Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Masterarbeit Informatik

Implementierung und Evaluierung von Algorithmen zur Berechnung von präferierten Diagnosen

Konstantin Grupp

14. Dezember 2015

Erstgutachter
Prof. Dr. Wolfgang Küchlin
Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Zweitgutachter
Prof. Dr. Peter Hauck
Arbeitsbereich Diskrete Mathematik
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Betreuer
Rouven Walter, M.Sc. Informatik
Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Grupp, Konstantin:
Implementierung und Evaluierung von Algorithmen zur Berechnung von präferierten Diagnosen
Masterarbeit Informatik
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 18. Juni 2015 - 18. Dezember 2015

Zusammenfassung

Bei vielen komplexen Produkten verwendet man SAT Solver zur Überprüfung der Konfiguration, so auch bei der Automobilkonfiguration. Diese liefern allerdings nur die Antwort, ob eine Konfiguration gültig ist oder nicht. Ist eine Konfiguration nicht gültig, dann interessiert den Kunden nur eines: Was kann ich ändern, damit meine Konfiguration gültig wird?

Es gibt bereits Algorithmen, die auf diese Frage eine Antwort geben können. Diese berücksichtigen allerdings keine Präferenz und sind somit in manchen Fällen unbrauchbar. Welcher Kunde aus einem heißen Land, beispielsweise Spanien, möchte auf seine Klimaanlage verzichten? Für ihn hat diese eine hohe Präferenz. Hingegen möchte ein Kunde aus Finnland eher nicht auf ein Anti-Blockier-System verzichten. Diese Beispiele beschreiben die Notwendigkeit, effiziente Algorithmen zur Berechnung von präferierten Diagnosen zu entwickeln. Exakt das untersucht diese Arbeit und analysiert dazu mehrere Algorithmen und Verbesserungen. Deren Effizienz wird durch einen umfangreichen Benchmark untermauert. Im Ergebnis liefert diese Arbeit Algorithmen, welche zudem die notwendige Geschwindigkeit für die praktische Anwendung liefern.

Danksagung

Ich möchte mich bei allen bedanken, die mich bei meiner Arbeit sowohl inhaltlich als auch moralisch unterstützt haben.

Dabei richte ich einen besonderen Dank an Prof. Dr. Wolfgang Küchlin aus dem Arbeitsbereich Symbolisches Rechnen, bei dem ich diese Masterarbeit abfassen durfte. Des Weiteren möchte ich mich bei Prof. Dr. Peter Hauck dafür bedanken, dass er sich zur Erstellung des Zweitgutachtens bereit erklärt hat.

Ebenso großer Dank geht an meinen Betreuer Rouven Walter, der mich mit vielfältigem wissenschaftlichen Knowhow und Tipps unterstützt hat.

Natürlich möchte ich mich auch explizit bei meiner Familie, meiner Freundin Melina und meinen Freunden für die Rücksichtnahme und Aufbauarbeit in den letzten sechs Monaten bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation und Zielsetzung

Zur Konfiguration von Kraftfahrzeugen oder anderen komplexen Produkten spielen präferierte Diagnosen eine große Rolle, um dem Kunden das bestmöglichste Produkt bieten zu können [WFK15b]. Ein Kunde hat bei der Auswahl seines Kraftfahrzeugs viele Optionen. Angefangen beim Motor, über die Farbe der Sitze bis hin zum Getränkebecherhalter ist theoretisch alles nach Kundenwunsch konfigurierbar. Aus technischer Sicht sind allerdings nicht alle Kombinationen möglich. Für jedes Fahrzeug gibt es daher sogenannte Baubarkeitsbedingungen. Anhand dieser Bedingungen kann man bestimmen, ob eine Konfiguration möglich ist. Die Komplexität dieser Bedingungen nimmt mit jedem neu entwickelten Automobil zu und kann daher heutzutage nicht mehr händisch überprüft werden. Um diese Bedingungen zu überprüfen, formuliert man diese als eine aussagenlogische Formel, die sogenannte Produktübersichtsformel. Mittels effizienter SAT Solving Algorithmen, wie dem CDCL-Algorithmus, kann dann geprüft werden, ob die gewählte Konfiguration möglich ist [SKK00].

Diese Algorithmen geben allerdings nur die Information zurück, ob die Konfiguration möglich ist oder nicht. Für den Kunden ist es aber ebenso interessant, welche Konfiguration denn seinem Wunsch am nächsten kommt, falls der Kundenwunsch nicht baubar ist. Er will möglichst wenig Wünsche aufgeben und hat in der Regel zudem noch eine Präferenz. So sind für ihn einige Wünsche wichtiger als andere, diese wird in Form einer totalen Ordnung ausgedrückt. Zum Beispiel wird ein Kunde in Spanien die Klimaanlage bevorzugen und eher die Sitzheizung aufgeben. Mit der dadurch notwendigen Beachtung der Präferenz wird das Problem komplexer. Die Lösung des Problems bezeichnet man als *präferierte Diagnose*.

Bisher gibt es nur wenige Algorithmen zur Lösung des Problems. Ein Algorithmus ist hinreichend geeignet, wenn er schnelle Lösungen liefert. Dies ist für die Beratung eines Kunden unabdingbar, denn nur damit können dem Kun-

den direkt Anpassungen seines Wunsches vorgeschlagen werden. Diese Arbeit analysiert verschiedene Vorschläge zur Berechnung solcher präferierter Diagnosen und evaluiert zusätzliche Optimierungen. Deren Effizienz wird von einem umfangreichen Benchmark validiert. Im Ergebnis kann so die Berechnung um mehrere Größenordnungen beschleunigt werden.

1.2 Gliederung

Die Arbeit gliedert sich in sechs Bereiche: Zunächst führt Kapitel 2 in die notwendigen Grundlagen ein. Dort wird die Aussagenlogik erklärt, erläutert mit welchen Algorithmen aussagenlogische Formeln gelöst werden und präferierte Diagnosen definiert. Kapitel 3 erläutert daraufhin die in der Literatur gängigen Algorithmen für präferierte Diagnosen. Mit diesen Grundlagen gerüstet, stellt Kapitel 4 neue Algorithmen vor, die zur Berechnung von präferierten Diagnosen verwendet werden können. Wie die Effizienz der Algorithmen durch die Implementierung maßgeblich verändert werden kann, erläutert Kapitel 5. Kapitel 6 schildert, welche Überprüfungen auf die Effizienz durchgeführt werden. Zum Abschluss präsentiert Kapitel 7 die Ergebnisse der Benchmarks und der Arbeit. Darüber hinaus gibt es einen Ausblick darauf, welche weitere Optimierungen künftig noch denkbar wären.

Kapitel 2

Grundlagen

In diesem Kapitel werden grundsätzliche Begrifflichkeiten erklärt. Diese sind für das Verständnis der vorgestellten Ansätze bedeutsam. Der Abschnitt 2.1 beginnt mit einer kurzen Einführung in die Aussagenlogik und in das Erfüllbarkeitsproblem. Der Abschnitt 2.2 erläutert einige grundlegende Begriffe, die über das Erfüllbarkeitsproblem hinaus gehen und für präferierte Diagnosen benötigen werden. Darauf aufbauend definiert Abschnitt 2.3, was präferierte Diagnosen sind. Damit werden alle notwendigen Grundlagen geschaffen, um ähnliche Arbeiten in Kapitel 3 analysieren zu können.

2.1 Aussagenlogik und das Erfüllbarkeitsproblem

Mithilfe der *Aussagenlogik* können komplexe Fragestellungen in *Formeln* dargestellt werden. Diese Formeln lassen sich zu einem *Wahrheitswert* auswerten. Das *Alphabet*, das die Formeln bildet, hat mehrere Symbole. Zum einen sind das unendlich viele *Variablen* $x, y, x_1, ...$, zum anderen die *Konstanten* $\top$ und $\bot$. Die Konstante $\top$ wertet zu wahr aus, während die Konstante $\bot$ zu falsch auswertet. Die *logischen Junktoren* $\wedge, \vee$ und $\neg$ verknüpfen die Variablen und Konstanten miteinander zu einer Formel. Dabei steht $\wedge$ für '*Und*', $\vee$ für '*Oder*' und $\neg$ für '*Negation*' [WHK04]. In Form einer *Belegung* bestimmt man welche Wahrheitswerte die Variablen haben. Diese sind entweder `wahr` (beziehungsweise 1) oder `falsch` (beziehungsweise 0). Belegt man alle Variablen einer Formel, so lassen sich diese zu einem Wahrheitswert auswerten. Eine Eigenschaft einer Formel ist die Erfüllbarkeit. Eine Formel gilt als *erfüllbar*, wenn mindestens eine Belegung existiert, für welche die Formel zu wahr auswertet. Alternativ kann sie *nicht erfüllbar* sein, das heißt, unabhängig von der Belegung wertet die Formel stets zu falsch aus. Eine solche unerfüllbare Formel befindet sich in Beispiel 2.1.

Beispiel 2.1. Die Formel $a \wedge \neg a$ ist nicht erfüllbar, unabhängig davon wie die Variable a belegt ist.

Bei einfachen Beispielen ist es direkt ablesbar, ob eine Formel erfüllbar ist, bei komplizierten Formeln ist es jedoch schwierig zu entscheiden, ob eine Formel erfüllbar ist oder nicht. Dieses Problem wird *Erfüllbarkeitsproblem* genannt. Es ist nicht mit einem polynomiellen Algorithmus lösbar sofern P $\neq$ NP gilt. Es ist *NP-vollständig* [Coo71], das heißt es ist in der Komplexitätsklasse *NP* und ist *NP-hart*. Ein Algorithmus, der eine Formel auf Erfüllbarkeit prüft, bezeichnet man üblicherweise als *SAT Solver*. Die meisten solcher Algorithmen erfordern die *konjunktive Normalform* der Eingabe, im Folgenden kurz *CNF* genannt. Diese besteht aus einer Konjunktion von Disjunktionen, $\bigwedge_i \bigvee_j (\neg) x_{ij}$ in mathematischer Notation. Die Disjunktionen bezeichnet man dabei als *Klauseln* c_i und $(\neg)x_{ij}$ als *Literal*. Ein Literal ist demzufolge eine Variable oder deren Negation. Diese Form hat den Vorteil, dass schnell erkannt werden kann, ob die Formel bei einer gegebenen Belegung unerfüllt wird. Es reicht bereits, dass *eine* Klausel nicht erfüllt ist und dies ist einfach und schnell bestimmbar. Des Weiteren gilt, wenn die Belegung die Formel erfüllt, so müssen alle Klauseln zu wahr auswerten.

Das Bespiel 2.2 verdeutlicht dieses Prinzip. Die Formel erfordert, dass die Variable a mit wahr belegt wird. Ansonsten ist die erste Klausel nicht erfüllbar und damit die ganze Formel nicht. Mit der Belegung $[a \mapsto 1, b \mapsto 0, c \mapsto 1, d \mapsto 1]$ sind alle Klauseln erfüllt und folglich auch die Formel. Damit ist die Formel erfüllbar.

Beispiel 2.2. Eine erfüllbare Formel in CNF-Form:

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

In der Regel verwendet man den *DPLL-* oder den *CDCL*-Algorithmus, um zu entscheiden, ob eine Formel erfüllbar ist. Der nächste Abschnitt beschreibt kurz den DPLL-Algorithmus, daran anschließend wird der CDCL-Algorithmus ausführlich erläutert, da dieser in der Arbeit verwendet wird.

2.1.1 DPLL-Algorithmus

Der DPLL-Algorithmus, benannt nach seinen vier Ideengebern M. Davis and H. Putnam, G. Logemann und D. Loveland, stellt die grundlegende Vorgehensweise für SAT Solving dar [DP60, DLL62]. Der Algorithmus führt zwei Vereinfachungen ein. Diese sind *Unit Propagation* und *Unit Subsumption*.

Ist bei einer nicht erfüllten Klausel nur noch eine Variable unbelegt, bezeichnet man diese als *Unit Klausel*. Eine solche Klausel bestimmt die Belegung der noch unbelegten Variable und verkleinert damit den Suchraum. Wird dies für alle Klauseln geprüft, so bezeichnet man dies als Unit Propagation.

Das Beispiel 2.2 verdeutlicht das Prinzip. Ausschließlich mit Unit Propagation lässt sich in diesem Beispiel die Erfüllbarkeit bestimmen. Wie bereits im letzten Abschnitt erwähnt, muss a mit wahr belegt werden. Das liegt daran, dass die Klausel (a) eine Unit Klausel ist. Durch die Belegung wird auch die zweite Klausel $(\neg a \vee \neg b)$ zu einer Unit Klausel. Die Variable b ist dort als einzige nicht belegt. Daraus resultiert die Belegung der Variable b zu falsch. Entsprechend ergibt sich damit, dass die dritte Klausel $(b \vee c)$ zur Unit Klausel wird. In der Konsequenz belegt man die Variable c mit wahr. Die letzte Klausel ist bereits erfüllt durch die Belegung der Variable b.

Eine belegte Variable erfüllt meist mehr als eine Klausel, so auch in Beispiel 2.2 die Variable b. Der DPLL-Algorithmus löscht diese Klauseln und verkleinert damit die Klauselmenge. Dieses Vorgehen bezeichnen die Autoren als Unit Subsumption. Führen Vereinfachungen nicht weiter, so testet der Algorithmus, ob bereits alle Klauseln erfüllt sind. Ist das nicht der Fall, so prüft er, ob eine Klausel nicht erfüllt ist. In diesem Fall existiert ein Konflikt und der Algorithmus muss eine andere Belegung testen; dies wird *Backtracking* genannt. Ist beides nicht der Fall, so testet der Algorithmus rekursiv beide Belegungen für eine bisher unbelegte Variable. Dieses systematische Vorgehen vereinfacht die Berechnung enorm im Vergleich zum Aufstellen einer Wahrheitstabelle. Er hat allerdings ein paar Schwachpunkte, die der CDCL-Algorithmus aufgreift. So kommt es dort zum Beispiel vor, dass lokale Konflikte wiederholt auftreten und damit unnötig viel Zeit benötigen. Angenommen der Algorithmus belegt die Variablen $x_1, ..., x_{i-1}$ und erst die Variable x_i ist relevant für den Konflikt. Dieser probiert alle Möglichkeiten 2^{i-1} der Variablen $x_1, ..., x_{i-1}$ aus, um jedes mal festzustellen, dass egal wie die Variable x_i belegt wird, die Klauselmenge unerfüllbar ist.

2.1.2 CDCL-Algorithmus

Der *Conflict-Driven Clause Learning-Algorithmus*, kurz CDCL, zieht die Konsequenzen aus den Schwächen des DPLL-Algorithmus und optimiert die Vorgehensweise an den entscheidenden Punkten. Erstmals stellten J. P Marques Silva und Karem A. Skallah ihn vor [SS96]. Stößt der DPLL-Algorithmus auf eine Belegung, die nicht erfüllbar ist, so kann er daraus keine Information ziehen. Er testet strikt alle $2^{|vars(\varphi)|}$ Kombinationen durch[1], falls die Formel φ unerfüllbar ist. Genau hier setzt die erste Optimierung an. Indem er aus einer Klausel lernt, verhindert der CDCL-Algorithmus die betroffenen Variablen nochmals gleich zu belegen. Dies ist vor allem dann relevant, wenn der Konflikt nur von wenigen Variablen abhängig ist und dieser somit auch in anderen Zweigen auftreten kann. Genau daraus folgt auch bereits der zweite Schwachpunkt des DPLL-Algorithmus. Selten ist die zuletzt gesetzte Variable abhängig für

[1] Der Algorithmus spart sich ein paar der Kombinationen ein, indem er Unit Propagation durchführt.

den gefundenen Konflikt. Meist liegt die Ursache viele Rekursionslevel zurück. Daher bietet es sich an, ohne Zwischenschritte so viele Level *backzutracken* wie nötig. Diese beiden Aspekte erhöhen die Geschwindigkeit enorm.

Um die Idee zu realisieren, muss sich der CDCL-Algorithmus merken, in welcher Reihenfolge er die Variablen belegt hat. Mit Hilfe von *Resolution* lernt der CDCL-Algorithmus aus der Konfliktklausel und der Reasonklausel[2] eine zusätzliche Klausel. Es lassen sich dabei verschiedene Arten von Klauseln lernen. In der Regel bevorzugt man die *First Unique Implication Point*-Klausel, kurz *1UIP*. Dies ist eine Klausel, die auf höchstem Level unit ist. Das bringt einem den Vorteil, dass man direkt nach dem Backtracking in jedem Fall durch Unit Propagation einen Fortschritt erzielen kann.

```
1  Input: Clauseset C
2  Output: SAT or UNSAT
3  level = 0
4  assignment = Set()
5  while true do {
6    unitPropagation(C, assignment)
7    if (C contains an empty clause) {
8      ec = empty clause
9      level = analyzeConf(ec, C)
10     if (level == -1) {
11       return UNSAT
12     }
13     backtrack(level)
14   } else {
15     if (no more variables undefined) {
16       return SAT
17     }
18     level++
19     x = next undefined var
20     assignment.add(x -> 0)
21   }
22 }
```

Abbildung 2.1: Der CDCL Algorithmus als Pseudo-Code [BBH+09, KZW13].

