Variable eine Reihe von Eigenschaften zugeordnet. Dazu gehören der Wahrheitswert der Variable, der Vorgänger und die Entscheidungsebene der Variable in eine Formel (ϕ). Formal ausgedrückt bedeutet das:

$$\nu(v_i) \in \{0, u, 1\}, \alpha(x_i) \in \phi \cup \text{NIL}, \text{und } \delta(x_i) \in \{-1, 0, 1, \ldots, |X|\}$$

Eine Variable x_i, die als Ergebnis der Unit Propagation einen Wahrheitswert erhält, wird als implizierte Variable bezeichnet. Die Einheitsklausel w, die dazu geführt hat, dass die Variable x_i impliziert wird, wird als Vorgänger von x_i notiert: $x_i, \alpha(x_i) = w$. Entscheidungsvariablen und Variablen, denen noch kein Wert zugeteilt wurde, haben als Vorgänger NIL. Die Entscheidungsebene gibt an, in welcher Tiefe des Entscheidungsbaums die Variable den Wert $\{0, 1\}$ zugewiesen bekommen hat. Die Entscheidungsebene einer noch nicht zugewiesenen Variable x_i hat die Entscheidungsebene $\delta(x_i) = -1$. Die Entscheidungsebene, die mit den für Verzweigungsschritte verwendeten Entscheidungszuweisungen verbunden ist, wird durch den Suchprozess festgelegt und gibt die aktuelle Tiefe des Entscheidungsbuams an. Daher ist eine Variable x_i, die mit einer Entscheidungszuweisung in Verbindung steht, dadurch gekennzeichnet, dass $\alpha(x_i) = \text{NIL}$ ist und $\delta(x_i) > 0$ ist. Formal ausgedrückt, ist die Entscheidungsebene von x_i durch den Vorgänger ω gegeben:

$$\delta(x_i) = \max(\{0\} \cup \{\delta(x_j) \,|\, x_j \in \omega \wedge x_j \neq x_i\})$$

Das bedeutet, die Entscheidungsebene eines implizierten Literals ist entweder die höchste Entscheidungsebene der implizierten Literale in einer Einheitsklausel oder beträgt 0, falls die Klausel eine Einheitsklausel ist. Die Notation $x_i = v@d$ wird verwendet, um auszudrücken, dass $\nu(x_i) = v$ und $\delta(x_i) = d$. Außerdem wird die Entscheidungsebene eines Literals durch die Entscheidungsebene seiner Variable definiert $\delta(l) = \delta(x_i)$, wenn $l = x_i$ oder $l = \neg x_i$ [Marques-Silva et al. (2021)].

Schauen wir uns folgende Beispielformel an:

$$(a \vee b \vee c) \wedge (a \vee e \vee \neg c) \wedge (\neg b \vee d) \tag{1}$$

Nehmen wir an, dass die Zuweisung in der ersten Entscheidungsebene $a = 0@1$ ist. Es entsteht nach dieser Entscheidung keine Einheitsklausel. Auf der zweiten Entscheidungsebene wird $b = 0@2$ zugewisen. Die Regel der Einheitsklausel setzt hier ein und $c = 1@2[\omega 1]$ in der ersten Klausel. Das $[\omega 1]$ symbolisiert, in welcher Klausel die Einheitsklausel entstanden ist. Dadurch dass $c = 1$ wegen Unit Propagation gesetzt wurde, wird die zweite Klausel auch zu einer Einheitsklausel. Schlussendlich evaluiert $e = 1@2[\omega 2]$.

4.5 Implikationsgraphen

Während der Ausführung können zugewiesene Variablen und deren Vorgänger als Implikationsgraphen bzw. azyklische gerichtete Graphen dargestellt werden: $I = (V_i, E_i)$. Ein Knoten im Implikationsgraphen steht für eine Variable, welchem ein Wahrheitswert zugewiesen wurde. Der Graph enthält außerdem einen speziellen Knoten κ, $V_i \subseteq X \cup \{\kappa\}$, wenn ein Konflikt entstanden ist. Die Kanten im Implikationsgraphen werden aus den Vorgängern jeder zugewiesenen Variable gewonnen: Wenn $\omega = \alpha(x_i)$, dann gibt es eine gerichtete Kante von jeder Variable in ω außer x_i zu x_i. Wenn durch Unit Propagation eine unerfüllte Klausel w_j als Ergebnis hervorbringt, so wird die unerfüllte Klausel mit einem speziellen Knoten κ dargestellt. In diesem Fall wird der Vorgänger von κ mit $\alpha(\kappa) = w_j$ definiert [Marques-Silva et al. (2021)].

Nehmen wir an, dass die Literale z, z1, z2 in unserer Menge von Variablen vorhanden sind. Um die Bedingungen für das Vorhandensein von Kanten in I abzuleiten, müssen zunächst eine Reihe von Prädikaten definiert werden. Das Prädikat λ(z, ω) nimmt den Wert 1 an, falls z oder $\neg$z in ω vorhanden ist.

$$\lambda(z,\omega) = \begin{cases} 1 & \text{if } z \in \omega \text{ or } \neg z \in \omega \\ 0 & \text{otherwise} \end{cases}$$

Das folgende Prädikat kann zur Überprüfung des Wertes eines Literals z in einer Klausel ω verwendet werden und gibt den Wert 1 zurück, wenn das Literal 0 ist.

$$\nu_0(z,\omega) = \begin{cases} 1 & \text{if } \lambda(z,\omega) \wedge z \in \omega \wedge \nu(z) = 0 \\ 1 & \text{if } \lambda(z,\omega) \wedge \neg(z \in \omega) \wedge \nu(z) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Im Falle, das ein Literal z in ω den Wert 1 hat, evaluert das Prädikat zu 1.

$$\nu_1(z,\omega) = \begin{cases} 1 & \text{if } \lambda(z,\omega) \wedge z \in \omega \wedge \nu(z) = 1 \\ 1 & \text{if } \lambda(z,\omega) \wedge \neg(z \in \omega) \wedge \nu(z) = 0 \\ 0 & \text{otherwise} \end{cases}$$

Als Resultat erhalten wir eine Kante von z_1 zu z_2, wenn das folgende Prädikat den Wert 1 hat.

$$\varepsilon(z_1,z_2) = \begin{cases} 1 & \text{if } z_2 = \kappa \wedge \lambda(z_1, \alpha(\kappa)) \\ 1 & \text{if } z_2 \neq \kappa \wedge \alpha(z_2) = \omega \wedge \nu_0(z_1,\omega) \wedge \nu_1(z_2,\omega) \\ 0 & \text{otherwise} \end{cases}$$

Die Menge aller vorhanden Kanten kann folgendermaßen dargestellt werden:

$$E_I = (z_1,z_2) \mid \varepsilon(z_1,z_2) = 1$$

[Marques-Silva et al. (2021)]

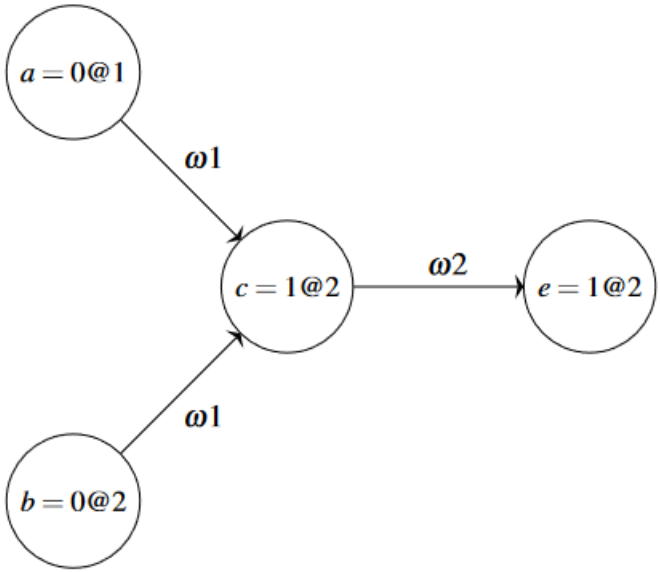


Abbildung 2: Implikationsgraph für Formel 4.4

4.6 Grundaufbau des Algorithmus

Der Algorithmus des CDCL-SAT Solvers sieht folgendermaßen aus:

Algorithm 1: CDCL algorithm

```
Input  : φ - CNF formula, ν - current assignment
Output : SAT or UNSAT
if UnitPropagation(φ, ν) == CONFLICT then // Unit Propagation
    return UNSAT;
dl ← 0 // Decision level
while (not AllVariablesAssigned(φ, ν)) do // Main Loop
    (x, v) = PickBranchingVariable(φ, ν) // Decide stage
    dl ← dl + 1 // Increment decision
    ν ← ν ∪ {(x, v)};
    if UnitPropagation(φ, ν) == CONFLICT then // Deduce stage
        β = ConflictAnalysis(φ, ν) // Diagnose stage
        if β < 0 then // Conflict Analysis Result
            return UNSAT;
        else
            Backtrack(φ, ν, β);
            dl ← β // Decrement decision level due to backtracking
return SAT;
```

Dieser Algorithmus folgt den Paradigmen des DPLL-Algorithmus mit dem Unterschied, dass die ConflictAnalysis aufgerufen wird, wenn ein Konflikt gefunden wurde und danach die

Zurückverfolgung des Konflikts stattfindet. Dies ermöglicht ein nicht chronologisches Zurückverfolgen des Konflikts, um so effizienter nach der Ursache zu suchen. Unitpropagation schaut, ob eine Einheitsklausel entdeckt wurde und löst diese in diesem Schritt auf. Dies wird auch zu Beginn des Algorithmus eingeleitet und falls vor der ersten Zuweisung ein Konflikt gefunden wurde, terminiert der Algorithmus und wertet die Formel als unerfüllbar aus. Eine Konfliktmeldung wird gegeben, wenn eine unerfüllbare Klausel durch Unit Propagation entdeckt wurde. Falls der Konflikt in der ersten Entscheidungsebene gefunden wurde, wird die Formel als unerfüllbar gewertet. PickBranchingVariable wählt eine Variable aus und setzt diesem einen Wert. Falls ein Konflikt entdeckt wurde, wird die conflict analysis aufgerufen, welche als Ergebnis eine neue gelernte Klausel liefert. Die neue Entscheidungsebene nach einem Konflikt wird durch Backtrack zurückgesetzt. AllVariablesAssigned testet, ob alle Variablen einen Wert besitzen. Wenn alle Variablen einen Wert zugewiesen bekommen haben und es zu keinem Konflikt mehr gekommen ist, so wird die Formel als erfüllbar gewertet.