Die Abbildung 2.1 beschreibt wie die Implementierung aussieht. Wie der DPLL-Algorithmus führt auch der CDCL-Algorithmus in jeder Iteration Unit Propagation durch. Entdeckt die Unit Propagation einen Konflikt, so berechnet die Methode `analyze` die Klausel, die gelernt werden muss und gibt gleichzeitig zurück bis zu welchem Level Backtracking durchgeführt werden muss. Ist der Level negativ, dann ist der Konflikt unabhängig von einer Belegung. Damit steht fest, dass die Formel unerfüllbar ist. Ansonsten führt die Methode `backtrack` das Backtracking bis zum berechneten Level durch. Das heißt die Belegung der Variablen wird zurückgesetzt bis zu dem Backtracklevel. Ist nach der Unit Propagation keine Klausel *empty* (nicht erfüllt) und alle Variablen

[2]Das ist die Klausel, welche dazu geführt hat, dass die konfliktauslösende Variable ursprünglich belegt wurde.

belegt, dann gibt der Algorithmus SAT zurück. Gibt es unbelegte Variablen, so belegt er eine dieser Variablen und beginnt wieder mit Unit Propagation.

2.2 Begriffsklärung

Wie in der Einleitung geschildert, gibt es eine ganz praktische Motivation zur Entwicklung eines solchen Algorithmus zur Berechnung von präferierten Diagnosen: die Automobilkonfiguration. Bei dieser gibt es Bedingungen die zwingend erforderlich sind - zum Beispiel die Baubarkeitsbedingungen - und solche, die der Nutzer definiert - zum Beispiel das optionale Vorhandensein einer Klimaanlage. Beim Versuch diese Bedingungen auf ein Entscheidungsproblem zu übertragen, fällt auf, dass man optionale und strikte Bedingungen benötigt. Formal spricht man dabei von der *Partial Satisfiability*, wonach es ausreicht, dass nur ein Teil der Klauseln erfüllt sein muss. Alle Klauseln, die zwingend erfüllt sein müssen, bezeichnet man als *harte Klauseln*. Überträgt man das Automobilkonfigurationsbeispiel auf die Partial Satisfiability, so formuliert man optionale Bedingungen als *weiche Klauseln* und strikte Bedingungen als *harte Klauseln*. Die Menge aller weichen Klauseln bezeichnet diese Arbeit mit φ_{soft} und die der harten Klauseln mit φ_{hard}.

Bei der Automobilkonfiguration möchte man die Klauselmenge mit dem minimale Korrekturvorschlag wissen. Entfernt man eine Klausel aus dem Korrekturvorschlag, so ist das Problem unerfüllbar. Damit lässt sich dann herausfinden, welche weichen Bedingungen gestrichen werden müssen, damit eine gültige Konfiguration zustande kommt. Diese Menge wird hier als *Minimal Correction Subset* bezeichnet.

Definition 2.1. Für ein Minimal Correction Subset $C \subseteq \varphi_{\text{soft}}$, kurz *MCS*, gilt, dass die Klauselmenge $\varphi_{\text{hard}} \cup (\varphi_{\text{soft}} \backslash C)$ erfüllbar ist. Weiter ist erforderlich, dass für jede echte Teilmenge C' von C die Klauselmenge $\varphi_{\text{hard}} \cup (\varphi_{\text{soft}} \backslash C')$ unerfüllbar ist. [MSHJ+13b]

Das Komplement zum MCS ist eine maximal erfüllbare Klauselmenge, welche man als *Maximal Satisfiable Subset* bezeichnet.

Definition 2.2. Eine Klauselmenge $S \subseteq \varphi_{\text{soft}}$ ist ein Maximal Satisfiable Subset, kurz *MSS*, falls $\varphi_{\text{hard}} \cup S$ erfüllbar ist. Dazu muss für alle Klauselmengen S' mit $S \subseteq S' \subseteq \varphi_{\text{soft}}$ gelten, dass $\varphi_{\text{hard}} \cup S'$ unerfüllbar ist.

Für die Beziehung zwischen MCS und MSS gilt dementsprechend folgendes: Betrachtet man das Problem mit den harten Klauseln φ_{hard} und den weichen Klauseln φ_{soft} und ist S ein MSS, so ist $\varphi_{\text{soft}} \backslash S$ ein MCS für das selbe Problem [LS08]. Entsprechend verhält es sich umgekehrt. Weiter ist zu beachten, dass mehrere verschiedene MCS für die Klauselmengen φ_{hard} und φ_{soft} existieren können. Analog gilt das auch für MSS.

2.3 Präferierte Diagnosen

Anhand des Beispiels der Automobilkonfiguration ist bereits bekannt, dass eine festgelegte Reihenfolge der einzelnen Komponenten nach Präferenz einen intuitiven Ansatz darstellt. Klauseln haben in der reinen Aussagenlogik keine Wertigkeit und keine Präferenz. Dies ist der entscheidende Punkt, der sich bei präferierten Diagnosen ändert. Wir betrachten jetzt eine totale Ordnung auf den Klauseln. Für die Klauselmenge φ_{soft} definiert man eine strikte totale Ordnung $<$. Im Folgenden wird diese wie folgt verwendet. Die Klauelmenge φ_{soft} enthält die Klauseln $c_1, ..., c_m$ und für diese gilt $c_i < c_{i+1}$ falls $1 \leq i < m$. Die Klausel c_1 ist dabei die wichtigste und die Klausel c_m hat die geringste Präferenz. Dazu definiert [MSP14] den Operator $<_{\text{lex}}$, der die lexikographische Ordnung wie folgt festlegt:

Definition 2.3. Für zwei Mengen $\psi_1 \subseteq \varphi$ und $\psi_2 \subseteq \varphi$ sagt man, dass ψ_1 lexikographisch präferiert ist zu ψ_2, geschrieben als $\psi_1 <_{\text{lex}} \psi_2$, falls $\exists_{1 \leq k \leq m} : c_k \in \psi_1 \backslash \psi_2 \wedge \psi_1 \cap \{c_1, ..., c_{k-1}\} = \psi_2 \cap \{c_1, ..., c_{k-1}\}$ gilt.

Im Gegensatz dazu definiert sich die anti-lexikographische Ordnung $<_{\text{antilex}}$ damit wie folgt:

Definition 2.4. Für zwei Mengen $\psi_1 \subseteq \varphi$ und $\psi_2 \subseteq \varphi$ sagt man, dass ψ_1 umgekehrt lexikographisch präferiert ist zu ψ_2, geschrieben als $\psi_1 <_{\text{antilex}} \psi_2$, falls $\exists_{1 \leq k \leq m} : c_k \in \psi_2 \backslash \psi_1 \wedge \psi_1 \cap \{c_{k+1}, ..., c_m\} = \psi_2 \cap \{c_{k+1}, ..., c_m\}$ gilt.

Die umgekehrte Ordnung ist durch den Operator $<^{-1}$ bezeichnet. Die Klausel c_1 hat dabei die geringste Präferenz und die Klausel c_m höchste. Mathematisch notiert gilt demzufolge $c_m <^{-1} c_1$. Aus der Definition resultierend bezeichnet man ein MSS als L-Preferred, falls es lexikographisch kleiner ist, als alle anderen existierenden MSS. Entsprechend bezeichnet man ein MCS als A-Preferred, falls es anti-lexikographisch kleiner ist, als alle anderen existierenden MCS. Des Weiteren ist wichtig zu bemerken, dass ein L-Preferred/A-Preferred MSS/MCS stets eindeutig ist. Spricht diese Arbeit von einem (Zwischen-) Ergebnis für die Berechnung eines A-Preferred MCS, so bezeichnet sie es als Δ. Entsprechend kennzeichnet sie das (Zwischen-) Ergebnis eines L-Preferred MSS als Γ.

Wie [MSP14] zeigt, gilt die Beziehung von MSS und MCS auch für L-Preferred MSS und A-Preferred MCS. Ist Γ das L-Preferred MSS, dann ist $\varphi_{\text{soft}} \backslash \Gamma$ das A-Preferred MCS für die umgekehrte Ordnung. Im Folgenden wird hier immer von der Berechnung von A-Preferred MCS gesprochen. Jeder Algorithmus und jede Optimierung für A-Preferred MCS mit umgekehrter Ordnung lassen sich auch für L-Preferred MSS anwenden, da lediglich das Komplement gebildet werden muss.

Beispiel 2.3. Betrachte die Klauselmenge φ_{hard} und die lexikographisch geordnete Klauselmenge $\varphi_{\text{soft}} = \{c_1, c_2, c_3, c_4\}$. Für diese gilt nach der Definition

$c_1 < c_2 < c_3 < c_4$.
Angenommen die Menge $\{c_1, c_3, c_4\}$ ist ein L-Preferred MSS, dann folgt daraus, dass das A-Preferred MCS bezüglich der Ordnung $<^{-1}$ die Menge $\{c_2\}$ ist.

Wie Abschnitt 2.1 erwähnt, ist das Erfüllbarkeitsproblem NP-vollständig. Die Berechnung des A-Preferred MCS ist allerdings ein schwierigeres Problem. Es liegt in der Komplexitätsklasse $\mathrm{FP}^{\mathrm{NP}}$ [MSP14]. Diese Klasse erfasst alle Funktionen, die aus Zeichenfolgen andere Zeichenfolgen berechnen und von einer deterministischen Turingmaschine mit SAT Orakel in Polynomialzeit berechnet werden können [Pap94]. Wie [WFK15a] zeigt, ist das Problem sogar $\mathrm{FP}^{\mathrm{NP}}$-hart und damit $\mathrm{FP}^{\mathrm{NP}}$-vollständig.

Kapitel 3

Verwandte Arbeiten

Nachdem in Kapitel 2 die notwendigen Grundlagen geschaffen wurden, widmet sich Kapitel 3 der Analyse verwandter Arbeiten. Dazu werden folgenden Standard Algorithmen erläutert: Die intuitive lineare Suche, welche in Abschnitt 3.1 erklärt wird, und der FastDiag Algorithmus, der in Abschnitt 3.2 dargestellt wird. Der letzte Abschnitt benennt weitere Ideen und Ansätze aus verwandten Arbeiten, die zur Lösung eines A-Preferred MCS verwendet werden können.

3.1 Lineare Suche

Die *lineare Suche* macht, was im schlimmsten Fall notwendig ist und benötigt $|\varphi_{\text{soft}}|$ viele SAT Aufrufe zur Berechnung. [WFK15b]. Sie prüft iterativ in einer Schleife einzeln für jede Klausel c_i von φ_{soft}, ob sie zum A-Preferred MCS gehört oder nicht [Jun04]. Angefangen wird mit der wichtigsten Klausel c_1. Für die Analyse baut der Algorithmus zwei Mengen auf: Zum einen ist das die Menge Γ, die alle Klauseln enthält, die erfüllbar sind. Γ entspricht dem L-Preferred MSS am Ende des Algorithmus. Zum anderen ist das die Menge Δ, die alle Klauseln enthält, die nicht erfüllbar sind. Δ ist am Ende das Ergebnis, also das A-Preferred MCS. In jedem Schritt prüft ein SAT Solver, ob die Klauselmenge $\varphi_{\text{hard}} \cup \Gamma \cup \{c_i\}$ erfüllbar ist oder nicht. Ist die Klauselmenge erfüllbar, dann fügt man die Klausel c_i ebenso Γ hinzu. Ist die Klauselmenge nicht erfüllbar, so gehört die betrachtete Klausel c_i zum A-Preferred MCS. Sie wird daher Δ hinzugefügt. Abbildung 3.1 gibt eben diesen Algorithmus als Pseudo-Code an.

3.2 FastDiag

Der FastDiag-Algorithmus [FSZ12] greift den Schwachpunkt der linearen Suche auf. Er basiert auf der gleichen Idee, wie der QuickXplain-Algorithmus, den Junker vorgestellt hat [Jun04]. Die lineare Suche hat den Nachteil, dass sie jede Klausel einzeln mit einem SAT Aufruf prüft, obgleich dies nicht im-

```
1  Input: φhard, φsoft = {c1, ..., cm} with c1 < ... < cm
2  Output: A-Preferred MCS Δ
3  Γ, Δ = ∅
4  for (i = 1 to m) {
5    if (SAT(φhard ∪ Γ ∪ {ci})) {
6      Γ = Γ ∪ {ci}
7    } else {
8      Δ = Δ ∪ {ci}
9    }
10 }
11 return Δ
```

Abbildung 3.1: Der lineare Suche Algorithmus für A-Preferred MCS als Pseudo-Code.

mer notwendig ist. Denn sind viele Klauseln in direkter Reihenfolge Teil des L-Preferred MSS, so benötigt man weniger SAT Aufrufe, um dies festzustellen. Es reicht, wenn man mehrere Klauseln gleichzeitig betrachtet. Dazu verwendet FastDiag die *Divide-and-Conquer* Strategie.

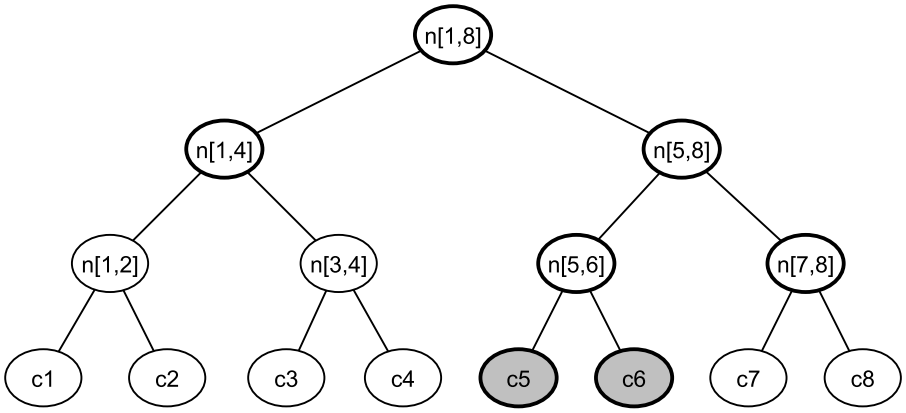


Abbildung 3.2: Der Suchbaum, den FastDiag bei acht Klauseln verwendet. Das A-Preferred MCS besteht aus den Klauseln c_5 und c_6.

Die Softklauseln teilt der Algorithmus pro Schritt in zwei Hälften und geht dabei ähnlich wie die binäre Suche vor. Dies illustriert Abbildung 3.2. FastDiag prüft zuerst, ob das Problem auch tatsächlich nicht erfüllbar ist. Ist das der Fall, dann teilt der Algorithmus die acht Klauseln in zwei Unteraufrufen auf, dabei berechnet der erste Unteraufruf, ob die Klauseln c_1, c_2, c_3, c_4 erfüllbar sind. Tritt die Erfüllbarkeit ein, so ist dieser Teil fertig berechnet. Dadurch müssen die einzelnen Klauseln nicht näher betrachtet werden. Der rechte Teilbaum $n_{[5,8]}$ prüft die restlichen Klauseln auf Erfüllbarkeit. Daraus, dass der Teilbaum $n_{[1,4]}$ erfüllt ist, kann gefolgt werden, dass eine Klausel in dem Teilbaum $n_{[5,8]}$ unerfüllbar sein muss. Die Überprüfung dessen erspart sich FastDiag deshalb. Es teilt das Teilproblem in zwei weitere Teilprobleme

$n_{[5,6]}$ und $n_{[7,8]}$ auf und verfährt nach demselben Prinzip weiter. Besteht das Teilproblem aus einer Klausel und ist nicht erfüllbar, so gehört die Klausel zum A-Preferred MCS. Der Algorithmus benötigt damit sechs SAT Aufrufe, um das A-Preferred MCS zu berechnen.

```
1  Input: φ_hard, φ_soft = {c_1, ..., c_m} with c_1 < ... < c_m
2  Output: A-Preferred MCS Δ
3  if (φ_soft = ∅ or SAT(φ_hard ∪ φ_soft)) {
4    return empty list
5  } else {
6    Γ = ∅
7    return FastDiagHelper(true, φ_soft)
8  }
9
10 func FastDiagHelper(isRedundant, φ): A-Preferred MCS {
11   if (!isRedundant and SAT(φ_hard ∪ Γ ∪ φ)) {
12     Γ = Γ ∪ φ
13     return empty list
14   } else if (|φ| = 1) {
15     return φ
16   }
17   split φ into φ_1 and φ_2
18   Δ_1 = FastDiagHelper(false, φ_1)
19   Δ_2 = FastDiagHelper(Δ_1 = ∅, φ_2)
20   return Δ_1 ∪ Δ_2
21 }
```

Abbildung 3.3: Der FastDiag Algorithmus für A-Preferred MCS. Pseudo-Code entnommen aus [FSZ12] und mit Änderungen in der Darstellungsform versehen.

Nach der intuitiven Vorgehensweise im letzten Absatz befasst sich dieser Absatz mit der praktischen Umsetzung. Dazu verschafft Abbildung 3.3 einen Überblick. Als Eingabe für den Algorithmus wird φ_{hard} und φ_{soft} benötigt, wobei φ_{soft} nach Präferenzen sortiert ist. φ_{soft} beginnt dabei mit der wichtigsten Klausel. Zuerst werden Trivialfälle überprüft, die ein sofortiges Ergebnis liefern. Dazu gehört zum einen der Fall, dass φ_{soft} keine Klauseln enthält. Daraus folgt, dass das A-Preferred MCS auch keine Klauseln enthält. Ebenso wird geprüft, ob das Problem erfüllbar ist. Eine weitere Berechnung macht nur für unerfüllbare Probleme Sinn. Tritt kein Trivialfall ein, so startet die eigentliche Berechnung. Dafür benötigt der Algorithmus die Variable Γ. Diese hält alle Klauseln, die zum L-Preferred MSS gehören. Dies benötigt man - wie schon bei der linearen Suche - für die Überprüfung auf Erfüllbarkeit (siehe Zeile 11). Der rekursiv aufgebaute Algorithmus steckt in der Hilfsmethode `FastDiagHelper`. Diese Methode erhält die Information, ob der aktuelle Fall redundant ist, durch die Variable `isRedundant`. Die Softklauseln, die aktuell zu betrachten sind, erhält der Algorithmus durch die Variable φ. In einem redundanten Fall kann auf ein SAT Aufruf verzichtet werden, was eine deutliche Zeitersparnis bringt. Im ersten Fall tritt dies ein, da in Zeile 3 bereits der exakt selbe Fall geprüft wurde. Ist der Fall nicht redundant und die Klau-

selmenge $\varphi_{\text{hard}} \cup \Gamma \cup \varphi$ erfüllbar, so fügt der Algorithmus alle Klauseln aus φ zu Γ hinzu - dieser Fall tritt in Abbildung 3.2 bei Knoten $n_{[1,4]}$ und $n_{[7,8]}$ ein. Ist die Klauselmenge nicht erfüllbar und φ besteht aus einer Klausel, so steht fest, dass diese Klausel zum A-Preferred MCS gehört (siehe Knoten c_5 und c_6). Tritt keiner der genannten Fälle auf, so teilt der Algorithmus das Problem in zwei gleich große Teilprobleme auf. Ist das erste Teilproblem erfüllbar, so kann der Algorithmus daraus folgern, dass im zweiten Teilproblem auf jeden Fall eine Klausel nicht erfüllt werden kann (siehe Knoten $n_{[5,8]}$). Mittels der Überprüfung $\Delta_1 = \emptyset$ ist das in Zeile 19 umgesetzt. Für das erste Teilproblem gibt es keine Informationen, die eine Redundanz ergibt; daher wird `false` übergeben. Nachdem beide Teilprobleme gelöst sind, konkateniert er beide Teillösungen und gibt diese zurück.

Im Ergebnis kommt FastDiag auf einen *worst case* an SAT Aufrufen von $2d \cdot \log_2(\frac{m}{d}) + 2d$ [FSZ12]. Wobei d die Größe des A-Preferred MCS ist und m die Anzahl der weichen Klauseln. Im Vergleich dazu benötigt die lineare Suche m Aufrufe. Setzt man das in das Verhältnis zueinander ergibt sich folgende Formel:

$$2d \cdot \log_2(\frac{m}{d}) + 2d < m \text{ setze } m = 1$$
$$\text{nummerisch gelöst ergibt das } 0 < d < 0,125$$

Daraus folgt: Gehören zum A-Preferred MCS weniger als etwa $12,5\%$ weichen Klauseln, so benötigt FastDiag garantiert weniger SAT Aufrufe. Liegen die Klauseln, die zum A-Preferred MCS gehören in Bezug auf die totale Ordnung beieinander, so kann es trotzdem noch effizienter sein, wenn mehr als $12,5\%$ zum A-Preferred MCS gehören.

3.3 Weitere Ideen

Während die unter 3.1 und 3.2 genannten Algorithmen bereits vollständige Lösungsmöglichkeiten bieten, gibt es noch weitere interessante Ideen, die zwar keine abschließende Lösung abbilden, bei denen sich aber ebenfalls eine Betrachtung lohnt. Zum einen gibt es den *CLD* Algorithmus der in [MSHJ+13b] vorgestellt wird. Dieser berechnet ein (nicht präferiertes) MCS, indem bei einer Belegung geprüft wird, ob die Disjunktion der unerfüllten Klauseln erfüllt werden kann, während die erfüllten Klauseln weiterhin erfüllt bleiben. Aus der Belegung folgt, welche Klauseln das sind. Dieser Vorgang wird iterativ wiederholt. Der Nachteil dieses Algorithmus liegt allerdings darin, dass er sich nicht zur Berechnung eines A-Preferred MCS übertragen lässt, da er keiner Reihenfolge folgt. Zum anderen stellt dieselbe Arbeit noch weitere Ideen zur Berechnung eines MCS vor. Diese überprüft [WFK15b] auf die Verwendung

für A-Preferred MCS. Einige der in Kapitel 4 vorgestellten Algorithmen und Ideen basieren darauf.

Kapitel 4

Neue Algorithmen und Verbesserungen

Dieses Kapitel befasst sich mit der Funktionsweise der Algorithmen und Verbesserungen, die diese Arbeit analysiert. In Abschnitt 4.1 werden dabei allgemeine Verbesserungen diskutiert und erläutert. Diese allein heben die Geschwindigkeit bereits auf ein neues Level. Eine Weiterentwicklung von FastDiag, der General Chunks Approach, wird in Abschnitt 4.2 vorgestellt. Eine weitere Variante modifiziert direkt den SAT Solver und berechnet durch einen einzigen Aufruf das Ergebnis. Wie dies funktioniert erläutert Abschnitt 4.3. Abschnitt 4.4 stellt den Adopted Branch and Bound Approach vor, der eine völlig andere Herangehensweise zur Berechnung von präferierten Diagnosen wählt.

4.1 Allgemeine Optimierungen

Im Kapitel 3 wurden zwei bekannte Algorithmen zur Berechnung von präferierten Diagnosen vorgestellt. Bei der Analyse stößt man auf mehrere Schwachpunkte und zusätzliche Ansätze. Mit diesen Punkten befasst sich dieser Abschnitt. Warum sich in der Regel der erste SAT Aufruf erspart werden kann, erläutert Abschnitt 4.1.1. Die letzten beiden Abschnitte erläutern Optimierungsansätze, die sich Informationen der Zwischenergebnisse zu Nutze machen.

4.1.1 Redundanter SAT Aufruf sparen

Algorithmen wie FastDiag prüfen im ersten Aufruf zuerst, ob $\varphi_{\text{hard}} \cup \varphi_{\text{soft}}$ erfüllbar ist. Dieser Aufruf ist in der Praxis redundant. Einen Solver für eine präferierte Diagnose wird in der Praxis nur verwendet, wenn feststeht, dass nicht alle weichen Klauseln φ_{soft} erfüllt werden können. Ein Aufruf, ob $\varphi_{\text{hard}} \cup \varphi_{\text{soft}}$ erfüllbar ist, findet demzufolge schon vor der Verwendung des Solvers statt.

4.1.2 Ergebnismodell ausnützen

Das Ergebnis des SAT Solvers lässt sich zu Nutze machen. Dies nennt man *Model Exploiting*. Für die Berechnung einer präferierten Diagnose benötigen die in Kapitel 3 vorgestellten Algorithmen mehrere SAT Aufrufe. Ist ein Test auf Erfüllbarkeit erfolgreich, so kann der SAT Solver die erfüllende Belegung speichern. Die Belegung bietet zusätzliche Informationen, wie [MSHJ+13b, WFK15b] vorstellt. Damit lässt sich etwa testen, ob die folgenden Klauseln bereits mit der Belegung erfüllt sind. Angenommen der SAT Aufruf testet $\varphi_{\text{hard}} \cup \Gamma \cup \{c_i\}$ auf Erfüllbarkeit. Dann können die folgenden Klauseln $c_{i+1}, ..., c_m$ schnell und ohne zusätzlichen SAT Aufruf auf ihre Erfüllbarkeit getestet werden. Die erste Klausel c_j (angenommen $i < j \leq m$), die nicht von der Belegung erfüllt wird, bricht die Suche ab. Alle Klauseln $c_{i+1}, ..., c_{j-1}$ können damit direkt zum A-Preferred MCS hinzugefügt werden. Über die Klausel c_j und alle folgenden kann allerdings keine weitere Aussage getroffen werden, ohne einen neuen SAT Aufruf zu tätigen.

Beispiel 4.1.

$$\varphi_{\text{hard}} = \{\{a, b\}, \{\neg a, c\}, \{\neg a, \neg b, d\}\}$$
$$\varphi_{\text{soft}} = \{\{\neg b\}, \{\neg c, a\}, \{\neg a, \neg d\}, \{d\}\}$$

Das Beispiel 4.1 veranschaulicht die Idee des Model Exploitings. Beim Aufruf $\text{SAT}(\varphi_{\text{hard}} \cup \{\neg b\})$ erhält man zum Beispiel das Modell $\tau = [a \leftarrow 1, b \leftarrow 0, c \leftarrow 1, d \leftarrow 1]$. Daraufhin ist nicht nur bekannt, dass die erste Klausel $\{\neg b\}$ zum L-Preferred MSS gehört, sondern es kann Model Exploiting angewendet werden. Dazu prüft man die zweite weiche Klausel $\{\neg c, a\}$. Zwar ist das erste Literal $\neg c$ mit der Belegung nicht erfüllt, aber das zweite Literal a ist es. Die zweite Klausel gehört damit auch zum L-Preferred MSS. Infolgedessen kann auch die dritte Klausel $\{\neg a, \neg d\}$ geprüft werden. Dort sind allerdings mit der gegebenen Belegung beide Literale nicht erfüllt. Über diese Klausel kann damit ohne einen weiteren SAT Aufruf keine Aussage getroffen werden. Zwei weitere SAT Aufrufe ergeben, dass die dritte Klausel auch erfüllbar ist, indem d mit falsch belegt wird. Die letzte Klausel ist damit unerfüllbar und bildet das A-Preferred MCS.

Das Model Exploiting lässt sich auch noch erweitern [WFK15b]. Angenommen man hat die Belegung τ für die Klauselmenge $\varphi_{\text{hard}} \cup \Gamma \cup c_i$. Mithilfe des Model Exploitings lassen sich die Klauseln $c_{i+1}, ..., c_{j-1}$ erfüllen. Die Klausel c_j ist mit der Belegung τ nicht erfüllbar. Dann lohnt es sich die Belegung τ zu speichern für den Fall, dass c_j tatsächlich nicht erfüllbar ist, also zum A-Preferred MCS gehört. Ist dies der Fall so kann die Belegung τ wiederverwendet werden. Es kann daraufhin Model Exploiting mit der Belegung τ für die Klauseln $c_{j+1}, ...$ verwendet werden.

4.1.3 Deltainformation ausnützen

Aus jeder Klausel, die zum A-Preferred MCS Δ hinzugefügt wird, lassen sich Informationen sammeln. Eine Variable, die in dieser Klausel vorkommt, wird garantiert so belegt, dass die Klausel nicht erfüllt wird. Diese Information kann dem SAT Solver direkt mitgeben werden und damit den Suchraum für folgende SAT Aufrufe einschränken. Stellt man nun fest, dass die Klausel c_i zum A-Preferred MCS gehört, so fügt man die Negation jedes seiner Literale als Klausel zu Γ hinzu. Diese Literale nennt man *Backbone* Literale [MSHJ+13b]. Damit werden bei den nächsten SAT Aufrufen direkt die Variablen so propagiert, dass die Klausel c_i nicht erfüllt ist. Dieser Schritt erspart dem SAT Solver eventuell auftretende Konflikte und er kann so schneller zum Ergebnis kommen.

Beispiel 4.2.

$$\varphi_{\text{hard}} = \{\{a, b\}, \{\neg a, c\}, \{\neg b\}, \{\neg a, b, d\}$$
$$\varphi_{\text{soft}} = \{\{\neg c, \neg d\}, \{\neg a, b, \neg c\}\}$$

Angenommen φ_{hard} und φ_{soft} sind wie in Beispiel 4.2 formuliert und man führt darüber den lineare Suche Algorithmus aus. Dann testet dieser im Ersten Schritt ob $\varphi_{\text{hard}} \cup \{\{\neg c, \neg d\}\}$ erfüllbar ist. Der Test ergibt, dass die Klauselmenge nicht erfüllbar ist, womit die Klausel $\{\neg c, \neg d\}$ zum A-Preferred MCS Δ gehört. Es folgt daraus, dass die Klauselmenge φ_{hard} nie erfüllbar ist, wenn die Variable c und d mit falsch belegt wird. Damit alle folgenden SAT Aufrufe diese Zusatzinformation berücksichtigen fügt man die beiden Klauseln $\{c\}$ und $\{d\}$ hinzu.

4.2 General Chunks Approach

Eine Verallgemeinerung von FastDiag stellt der von [WFK15b] vorgestellte *General Chunks Algorithmus* dar. Dieser funktioniert im Prinzip wie FastDiag allerdings mit dem wesentlichen Unterschied, dass der General Chunks Algorithmus die Klauselmenge nicht in jedem Schritt halbiert. Abhängig von Rekursionstiefe und Anzahl der Klauseln wählt er eine beliebige Anzahl an Teilmengen (Chunks). Hierfür können verschiedene Strategien gewählt werden auf welche in Abschnitt 4.2.2 eingegangen wird. Eine einfache Strategie ist es zum Beispiel, die Menge immer in vier gleich große Teilmengen aufzuteilen.

Die Implementierung des General Chunks Algorithmus zeigt Abbildung 4.1. Die Optimierungen aus Abschnitt 4.1 sind für den Algorithmus optional verwendbar. Die Annahme zur Unerfüllbarkeit von $\varphi_{\text{hard}} \cup \varphi_{\text{soft}}$, die in Abschnitt 4.1.1 beschrieben wurde, drückt sich durch die Variable `assumeUNSAT` aus. Dies zeigt Zeile 4 in der Abbildung. Die anderen beiden Optimierungen beschreibt Abschnitt 4.2.1 genauer. Dazu gehört die Funktion der Variable `skip`.

```
1  Input: φhard, φsoft = {c1, ..., cm} with c1 < ... < cm
2  Output: A–Preferred MCS Δ
3  Γ, Δ = ∅
4  ChunksHelper(1, assumeUNSAT, 1, m)
5  return Δ
6
7  func ChunksHelper(recursion, isRedundant, start, end) {
8    if (¬isRedundant ∧ SAT(φhard ∪ Γ ∪ {cstart, ..., cend})) {
9      Γ = Γ ∪ {cstart, ..., cend}
10     numberOfClausesSatisfied = 0
11     // insert model exploiting
12     return (true, numberOfClausesSatisfied)
13   } else if (end = start) {
14     Δ = Δ ∪ cstart
15     // insert backbone handling
16     return (false, 0)
17   } else {
18     partitionMaker = new PartitionMaker(start, end, recursion, strategy)
19     areSubCallsSAT = true; skip = 0
20     while (partitionMaker.hasNext(skip)) {
21       (recStart, recEnd) = partitionMaker.nextPartition(skip)
22       recIsRedundant = areSubCallsSAT ∧ ¬partitionMaker.hasNext()
23       (subCallIsSAT, skip) = ChunksHelper(recursion + 1, recIsRedundant, recStart,
               recEnd)
24       areSubCallsSAT = areSubCallsSAT ∧ subCallIsSAT
25     }
26     return (areSubcallsSAT, partitionMaker.recursiveSkip(skip))
27   }
28 }
```

Abbildung 4.1: Der General Chunks Algorithmus für A-Preferred MCS als Pseudo-Code.