4.7 Klauselerlernung

Nachdem ein Konflikt durch die Unit Propagation gefunden wurde, kommt die Technik der Konfliktanalyse zum EinSATz. Aus der Struktur der Unit Propagation wird entschieden, welche Literale genutzt werden sollen, um eine neue Klausel zu bilden, damit der gleiche Konflikt vermieden wird. Diese neu gebldeten Klauseln werden schließlich in die Formel der schon vorhandenen Klauseln dazu addiert. Außerdem wird in der Konfliktanalyse eine neue Entscheidungseben für die Rückverfolgung berechnet. Die mit den zugewiesenen Variablen verbundenen Entscheidungsebenen definieren eine Anordnung der Variablen. Ausgehend von einer gegebenen unerfüllten Klausel durchläuft das Konfliktanalyseverfahren diejenigen Variablen, die auf der zuletzt getroffenen Entscheidungsebene impliziert wurden. Identifiziert die Vorgänger der besuchten Variablen und behält von den Vorgängern die Literale bei, welche auf Entscheidungsebenen zugewiesen sind, die kleiner sind als die zuletzt getroffene Entscheidungsebene. Dieser Prozess wird wiederholt, bis die zuletzt getroffene Entscheidungsvariable erreicht wurde. Die Prozedur der Klauselerlernung kann durch eine Reihe von Resolutionsoperationen beschreiben werden [Audemard & Simon (2018)].

Sei d die aktuelle Entscheidungsebene, x_i die Entscheidungsvariable, $\nu(x_i) = \mathrm{v}$ die Entscheidungszuweisung und ω_j eine unerfüllte Klausel, die durch Unit Propagation identifiziert wurde. In Bezug auf den Implikationsgraphen ist der Konfliktknoten κ so beschaffen, dass $\alpha(\kappa) = \omega_j$. Darüber hinaus steht das Symbol $\odot$ für den Resolution-Operator. Für zwei Klauseln ω_j und ω_k, bei denen es eine eindeutige Variable x gibt, so dass eine Klausel das Literal x und die andere das Literal $\neg x$ enthält, so enthält $\omega_j \odot \omega_k$ alle Literale von ω_j und ω_k mit Ausnahme von x und $\neg x$.

$$\xi(\omega, l, d) = \begin{cases} 1 & \text{if } l \in \omega \wedge \delta(l) = d \wedge \alpha(l) \neq \text{NIL} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Gleichung 2 beschreibt, ob eine Klausel ω ein impliziertes Literal l auf der aktuellen Entscheidungsebene d hat [Marques-Silva et al. (2021)].

$$\omega_{d,i}L = \begin{cases} \alpha(\kappa) & \text{if } i = 0 \\ \omega_L^{d,i-1} \odot \alpha(l) & \text{if} i \neq 0 \text{ and } \xi(\omega_L^{d,i-1}, l, d) = 1 \\ \omega_L^{d,i-1} & \text{if } i \neq 0 \text{ and } \forall l \xi(\omega_L^{d,i-1}, l, d) = 0 \end{cases} \tag{3}$$

Der Prozess der Klauselerlernung wird durch die Gleichung 3 formalisiert. Die erste Bedingung $i = 0$ führt den ersten Zwischenschritt der Klauselerlernung an, indem alle Literale aus der unerfüllten Klausel zu einer Zwischenklausel geformt werden. Anschließend wird in jedem Schritt i ein Literal l gewählt, welches auf der aktuellen Entscheidungsebene d zugewiesen ist. Die erhaltene Zwischenklausel wird dann mit dem Vorgänger von l aufgelöst. Wenn keine Resolution mehr durchgeführt werden kann, wird die Zwischenklausel zu einer gelernten Klausel und wird in der Formel der noch zu lösenden Klauseln hinzugefügt [Marques-Silva et al. (2021)].

Die Verbesserung der Effizienz und der Leistungssteigerung ist einer der Hauptgründe, warum gelernte Klauseln gebildet werden, denn durch gelernte Klauseln, lassen sich bekannte Konflikte vermeiden. Ein weiterer Grund, weshalb gelernte Klauseln eine entscheidende Rolle spielen, ist die mögliche Unterstützung anderer SAT-Solver, die mit gleichen KNF Formeln arbeiten. Diese können die gelernten Klauseln nutzen, um schon durchgegangene Konflikte zu vermeiden. Dies führt in erster Linie zu einer Verbesserung und einer effizienteren Lösung des Erfüllbarkeitsproblem. Zusätzlich ermöglichen gelernte Klauseln auch die Identifizierung einer Teilmengen von Klauseln, die ebenfalls unerfüllbar sind.

4.8 Überblick des Glucose SAT Solvers

Dieser Prozess des Klauselaustausches wurde auch in dieser Arbeit verwendet. Eine in Rust implementierte BDD Datenstruktur wurde mit Glucose, einem CDCL-SAT-Solver in Verbindung gebracht. Das Ziel war es, den CDCL-SAT-Solver in seiner Berechnung zu unterstützen, indem die BDDs die gleiche KNF als Input bekommen und daraus gelernte Klauseln hervorbringen. Diese Klauseln wurden anschließend in die Formel des Glucose SAT-Solvers hinzuaddiert, sodass der CDCL-Solver schon gelernte Konflikte aus den BDDs vermeidet. Der von Gilles Audemard und Laurent Simon implementierte CDCL-SAT Solver Glucose [Audemard & Simon (2018)] ist einer der nominiertesten SAT Solver in der heutigen Zeit. Neben den üblichen Funktionen eines jeden CDCL-SAT Solvers, wurden in Glucose Methoden verbunden und entwickelt, um den Prozess so effizient wie möglich zu halten. Ein wichtiger Indikator, den Glucose ausmacht, ist das aggressive Löschen von neu gelernten Klauseln, die als nicht bedeutsam eingestuft werden. Auch wenn gelernte Klauseln notwendig sind, um schon erkannte Konflikte zu vermeiden, sollten nicht verwendete Klauseln gelöscht werden, um so den Prozess der Unit Propagation nicht zu verlangsamen. Deshalb ist es wichtig, eine geeignete Methode zu finden, die gute von schlechten Klauseln unterscheidet. Hierfür wurde in Glucose die Literal Block Distance (LBD) implementiert, eine Heuristik welche die Aktivität der neu gelernten Klauseln auswertet. Die Bewertung findet dabei direkt nach der Bildung der Klausel statt und evaluiert dabei die Wichtigkeit der Klausel, um eine mögliche Löschung zu verhindern. Damit der Solver sich nicht unnötig lange am gleichen Problem aufhält und zum Stillstand kommt, gibt es in Glucose die Möglichkeit, den Algorithmus neu zu starten, sowie schon gelernte Klauseln zu löschen. Wenn die Evaluation der LBD, von den neue gebildeten Klauseln einen bestimmten Wert überschreitet, treten in Glucose Neustarts auf, um die Bildung von guten Klauseln zu begünstigen. Auch wenn die Klauseleliminierung und die Neustarts wichtig sind, um schlechte Klauseln zu entfernen, kann es vorkommen, dass wichtige Klauseln entfernt werden, die es ermöglicht hätten, die Erfüllbarkeit der Formel zu zeigen [Audemard & Simon (2018)].

In der neuesten Version von Glucose wurde deshalb die Möglichkeit hinzugefügt, Neustarts zu verschieben. Eine blockierung des Neustarts wird erst dann eingeleitet, wenn eine große Anzahl

Zuweisungen in kurzer Zeit stattfinden, ohne dass es zu einem Konflikt kommt.