Zunächst legt der Algorithmus die zwei globalen Variablen für das A-Preferred MCS Δ und für alle erfüllbaren Klauseln Γ an. Die Verwendung von globalen Variablen für Γ und Δ bringt eine erhöhte Effizienz, da so auf Operationen über Mengen oder Listen verzichtet werden kann [WFK15b]. Daraufhin ruft der Algorithmus die Hilfsfunktion `ChunksHelper` auf, die rekursiv das A-Preferred MCS berechnet. Diese Funktion berechnet mithilfe der Rekursionstiefe, der Information, ob der SAT Aufruf redundant ist, und den Grenzen der aktuell betrachteten Klauseln die präferierte Diagnose. Sie gibt zwei Werte zurück, auf die später eingegangen wird. Die Übergabe der Grenzen bietet einen Effizienzvorteil und impliziert auch die Verwendung eines Arrays für den Zugriff auf die weichen Klauseln. Ähnlich wie bei FastDiag existieren drei Fälle, die bestimmen, was passiert. Im ersten Fall prüft die Funktion, ob die betrachtete Klauselmenge $c_{\text{start}}, ..., c_{\text{end}}$ erfüllbar ist. Trifft dies zu, so kann die Klauselmenge zu Γ hinzugefügt werden. Ist durch die Variable `isRedundant` bekannt, dass der Fall redundant ist, dann erspart sich die Funktion die Überprüfung. Ist der Test nicht erfolgreich oder redundant, dann prüft die Funktion den zweiten Fall. Handelt es sich um exakt *eine* Klausel, die überprüft wurde, dann steht fest, dass die Klausel zum A-Preferred MCS Γ gehört. Der dritte Fall wendet das Teile-und-Herrsche Prinzip an. Die Klasse `PartitionMaker` übernimmt

dabei das Aufteilen der Klauselmenge in k Partitionen. Sind die ersten $k-1$ Partitionen erfüllbar, so ist der letzte SAT Aufruf redundant. Für diese Funktionalität verwendet die Funktion die Variable `areSubCallsSAT` und gibt in jedem Berechnungschritt diese Information zurück. Zu sehen ist das in der Abbildung in Zeile 12, 16 und 26. Die Funktion durchläuft iterativ in einer Schleife die Partitionen. Die Partition, welche die Methode `nextPartition()` berechnet hat, legt fest welche Klauseln in der nächsten Iteration betrachtet werden. Mittels der Variable `areSubCallsSAT` und der Methode `partitionMaker()` kann bestimmt werden, ob es die letzte Partition ist und ob der SAT Aufruf darin redundant ist.

4.2.1 Model Exploiting und Backbone Literale

Die Optimierungen Model Exploiting und Backbone Literale lassen sich - wie bereits erwähnt - in den General Chunks Algorithmus integrieren. Die Kommentare in Abbildung 4.1 Zeile 11 und 15 markieren, an welchen Stellen diese zu implementieren sind.

```
1 if (useModelExploiting) {
2   for (j <- end + 1 to m) {
3     if (c_j is satisfied with assignment) {
4       Γ = Γ ∪ {c_j}
5       numberOfClausesSatisfied += 1
6     } else break
7   }
8 }
```

Abbildung 4.2: Das Model Exploiting in General Chunks Algorithmus für A-Preferred MCS als Pseudo-Code.

Die Abbildung 4.2 zeigt die notwendige Ergänzung, die für Model Exploiting notwendig ist. Durch die Variable `useModelExploiting` kann das Model Exploiting auch optional ausgeschaltet werden. Iterativ prüft die Methode die Klausel c_{end+1} und die folgenden darauf, ob sie mit der zuletzt berechneten Belegung erfüllt werden. Die Klauseln, auf die es zutrifft, werden - wie in Abschnitt 4.1.2 beschrieben - zu Γ hinzugefügt. Sobald die erste Klausel nicht erfüllt ist, bricht die Suche ab und der Algorithmus läuft normal weiter. Zu beachten hat der General Chunks Algorithmus allerdings, dass einige Klauseln nun nicht mehr betrachtet werden müssen. Dies geschieht durch die Variable `numberOfClausesSatisfied`. Sie zählt, wie viele Klauseln erfüllt wurden, und gibt die Anzahl an den nächst höheren Rekursionsschritt zurück. Genau hierfür ist der zweite Rückgabewert der Funktion `ChunksHelper` vorhanden. Für die Überprüfung, ob noch eine weitere Partition existiert, benötigt man exakt diesen Wert, den die Variable `skip` speichert. Die Anzahl der zusätzlich erfüllten Klauseln kann über mehrere Rekursionsschritte hinweg überlappen.

Diese Überlappung berücksichtigt die Methode `recursiveSkip` und berechnet die Anzahl der Klauseln, die der nächste höhere Rekursionsschritt noch überspringen kann.

Die Implementierung der Backbone Literale benötigt keine besondere Beachtung für den restlichen Algorithmus. Abbildung 4.3 zeigt die Implementierung als Pseudo-Code. Die Variable `useBackbone` ermöglicht auch hier wieder eine optionale Verwendung der Optimierungsform. Eine For-Schleife iteriert über die Literale der Klausel und fügt diese in negierter Form als Klauseln zu Γ hinzu.

```
1 if (useBackbone) {
2   for (lit <- c_start) {
3     Γ = Γ ∪ {¬lit}
4   }
5 }
```

Abbildung 4.3: Die Backbone Literale im General Chunks Algorithmus für A-Preferred MCS als Pseudo-Code.

4.2.2 Partitionsstrategie

Der General Chunks Algorithmus hat die vorteilhafte Eigenschaft, dass er mit verschiedenen Partitionsstrategien funktioniert. Vor allem damit hebt er sich vom Standardalgorithmus FastDiag ab. Mittels einer Funktion legt der Anwender die Strategie des Algorithmus fest. Diese Funktion berechnet - abhängig von der Rekursionstiefe und der Anzahl der Klauseln - die Anzahl der Partitionen, die der Algorithmus bilden soll. FastDiag kann mit der Funktion simuliert werden, die konstant zwei zurückgibt. Ein einfacher Ansatz ist es, größere Zahlen auszuprobieren. Alternativ bietet es sich an, die Partitionen möglichst klein zu halten. Damit kann man die Wahrscheinlichkeit erhöhen, dass direkt alle Klauseln in einer Partition erfüllt sind und keine weiteren SAT Aufrufe notwendig sind. Weiter ist auch eine Kombination dieser Idee mit anschließendem Fast Diag denkbar. Theoretisch könnte man die Partitionsstrategie auch komplett anders konzipieren und verschieden große Partitionen bilden. Eine Auswertung der verschiedenen Strategien erfolgt in Kapitel 6.

4.3 Modifizierter SAT Solver

Bisher vorgestellte Algorithmen haben unter anderem stets das Ziel verfolgt, die Anzahl der SAT Aufrufe zu minimalisieren, um die Berechnung von präferierten Diagnosen zu beschleunigen. Der *Modifizierter SAT Solver* Ansatz geht einen anderen Weg und verändert direkt den CDCL SAT Solver. Diese Vorgehensweise beschreibt [WFK15b]. Dazu wird jede Klausel c_i aus φ_{soft} um

das zusätzliche Literal $\neg s_i$ ergänzt. Die neuen Variablen $s_1, ..., s_m$ werden hier als *First Decision* Variablen bezeichnet und werden im SAT Solver gesondert markiert. Dabei ist daran zu denken, dass jede dieser Variablen eine weiche Klausel repräsentiert. Alle Klauseln werden daraufhin zur SAT Solver Instanz hinzugefügt. Während dem SAT Solving muss nur die Variablenwahl geändert werden.

Kommt der SAT Solver durch Unit Propagation nicht weiter, so sucht er sich bisher eine beliebige Variable und belegt diese. Bei diesem Ansatz ändert sich das Vorgehen, wenn nicht alle First Decision Variablen schon belegt sind. Der Solver belegt bei einer Entscheidung zuerst eine der First Decision Variablen mit wahr. Ist diese First Decision Variable vor dem letzten Backtracking (durch eine Entscheidung) mit wahr belegt worden, so wird sie nun mit falsch belegt. Dabei wird die wichtigste First Decision Variable zuerst belegt. Die wichtigste First Decision Variable ist die Variable s_1, welche die Klausel c_1 repräsentiert. Die unwichtigste First Decision Variable ist damit s_m, welche die Klausel c_m repräsentiert. Es ist dabei zu beachten, dass First Decision Variablen auch durch Unit Propagation belegt werden können. Das führt dazu, dass der Decision Level des SAT Solvers nicht bestimmen kann, ob alle First Decision Variablen belegt sind. Die First Decision Variablen benötigen eine zusätzliche Form der Überprüfung, ob alle Variablen schon belegt sind. Sind alle m First Decision Variablen belegt, so kann der SAT Solver nach beliebiger Heuristik die restlichen Variablen belegen und ein normales SAT Solving durchführen. Entgegen dem Vorschlag von [WFK15b] kann zusätzlich auf eine geänderte Form des Backtrackings verzichtet werden. Es reicht die strikte Variablenwahl aus, um das A-Preferred MCS zu finden.

Findet der SAT Solver eine erfüllende Belegung steht direkt fest, wie die präferierte Diagnose aussieht. Sie kann unmittelbar aus den First Decision Variablen abgelesen werden. Ist die First Decision Variable s_i erfüllt, so gehört die zugehörige Klausel zum L-Preferred MSS. Ist sie nicht erfüllt, so gehört sie zum A-Preferred MCS.

Dieser Algorithmus besticht vor allem dadurch, dass nur `ein` SAT Aufruf notwendig ist. Im Gegenzug ist eine eventuell schlechtere Variablenwahl während der Ausführung des Algorithmus in Kauf zu nehmen. Je nachdem, wie die Klauseln zueinander korrelieren, könnte sich der SAT Solver in lokalen Maxima verhängen. Letztendlich kommt er aber garantiert zur Lösung.

4.4 Adopted Branch and Bound

Eine Variante zur Berechnung von präferierten Diagnosen, die frei von SAT Solver ist, stellt der *Adopted Branch and Bound* Ansatz dar. Dieser Ansatz wird zur Optimierung von verschiedensten Problemen verwendet, so zum Beispiel auch für die Berechnung von MaxSAT-Problemen [BBH+09]. Dabei werden

die Zweige (Branches) eines Entscheidungsbaums durchlaufen und deren Zwischenergebnisse verglichen. Ist das Zwischenergebnis schlechter als das bisherige, so wird der Zweig nicht weiter verfolgt (Bound). Der Ansatz für A-Preferred MCS wird von [WFK15b] erstmals vorgestellt.

In der Variante für präferierte Diagnosen ist dabei der *Upper Bound* das aktuell beste Zwischenergebnis für das A-Preferred MCS. Initial kann man zum Beispiel die Belegung nehmen, die die harten Klauseln erfüllt, und die Menge aller erfüllten weichen Klauseln als Upper Bound hernehmen. Jeder Algorithmus der ein MCS oder eine Approximation davon berechnet, ist auch verwendbar für die Bestimmung des initialen Upper Bounds. Eine einfachste Variante eines Upper Bounds ist φ_{soft}.

```
1  Input: φ_hard, φ_soft = {c_1, ..., c_m} with c_1 < ... < c_m
2  Output: A–Preferred MCS Δ
3  upperBound = calculate(φ_hard, φ_soft)
4  return adoptedBAB(φ_hard, φ_soft, ∅, upperBound)
5
6  func adoptedBAB(φ'_hard, φ'_soft, φ'_soft∅, upperBound) {
7    (hardClausesAreEmpty, φ''_hard, φ''_soft, φ''_soft∅) = simplify(φ'_hard, φ'_soft, φ'_soft∅)
8    if (hardClausesAreEmpty) return upperBound
9    if (φ''_hard = ∅ ∧ φ''_soft = ∅) return φ''_soft∅
10   lowerBound = underestimation(φ''_soft, φ''_soft∅)
11   if (upperBound ≤_antilex lowerBound) return upperBound
12   x = next undefined variable
13   assign(x, true)
14   resultPos = adoptedBAB(φ''_hard, φ''_soft, φ''_soft∅, upperBound)
15   assign(x, false)
16   if (resultPos ≤_antilex upperBound) upperBound = resultPos
17   resultNeg = adoptedBAB(φ''_hard, φ''_soft, φ''_soft∅, upperBound)
18   unassign(x)
19   if (resultNeg ≤_antilex upperBound) return resultNeg
20   else return upperBound
21 }
```

Abbildung 4.4: Der Adopted Branch and Bound Algorithmus für A-Preferred MCS als Pseudo-Code.

Den Adopted Branch And Bound Algorithmus in seiner grundsätzlichen Form beschreibt der Pseudo-Code in Abbildung 4.4. Zuerst berechnet er ein Upper Bound in der Methode `calculate` und ruft damit die rekursive Hilfsfunktion `adoptedBAB` auf. Die Methode `simplify()` verkleinert die Menge der Klauseln anhand der bisherigen Belegung der Variablen. Erfüllte Klauseln werden aus den Mengen gelöscht und alle weichen unerfüllbaren Klauseln werden zu der Menge $\varphi''_{\text{soft}\emptyset}$ hinzugefügt. Die Menge der noch nicht erfüllten weichen Klauseln und die Menge der nicht erfüllbaren harten sowie weichen Klauseln wird zurückgegeben. Zusätzlich bestimmt die Funktion, ob eine der harten Klauseln nicht erfüllbar ist. Trifft dies zu, so befindet sich der aktuelle Zweig in einer Sackgasse, denn alle harten Klauseln müssen erfüllt sein. Im Folgenden prüft die Funktion, ob bereits alle harten und weichen Klauseln erfüllt worden sind. Ist dies der Fall, so ist $\varphi''_{\text{soft}\emptyset}$ ein Kandidat für das A-Preferred MCS.

Indem das Ergebnis der Methode `underestimation()` mit dem aktuellen Upper Bound verglichen wird, kann der Algorithmus frühzeitig Zweige aufgeben, die keine guten Kandidaten für ein A-Preferred MCS liefern. Die Methode `underestimation()` liefert eine Menge U zurück für die gilt $\varphi''_{\text{soft}\emptyset} \subseteq U$. Im Standardfall liefert die Methode einfach die Menge $\varphi''_{\text{soft}\emptyset}$ zurück. Hier bleibt ein Freiraum für weitere Optimierungen. Tritt keiner der genannten Fälle ein, dann sucht sich der Algorithmus eine unbelegte Variable aus und belegt diese. Für den ersten Zweig wird diese mit wahr belegt und für den zweiten Zweig entsprechend mit falsch. Das Ergebnis des ersten Zweigs vergleicht die Funktion mit dem derzeitigen Upper Bound und nimmt das jeweils bessere für die weitere Verwendung. Damit können im zweiten Zweig bereits einige Fälle eingespart werden. Entsprechend geht der Algorithmus mit dem Ergebnis des zweiten Zweigs um.

4.4.1 Weitere Optimierungen

Intuitiv wählt man bei jedem Aufruf der Methode `simplify()` eine Iteration über alle noch unerfüllten harten und weichen Klauseln. Dies nimmt allerdings unnötig viel Zeit in Anspruch. Die Methode `simplify` kann deutlich beschleunigt werden, wenn die klassischen Optimierungen eines SAT Solvers eingebaut werden. Dazu gehören Unit Propagation, *Watched Literals* und eine geeignete Heuristik für die Variablenwahl. In der Praxis zeigt sich jedoch, dass das Berechnen der Mengen φ''_{hard}, $\varphi''_{\text{soft}\emptyset}$ und φ''_{soft} selbst mit den gerade genannten Optimierungen recht aufwändig ist. Die Berechnung und das Mitführen der Mengen φ''_{hard} und φ''_{soft} kann aber komplett eingespart werden, wenn Watched Literals verwendet werden. Die Überprüfung in Zeile 9 entfällt dadurch. Dafür muss geprüft werden, ob es noch eine Variable gibt, die noch belegt werden kann. Falls nicht, so ist $\varphi''_{\text{soft}\emptyset}$ das optimale MCS für diesen Zweig. Damit zumindest der Vergleich der Klauseln schnell verläuft, empfiehlt es sich hier nur mit den Indexen die Mengen zu berechnen.

Kapitel 5

Implementierung

Im Kapitel 3 und 4 wurden mehrere Algorithmen zur Berechnung von präferierten Diagnosen vorgestellt. Bei der Implementierung und Analyse dieser Algorithmen stößt man auf verschiedene Herausforderungen. Mit diesen Punkten befasst sich Kapitel 5. Der Abschnitt 5.1 nimmt eine kurze Einführung in das *Warthog*-Projekt vor. Dieses stellt den SAT Solver sowie die nötigen Klassen zum Berechnen mit Klauseln und Variablen zur Verfügung. Je nachdem, wie die Algorithmen und deren Interfaces umgesetzt sind, ändert sich die Effizienz. Welche Optimierungen diesbezüglich möglich sind, erläutert Abschnitt 5.2. Die beiden Abschnitte 5.3 und 5.4 befassen sich mit der Effizienz der Anbindung zum SAT Solver und mit dem Aufwand, der bei jedem Test auf Erfüllbarkeit entsteht.