5 Integrierung binärer Entscheidungsdiagramme mit Glucose

Um die Kommunikation zwischen Glucose und den BDDs zu ermöglichen, war es notwendig, beide Architekturen miteinander zu verbinden. Die BDDs und die dazugehörigen Methoden und Funktionen in diesem Projekt wurden von Kondylidou [Kondylidou (2023)] implementiert. Der Unterschied zwischen der neuen und ursprünglichen Implementierung liegt in der Reihenfolge, in welche die Solver die Berechnung durchführen, sowie der Klauselaustausch beider Solver. Die Gundidee lag darin, die Solver sequenziell zu implementieren, aber wegen komplexen Formeln und langer Berechnungsschritte, ist die Idee der parallelen Berechnung für das SAT-Problem entstanden. Threads ermöglichen diese Parallelität, und sorgen dafür, dass beide Solver unabhängig voneinander Berechnungen durchführen können. Dadurch dass beide Solver nicht auf den Datenaustausch angewiesen sind, können längere Berechnungsschritte von beiden Solvern durchgeführt werden.

Zu Beginn der Berechnung sind die BDDs kleiner und werden mit der Zeit komplexer, da größere Teile der Formel analysiert werden. Gleichzeitig dauert es länger, bis neue Konfliktklauseln gefunden werden. Durch eine parallele Berechnung, können die anfangs kleinen und informativen Klauseln schneller erstellt und übertragen werden. Kleine Klauseln sind deswegen informativer, da weniger Entscheidungen gemacht werden müssen, damit daraus eine Einheitsklausel entsteht und somit durch Unit Propagation aufgelöst werden kann. In welchem Status sich die Solver befinden, wird untereinander nicht ausgetauscht. Stattdessen wurde sichergestellt, dass wenn in Glucose die Suche nach einem Konflikt erfolgreich war, die gefunden Klauseln aus den BDDs in diesem Schritt hinzuaddiert werden. Das hat versichert, dass Glucose sich in einem Status befinden, in dem die Klauseln problemlos eingefügt werden können. Der Ablauf einer Lösungsfindung lief so ab, dass Glucose als Hauptprogramm gestartet wird und während Glucose schon die ersten Berechnungen durchlaufen hat, die Konfliktklauseln aus den BDDs in die Formel der gefundenen Konfliktklauseln eingefügt werden.

Die ursprünglich implementierte Methode der Klauseltransferierung besaß die Eigenschaft, dass eine Glucose Instanz aus den BDDs gestartet wird, sowie eine Schnittstelle zwischen den beiden Solver erstellt wurde. Diese Schnittstelle war nach der Umstrukturierung nicht mehr notwendig und der EinSATz von Pointern hat die Übertragung der Konfliktklauseln verbessert. Nachdem Glucose mit der Berechnung begonnen hat, wurden in den BDDs schon die ersten gelernten Konfliktklauseln gebildet. Das heißt, es werden nur die Klauseln aus den BDDs genommen, die zu einem Konflikt geführt haben. Der Grundgedanke bei dieser Methode ist, dass Glucose die gleichen Konflikte nicht mehr durchlaufen muss und somit effizienter auf ein Ergebnis kommt.

BDDs sind effizient, wenn die Formeln nicht zu groß sind. Ein aufteilen der Formel in kleinere Teilformeln, hat eine gute und schnelle Konstruierung der Diagramme ermöglicht. CDCL-Solver sind gut darin, große und komplexe Formeln zu lösen und durch die Zusammenarbeit beider Solver, besteht die Möglichkeit, dass die Berechnung der Erfüllbarkeit dadurch effizienter gestaltet wird.

5.1 Parallelität

Parallelität bedeutet, dass mehrere Aufgaben gleichzeitig ausgeführt werden. Parallelität in der Programmierung bezeichnet die gleichzeitige Ausführung von mehreren Teilen eines Computer-

programms, um die Effizienz und Leistung zu steigern. In der Erfüllbarkeitsberechnung, bedeutet Parallelität, dass zwei oder mehr Solver an einer Lösung für eine gegebene Formel arbeiten. Hierfür ist ein Informationsaustausch beider Solver notwendig, damit die gelernten Daten untereinander ausgetauscht werden können. Der Austausch der Daten kann über verschiedene Wege geschehen. Einer davon ist, dass gelernten Klauseln aus den Solvern, in eine gemeinsame Klauseldatenbank geladen werden. So eine Datenbank dient als Zugriffspunkt, indem die Solver ihre Fortschritte dokumentieren. Ein anderer Weg, der in dieser Arbeit verwendet wurde, ist, dass die Daten aus den BDDs direkt in den parallel mitlaufenden Glucose SAT-Solver integriert werden. Das entscheidende hierbei ist, dass der Solver, in diesem Fall die BDDs, unabhängig Konflikt Klauseln generieren können. Das bedeutet, dass die Solver untereinander nicht angewiesen sind. Um die Klauseln von einem Solver zum anderen zu schicken, kann eine Klauseldatenbank verwendet werden oder mithilfe einer direkten Verbindung von Pointern im internen Speicher der Solver übertragen werden.

5.2 Threads

Damit zwei Programme gleichzeitig laufen können, wird die Funktionalität von Prozessoren in modernen Computern verwendet. Diese bieten die Möglichkeit, dass Threads erstellt werden können. Das hat den Vorteil, wenn einer der SAT-Solver länger für die Berechnung braucht, im Hintergrund die Auswertung des anderen SAT-Solvers weiter ausgeführt wird. Das grundlegende Ziel von Threading ist es, Programme zu gestalten, die mehrere Aufgaben gleichzeitig und effizient lösen können. Außerdem ist es wichtig, dass Glucose unabhängig und ohne Unterbrechung von den BDDs an einer Lösung arbeiten kann. Threads eigenen sich hierfür am besten, da sie die Auslastung eines Prozessors optimieren können. Hierfür wurde im Glucose Code der Thread für die BDDs gestartet. Das hatte den Grund, dass Glucose die Aufgabe hatte, die Lösung der KNF-Formel zu liefern und die BDDs hauptsächlich nur Teile der ganzen Formel betrachtet haben und dementsprechend nur die gefundenen Konfliktklauseln aus den Teilformeln an Glucose übermittelt haben.

5.3 Generierung der Konfliktklauseln in den BDDs

Nachdem Gluose eine DIMACS Datei als Input erhalten hat, gibt dieser die gleiche Datei an die BDDs weiter. Eine DIMACS Datei enthält eine Text basierte Repräsentation einer KNF Formel. Die BDDs werden dann mithilfe einer gemeinsamen Bibliothek initialisiert und eine Variablenanordnung für die erhaltene Formel wird zeitgleich miterstellt. Da die Formeln in den meisten Fällen sehr groß sind und das Finden einer guten Variablenreihenfolge entscheidend ist, wird nicht die komplette Formel als BDD dargestellt, sondern nur einzelne Teile davon. Da die Klauseln miteinander konjugiert sind, können in Teilformeln gefundene Konflikte auf die ganze Formel angewendet werden. Die gelernten Konflikte werden dann an Glucose übermittelt und im internen Speicher der schon gelernten Klauseln gespeichert.

5.4 Einbindung von Konfliktklauseln

Um die Klauseln aus den BDDs an Glucose zu übergeben, wurde der Thread für die BDDs in Glucose integriert. Weitere Methoden, wie die Initialisierung der Klauseldatenbank werden hier gestartet. Da die neu gelernten Klauseln zu lang werden können oder Duplikate sind, müssen

diese einen Filter passieren. Erst dann dürfen die Klauseln in den Vektor der gelernten Klauseln hinzugefügt werden.

Während Glucose schon die ersten Konflikte findet, laufen die BDDs im Hintergrund und bilden unabhängig davon eigene Konfliktklauseln. Die Übertragung des Vektors mit den gefundenen Konfliktklauseln wurde mithilfe von Pointern auf den internen Speicher ermöglicht. Dies hat den Vorteil, dass keine externe Schnittstelle zwischen den beiden Solvern notwendig ist und somit schneller darauf zugegriffen werden kann. Nachdem die einzelnen Klauseln in Glucose übertragen werden, ist der nächste Schritt, die Klauseln in den Vektor der gelernten Konfliktklauseln von Glucose zu addieren. Um die Klauseln einzubinden wurden Methoden benutzt, die schon in Glucose vorhanden waren. Nach der Einbindung läuft der Solver normal weiter mit dem Unterschied, dass neue Klauseln dazugekommen sind. Ob Glucose nach einer gewissen Zeit die Klauseln löscht oder behält, hängt von der berechneten Wichtigkeit der einzelnen Klauseln ab. Da Glucose sehr aggressiv mit der Löschung von Klauseln umgeht, kann sich das Behalten der BDD-Klauseln als schwierig erweisen [Audemard & Simon (2018)]. In dieser Arbeit konnte Glucose die Löschung der BDD Klauseln vornehmen, da ein zu starkes Eingreifen und das Aufrechterhalten von eventuell schlechten Klauseln den Solver beeinträchtigen könnte.

6 Daten- und Leistungsauswertung

Für die Arbeit war es notwendig, die Auswirkungen aus der Vereinigung beider SAT-Solver hervorzuheben und visuell darzustellen. Es ist zu zeigen, dass durch den Austausch der Konfliktklauseln, Glucose schneller zu einer Lösung kommt. Eine zu hohe Dichte an neu gelernten Klauseln oder eine Anhäufung an schlechten Klauseln könnte den Solver in Suchräume lenken und zu einer Verlangsamung der Berechnung führen. Allerdings könnten die übermittelten Konfliktklauseln den Solver in die richtige Richtung lenken und so die Berechnung verbessern. Es kann nicht von Anfang an gesagt werden, ob die Verbindung der beiden SAT-Solver zu einer Leistungssteigerung führt. Insgesamt wurden 510 KNF Formeln ausgewertet und die wichtigsten Ergebnisse zusammengefasst. Es wurden 266 ”hard combinatorial” und 244 Application Test genommen. Hard combinatorial Dateien sind für den SAT-Solver nicht einfach zu berechnen, aber dennoch relativ überschaubar in ihrer Formelgröße. Application Tests sind große Dateien, die aufgrund ihrer Formelgröße an Komplexität gewinnen. Für die Auswertung der erhaltenen Ergebnisse wurde eine Python Script entwickelt, welches die erhaltenen Daten auswertet und visuell darstellt.

6.1 Python Bibliothek

Für die visuelle Darstellung der Daten fiel die Wahl auf Python. Python ist eine Sprache, die es ermöglicht, große Datenmengen einfach und unkompliziert zu verarbeiten. Für die Erstellung von Plots und der Datenverarbeitung von einigen Millionen Datenpunkten war Pythons Performance immer noch überzeugend. Zur Erstellung der Diagramme wurde Matplotlib verwendet, eine Bibliothek zur Visualisierung von Datenmengen. Die Auswertung der Daten geschah erst dann, nachdem alle Tests durchgeführt wurden und alle Logdateien vorhanden waren.

6.2 Benchmarks

Um zu zeigen, dass die Vereinigung beider SAT-Solver einen Einfluss auf die Leistung hat, war es wichtig, genügend Tests mit verschiedenen Formeln durchlaufen zu lassen. Die nötigen DIMACS-Dateien stammen aus der offiziellen Seite der SAT Competition. Der Veranstaltungsgrund für diesen Wettbewerb ist das Identifizieren von herausfordernden Testfällen, das testen von neuen SAT-Solvern und sowie die Förderung von neuen Lösungsansätzen zur Erfüllbarkeitsberechnung. Die Ergebnisse aus diesem Wettbewerb sind ein guter Indikator für die aktuelle Umsetzbarkeit der Erfüllbarkeitsberechnung. In der Arbeit wurden die Hard Combinatorial, sowie die Applicationtests aus unterschiedlichen Jahren verwendet. Die Solver haben hierfür ein Zeitlimit von 15 Minuten pro Datei, um auf ein Endergebnis zu kommen. Falls das Zeitfenster überschritten wird, ist ”unbewertet” als Ergebnis einzutragen.

Getestet wurde auf einer Maschine mit einem Intel i5-7600K CPU und einer NVIDIA GeForce GTX 1060 Grafikkarte. Die Test verliefen folgendermaßen: Zu Beginn der Tests liefen die Hard Combinatorials und Application Tests nur auf Glucose, ohne der parallelen Berechnung von den BDDs. Als Vergleich wurden die gleichen KNF Formeln in Glucose in Verbindung mit den BDDs gestartet. Hierfür wurde ein selbst geschriebenes Python Script verwendet, welches Glucose mit dem Pfad der DIMACS Datei als Argument aufruft. Pro Datei wurde eine Instanz von Glucose gestartet. Im Vorfeld kann der Code modifiziert werden, sodass Glucose entweder mit oder ohne den BDDs gestartet wird. Die Daten für die Auswertung wurden schon während Glucose läuft gesammelt, um so den Verlauf des Solvers nachvollziehen zu können.

Die Anzahl der Konfliktliterale, Konflikte, Unit Propagationen, Neustarts, Entscheidungen, geblockte Neustarts, Reduzierungen der Datenbank und die benötigte Zeit für die Berechnung wurde für jede Formel dokumentiert. Nach jeder Berechnung oder Überschreitung des Zeitlimits wurden die gesammelten Daten in eine Logdatei für die jeweilige Formel gespeichert. Außerdem wurde eine eigene Struktur der Datenspeicherung überlegt, indem nach der Terminierung des Solvers die wichtigsten Informationen in der ersten Zeile der Logdatei gespeichert werden. Name der gelösten Formel, die benötigte Zeit und ob die Datei erfüllbar war oder nicht, werden in dieser ersten Zeile zusammengeführt. Nach der ersten Zeile folgen die genau erfassten Daten der einzelnen Datensätze mit dem dazugehörigen Zeitstempel, um so die Entwicklungen im Solver genauer darzustellen.

Da die Logdateien viel Speicherplatz verbrauchen, ist es notwendig, genügend Ressourcen zur Verfügung zu stellen. Außerdem ist es wichtig, dass die Testmaschine, die dafür verwendet wird, nicht zu schwach ist, da große oder komplizierte Formeln öfter zum Timeout kommen. Außerdem sollte die Maschine nur für die Testläufe laufen, da sich fremde Einwirkungen negativ auf die Leistung des Computers und auch des SAT-Solvers auswirken können. Um einen Vergleich und eine Veränderung zwischen Glucose und BDD + Glucose zu sehen, wurden zwei Arten von Tests durchgeführt.

6.2.1 Application Tests

Für die erste Testreihe wurden KNF-Formeln genommen, die aufgrund ihrer Formelgröße ein Problem darstellen. Insgesamt wurden 244 Formeln von dieser Art in Glucose getestet. Die gleichen Formeln wurden auch in Glucose in Verbindung mit den BDDs ausgeführt. Im folgenden Diagramm wird die Anzahl an gelösten und ungelösten Formeln von Glucose ohne BDDs gezeigt. Es ist zu sehen, wie viele Formeln als 'indeterminate' bzw. unbestimmt ausgewertet sind. Unbestimmt bedeutet, dass eine Formel in einem gegebenen Zeitlimit zu keinem Ergebnis gekommen ist, heißt die Formel wurde weder als erfüllbar noch unerfüllbar ausgewertet.

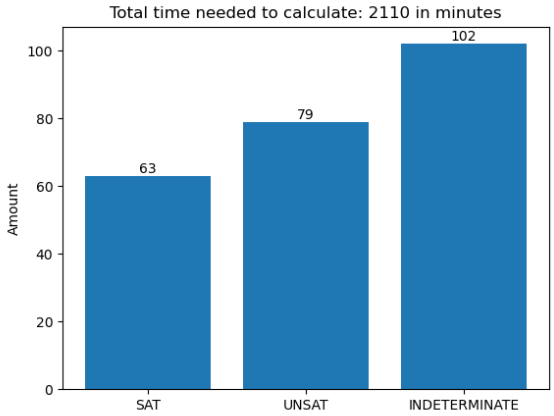


Abbildung 3: Anzahl der Erfüllbaren und unerfüllbaren Formeln von Glucose

In diesem Diagramm ist zu sehen, dass Glucose 63 Formeln als erfüllbar, 79 als unerfüllbar und 102 als unbestimmt evaluiert hat. Insgesamt hat die Berechnung der Formeln 2110 Minuten gedauert. Glucose hat hier eine relativ gleichmäßige Verteilung von erfüllbaren und unerfüllbaren Formeln errechnet. Das bedeutet, dass Glucose sowohl die Erfüllbarkeit als auch die Unerfüllbarkeit konsistent überprüfen kann. Im GegenSATz dazu werden im nächsten Diagramm die Daten aus der Verbindung von Glucose und den BDDs gezeigt:

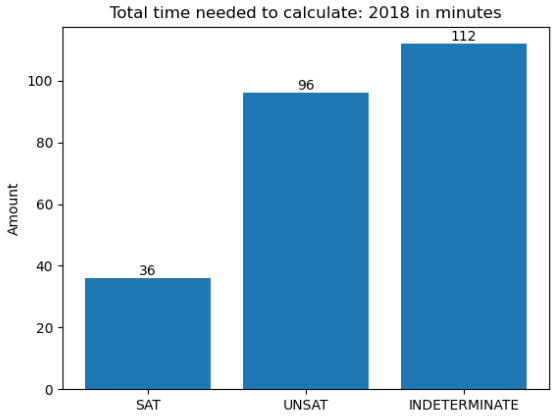


Abbildung 4: Anzahl der Erfüllbaren und unerfüllbaren Formeln von Glucose + BDD

Die Summe der erfüllbaren, unerfüllbaren und unbestimmten Formeln liegen bei 36, 96 und 112. Es ist zu erkennen, dass Glucose + BDD weniger Formeln als erfüllbar und mehr als unerfüllbar ausgewertet hat. Auch die Anzahl an unbestimmten Formeln ist leicht angestiegen. Die Verbindung der beiden Architekturen hat bessere Ergebnisse für die Unerfüllbarkeit von Formeln geliefert. Interessant zu sehen ist, dass Glucose + BDD 92 Minuten weniger Zeit in Anspruch genommen hat, auch wenn die Anzahl der unbestimmten Formeln höher liegt. Bedeuted jedoch, dass Glucose mehr Formeln gelöst hat als die Vereinigung der beiden Solver. Auch wenn der Zeitunterschied überschaubar ist und Glucose + BDD eine erhöhte Anzahl an unbestimmten Formeln erzielt hat, kann das als leichte Verbesserung gegenüber dem Glucose Solver angesehen werden. Da die Frage auf Erfüllbarkeit jedoch eine größere Rolle spielt, muss hier angemerkt werden, dass Glucose, auch wenn die verwendete Zeit höher lag, öfter zu einem Ergebnis kam als BDD + Glucose. Der Unterschied ist zwar gering, aber auf die Frage der Erfüllbarkeit von aussagenlogischen Formeln liefert Glucose ein besseres Ergebnis. In Hinsicht auf die Gesamtdauer hat Glucose + BDD besser abgeschnitten.