5.1 Warthog

Warthog ist ein Framework, das eine Sammlung an Tools bezüglich Aussagenlogik, Prädikatenlogik, Higher-Order-Logic, Temporale Logik und Beschreibungslogik beinhaltet [KZW15]. Es ist in weiten Teilen in Scala geschrieben. Für die Berechnung von präferierten Diagnosen benötigt diese Arbeit Klauseln, Variablen und Literale in Form der Aussagenlogik. Klauseln in der Aussagenlogik repräsentiert Warthog durch das Interface `ClauseLike[PL, PLLiteral]`. Die Klasse `PL` definiert, dass es sich um Klauseln in der Aussagenlogik handelt und `PLLiteral` die Art, wie Literale repräsentiert werden. Die Klasse `PartialWeightedMaxSATReader` bietet die Methode `read()` an, die WDIMACS-Dateien[1] einlesen kann. Das Ergebnis sind Klauseln, die das Interface `ClauseLike` implementieren. Für präferierte Diagnosen gibt es bisher keinen offiziellen Standard, daher verwendet diese Arbeit MaxSAT Probleme und interpretiert sie entsprechend. Dabei wird angenommen, dass die Reihen-

[1] Die Definition, wie diese Dateien aussehen befindet sich auf der Webseite http://www.maxsat.udl.cat/15/requirements/index.html

folge der Klauseln in den WDIMACS-Dateien die präferierte Reihenfolge ist. Die angegebenen Gewichte werden ignoriert.

Für alle Algorithmen zur Berechnung von präferierten Diagnosen wird ein allgemeines Interface `APreferredMCSSolver` definiert. Abbildung 5.1 präsentiert ein abgekürztes Klassendiagramm. Die Methode `addHardConstraint()` fügt die strikten Bedingungen hinzu und die Methode `solve()` bestimmt die präferierte Diagnose. Dazu testet sie zuerst die strikten Klauseln auf ihre Erfüllbarkeit mittels der Methode `areHardConstraintsSatisfiable` und gibt `None` zurück, falls sie nicht erfüllbar sind. Andernfalls ruft sie die Methode `solveImpl()` auf. Die tatsächliche Algorithmen, wie FastDiag oder die Lineare Suche, sind in der Methode `solveImpl()` zu implementieren. Unter anderem die gerade genannten Algorithmen benötigen einen SAT Solver. Die abstrakte Klasse `SATBasedApreferredMCSSolver` übernimmt die Anbindung des SAT Solvers und vermeidet redundanten Code.

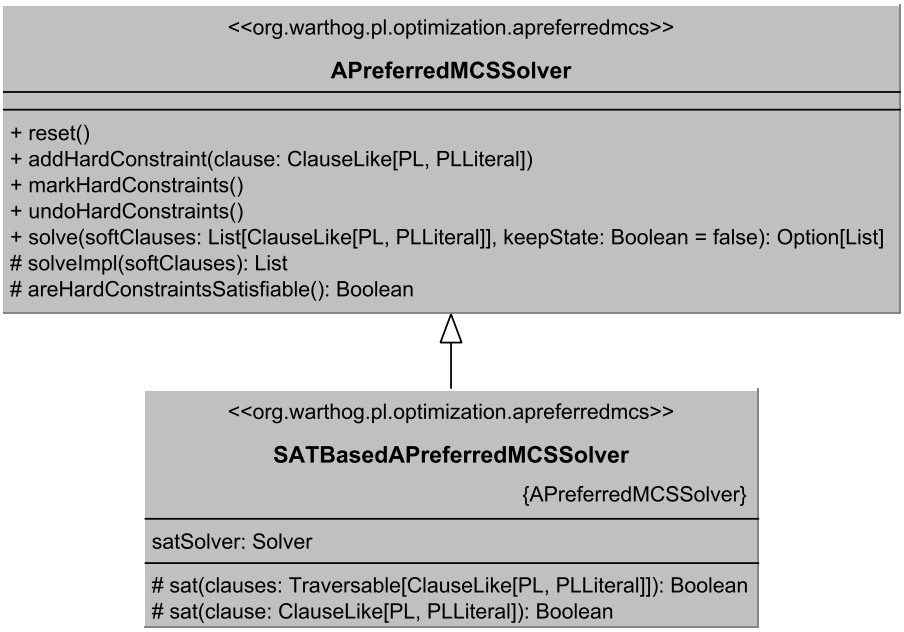


Abbildung 5.1: Das abgekürzte Klassendiagramm für das Interface `APreferredMCSSolver`.

Warthog bringt die zwei SAT Solver *PicoSat* als angebundene DLL und *MiniSat* als Java Übersetzung der C++ Implementierung mit. Da Änderungen am Solver für mehrere Ansätze notwendig sind, beschränkt sich diese Arbeit auf die Verwendung des MiniSat Solvers. PicoSat erfüllt diese Voraussetzung nicht. MiniSat ist ursprünglich in der Programmiersprache C von Niklas Eén und Niklas Sörensson geschrieben. Der MiniSat Algorithmus, der in der Klasse `MSJCoreProver` implementiert ist, kann mit dem Adapter `MiniSatJava`

verwendet werden. Dieser implementiert das allgemein formulierte Interface `Solver`. Mit der Methode `add()` können Klauseln hinzugefügt werden. Der Aufruf der Methode `sat()` führt die tatsächliche Berechnung aus und das SAT Solving durch. Der SAT Solver kann mit abwechselnden Aufrufen der Methoden `add()` und `sat()` umgehen. Allerdings können von einer Instanz keine Klauseln gelöscht werden. Der Adapter simuliert mittels der Methoden `mark()` und `undo()` das Löschen von Klauseln.

5.2 Spezifische Optimierungen der Implementierung

Bei der Formulierung als Pseudo-Code bleiben manchmal Aspekte unbedacht, die für die Implementierung wichtig sind. Die Formulierung in Form von Mengen ist in den neueren Sprachen ohne Probleme möglich. Auch Scala bietet hierfür eine Vielzahl an Funktionen an. Mit dem Wissen über die Art der Daten lässt sich die Verwendung von teuren Operationen wie der Vereinigung $\cup$ vermeiden. Bei FastDiag erhält man die Zwischenergebnisse zur Berechnung des A-Preferred MCS in geordneter Form; zudem kann keine Klausel doppelt vorkommen. Die Vereinigung der Zwischenergebnisse (Abbildung 3.3 Zeile 20) ist also nichts anderes als eine Konkatenation zweier sortierter Listen.

Das Aufteilen einer Liste in zwei Hälften (Abbildung 3.3 Zeile 17) und das entsprechende Aufbauen zwei neuer Listen ist relativ teuer. Eine Liste der Länge n benötigt mindestens n Berechnungsschritte, um die Liste in zwei weitere Listen aufzuteilen. Da diese Anzahl an Klauseln in den Arbeitsspeicher passt, kann schlicht der Index von Start und Ende übergeben werden. Das kostet einen Berechnungsschritt. Speichert man nun die Klauseln als Array ab, so erhält man einen ebenso schnellen Zugriff wie bei einer Iteration über die Liste. Man spart sich in diesem Fall aber das Erstellen der Listen in jedem Rekursionschritt.

Indem man die Ergebnisse nur als Indizes zurückgibt, wird dem Nutzer eine schnellere Analyse dessen, wie wichtig die einzelnen Klauseln der präferierten Diagnose sind, ermöglicht. Zusätzlich muss der Algorithmus nicht unnötigerweise mit den tatsächlichen Klauseln arbeiten.

5.3 Aufwand von SAT-Aufrufen minimieren

Jeder Aufruf des SAT Solvers kostet Rechenzeit. Analysiert man diese, zeigt sich, dass zu viel Zeit dafür verbraucht wird, in jedem Aufruf alle Klauseln neu hinzuzufügen. Jeder Aufruf beinhaltet stets alle harten Klauseln. Zudem sind alle Klauseln, welche die Menge Γ speichert, ebenfalls harte Klauseln. Dazu erhält das Interface `Solver` die zusätzliche Methode `addHard()`. Klauseln

die damit hinzugefügt werden, sind von den Methoden `mark()` und `undo()` nicht betroffen. So sparen die Algorithmen das Aufbauen der Menge Γ ein und müssen die Menge Γ nicht weitere Male hinzufügen. Vor jedem Test auf Erfüllbarkeit mit einer neuen Klausel merkt sich der Algorithmus mit der Methode `mark()` den Zustand. Danach fügt er die zu überprüfenden Klauseln hinzu und führt das SAT Solving durch. Schließlich führt er die Methode `undo()` aus und der SAT Solver ist wieder in seinem ursprünglichen Zustand.

Eine tiefergehende Analyse ergibt, dass vor allem die Methode `undo()` viel Zeit kostet. Jeder Aufruf der Methode `undo()` führt dazu, dass eine komplett neue Instanz der Klasse `MSJCoreProver` erstellt wird und alle bisher hinzugefügten Klauseln erneut hinzugefügt werden. Eine weitere Optimierungsmöglichkeit besteht somit darin, die Anzahl der `undo()` Aufrufe zu minimieren. Dabei fällt auf, dass jede Klausel, die zum L-Preferred MSS Γ gehört direkt im SAT Solver verbleiben kann und der letzte Aufruf von `mark()` ignoriert werden kann. Damit überflüssige Markierungen den SAT Solver Adapter nicht verlangsamen erhält das Interface `Solver` noch die Methode `forgetLastMark()`. Alle Klauseln, die nach einer Markierung hinzugefügt werden, werden mit dem Aufruf dieser Methode zu harten Klauseln.

```
1  Input: φ_hard, φ_soft = {c_1,...,c_m} with c_1 < ... < c_m
2  Output: A-Preferred MCS Δ
3  Δ = ∅
4  satSolver.addHard(φ_hard)
5  for (i = 1 to m) {
6    satSolver.mark()
7    satSolver.add(c_i)
8    if (satSolver.sat())) {
9      satSolver.forgetLastMark()
10   } else {
11     satSolver.undo()
12     Δ = Δ ∪ {c_i}
13   }
14 }
15 return Δ
```

Abbildung 5.2: Der lineare Suche Algorithmus für A-Preferred MCS als Pseudo-Code in optimierter Form.

Die gerade beschriebenen Optimierungen zeigt der Pseudo-Code der Linearen Suche in Abbildung 5.2. Der Quellcode nimmt dabei an, dass der Adapter in der Variable `satSolver` repräsentiert wird. Entsprechend der obigen Erläuterung entfällt die Verwendung der Variable Γ. In der Schleife wird in jeder Iteration mit der Methode `mark()` eine Markierung gesetzt. Je nachdem, ob der Test erfolgreich ist oder nicht, folgt das Löschen der letzten Klausel aus dem SAT Solver. Alternativ wird die Klausel behalten und die letzte Markierung wird mittels der Methode `forgetLastMark()` gelöscht. Dies hat zur Folge, dass die lineare Suche nur noch exakt so viele Aufrufe der teuren Methode `undo()` benötigt, wie es Klauseln im A-Preferred MCS gibt.

5.4 Optimierung bei der Verwendung des SAT Solvers

Der letzte Abschnitt thematisiert unter anderem das Problem, dass die Berechnung maßgeblich durch den Aufruf der Methode `undo()` gebremst wird. Das Ziel ist es, dass beim Löschen der Klauseln keine neue Instanz der SAT Solvers erstellt werden muss. Dazu bieten sich die Assumptionvariablen der MiniSat Implementierung an [ES04]. Das Vorgehen ähnelt dabei dem des modifizierten SAT Solvers, der in Abschnitt 4.3 beschrieben wird. Jede weiche Klausel c_i, die eventuell wieder entfernt werden muss, erhält eine sogenannte *Assumptionvariable* a_i. Die Klausel c_i erhält damit a_i als zusätzliches Literal. Der SAT Solver erhält die Anweisung, die Assumptionvariable a_i entsprechend mit falsch zu belegen. Damit hat die Assumptionvariable zunächst keinen Einfluss auf das Ergebnis. Sollte die Klausel nicht erfüllbar sein und muss gelöscht werden, so kann die Assumptionvariable a_i auf wahr gesetzt werden. Damit ist die Klausel c_i sofort erfüllt und wird vom SAT Solver nicht mehr beachtet. Das eingangs erwähnte Ziel ist erreicht.

Die MiniSat Implementierung in Warthog übernimmt dabei einen großen Teil des Aufwands; denn diese ermöglicht es bereits beim Aufruf der Methode `solve()`, eine Variablenbelegung in Form eines Vektors mit zu übergeben. Der optimierte Adapter, der genau diese Funktionalität einsetzt, ist in der Klasse `MiniSatAssumption` implementiert. Dieser verwendet HashMaps, um sich die Beziehung zwischen Assumptionvariablen und Klauseln zu speichern und einen schnellen Zugriff zu ermöglichen. Dabei beachtet er, dass jede Klausel auch tatsächlich nur einmal hinzugefügt wird. Problematisch hierbei ist allerdings der sich stets vergrößernde Vektor mit den Assumptionvariablen. Dazu bietet es sich an zusätzlich den Adapter nach mehreren SAT Aufrufen zurückzusetzen und so die Anzahl an Assumptionvariablen klein zu halten. Der Zurücksetzen Vorgang verläuft ähnlich dem, was die `MiniSatJava` Implementierung bei jedem Aufruf der Methode `undo()` macht. Es wird eine neue Instanz des SAT Solvers erstellt und die Klauseln neu hinzugefügt. Weiche Klauseln, die in der Zwischenzeit hart geworden sind, werden als solche hinzugefügt. Entsprechend werden Klauseln, die mit den letzten `undo()` Aufrufen entfernt wurden komplett gelöscht. Damit bleiben nur noch Assumptionvariablen für die verbleibenden Markierungen übrig. Womit der Assumptionvektor auch klein bleibt und auch die Anzahl der Variablen nicht durch die Assumptionvariablen unnötig ansteigt.

Weiter testet diese Arbeit eine neue Idee für einen optimierten Adapter. Dieser ist in der Klasse `MiniSatOpt` implementiert. Dieser Ansatz verwendet ebenso Assumptionvariablen; allerdings prüft dieser Ansatz nicht, ob eine Klausel schon einmal hinzugefügt wurde. Damit erspart sich der Adapter die Verwendung von HashMaps für die Zuordnung der Assumptionvariablen und Klauseln. Die Folge ist, dass es bei FastDiag und General Chunks Algorith-

mus vorkommt, dass eine Klausel in Kombination mit verschiedenen Assumptionvariablen existiert. Angenommen die Klausel c_i wird über mehrere SAT Aufrufe hinweg zweimal hinzugefügt, dann existieren im SAT Solver die Klauseln $c_i \vee a_1$ und $c_i \vee a_2$. Der SAT Solver muss also deutlich mehr Klauseln verarbeiten. Dadurch, dass eine feste Ordnung besteht können allerdings bei jedem Aufruf der Methode `undo()` die zugehörigen Assumptionvariablen als Unit Klausel übergeben werden. Daraus folgt, dass der Vektor mit den Assumptionvariablen durchgängig klein bleibt. Außerdem lässt sich bei diesem Ansatz auch die Anzahl der Assumptionvariablen verkleinern. So reicht es aus für jede Markierung eine Assumptionvariable zu verwenden. Alle Klauseln, die nach der Markierung hinzugefügt werden, erhalten dieselbe Assumptionvariable zugeordnet. Damit muss beim Aufruf der Methode `undo()` nur diese eine Assumptionvariable als Unit Klausel hinzugefügt werden und alle Klauseln sind direkt erfüllt.

5.5 Model Exploiting

Das Model Exploiting ist eine Optimierungsansatz, der für mehrere Algorithmen verwendet werden kann. Für die Umsetzung benötigen wir das Model des SAT Solvers. Durch das Model Exploiting werden nach jedem SAT Aufruf nur wenige paar Klauseln geprüft, weswegen es zu aufwendig ist jedes Mal ein vollständiges Model zu erstellen. Weiter benötigt der SAT Solver dafür die Zuordnung der Variablen und der internen Repräsentation der Variablen im SAT Solver. Dafür muss der SAT Solver eine zusätzliche HashMap aufbauen. Um den Zustand der einzelnen Variablen abzufragen, ist die zusätzliche Methode `getVarState(variable)` für das Interface `Solver` erforderlich. Diese gibt dem Anwender nach einem erfolgreichen SAT Aufruf den Zustand der mit übergebenen Variable zurück. Die Klasse `ModelExploiting` bietet für die Überprüfung einer Klausel auf ihre Erfüllbarkeit die Methode `isSat()` an. Diese ruft für jedes in der Klausel vorkommende Literal die Methode `getVarState()` des SAT Solvers auf. Ist das Literal mit der erhaltenen Belegung erfüllt, so gibt es wahr zurück.

Kapitel 6

Experimentelle Evaluierung

Nach der ausführlichen Analyse der Algorithmen und Optimierungen zur Berechnung von präferierten Diagnosen in den letzten Kapiteln befasst sich dieses Kapitel mit der Evaluierung dieser Vorschläge. Dazu verwendet diese Arbeit einen Benchmark. Wie dessen Konfiguration aussieht und welche Dinge zu beachten sind, bespricht Abschnitt 6.1. Daraufhin werden die verschiedenen SAT Solver Adapter in Abschnitt 6.2 unter Effizienzgesichtspunkten betrachtet. Im Anschluss daran werden die Ergebnisse der Algorithmen lineare Suche und FastDiag in den Abschnitten 6.3 und 6.4 untersucht. Danach analysiert Abschnitt 6.5 den General Chunks Algorithmus und betrachtet verschiedene Strategien dafür. Die Abschnitte 6.6 und 6.7 ermitteln die Effizienz der restlichen beiden Algorithmen: Der modifizierte SAT Solver und der Adopted Branch and Bound Algorithmus. Im Anschluss daran werden die besseren Algorithmen mit einer Anzahl an größeren Instanzen auf die Probe gestellt. Abschnitt 6.8 analysiert deren Ergebnisse.