Um einen Überblick der aufgewendeten Zeit zu erhalten, wurden Cactus Plots für die Combinatorial und Application Tests erstellt. Cactus Plots sind Abbildungen, welche die Effizienz von SAT Solvern darstellen sollen. Auf der X-Achse kann abgelesen werden, wie viele Dateien der Solver gelöst hat. Auf der Y-Achse ist die verwendete Zeit für die Lösung der Formel beschrieben. Als erstes wird der Cactus Plot für die Application Tests gezeigt:

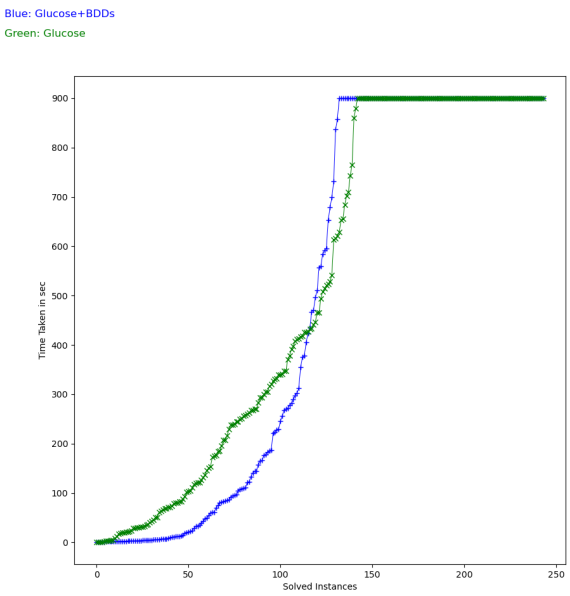


Abbildung 5: Cactus Plot Application Files

In diesem Diagramm wurden die Tests der gebrauchten Zeit nach sortiert und visuell dargestellt. Wie man im Balkendiagramm der Applicationtests von Glucose + BDD sehen kann, hat Glucose + BDD weniger Zeit in Anspruch genommen als Glucose. Außerdem ist zu erkennen, dass in diesem Diagramm die Kurve von Glucose + BDD einige Zeit lang unter der Kurve von Glucose lag, dann jedoch rapide angewachsen ist und über die Kurve von Glucose gelandet ist. Das hat den Grund, dass Glucose für einige Formeln ein Ergebnis erbracht hat und BDD + Glucose dies nicht getan

hat. Dass Glucose in einer höheren Anzahl an Formeln länger Zeit gebraucht hat als die parallele Berechnung, weist darauf hin, dass die Vereinigung der beiden Solver eine verbesserte Wirkung auf das Lösen der Erfüllbarkeit zeigt.

6.2.2 Combinatorial Tests

Als Vergleich zu den Applicationtests, wurden die Combinatorialtests ausgewertet. Diese sind schwieriger zu berechnen, da die erhöhte Formelkomplexität den Solver verhindern können. Hierfür wurden 266 combinatorial KNF Formeln ausgewertet. Im folgenden Abbild ist das Ergebnis aus besagten Formeln von Glucose zu sehen.

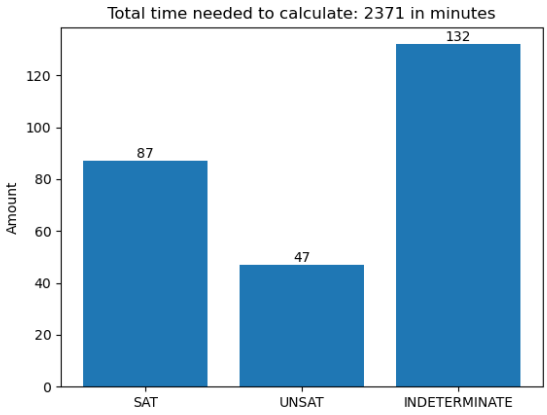


Abbildung 6: Anzahl der Erfüllbaren und unerfüllbaren Formeln - Combinatorials Glucose

Im Vergleich zu den Application Test, hat Glucose bei diesen Formeln fast die doppelte Menge an erfüllbaren als unerfüllbaren Formeln gefunden. Bei beiden Tests kann man sehen, dass Glucose gut darin ist, erfüllbare Formeln zu erkennen. Bei den Combinatorials befindet sich der Wert der erfüllbaren Formeln deutlich höher als der Wert der unerfüllten Formeln. Wie BDD + Glucose abgeschnitten hat, sieht man an der folgenden Abbildung:

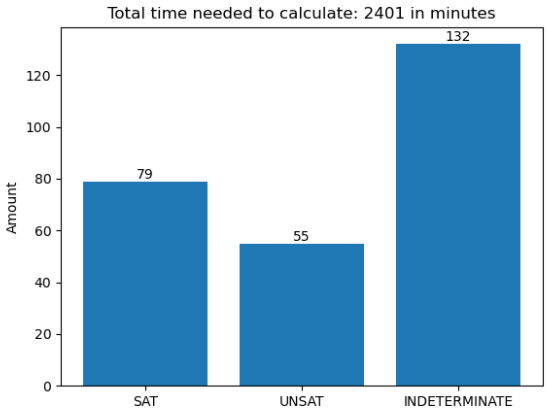


Abbildung 7: Anzahl der Erfüllbaren und unerfüllbaren Formeln - Combinatorials Glucose + BDD

Als erstes kann man erkennen, dass BDD + Glucose länger gebraucht hat, die gleichen Formeln zu lösen. Die erhöhte Komplexität der Formeln, sowie die Addierung von zusätzlichen Klauseln in Glucose sind ausschlaggebend dafür, ob Glucose in nicht optimale Suchräume gelenkt wird und der Berechnungsprozess dadurch verlangsamt wird. Die aufgewendete Zeit ist jedoch nah beieinander und eine Verbesserung des BDD Algorithmus könnte bessere Klauseln produzieren, um Glucose in bessere Suchräume zu lenken und somit den Prozess beschleunigen. Mit der jetzigen Implementierung ist Glucose und Glucose + BDD in einigen Aspekten schneller, aber dennoch ausgeglichen. Bei diesen Tests haben beide Solver ungefähr die gleiche Anzahl an erfüllbaren, unerfüllbaren und unbestimmten Formeln ausgewertet.

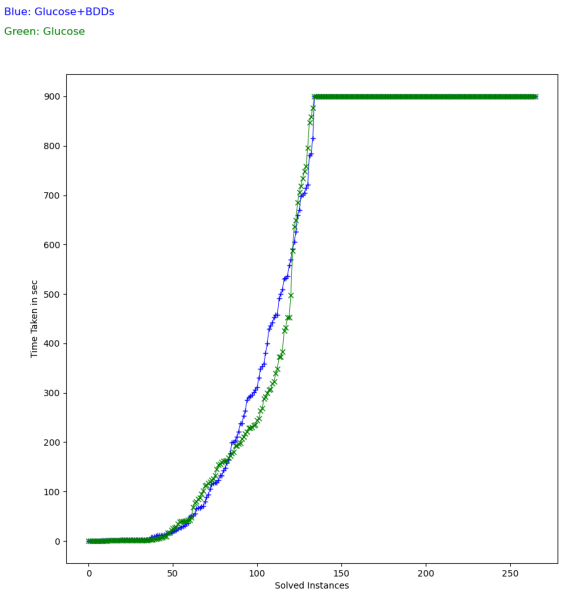


Abbildung 8: Combinatorials Cactus Plot

Schaut man sich nun den Cactus Plot 8 von den Combinatorial Daten an, so kann man erkennen, dass der Unterschied zwar vorhanden ist, aber marginal ist. Beide Solver sind sich dabei ebenbürtig. Es lässt sich nicht genau sagen, welcher Solver hier besser ist.

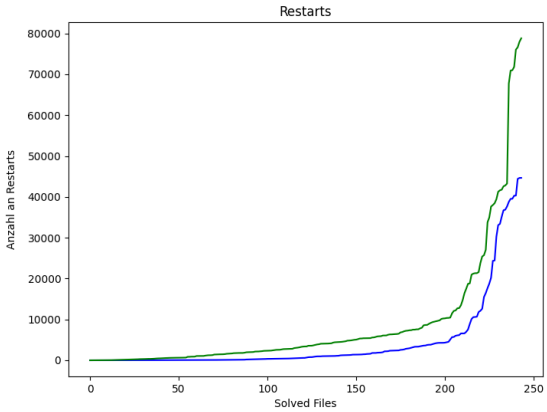
6.3 Interne Datenerhebungen

Um der Frage nachzugehen, warum ein Unterschied beider Solver vorhanden ist, wurden auch während der Berechnung Daten gesammelt. Sowohl die Anzahl an geblockten und nicht geblockten Neustarts als auch Entscheidungen, Konflikte und Unit Propagations können Hinweise darauf geben, warum beide Solver verschiedene Ergebnisse liefern. Eine Auswertung des Durchschnitts über die ganzen gesammelten Daten Anzahl dieser Daten zeigen, dass die Vereinigung beider Solver einen Einfluss auf die Berechnung der Formeln hat.