6.1 Benchmarkkonfiguration

Für die Analyse der Algorithmen verwendet diese Arbeit mehrere Instanzen des *ijcai13* Benchmarks [MSHJ+13a]. Der Benchmark ist konzipiert für die Berechnung allgemeiner MCS. Mit der bereits in Abschnitt 5.1 erläuterten Interpretation lässt sich der Benchmark auch für die Analyse von Algorithmen zur Berechnung von präferierten Diagnosen verwenden. Die einzelnen Instanzen kommen von verschiedenen SAT und MaxSAT sowie Partial MaxSAT Wettbewerben. Zusätzlich sind einige der Instanzen aus originalen industriellen CNF Instanzen generiert. Diese Arbeit teilt die aus dem Benchmark ausgewählten Instanzen in zwei Bereiche auf.

Zum einen sind das die kleineren Instanzen[1], welche für alle zu analysie-

[1] Die kleineren Instanzen umfassen die Dateien aus den Ordnern *mm-s11*, *mm-s9* und *pms-bcp-fir-1*.

renden Algorithmen verwendet werden. Insgesamt sind dies 63 verschiedene Instanzen. Diese Instanzen haben zwischen 10.000 und 42.000 harte Klauseln und zwischen 400 und 8700 weiche Klauseln. Die Anzahl der Variablen liegt bei diesen Instanzen zwischen 22.000 und 109.000. Zum Anderen gibt es die größeren Instanzen[2], gegen die nur die besseren Algorithmen antreten. Die Instanzen haben zwischen 80.000 und 500.000 harte Klauseln und zwischen 160 und 100.000 weiche Klauseln. Dabei benötigen sie zwischen 204.000 und knapp 1.600.000 Variablen pro Instanz. Ausgelassen werden lediglich die Instanzen, die zu groß sind für die Berechnung, und die, die zu klein sind, um tatsächlich relevant zu sein. Das Zeitlimit für die kleineren Instanzen beträgt dabei 60 Sekunden und für die größeren Instanzen 180 Sekunden. Diese Beschränkung ist nötig, um die Dauer des Benchmarks in einem zumutbaren Rahmen zu halten.

Ausgeführt wurde der Benchmark auf einem Windows 10 Rechner mit einem Intel Core i5-3570K mit 3,4 GHz und 8 GB Arbeitsspeicher. Der Benchmark misst die Millisekunden, die verstreichen, während die Methode `solve()` des A-Preferred MCS Solvers ausgeführt wird. Verglichen wird dabei die Zeit, die pro Instanz im Durchschnitt benötigt wurde. Zur Ausführung wird Java in der Version *1.8.0_51* und Scala in der Version *2.10.5* eingesetzt.

6.2 Analyse der SAT Solver Adapter

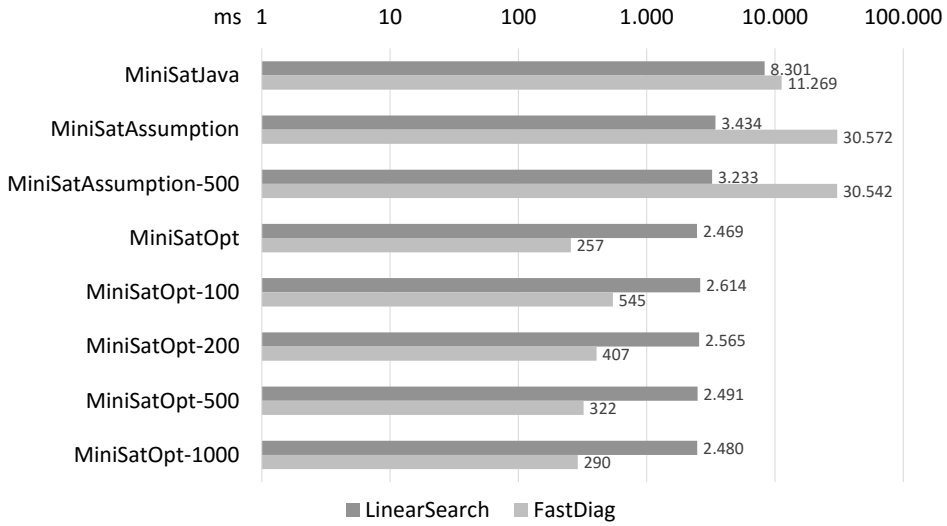


Abbildung 6.1: Die Benchmark Ergebnisse für die verschiedenen SAT Solver Adapter in logarithmischer Skala.

Bevor die einzelnen Algorithmen auf ihre Effizienz verglichen werden, ana-

[2]Die größeren Instanzen umfassen 59 Dateien aus den Ordnern *mm-sat-mes-1*, *pms-bcp-fir-2*, *pms-bcp-hipp-yRa1-su-1* und *pms-bcp-hipp-yRa1-su-2*.

lysiert dieser Abschnitt die Effizienzunterschiede der SAT Solver Anbindung. Die Algorithmen für lineare Suche und FastDiag, die hier getestet werden sind bereits optimiert. Alle in Abschnitt 5.3 erläuterten Optimierungen - zum Beispiel das Vermeiden von unnötigen `undo()` Operationen - sind bereits berücksichtigt. Wie Abbildung 6.1 zeigt, macht die Wahl der SAT Solver Anbindung einen großen Unterschied. Bei der Betrachtung des Schaubildes ist zu beachten, dass dieses eine logarithmische Skala verwendet, um die Differenz der schnelleren Varianten zu veranschaulichen.

Der Adapter `MiniSatJava` ist die Standardvariante einer SAT Solver Anbindung und ist - wie zu erwarten war - eine der langsameren. Zu Beachten ist allerdings, dass FastDiag bei drei Instanzen die Berechnung der maximalen Berechnungszeit überschritten hat und somit theoretisch mehr Zeit benötigt hätte. Der Adapter vergeudet bei jedem Aufruf der Methode `undo()` viel Zeit. Die Analyse zeigt, dass der lineare Suche Algorithmus 58% seiner benötigten Zeit in der `undo()` Operation verbringt. Beim FastDiag Algorithmus sind es sogar 83%.

Der Adapter `MiniSatAssumption`, der in Abschnitt 5.4 eingeführt wurde, geht genau auf diesen Schwachpunkt ein. Die lineare Suche zeigt hier bereits eine deutliche Beschleunigung der Berechnung und benötigt nur noch $3,4$ Sekunden. Der Aufruf der Methode `undo()` nimmt $2,4\%$ der Rechenzeit in Anspruch. Problematisch ist allerdings das Ergebnis, dass der FastDiag Algorithmus mit dem `MiniSatAssumption` Adapter erzielt. Dieser übertritt bei 29 Instanzen das Zeitlimit und ist schlechter als die Standardvariante, was daran liegt, dass die Methode `add()` des Adapters zu viel Zeit verschlingt. Liegt die Anzahl an weichen Klauseln über 2000, so benötigt die HashMap der Methode `add()` zu viel Zeit. Da bei FastDiag alle Klauseln zuerst auf Erfüllbarkeit geprüft werden, bremst das bereits stark aus. Die Prüfung, ob Klauseln doppelt hinzugefügt werden, ist demnach eher bremsend. Bei der linearen Suche wird jede Klausel nur einmal hinzugefügt, daher fallen die Aufrufe der Methode `add` nicht ins Gewicht. Die Variante `MiniSatAssumption-500` verfügt noch über eine zusätzliche Funktionalität. Sie führt das vollständige Zurücksetzen des Adapters alle 500 Aufrufe der Methode `undo()` durch, wie Abschnitt 5.4 beschrieben hat. Diese Variante zeigt zugleich leichte Geschwindigkeitsfortschritte für beide Algorithmen, da der Vektor mit den Assumptionvariablen tendenziell kleiner bleibt und die Überprüfung, welche Klauseln bereits hinzugefügt wurden, von der kleineren HashMap profitiert.

In dem zuletzt vorgeschlagenen Adapter sind die restlichen Optimierungen eingebaut. Die Variante `MiniSatOpt` spart sich die HashMap und verwendet nur eine Assumptionvariable pro Markierung. Diese Optimierung bringt selbst bei der linearen Suche eine Beschleunigung von etwa einer Sekunde. Diese profitiert lediglich aus der Einsparung der HashMap. Mit dem FastDiag Algorithmus kann die Berechnungszeit sogar um mehr als das 40-fache beschleunigt werden. Die Varianten mit Zurücksetzen sind stets schlechter als

die Varianten ohne Zurücksetzen. Dies hat mehrere Gründe. Zum einen ist der Assumptionvektor stets klein und besteht aus nur einer Variable. Zum anderen übersteigt der Aufwand, der durch wenige zusätzliche Variablen im SAT Solver entsteht, den Aufwand des vollständigen Zurücksetzens. Des Weiteren werden gelernte Klauseln beim Erstellen einer neuen Instanz vergessen. Diese ersparen Konflikte bei weiteren Aufrufen und können daher die Berechnung beschleunigen. Weiter zeigt dieser Benchmark, dass sich die eingesparten SAT Aufrufe bei einer guten Implementierung des SAT Solver bezahlt machen. FastDiag benötigt bei den gewählten Instanzen im Durchschnitt 30% weniger SAT Aufrufe als die lineare Suche. Im Ergebnis stellt sich der Adapter `MiniSatOpt` als der Schnellste heraus. In den folgenden Benchmarks wird daher nur noch dieser für alle Algorithmen verwendet, die eine SAT Solver Anbindung benötigen.

6.3 Analyse des linearen Suche Algorithmus

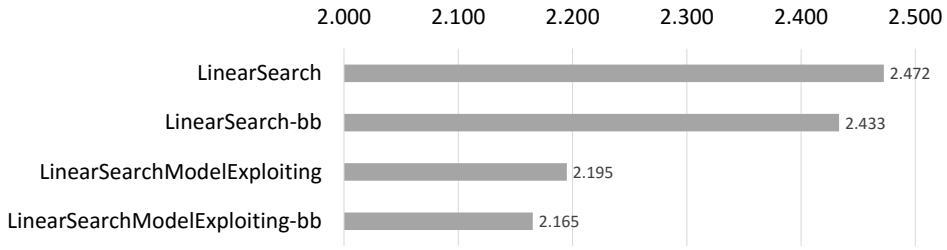


Abbildung 6.2: Die Benchmark Ergebnisse für Optimierungen des linearen Suche Algorithmus.

Die lineare Suche stellt das Standardvorgehen zur Lösung eines A-Preferred MCS dar. Durch Backbone Literale, bezeichnet als `bb`, und Model Exploiting lässt sich dieser Algorithmus ein wenig beschleunigen. Während die Backbone Literale die Effizienz nur um lediglich $1,5\%$ verbessern, kann mit Model Exploiting die Berechnungszeit immerhin um 11% beschleunigt werden. Beim Betrachten der SAT Aufrufe, wird der Grund ersichtlich: Die lineare Suche benötigt durchschnittlich 2.656 SAT Aufrufe ohne Optimierungen. Durch Model Exploiting können durchschnittlich 761 SAT Aufrufe vermieden werden, was einer Einsparung von etwa 28% entspricht. Die Kombination von beiden Optimierungen bringt für die lineare Suche das beste Ergebnis.

6.4 Analyse des FastDiag Algorithmus

Der zweite Algorithmus, der auf seine Optimierungen überprüft wird, ist FastDiag. Hierfür kombinieren wir jede Möglichkeit der einzelnen Optimierungen.

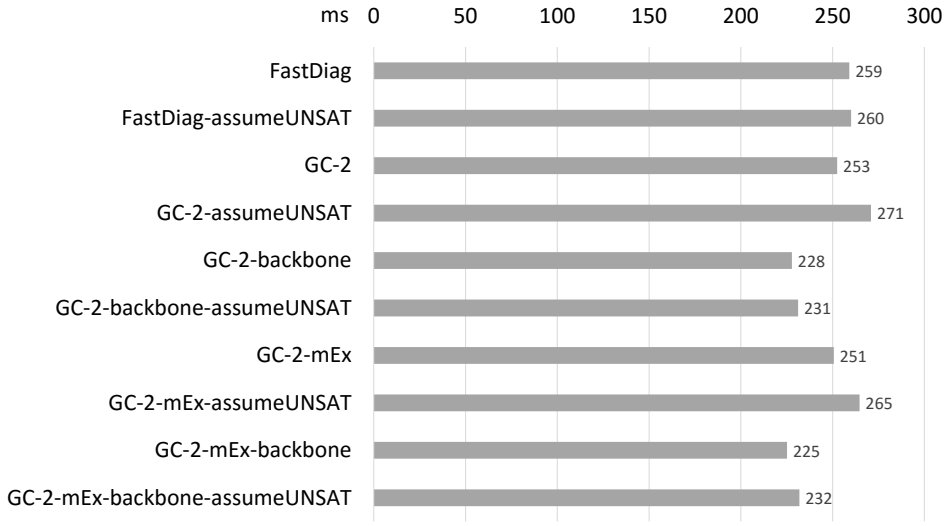


Abbildung 6.3: Die Benchmark Ergebnisse für die Varianten von FastDiag.

Dazu gehört das Einsparen des ersten SAT Aufrufs, welcher in der Abbildung 6.3 mit `assumeUNSAT` bezeichnet wird. Des Weiteren werden Backbone Literale verwendet, die mit `backbone` bezeichnet werden. Die Optimierung Model Exploiting kürzt die Abbildung mit `mEx` ab. Jegliche Optimierungen wurden aus praktischen Gründen nur in den General Chunks Algorithmus integriert. Wie in Abschnitt 4.2.2 gezeigt, kann mit der geeigneten Strategie die Funktionalität von FastDiag simuliert werden. Weiter bringt der General Chunks Algorithmus die kleine Optimierung mit, dass dieser nur mit den Indexen über Arrays arbeitet und somit Operationen über Listen einspart. Diese Optimierung erspart im Durchschnitt $6ms$ und ist damit ein kleiner Schritt zu einer besseren Performance.

Es zeigt sich, dass die Optimierung `assumeUNSAT` keine Einsparung bringt. Bei jeder Kombination ist die Variante mit der Optimierung langsamer. Der Grund kann auch durch zusätzliche Tests nicht benannt werden. Die Analyse zeigt, dass die Zeit steigt, die der SAT Solver mit dem Lösen verbringt. Dies ist nicht bei allen Instanzen der Fall, bei manchen Instanzen bringt der eingesparte SAT Aufruf auch eine Zeitersparnis. Vermutlich lernt der SAT Solver im ersten Aufruf bei manchen Instanzen bereits eine Vielzahl wichtiger Klauseln, die für alle folgenden Aufrufe Einsparungen mit sich bringt. Mit den anderen beiden SAT Solver Adaptern `MiniSatJava` und `MiniSatAssumption` bringt diese Optimierung eine messbare Verkürzung der Berechnungszeit. Dort verursacht jeder SAT Aufruf erhebliche Kosten durch die Methoden `add()` und `undo()`, wie bereits in Abschnitt 6.2 erläutert.

Die Backbone Literale bringen eine erhebliche Beschleunigung der Berechnung mit sich (etwa 10%). Die folgenden SAT Aufrufe profitieren demzufolge enorm davon, dass mehrere Variablen bereits im Voraus belegt sind. Im Gegen-

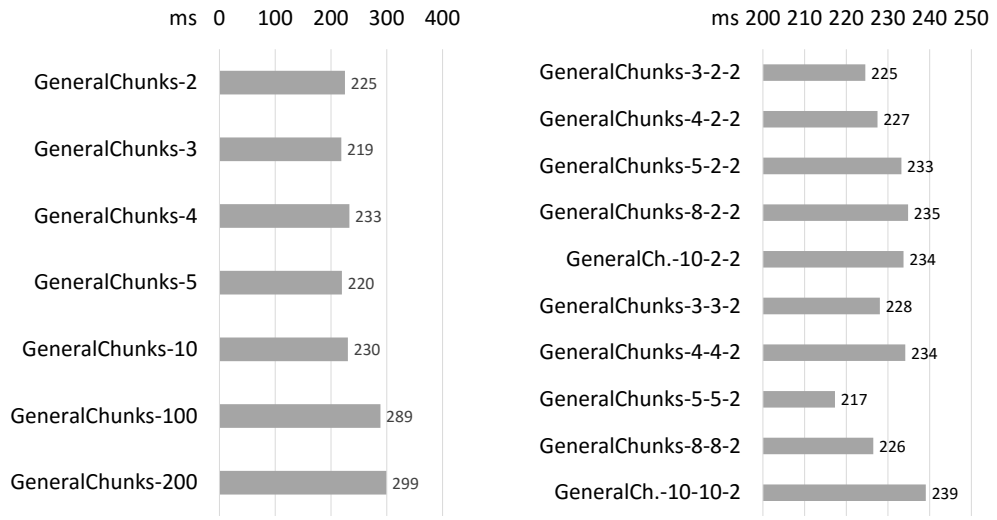
satz zur linearen Suche kann mit dem Model Exploiting die durchschnittliche Berechnungszeit nur um zwei Millisekunden erhöht werden. Die Einsparung an SAT Aufrufen durch Model Exploiting liegt bei durchschnittlich 30. Dies sind nur $1,5\%$ der durchschnittlich notwendigen SAT Aufrufe; daher verbessert sich die Geschwindigkeit nur minimal. Dies liegt unter anderem daran, da durch das Aufteilen des Problems in zwei Teilprobleme die Verkleinerung durch Model Exploiting nur minimal ist. Oft werden bereits mehrere Klauseln gleichzeitig als SAT erkannt und die Wahrscheinlichkeit, dass danach noch weitere erfüllbare Klauseln folgen, sinkt folglich. Die Kombination der beiden Optimierungen bringt auch bei FastDiag die besten Ergebnisse hervor und nimmt im Durchschnitt 225 Millisekunden in Anspruch.

6.5 Analyse des General Chunks Algorithmus

Die Effizienz des General Chunks Algorithmus hängt stark von der gewählten Strategie ab. Die Strategie des FastDiag Algorithmus wurde bereits im letzten Abschnitt erläutert. Verschiedene weitere Strategien wurden in Abschnitt 4.2.1 vorgestellt. Diese sollen nun genauer untersucht werden. Mit dem Model Exploiting und den Backbone Literalen kann die Effizienz deutlich gesteigert werden, wie mehrere Tests gezeigt haben. Grundsätzlich werden daher nur noch unterschiedliche Strategien betrachtet, welche beide dieser Optimierungen verwenden. Die Optimierung mit der Einsparung des ersten SAT Aufrufs kann - je nach Wahl der konkreten Strategie - die Rechenzeit beschleunigen, aber in manchen Fällen auch die Rechenzeit verlängern. Im Folgenden werden daher nur die Ergebnisse ohne diese Optimierung betrachtet.

Die Vorgehensweise von FastDiag legt nahe, dass eine Aufteilung in eine feste Anzahl an Teilmengen sinnvoll erscheint. Die Abbildung 6.4a zeigt die Ergebnisse des Benchmarks. Die Abbildung lässt erkennen, dass eine Aufteilung in mehrere Teilmengen die Geschwindigkeit erhöhen kann. Ist die Anzahl der Teilmengen allerdings zu hoch, so sinkt die Geschwindigkeit wieder. Die optimale Aufteilung wird mit drei Teilmengen erreicht, bei der im Durchschnitt nur 219 Millisekunden benötigt werden. Dass die Aufteilung in viele Teilmengen verlangsamend wirkt, ist unerwartet. Allerdings haben die Strategien, die viele Teilmengen bilden, einen Schwachpunkt bei Instanzen, bei denen unter 2% der weichen Klauseln zum A-Preferred MCS gehören. Da zwischen den einzelnen Klauseln, die zum A-Preferred MCS gehören, viele erfüllbare Klauseln liegen, kann eine Strategie, die wenige Teilmengen bildet, frühzeitig große Teilmengen als erfüllbar erkennen. Daraus lässt sich folgern, dass das Größenverhältnis zwischen des A-Preferred MCS und den weichen Klauseln bestimmt, welche Strategie optimal ist.

Alternativ zu der festen Aufteilung schlägt [WFK15b] die Strategie vor, im ersten Rekursionsschritt mehrere Teilmengen und in allen darauf folgenden



(a) Die Strategie `constant`. **(b)** Die Strategie `hierarchized`.

Abbildung 6.4: Die Benchmark Ergebnisse für den General Chunks Algorithmus.

kleinere Teilmengen zu bilden. Die Abbildung 6.4b zeigt die Ergebnisse dieser Variante. Der Benchmark umfasst zwei Varianten der Strategie. So werden zum Einen im ersten Rekursionsschritt mehrere Teilmengen gebildet und zum Anderen werden in den ersten beiden Rekursionsschritten mehrere Teilmengen gebildet. Dabei gibt die Bezeichnung jeweils an, wie viele Teilmengen gebildet werden. Zum Beispiel bildet die eher langsame Strategie `GeneralChunks-8-8-2` im ersten und zweiten Rekursionsschritt acht Teilmengen und in allen darauf folgenden bildet sie nur noch zwei Teilmengen. Im Ergebnis liefern alle gewählten Strategien relativ gute Ergebnisse. Das beste Ergebnis liefert dabei die Strategie `5-5-2` mit 217 Millisekunden. Es zeigt sich, dass es sich lohnt in den ersten beiden Rekursionsschritten mehrere Teilmengen zu bilden. Hier ist ebenso zu beobachten, dass die durchschnittliche Effizienz am höchsten ist, wenn eine Strategie gewählt wird, die nicht allzu viele und nicht allzu wenige Teilmengen bildet.

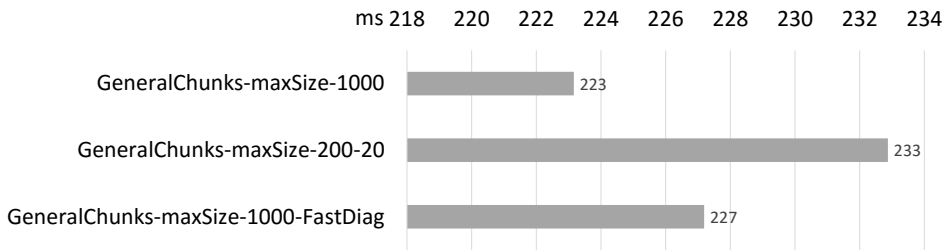


Abbildung 6.5: Die Benchmark Ergebnisse für den General Chunks Algorithmus mit verschiedenen Strategien.

Schließlich gibt es auch noch weitere Strategien. Diese reichen jedoch alle nicht an die Effizienz der bisher genannten heran. Abbildung 6.5 zeigt die besten Ergebnisse der weiteren Strategien an. Die erste Strategie `maxSize-1000` bildet so viele Teilmengen wie nötig sind, damit jede Teilmenge 1.000 Klauseln enthält. Im Gegensatz zur vorherigen Strategie ist nicht die Anzahl der Teilmengen, sondern die Größe der Teilmengen festgelegt. Alle Folgenden Klauseln werden einzeln behandelt. Die zweite Strategie bildet im ersten Rekursionsschritt Teilmengen der Größe 200 und im zweiten Teilmengen der Größe 20. Daraus folgt, dass es im zweiten Rekursionsschritt nie mehr als 10 Teilmengen gibt. Die dritte Variante, die `maxSize-1000-FastDiag` genannt wird, teilt erst die Klauseln in Teilmengen der Größe 1.000 auf und führt danach die FastDiag Strategie durch. Sie bildet also ab dem zweiten Rekursionsschritt zwei Teilmengen. Ruft man sich ins Gedächtnis, dass die Instanzen zwischen 400 und 8700 Softklauseln haben, so bemerkt man, dass eine ähnliche Aufteilung erfolgt wie bei den obigen Strategien. Es werden im ersten Rekursionsschritt zwischen zwei und maximal neun Teilmengen gebildet. Als Ergebnis kann also festgehalten werden, dass der General Chunks Algorithmus effizient ist, wenn die Anzahl der Buckets unter zehn liegt.

6.6 Analyse des modifizierten SAT Solver Algorithmus

Der modifizierte SAT Solver Algorithmus erfordert eine differenzierte Betrachtung. Er benötigt im Durchschnitt relativ schlechte 2, 4 Sekunden und bei einer Instanz überschreitet er das Zeitlimit.

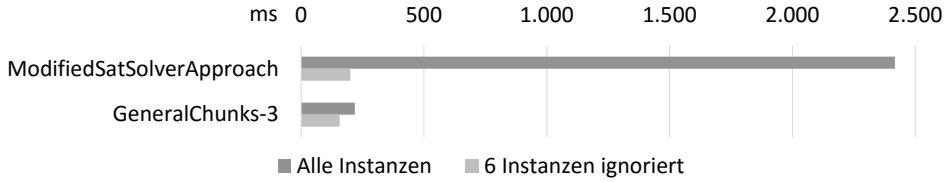


Abbildung 6.6: Ein Vergleich der Ergebnisse des modifizierten SAT Solver mit denen des General Chunks Algorithmus.

Vergleicht man den modifizierten SAT Solver mit einer guten Strategie des General Chunks Algorithmus, so fällt auf, dass bei etwa 60% der Instanzen der modifizierte SAT Solver schneller oder mindestens gleich schnell ist. Bei diesen 60% der Instanzen ist er überdies nicht nur ein wenig, sondern mit Abstand schneller. Durchschnittlich spart der modifizierte SAT Solver 58% der Rechenzeit ein. Zusätzlich spart er bei zwölf Instanzen sogar zwischen 80% und 90%.

Ignoriert man alle sechs Instanzen, die Ergebnisse für den modifizierten SAT Solver liefern, welche über vier Sekunden liegen, dann sieht das Verhältnis bereits ganz anders aus. Die Abbildung 6.6 zeigt das Ergebnis dieser alternativen Betrachtung. Der modifizierte SAT Solver ist damit um mehr als das zehnfache schneller während die Effizienz des General Chunks nur um relativ wenig steigt. Einige wenige Instanzen beeinflussen das Ergebnis damit mehr als überproportional. Um den tatsächlichen Schwachpunkt zu erkennen, ist eine tiefergehende Analyse der einzelnen Instanzen notwendig. Zusammengefasst erreicht dieser Algorithmus in manchen Fällen herausragende Ergebnisse und eine weitere Betrachtung erscheint sinnvoll.

Zusätzlich zu den genannten Optimierungen wurden noch weitere Optionen auf ihre Effizienz überprüft. Die MiniSat Implementierung verwendet verschiedene Ansätze zur Optimierung. So benutzt MiniSat sogenannte Restarts, um zu verhindern, dass sich der SAT Solver in lokalen Minima verfängt. Da der modifizierte SAT Solver größtenteils eine feste Variablenwahl vorgibt, kann dieser nur wenig davon profitieren. Die Ergebnisse zeigen, dass es sich lohnt, keine Restarts zu verwenden. Weiter gibt es Heuristiken, welche gelernte Klauseln löschen, wenn diese eine zu niedrige Aktivität aufweisen. An und für sich wäre zu erwarten, dass bei einer festen Variablenwahl diese Funktionalität nicht förderlich sein kann. Hier zeigt aber das Ergebnis, dass sich diese Optimierung auch für die modifizierte Variante lohnt und das Löschen von wenig verwendeten gelernten Klauseln die Effizienz steigert.

6.7 Analyse des Adopted Branch and Bound Algorithmus

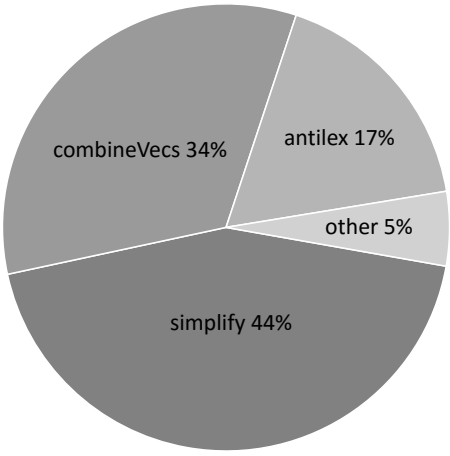


Abbildung 6.7: Die relative Zeit, die der Adopted Branch and Bound Algorithmus in einzelnen Methoden während des Benchmarks verbringt.

Der Adopted Branch and Bound Algorithmus benötigt im Durchschnitt 40 Sekunden und überschreitet bei 41 Instanzen das Zeitlimit. Er ist damit der langsamste aller bisher betrachteten Algorithmen und dies obwohl viele Optimierungen umgesetzt wurden, die den Algorithmus um mehrere Größenordnungen beschleunigt haben. Die Reduktion, dass nur noch die Menge, beziehungsweise die Liste, $\varphi'_{\text{soft}\emptyset}$ gebildet werden muss, sowie die Verwendung von Watched Literals bringen zwar eine große Beschleunigung, reichen aber am Ende nicht aus. Wie Abbildung 6.7 zeigt, verbringt der Algorithmus viel Zeit in wenigen Methoden. Die Methode `combineVecs` benötigt so zum Beispiel 34% der Berechnungszeit und ist lediglich dafür zuständig die von `simplify` erkannten leeren Klauseln in die sortierte Liste der bereits leeren Klauseln einzusortieren. Ebenso benötigt der Vergleich, welchen die Methode `antilex` durchführt, zu viel Zeit. Dies ist vor allem dann relevant, wenn die Anzahl der weichen Klauseln hoch ist. Wie zu erwarten ist nimmt die Methode `simplify` einen Großteil der Berechnungszeit in Anspruch. Im Ergebnis benötigt der Algorithmus noch eine entscheidende weitere Optimierung, damit er mit den anderen Vorschlägen konkurrieren kann.

6.8 Allgemeine Analyse mit größeren Instanzen

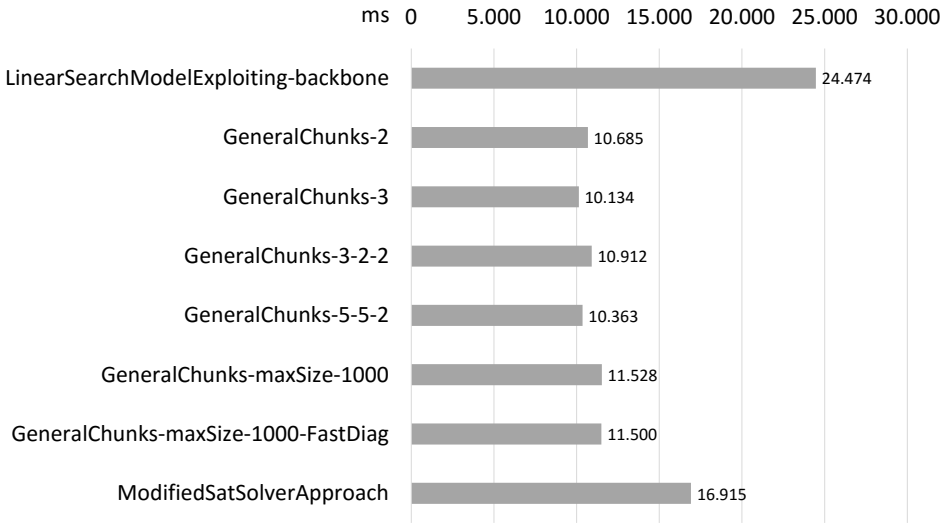


Abbildung 6.8: Die Benchmark Ergebnisse bei den größeren Instanzen.

Die Unterschiede zwischen verschiedenen Optimierungen fallen bei den kleineren Instanzen eher gering aus. Deshalb folgt nun ein Vergleich der schnellsten Algorithmen auf den größeren Instanzen. Dabei werden verschiedene Strategien des General Chunks Algorithmus, sowie die lineare Suche und der modifizierten

SAT Solver Algorithmus betrachtet. Alle Algorithmen verwenden wie bisher die Optimierung durch Model Exploiting und Backbone Literale. Der General Chunks Algorithmus mit der Strategie, in jedem Schritt drei Teilmengen zu bilden, kann sich dabei behaupten, wie Abbildung 6.8 zeigt. Im Vergleich zu den anderen Strategien benötigt diese im Durchschnitt deutlich weniger SAT Aufrufe. Die anderen Strategien, die eine feste Anzahl an Teilmengen bilden, sind ähnlich schnell. Die `maxSize` Strategien fallen hingegen deutlich und liefern eine schlechtere Performance. Die lineare Suche schneidet mit etwa 24 Sekunden im Durchschnitt am schlechtesten ab und übertritt bei vier Instanzen das Zeitlimit. Damit bleibt sie auch mit den kleineren Optimierungen noch das Schlusslicht.

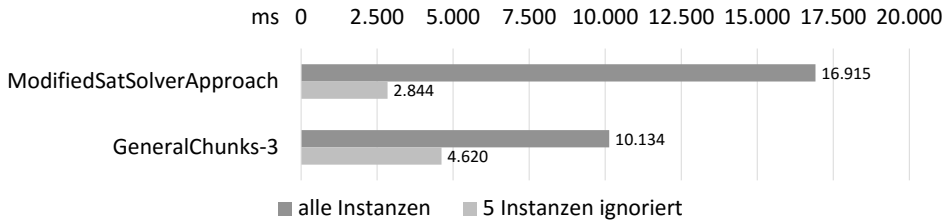


Abbildung 6.9: Vergleich der Ergebnisse des modifizierten SAT Solver mit denen des General Chunks Algorithmus bei größeren Instanzen.