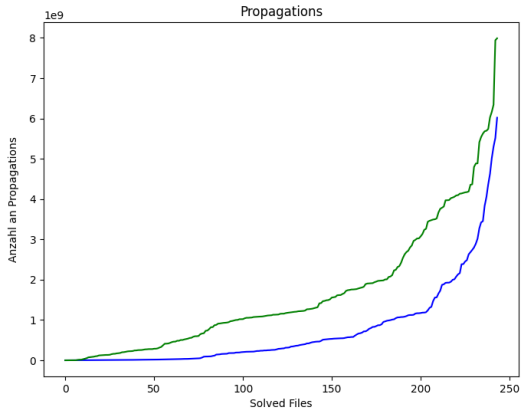
	BDD+Glucose	Glucose
Restarts	4,356	8,593
Conflicts	1,217,732	2,609,360
Decisions	13,354,373	20,960,420
Conflict Literals	86,781,530	252,381,202
Blocked Restarts	3,515	7,929
Reduced Databases	21	36
Propagations	724,798,804	1,643,925,945

Abbildung 9: Durchschnitt Application Tests

In 9 ist zu erkennen, dass BDD+Glucose in jedem Aspekt fast um die Hälfte weniger Daten produziert hat als Glucose. Dies spiegelt sich auch bei bei der benötigten Zeit wieder, die BDD + Glucose gebraucht hat, um die gleiche Menge an Formeln zu lösen.



(a) Anzahl an Restarts

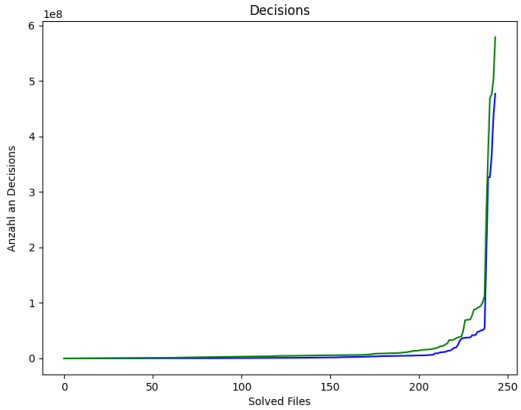


(b) Anzahl an Propagations

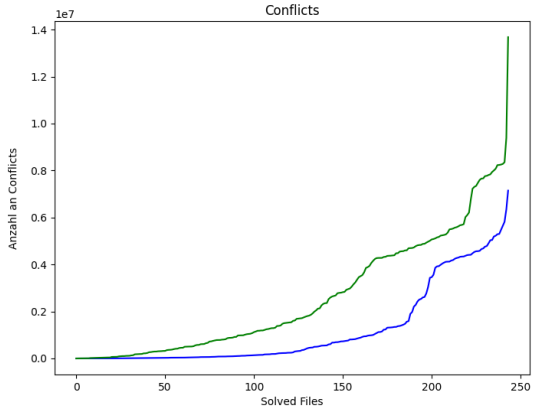
Ein Neustart erfolgt in der Regel, wenn der Solver auf Schwierigkeiten stößt oder nicht in der Lage ist, eine Lösung zu finden. Alle errechneten Heuristiken werden beibehalten und die Berechnung fängt an einen neuen Punkt an, um mögliche Verlangsamung durch schlechte Suchräume zu vermeiden. Dafür werden die Abhängigkeiten zwischen den einzelnen Variablen neu geordnet. Ein SAT-Solver, der weniger Neustarts benötigt, könnte als robuster betrachtet werden, da schwierige Situationen besser bewältigt werden [Audemard & Simon (2018)]. Schwierige Konfliktsituationen könnten schon durch die BDDs berechnet worden sein. Somit musste Glucose + BDD seltener den Berechnungsprozess beenden und neu anfangen. Ein Solver mit weniger Neustarts könnte effizienter sein, da weniger Zeit mit dem erneuten Starten von Lösungsversuchen verbraucht wird und stattdessen kontinuierlich an der aktuellen Lösung weitergearbeitet wird.

Schaut man sich die Anzahl der Unit Propagations 10b an, kann man erkennen, dass die durchschnittliche Menge der Unit Propagations des BDD + Glucose Solvers, die Hälfte der von Glucose ist. Die Konfliktklauseln aus den BDDs sorgen dafür, dass schon gelernte Konflikte der Solver nicht noch mal berechnen muss. Deswegen ist auch die Zahl der Propagations bei BDD +

Glucose geringer, um eine Formel als erfüllbar oder unerfüllbar zu werten. Je weniger Propagations gemacht werden, um ein endgültiges Ergebnis zu erhalten, desto früher konnte der Solver die Unerfüllbarkeit bzw. Erfüllbarkeit einer Formel bestätigen.



(a) Anzahl an Decisions



(b) Anzahl an Propagations

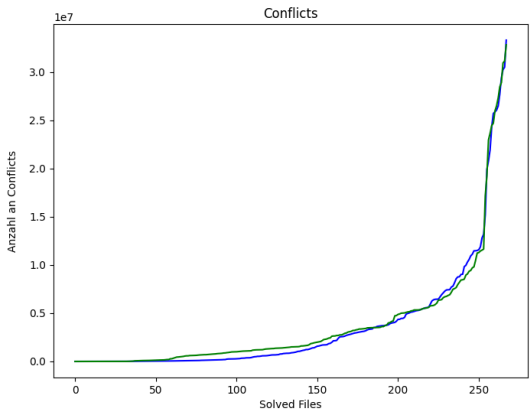
Da die Propagations auch in Verbindung mit den Decisions stehen, ist es nur logisch, dass je weniger Entscheidungen getroffen werden, desto seltener wird auch die Unit Propagation aufgerufen und weniger Neustarts sind erforderlich. Da bei den Applicationtests die Unerfüllbarkeit häufiger als Ergebnis aufgetreten ist, konnte der Solver mithilfe der gelernten Konflikte aus den BDDs schneller Konflikte finden, die auf der ersten Ebene des Implikationsgraphen lagen und somit unwichtige Berechnungen auslassen. Dies wirft jedoch die Frage auf, warum die Menge der gefunden erfüllbaren Formeln so gering ist. Die hinzugefügten Klauseln aus den BDDs, erhöhen nach jeder neu hinzugefügten Klausel die Komplexität der Formel, der Solver verwendet die neu gelernten Klauseln und betritt dabei neue Suchräume. Weil die Unerfüllbarkeit in erfüllbaren Formeln nicht nachgewiesen werden kann, verlängert sich die Suche nach einem Modell. Im Falle der Application Tests, konnte der Solver viele Formeln als unerfüllbar einordnen, da die gelernten Klauseln die Suchräume soweit eingeschränkt haben, dass die Konflikte, die daraus entstanden sind, zur Unerfüllbarkeit geführt haben.

Das bedeutet, dass je weniger Conflicts, decisions und Propagations durchgeführt werden, der Solver schneller zu einem Ergebnis kommt. Schaut man sich nun die Plots der Propagations, Decision, Conflicts und Restarts an, kann man erkennen, dass die Linie von BDD+Glucose unter der Linie von Glucose liegt. Wenn die verbrauchte Zeit von den ungelösten Formeln von der Gesamtdauer subtrahiert wird, so liegt der Wert für Glucose bei 580 Minuten und für Glucose + BDD bei 338 Min. Auch wenn Glucose + BDD weniger Formeln gelöst hat, liegt die Gesamtdauer der gelösten Formeln fast bei der Hälfte von dem, was Glucose gebraucht hat.

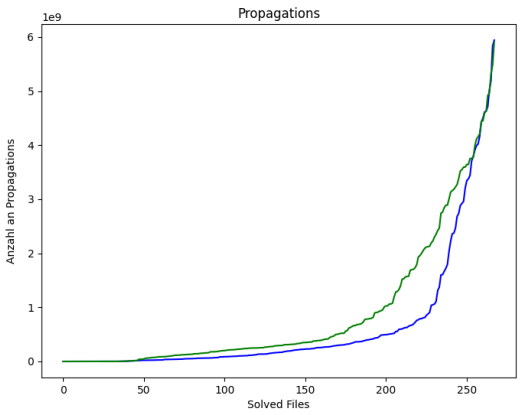
	BDD+Glucose	Glucose
Restarts	5,511	6,513
Conflicts	3,511,578	3,755,092
Decisions	14,791,277	15,819,218
Conflict Literals	201,770,829	366,219,694
Blocked Restarts	10,559	11,399
Reduced Databases	35	42
Propagations	618,646,246	890,287,376

Abbildung 12: Durchschnitt Combinatorial Tests

Bei den Combinatorialtests, lag der Unterschied nicht weit auseinander. Dies spiegelte sich auch bei den erhobenen Daten während der Berechnung wieder. Die durchschnittlichen Werte für die erhobenen Daten lagen sehr nah beieinander. Der Unterschied in der Gesamtdauer lag bei 30 Minuten. Auch wenn beide Solver die gleiche Anzahl an ungelösten Formeln ergeben haben, hat in diesem Fall Glucose + BDD schlechter in der Zeit abgeschnitten. Jedoch liegen die erhobenen Daten 12 für Glucose + BDD immer noch unter denen von Glucose. Ein Trend, der sich in diesen Formeln zeigt, ist dass diese eine größere Anzahl an erfüllbaren Formeln hatten. Sowohl bei den Application als auch bei den Combinatorial Tests, hat Glucose + BDD mehr Formeln als unerfüllbar bewertet als Glucose. Es ist anzunehmen, dass diese Art von paralleler Berechnung besser für unerfüllbare Formeln geeignet ist. Glucose + BDD hat bei den Applicationtests deutlich mehr Formeln als unerfüllbar berechnet. Die Gesamtdauer lag niedriger als bei Glucose, welcher eine gleichmäßigere Verteilung an erfüllbaren und unerfüllbaren Formeln erbracht hat. Dahingegen konnten Glucose + BDD bei den Combinatorialtests in weniger Formeln die Unerfüllbarkeit prüfen. Die Gesamtdauer der Berechnungen lag höher als bei Glucose. Somit ist anzunehmen, dass die Vereinigung beider Solver bei unerfüllbaren Formeln bessere Ergebnisse erzielen als bei erfüllbaren.



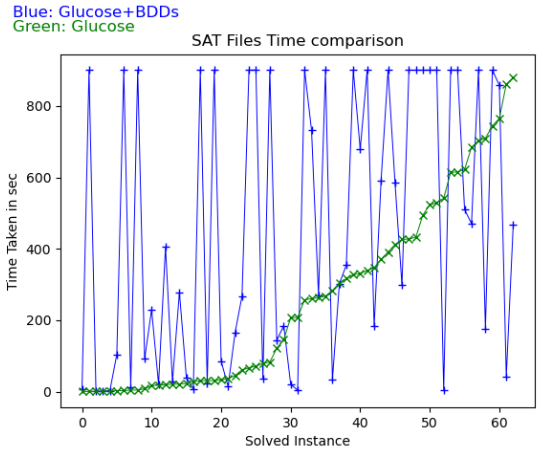
(a) Conflicts Combinatorials



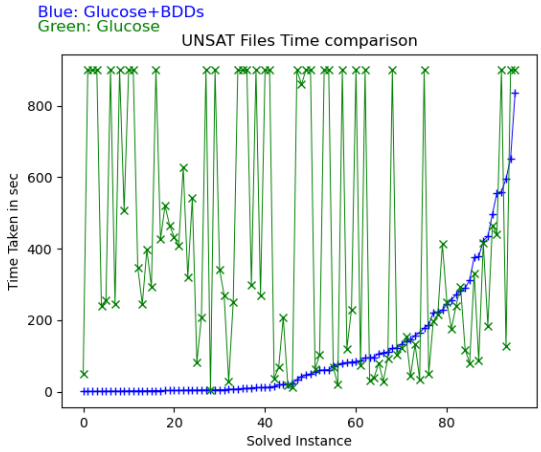
(b) Propagations Combinatorials

Vergleicht man den Graphen der Anzahl an Conflicts 13a, so ist zu erkennen, dass die beiden Kurven sehr identisch aussehen. Tabelle 12 biete schon einen Überblick darüber, wie die Plots zu

diesen Daten aussehen könnten. Die Unterschiede sind marginal und nicht aussagekräftig. Nur bei den Propagations lässt sich ein kleiner Unterschied bemerken. Hier hat Glucose + BDD überwiegend weniger Propagations durchgeführt. Bei den restlichen Kategorien sehen die Plots alle sehr identisch aus und lassen keine Interpretation offen. Nichtsdestotrotz wurden weniger Daten in Glucose + BDD produziert, auch wenn die Gesamtdauer der Berechnung höher lag, als wenn Glucose allein läuft. Es ist zu vermerken, dass die Verbindung aus zwei SAT Solvern einen positiven Effekt erbracht hat. Auch wenn die Leistung bei den Combinatorial Test nicht die von Glucose überschritten wurde, konnten in den Application Tests deutliche Verbesserungen erkannt werden. In komplexeren Formeln konnte Glucose + BDD nicht die erhofften Ergebnisse erzielen, doch die Vereinigung beider Solver kann bei bestimmten Formeln zu einer Verbesserung der Erfüllbarkeitsberechnung beitragen.

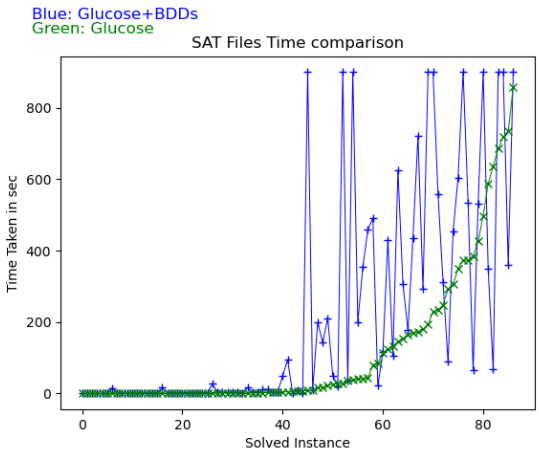


(a) Application Tests, SAT Files Vergleich

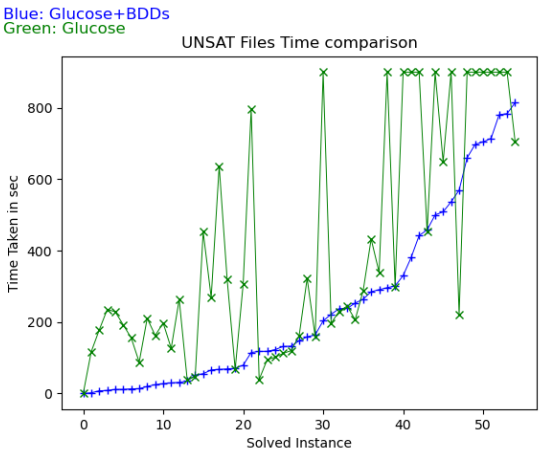


(b) Application Tests, UNSAT Files Vergleich

Abbildung 14: Application Tests, Zeitunterschiede der SAT und UNSAT Formeln



(a) Combinatorial Tests, SAT Files Vergleich



(b) Combinatorial Tests, UNSAT Files Vergleich

Abbildung 15: Combinatorial Tests, Zeitunterschiede der SAT und UNSAT Formeln

Eine interessante Beobachtung, die sich nach den gesammelten Tests gezeigt hat, ist der Unterschied in den Berechnungszeiten der SAT und UNSAT Formeln. Zum einen kann abgelesen werden, dass Formeln die erfüllbar sind, von Glucose schneller berechnet wurden. Zum anderen wurden die unerfüllbaren Formeln aus der parallelen Berechnung schneller gelöst. Eine Mögliche Ursache dafür, könnte in der Natur der Konfliktklauseln liegen. Werden mehr Konfliktklauseln in den Solver hinzuaddiert, so stößt der Solver in unerfüllbaren Formeln schneller zu neuen und wichtigen Konflikten. Die Eliminierung von Suchräumen aus den Konfliktklauseln von den BDDs, kann Glucose effizienter über die Formel führen und so schneller endgültige Konflikte zur Unerfüllbarkeit finden. Umgekehrt bedeutet das aber, dass durch die Anzahl der hinzugefügten Konfliktklauseln in den erfüllbaren Formeln, Glucose öfter in ungünstige Suchräume gelenkt wird und deswegen eine Verlangsamung der Modellfindung stattfindet. Diese Erkenntnis zeigt, dass die Verbindung der beiden SAT-Solver besser für unerfüllbare Formeln geeignet ist. Glucose hingegen, kann die Erfüllbarkeit ohne die BDDs besser überprüfen. Wenn Anzeichen entdeckt werden können, welcher Solver für eine bestimmte Formel effizienter ist, dann würde das SAT-Problem weiter reduziert werden. Einen solchen Mechanismus zu finden ist jedoch schwierig, da von Anfang an entschieden werden muss, ob eine Formel Anzeichen auf Erfüllbarkeit bzw. Unerfüllbarkeit gibt. Eine mögliche Verbesserung die nicht getestet wurde, könnte der EinSATz von Glucose, welcher parallel mit der Berechnung von Glucose + BDD läuft. Wenn entweder Glucose oder Glucose + BDD zu einem Ergebnis kommt, stoppen alle Solver und das Ergebnis des zuerst terminierten Solvers wird übernommen. Die Anzahl an blocked Restarts kann außerdem genutzt werden, um zu entscheiden ob der Solver auf Erfüllbarkeit oder Unerfüllbarkeit prüft, um somit zwischen der parallelen Berechnung von BDD + Glucose und nur Glucose zu entscheiden.

7 Schlussfolgerung

Die vorliegende Arbeit konzentrierte sich darauf, die Auswirkungen der parallelen Berechnung des SAT-Problems mit zwei unterschiedlichen SAT Solvern zu zeigen. Die Verbindung aus einem BDD-Solver, der nur die gefundenen Konfliktklauseln an einen CDCL-SAT Solver übermittelt, wurde analysiert und visuell dargestellt. Die Analyse der erhaltenen Daten zeigte, dass diese Art der parallelen Berechnung in einigen Aspekten positiv aufgefallen ist. Vor allem in den Application Tests waren die Unterschiede deutlich zu erkennen. Die erhobenen Datenmengen, siehe 9, ist in der parallelen Berechnung in fast allen Aspekten geringer als bei Glucose. Eine geringere Menge an Daten führt zu einer effizienteren Ausführung des Algorithmus. Dies trägt dazu bei, die Laufzeit des Systems zu optimieren und die Gesamtleistung des Algorithmus zu verbessern. Jedoch muss an dieser Stelle erwähnt werden, dass Glucose in diesen Tests weniger Formeln mit Zeitüberschreitungen hatte. Außerdem kann gesagt werden, dass BDD + Glucose in 4 besser auf Unerfüllbarkeit überprüft hat. Der Unterschied in der Anzahl an gefunden erfüllten und unerfüllten Formeln lag bei 36 zu 96. Übertragene Konfliktklauseln haben die Eigenschaft, dass Konflikte, die Glucose erst später in seiner Berechnung gefunden hätte, früher angetroffen werden. Dies sorgt dafür, dass Formeln effizienter auf Unerfüllbarkeit geprüft werden können. Aber warum die Anzahl der gefundenen erfüllbaren Formeln so gering ist, lässt sich nicht genau sagen. Die aggressive Löschung von Glucose sowie die erhöhte Komplexität durch die hinzugefügten Konfliktklauseln könnte den Solver in ungünstige Suchräume lenken. In erfüllbaren Formeln kann die Unerfüllbarkeit nicht belegt werden, deswegen könnte eine weniger optimierte Zuweisung der Variablen und das

Aufhalten in schlechten Suchräumen ausschlaggebend dafür sein, dass das Finden eines Modells sich schwieriger erweist. Nichtsdestotrotz war dies ein erfolgreicher Test, um die Leistungssteigerung gegenüber Glucose zu zeigen.

Im zweiten Test wurden komplexere Formeln verwendet, die durch ihre Struktur den Schwierigkeitsgrad erhöhen. Hier liegen die Ergebnisse beider Methoden nah beieinander. Glucose hat bei diesen Tests Zeit gebraucht, die Formeln auszuwerten. Dennoch hat Glucose + BDD bei diesen Tests während der Berechnungen wieder weniger Daten erhoben, auch wenn der Unterschied bei diesen Formeln nicht so groß wie bei den Application Tests war. Diese waren für Glucose schwieriger zu lösen. Dies kann man an den Tabellen 9 und 12 erkennen, da die Anzahl der Propagations niedriger, die gefundenen Conflicts höher und die Blocked Restarts deutlich angestiegen sind. Weniger Propagations bedeutet, dass weniger Einheitsklauseln generiert wurden und somit ein Großteil der Berechnungszeit der Zuweisung von Variablen geschuldet ist. Eine höhere Ansammlung an Blocked Restarts reflektiert die Summe an gefundenen erfüllbaren Formeln. Die Balkendiagramme 6 und 7 zeigen, dass ungefähr die gleiche Anzahl an erfüllbaren Formeln gefunden wurde. Wenn viele Entscheidungen in kurzer Zeit getroffen werden, ohne dass ein Konflikt entsteht, dann befindet sich der Solver in einem Suchraum, der in den meisten Fällen nah an einer Lösung für die Erfüllbarkeit ist. Deshalb kann es passieren, dass der Fortschritt durch einen Neustart in solch einem Suchraum verloren geht. Deswegen werden in bestimmten Fällen Neustarts blockiert. Blocked Restarts sind also ein Indiz dafür, dass der Solver in kurzer Zeit viele Entscheidungen für Variablen trifft, die zu keinem Konflikt führen. Dies spiegelt sich auch in der Anzahl der gefundenen erfüllbaren Formeln wieder. In den Combinatorial Tests hat Glucose + BDD wieder mehr Formlen als unerfüllbar bewertet als Glucose. Da aber in diesen Tests eine überwiegende Menge an erfüllbaren Formeln vorhanden war, ist es Glucose + BDD schwerer gefallen, diese zu berechnen, was man auch bei der Gesamtdauer ablesen kann.

Eine interessante Beobachtung, die sich nach den gesammelten Tests gezeigt hat, ist der Unterschied in den Berechnungszeiten der SAT und UNSAT Formeln. In den Abbildungen 14 und 15 kann abgelesen werden, dass Formeln die erfüllbar sind, von Glucose schneller berechnet weurden. Andererseits, wurden die unerfüllbaren Formeln aus der parallelen Berechnung schneller gelöst. Eine Mögliche Ursache dafür, könnte in der Natur der Konfliktklauseln liegen. Werden mehr Konfliktklauseln in den Solver hinzuaddiert, so stößt der Solver in unerfüllbaren Formeln schneller zu neuen Konflikten. Die Eliminierung von Suchräumen aus den Konfliktklauseln, kann Glucose effizienter über die Formel führen und so früher endgültige Konflikte zur Unerfüllbarkeit finden. Umgekehrt bedeutet das aber, dass durch die Anzahl der hinzugefügten Konfliktklauseln in den erfüllbaren Formeln, Glucose öfter in schlechtere Suchräume gelenkt werden kann und deswegen eine Verlangsamung der Modellfindung stattfindet. Dies zeigt, dass die Vereinigung der beiden Solver besser für unerfüllbare Formeln geeignet ist. 14 15 Wenn man all diese Punkte zusammenfasst, kann gesagt werden, dass die parallele Berechnung von Glucose + BDD gegenüber Glucose in einigen Aspekten besser ist. Vor allem in der Berechnung von unerfüllbaren Formeln ist Glucose + BDD überlegen. Für erfüllbare Formeln ist dennoch Glucose im Vorteil. Das Projekt hat gezeigt, dass die parallele Berechnung nicht immer zu einer Leistungssteigerung geführt hat, aber ein Unterschied definitiv zu vermerken ist.

8 Code und Implementierung

Der verwendete und geschriebene Code kann auf https://github.com/kondylidou/CDCL-support-by-BDD-methods/tree/raykov/bachelorarbeit. Der verwendete Code zur Visualisierung ist im Branch raykov/bachelorarbeit zu finden. Führ nähere Details, ”starter.py” File anschauen.

Abbildungsverzeichnis

Literatur

ANDERSEN, HENRIK REIF. 1997. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen* 5.

AUDEMARD, GILLES, und LAURENT SIMON. 2018. On the glucose sat solver. *International Journal on Artificial Intelligence Tools* 27.1840001.

BALBACH, FRANK J. 2023. The cook-levin theorem. *Archive of Formal Proofs*.

BIERE, ARMIN; MARIJN HEULE; und HANS VAN MAAREN. 2009. *Handbook of satisfiability*, Ausg. 185. IOS press.

BIRRELL, ANDREW D. 1989. *An introduction to programming with threads*. Digital Systems Research Center.

BOLLIG, BEATE, und INGO WEGENER. 1996. Improving the variable ordering of obdds is np-complete. *IEEE Transactions on computers* 45.993–1002.

FITTING, MELVIN, und RICHARD L MENDELSOHN. 2023. *First-order modal logic*, Ausg. 480. Springer Nature.

GU, JUN; PAUL W PURDOM; JOHN FRANCO; und BENJAMIN W WAH. 1996. Algorithms for the satisfiability (sat) problem: A survey. *Satisfiability problem: Theory and applications* 35.19–152.

HILTON, PR HALMOS PJ; R REMMERT; und B SZOKEFALVI-NAGY. 1968. Ergebnisse der mathematik und ihrer grenzgebiete.

KLABNIK, STEVE, und CAROL NICHOLS. 2023. *The rust programming language*. No Starch Press.

KONDYLIDOU, LYDIA. 2023. Supporting a cdcl sat solver by bdd methods. URL `https://kondylidou.github.io/assets/pdf/Master-Thesis-Kondylidou.pdf`.

LEE, EDWARD A. 2006. The problem with threads. *Computer* 39.33–42.

MARQUES-SILVA, JOAO; INÊS LYNCE; und SHARAD MALIK. 2021. Conflict-driven clause learning sat solvers. *Handbook of satisfiability*, 127–145. IOS press. URL `https://www.cs.princeton.edu/~zkincaid/courses/fall18/readings/SATHandbook-CDCL.pdf`.

MOEINZADEH, HOSSEIN; MEHDI MOHAMMADI; HOSSEIN PAZHOUMAND-DAR; ARMAN MEHRBAKHSH; NAVID KHEIBAR; und NASSER MOZAYANI. 2009. Evolutionary-reduced ordered binary decision diagram. *2009 third asia international conference on modelling simulation*, 142–145.

NIEUWENHUIS, ROBERT; ALBERT OLIVERAS; und CESARE TINELLI. 2006. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)* 53.937–977.

PIECHA, THOMAS. Philosophische logik: Ausgewahlte themen.

SCHULER, RAINER. 2005. An algorithm for the satisfiability problem of formulas in conjunctive normal form. *Journal of Algorithms* 54.40–44.

SILVA, JOAO P MARQUES, und KAREM A SAKALLAH. 1996. Grasp-a new search algorithm for satisfiability. *Iccad*, Ausg. 96, 220–227.

VAN ROSSUM, GUIDO, und FRED L DRAKE. 2003. *An introduction to python*. Network Theory Ltd. Bristol.

Akronyme

BDD Binary Decision Diagram. 1, 2, 5–7, 13–27

CDCL Conflict Driven Clause Learning. 1, 5–9, 11, 13, 14, 26

DPLL Davis–Putnam–Logemann–Loveland. 7, 8, 11