Deutlich besser im Vergleich zu den kleinen Instanzen, jedoch immer noch schlechter als eine gute General Chunks Strategie schneidet der modifizierte SAT Solver Algorithmus ab. Auch dieser übertritt bei vier Instanzen das Zeitlimit. Eine genaue Analyse zeigt aber auch hier, dass wenige Instanzen überproportional stärker relevant sind für das Ergebnis. Abbildung 6.9 stellt den Unterschied dar, der entsteht, wenn die fünf Instanzen mit den schlechtesten Ergebnissen des jeweiligen Algorithmus ignoriert werden. Der modifizierte SAT Solver profitiert deutlich mehr von der Betrachtungsweise und kommt auf $2,8$ Sekunden während der General Chunks Algorithmus nur $4,6$ Sekunden erreicht. Der modifizierte SAT Solver erreicht dabei bei 86% der Instanzen bessere Ergebnisse. Ebenso wie bei den kleineren Instanzen spart der modifizierte SAT Solver im Durchschnitt 55% der Rechenzeit ein und hängt den General Chunks Algorithmus bei diesen Instanzen klar ab.

Kapitel 7

Zusammenfassung und Ausblick

Dieses Kapitel bildet den Abschluss dieser Arbeit. Es fasst in Abschnitt 7.1 die Optimierungen und Algorithmen sowie deren Ergebnisse zusammen. Daraufhin geht Abschnitt 7.2 auf weitere Optimierungsmöglichkeiten ein.

7.1 Zusammenfassung

Diese Arbeit hat die effiziente Implementierung von fünf verschiedenen Algorithmen zur Berechnung von präferierten Diagnosen erläutert. Die Analyse durch den Benchmark machte mehrere Schwachpunkte sichtbar und zeigte, welche Optimierungen sich behaupten können.

Die Verwendung einer effizienten SAT Solver Anbindung ist dabei das Herzstück. Diese muss explizit darauf ausgelegt sein, mehrere SAT Aufrufe effizient verarbeiten zu können. Mit dem Adapter `MiniSatOpt` wurde dies erreicht. Wie präsentiert, spart dieser bei der Berechnung präferierter Diagnosen mit der linearen Suche mindestens 70% der Zeit und bei FastDiag sogar 97% der Zeit ein.

Model Exploiting und Backbone Literale bringen bei allen Algorithmen, die eine Verwendung ermöglichen, einen klaren Effizienzvorteil. Des Weiteren hat sich herausgestellt, dass - entgegen der ursprünglichen Annahme - das Einsparen des ersten SAT Aufrufs nicht in jedem Fall die Berechnung beschleunigt, da Informationen aus der Berechnung in die weitere Berechnung einfließen. Die lineare Suche ist auch mit den genannten Optimierungen gegenüber den anderen Algorithmen deutlich abgeschlagen. Diesem entgegengesetzt kann sich der FastDiag Algorithmus behaupten. Mit den Optimierungen sowie einer effizienteren Implementierung, die er dem verallgemeinerten General Chunks Algorithmus verdankt, kann er gute Zeiten aufzeigen. Ebenso zeigten verschiedene Strategien des General Chunks Algorithmus, dass mit einer Aufteilung von drei bis fünf Teilmengen die besten Zeiten erreichbar waren. Dieser erreicht auch bei größeren Instanzen noch konstant gute Zeiten und ist im Durchschnitt etwa 16% schneller als FastDiag. Der Benchmark zeigt, dass die Effizienz

des modifizierten SAT Solver Algorithmus von der spezifischen Instanz stark abhängig ist. Bei einigen Instanzen ist dieser unangefochten schneller als alle anderen Algorithmen. Der Adopted Branch And Bound Algorithmus zeigt sich ebenso als nicht praktikabel, obwohl bereits mehrere Optimierungen eingebaut wurden. Dieser benötigt noch einige weitere tiefgehende Verbesserungen, damit er mit den anderen Algorithmen mithalten kann.

Mit dem General Chunks und dem modifizierten SAT Solver Algorithmus sind große Instanzen mit einer halben Million Klauseln berechenbar und liefern in einer akzeptablen Zeit das Ergebnis. Demzufolge sind diese Algorithmen auch in der Praxis einsetzbar und können zum Beispiel bei der Kraftfahrzeugkonfiguration verwendet werden.

7.2 Weitere Optimierungsmöglichkeiten

Eine Vielzahl an Optimierungsmöglichkeiten und Algorithmen werden in dieser Arbeit bereits implementiert und analysiert. Im Folgenden werden noch Möglichkeiten zur Berechnung von präferierten Diagnosen erläutert, die bisher nicht berücksichtigt wurden. Dazu gehören Weiterentwicklungen von verschiedenen Algorithmen. Diese werden im ersten Unterabschnitt erläutert. Im zweiten Unterabschnitt wirft die Arbeit ein Licht auf die Idee zweier weiterer Algorithmen.

7.2.1 Algorithmen weiterentwickeln

Jeder Algorithmus wurde in vielerlei Hinsicht optimiert. Nichts desto trotz gibt es weitere Möglichkeiten, die einer näheren Betrachtung lohnend erscheinen. Der modifizierte SAT Solver Ansatz brachte, wie bereits erwähnt, für manche Instanzen exzellente Ergebnisse. Bei anderen Instanzen versagt der Algorithmus allerdings. Eine Analyse der Gründe bei diesen Instanzen kann hier noch weitere Optimierungsmöglichkeiten eröffnen. Ein Schwachpunkt ist in jedem Fall die vorbestimmte Variablenbelegung. Genau hier ist ein Ansatz, wie der Adopted Branch and Bound Ansatz ihn anstrebt, hilfreich. Hier kann die Variablenbelegung beliebig angepasst werden. Eine ähnliche Variante beschreibt auch [DRGM08]. Die Kombination aus beiden Vorschlägen stellt [DRG13] vor und könnte die Effizienz nochmal deutlich steigern.

Ebenso weist der Adopted Branch and Bound Algorithmus im Vergleich zu einem üblichen SAT Solver noch einen deutlichen Nachteil auf. Er kann nämlich keine Klauseln lernen. So ist es denkbar, dass er aus Teilbäumen, bei denen er auf ein schlechteres Ergebnis trifft, eine zusätzliche harte Klausel bildet. Hängt der Grund dafür nur von wenigen Variablen ab, so können dieselben schlechten Vorschläge mehrfach auftreten. Dies ließe sich allerdings durch eine Entscheidungsklausel verhindern. Weiter ist auch eine gute Abschätzung für

die untere Grenze hilfreich. Wie bereits in Abschnitt 4.4 erwähnt, kann die Methode `underestimation()` mit einer guten Abschätzung verhindern, dass Teilbäume, die keine guten Ergebnisse liefern, nicht besucht werden. Mehrere Vorschläge für MaxSAT Algorithmen bietet dafür [BBH+09], welche eine Anpassung wegen der anderen Betrachtungsweise von präferierten Diagnosen erfordert.

Die Funktionalität des SAT Solver Adapters `MiniSatAssumption` lässt sich weiter optimieren. Gerade das erste Hinzufügen aller Klauseln bei Algorithmen wie FastDiag oder dem General Chunks fällt schwer ins Gewicht. So können durch das Hinzufügen aller Klauseln in einem Zug unnötige Aufrufe der HashMap eliminiert werden. Durch veränderbare Klauselobjekte kann man auch deren Assumptionvariablen speichern und sich so das Verwenden der HashMap ersparen, obwohl Klauseln nicht doppelt hinzugefügt werden. Weiter kann - ähnlich der Idee den SAT Solver nach einigen Aufrufen zurücksetzen - auch der Vektor mit den Assumptionvariablen verkleinert werden. So kann der Vektor im Schnitt stets klein gehalten werden.

7.2.2 Weitere Algorithmen

Es gibt noch zwei weitere Algorithmen, die [WFK15b] vorschlägt. Dies ist zum Einen der *MCS gesteuerte Ansatz* und zum Anderen der *L-Preferred MUS gesteuerte Ansatz*. Da beide Algorithmen mehrere komplexe Aufrufe für einen MCS Solver beziehungsweise L-Preferred MUS Solver benötigen, ist die Effizienz dieser Algorithmen vermutlich schlecht. Beide werden im Folgenden nur kurz erläutert.

Der MCS gesteuerte Ansatz benötigt einen MCS Solver, welcher aus den zwei Klauselmengen φ_{hard} und φ_{soft} ein (nicht präferiertes) MCS $C \subseteq \varphi_{\text{soft}}$ berechnet. Die weitere Vorgehensweise entspricht dem der linearen Suche mit Model Exploiting. In jedem Schritt bildet der MCS Solver aus der harten Klauseln $\varphi_{\text{hard}} \cup \Gamma \cup \{c_i\}$ und den weichen Klauseln $\varphi_{\text{soft}} \backslash (\Gamma \cup \Delta \cup \{c_i\})$ ein MCS. Dadurch, dass ein MCS gebildet wird, erhofft sich der Algorithmus, dass das Model Exploiting möglichst viele der folgenden Klauseln ebenso erfüllt. Der Nachteil des Algorithmus liegt allerdings darin, dass das Bilden eines (nicht präferierten) MCS bereits Σ_2^{P}-vollständig [MSP14] und damit komplexer als die Berechnung des Erfüllbarkeitsproblems ist welches - wie bereits erwähnt - NP-vollständig ist. Die Berechnung des MCS benötigt demzufolge mehr Zeit als die Berechnung der Erfüllbarkeit; allerdings ist sie immer noch deutlich schneller als die Berechnung präferierten Diagnosen. Die Aussicht, dass in jedem Schritt mehrere Klauseln erfüllt werden, ist nicht sehr hoch. Deshalb noch ein alternativer Vorschlag. Es ist auch denkbar den selben Ansatz zu wählen und einen Weighted MaxSAT Solver zu verwenden. Die nächsten hoch präferierten Klauseln belegt man dabei mit einem höheren Gewicht. So ist die Wahrscheinlichkeit hoch, dass diese erfüllt werden und damit mit Model Ex-

ploiting viele Klauseln erfüllt werden können. Allerdings ist dieses Problem sogar $\mathrm{FP}^{\mathrm{NP}}$-vollständig [Pap94] und folglich so komplex wie die Berechnung von präferierten Diagnosen selbst. Damit benötigt die Berechnung der Zwischenergebnisse zu lange.

Des Weiteren schlägt [WFK15b] auch noch einen Ansatz vor, der mit Hilfe eines *L-Preferred MUS* die präferierte Diagnose berechnen möchte. Ein *Minimal Unsatisfiable Subset* (MUS) ist dabei wie folgt definiert:

Definition 7.1. Eine Menge $U \subseteq \varphi_{\text{soft}}$ ist eine *unerfüllbare Teilmenge*, wenn $\varphi_{\text{hard}} \cup U$ unerfüllbar ist. Eine Menge $M \subseteq \varphi_{\text{soft}}$ ist ein Minimal Unsatisfiable Subset, falls M eine unerfüllbare Teilmenge ist und für alle $M' \subseteq \varphi_{\text{soft}}$ mit $M' \subset M$ gilt, dass die Vereinigung $\varphi_{\text{hard}} \cup M'$ erfüllbar ist.

Die Definition eines L-Preferred MUS ist analog zu der eines L-Preferred MSS/A-Preferred MCS. Aus einem MUS kann nicht geschlossen werden, welche Klausel dabei Teil des A-Preferred MCS ist. Ebenso verhält es sich bei einem L-Preferred MUS [WFK15b]. Der vorgeschlagene Algorithmus berechnet solange ein L-Preferred MUS, bis die Klauselmenge $\varphi_{\text{hard}} \cup \Gamma\varphi$ erfüllbar ist, demzufolge kein MUS mehr existiert. Die Menge φ enthält dabei alle restlichen Klauseln, die noch nicht zum A-Preferred MCS Δ oder zum L-Preferred MSS Δ hinzugefügt wurden. Zu Beginn gilt also $\phi = \varphi_{\text{soft}}$. Angenommen c^* ist die wichtigste Klausel des berechneten L-Preferred MUS, dann wird $\varphi_{\text{hard}} \cup \Gamma \cup \{c^*\}$ auf Erfüllbarkeit geprüft. Ist es erfüllbar, so gehört c^* zum L-Preferred MSS Γ und wird diesem hinzugefügt. Andernfalls wird es zum A-Preferred MCS Δ hinzugefügt. In jedem Schritt verkleinert sich die Klauselmenge φ um eine Klausel. Im besten Fall hat der Algorithmus damit nach $2 \cdot |\Delta| + 1$ vielen SAT Aufrufen das A-Preferred MCS berechnet. Im schlechtesten Fall benötigt er $2m$ viele SAT Aufrufe[1]. Zu beachten ist allerdings, dass es bei der Hälfte der Aufrufe notwendig ist, zusätzlich das L-Preferred MUS zu berechnen, was weiteren Aufwand erfordert. Die Berechnung des L-Preferred MUS liegt in der Komplexitätsklasse $\mathrm{FP}^{\Sigma_2^{\mathrm{P}}}$ [MSP14] und ist damit deutlich komplexer als die Berechnung des Erfüllbarkeitsproblems.

Weiter schlagen Walter et al. [WFK15b] vor, den SAT Solver der ein L-Preferred MUS liefert, auch für die anderen vorgestellten Algorithmen zu verwenden. Das MUS liefert einem die erste Klausel, die überprüft werden muss und begrenzt damit den Suchraum.

[1] m ist dabei die Anzahl der weichen Klauseln.

Literaturverzeichnis

[BBH+09] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009. (4 Referenzierungen auf den Seiten v, 6, 23 und 47).

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. (auf Seite 4 referenziert).

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. (auf Seite 4 referenziert).

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960. (auf Seite 4 referenziert).

[DRG13] Emanuele Di Rosa and Enrico Giunchiglia. Combining approaches for solving satisfiability problems with qualitative preferences. *AI Commun.*, 26(4):395–408, October 2013. (auf Seite 46 referenziert).

[DRGM08] Emanuele Di Rosa, Enrico Giunchiglia, and Marco Maratea. A new approach for solving satisfiability problems with qualitative preferences. In *Proceedings of the 2008 Conference on ECAI 2008: 18th European Conference on Artificial Intelligence*, pages 510–514, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press. (auf Seite 46 referenziert).

[ES04] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer Berlin Heidelberg, 2004. (auf Seite 31 referenziert).

[FSZ12] A. Felfernig, M. Schubert, and C. Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. *Artif. Intell. Eng. Des. Anal. Manuf.*, 26(1):53–62, February 2012. (4 Referenzierungen auf den Seiten v, 11, 13 und 14).

[Jun04] Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, pages 167–172. AAAI Press, 2004. (auf Seite 11 referenziert).

[KZW13] Wolfgang Küchlin, Christoph Zengler, and Rouven Walter. SAT-Solving und Anwendungen. Technical report, Universität Tübingen, Novembre 2013. (2 Referenzierungen auf den Seiten v und 6).

[KZW15] Andreas J. Kübler, Christoph Zengler, and Rouven Walter. Warthog. Website, 2015. Online erhältlich unter https://github.com/warthog-logic/warthog; abgerufen am 10. November 2015. (auf Seite 27 referenziert).

[LS08] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reason.*, 40(1):1–33, January 2008. (auf Seite 7 referenziert).

[MSHJ+13a] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. MCS computation and enumeration tool. Website, 2013. Online erhältlich unter http://logos.ucd.ie/wiki/doku.php?id=mcsls; abgerufen am 25. November 2015. (auf Seite 33 referenziert).

[MSHJ+13b] Joao Marques-Silva, Federico Heras, Mikolas Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 615–622. AAAI Press, 2013. (4 Referenzierungen auf den Seiten 7, 14, 18 und 19).

[MSP14] Joao Marques-Silva and Alessandro Previti. On computing preferred muses and mcses. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing ? SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 58–74. Springer International Publishing, 2014. (5 Referenzierungen auf den Seiten 8, 9, 47 und 48).

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, Inc, United States of America, 1994. (2 Referenzierungen auf den Seiten 9 und 48).

[SKK00] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Sat-based consistency checking of automotive electronic product data. In *ECAI Workshop*, 2000. (auf Seite 1 referenziert).

[SS96] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society. (auf Seite 5 referenziert).

[WFK15a] Rouven Walter, Alexander Felfernig, and Wolfgang Küchlin. Inverse quickxplain vs. maxsat - a comparison in theory and practice. In *Proceedings of the 17th International Configuration Workshop*, 2015. (auf Seite 9 referenziert).

[WFK15b] Rouven Walter, Alexander Felfernig, and Wolfgang Küchlin. Suggestions for improvements of preferred minimal correction subset computation. (nicht veröffentlicht), 2015. (15 Referenzierungen auf den Seiten 1, 11, 14, 18, 19, 20, 22, 23, 24, 38, 47 und 48).

[WHK04] Manfred Wolff, Peter Hauck, and Wolfgang Küchlin. *Mathematik für Informatik und BioInforamtik*. Springer, Berlin Heidelberg, 2004. (auf Seite 3 referenziert).

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift