Institut für theoretische Informatik

Leibniz Universität Hannover

Bachelorarbeit

Algorithmen für quantifizierte Boole'sche Formeln

Joscha Snakker
3038260

Mai 2020

Betreuer: Dr. Maurice Chandoo
Erstprüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Arne Meier

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 19. Mai 2020

Joscha Snakker

Für mein Netzteil, das alles gab, bis es es nicht mehr gab.

Inhaltsverzeichnis

1 Einführung

Im Rahmen dieser Arbeit werden Algorithmen für quantifizierte Boole'sche Formeln vorgestellt, hinsichtlich ihrer Laufzeit sowie ihrem Speicherbedarf untersucht und verglichen.

Kapitel 2 widmet sich zunächst zugrundeliegenden Konzepten und Verfahren. Theoretische Grundlagen von aussagenlogischen und quantifizierte Boole'sche Formeln werden eingeführt. Dies beinhaltet Konventionen zu Schreibweisen, wichtige Normalformen und bestimmte Eigenschaften. Außerdem werden die Erfüllbarkeitsprobleme für aussagenlogische- und quantifizierte Boole'sche Formeln vorgestellt. Zuletzt wird auf Konsequenzen eines hypothetischen Durchbruchs hinsichtlich Algorithmen zum Lösen dieser Probleme für die theoretische Informatik und Anwendungen in der Praxis eingegangen.

In Kapitel 3 werden Verfahren zum zielgerichteten Widerlegen von Formeln betrachtet. Für aussagenlogische und quantifizierte Boole'schen Formeln werden Resolutionsverfahren eingeführt. Die Resolution für quantifizierte Boole'sche Formeln, Q-Resolution genannt, wird verstärkt betrachtet und der Beweis ihrer Korrektheit und Widerlegungsvollständigkeit nachvollzogen.

Sobald die Resolutionsverfahren theoretisch betrachtet wurden, stehen in Kapitel 4 Algorithmen zur Resolution im Mittelpunkt. Mithilfe dieser Algorithmen sollen später Erfüllbarkeitsprobleme entschieden werden können. Ziel ist es, Algorithmen zu implementieren, die Erfüllbarkeitsprobleme lösen, dabei behilflich sind oder Erkenntnisse dazu sammeln. Zunächst werden diese Algorithmen vorgestellt. Außerdem wird auf die Generierung von logischen Problemen eingegangen. Anhand dieser Problemee sollen die Algorithmen in möglichst faire Vergleiche gestellt werden können.

Im anschließenden Kapitel 5 soll die Leistungsfähigkeit der Algorithmen analysiert werden. Zunächst werden die, durch die Generierung logischer Probleme, erzeugten Testmengen untersucht. Dabei wird versucht festzustellen, welche Eigenschaften oder Parameter logischer Formeln in den Testmengen wie stark vertreten sind. Es werden Testmengen gesucht, die einen möglichst fairen Wettbewerb ermöglichen. Hierzu werden schließlich die zuvor präsentierten Algorithmen auf die erzeugten logischen Probleme angewendet. Laufzeit und Speicherbedarf werden gemessen sowie die Algorithmen anhand dieser Werte untereinander verglichen und abschließend bewertet.

2 Grundlagen

Grundlage bildet die Lektüre von Hans Kleine Büning und Theodor Lettmann „Propositional Logic: Deduction and Algorithms“ (KBL94). Die Definitionen in dieser Arbeit sind (KBL94) entnommen, sofern nicht anderweitig deklariert.

2.1 Aussagenlogik

Aussagenlogische Formeln bestehen aus **Atomen**. Diese stehen für eine elementare Aussage, die entweder wahr oder falsch ist. Die Wahrheitswerte werden auch durch 1 für wahr und 0 für falsch repräsentiert. Eine aussagenlogische Formel setzt sich aus Atomen und durch ein **Nicht** ($\neg$) negierte Atome zusammen. Diese werden durch die logischen Operatoren **Und** ($\wedge$) und **Oder** ($\vee$) verknüpft. Die folgende Definition formalisiert diese Regeln.

Definition 2.1 (Klasse der aussagenlogischen Formeln, $\mathcal{AL}$)**.** *Die Klasse der aussagenlogischen Formeln $\mathcal{AL}$ wird induktiv definiert durch die folgenden vier Schritte.*
1. Jedes Atom ist eine Formel.
2. Ist α eine Formel, so ist auch $\neg\,\alpha$ eine Formel.
3. Falls α und β Formeln sind, so sind auch $(\alpha \wedge \beta)$ und $(\alpha \vee \beta)$ Formeln.
4. Nur mit 1.-3. gebildete Ausdrücke sind Formeln.

In Definition 2.1 wurden die für diese Arbeit relevanten Logikoperatoren bereits definiert. Ein „$\neg$“ negiert die Aussage des zugehörigen Atoms. Die Negation „$\neg\alpha$“ ist genau dann wahr, wenn „α“ nicht wahr ist.
Ein Atom oder dessen Negation wird als **Literal** bezeichnet. Eine Verknüpfung von Literalen „$\alpha \wedge \beta$“ entspricht „α und β“. Diese Formel ist genau dann wahr, wenn sowohl α als auch β wahr sind. Eine Verknüpfung „$\alpha \vee \beta$“ entspricht „α oder β“ und ist genau dann wahr, wenn mindestens eine der verknüpften Aussagen wahr ist.

Die folgende Wahrheitstabelle zeigt die aus den logischen Operationen resultierenden Werte, abhängig von den Werten, der durch sie verknüpften Atome. Atomen einen Wahrheitswert zuzuordnen nennt man **Belegung**. In Tabelle 2.1 werden alle Kombinationen von Belegungen für x und y betrachtet.

x	y	$\neg x$	$x \wedge y$	$x \vee y$
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Tabelle 2.1: Wahrheitstabelle für Negation, Und, Oder

Eine aussagenlogische Formel, für die mindestens eine Belegung existiert, sodass sie wahr ist, heißt **erfüllbar**. Eine Formel, die für jede Belegung wahr ist, wird **Tautologie** genannt. Eine aussagenlogische Formel, für die keine Belegung existiert, sodass sie wahr ist, wird **unerfüllbar** oder **inkonsistent** genannt.

Unter einigen Bedingungen lassen sich Formeln vereinfachen bzw. Literale weglassen. Die folgenden Lemmata zeigen die grundlegenden Möglichkeiten hierfür. Das Zeichen $\equiv$ steht für logische Äquivalenz, das heißt auf beiden Seiten des Zeichens stehen logische Ausdrücke, die für jede Belegung den gleichen Wahrheitswert annehmen.

Definition 2.2 (Logische Äquivalenz)**.** *Zwei Formeln α und β heißen (logisch) äquivalent, abgekürzt $\alpha \equiv \beta$, genau dann, wenn sie für jede Belegung den gleichen Wahrheitswert annehmen.*

Lemma 2.3. *Es gilt:* $x \wedge 0 \equiv 0, \quad x \wedge 1 \equiv x,$
$x \vee 0 \equiv x, \quad x \vee 1 \equiv 1.$

Lemma 2.4. *Es gilt:* $x \wedge x \equiv x, \quad x \wedge \neg x \equiv 0,$
$x \vee x \equiv x, \quad x \vee \neg x \equiv 1.$

Die Äquivalenzen, aus denen diese Lemmata bestehen, lassen sich einfach durch die folgenden Wahrheitstabellen (Tabellen 2.2 und 2.3) zeigen.

x	$x \wedge 0$	$x \wedge 1$	$x \vee 0$	$x \vee 1$
0	0	0	0	1
1	0	1	1	1

Tabelle 2.2: Wahrheitstabelle zu Lemma 2.3

x	$x \wedge x$	$x \wedge \neg x$	$x \vee x$	$x \vee \neg x$
0	0	0	0	1
1	1	0	1	1

Tabelle 2.3: Wahrheitstabelle zu Lemma 2.4

Konjunktive Normalform

Eine wichtige Darstellungsform einer aussagenlogischen Formel ist die Konjunktive Normalform. Eine Formel in dieser Form besteht aus einer Und-Verknüpfung von Oder-Verknüpfungen, auch als Konjunktion von Disjunktionen bezeichnet. Disjunktionen, auch Klauseln genannt, sind Oder-Verknüpfungen bestehend aus beliebig vielen Literalen.

Definition 2.5 (Klausel)**.** *Eine Formel* $\alpha = (x_1 \vee ... \vee x_n)$ *mit den Literalen* x_i $(1 \leq i \leq n)$ *bezeichnen wir als Klausel.*

Die Natur von $\vee$ bedeutet für Klauseln, dass sie genau dann wahr sind, wenn mindestens eines ihrer Literale wahr ist.
Eine Menge von Klauseln, durch $\wedge$ verknüpft, ist eine Formel in KNF. Damit eine Formel aus der Formelklasse $\mathcal{KNF}$ zu wahr resultiert, muss jede Klausel wahr sein, was bedeutet, dass in jeder Klausel mindestens ein Literal wahr sein muss.

Definition 2.6 (Konjunktive Normalform, $\mathcal{KNF}$ (KBL94))**.** *Eine Formel* α *ist in Konjunktiver Normalform (KNF, engl. CNF) genau dann, wenn* α *eine Konjunktion von Klauseln ist, d.h.* $\alpha = \alpha_1 \wedge ... \wedge \alpha_n$ *mit Klauseln* α_i $(1 \leq i \leq n)$*. Die entsprechende Formelklasse wird als* $\mathcal{KNF}$ *bezeichnet.*

Diese Formeln lassen sich ihrer Struktur in Mengenschreibweise darstellen. Eine Klausel ist eine Menge von Literalen. Eine KNF-Formel ist eine Menge von Klauseln. Beispiel 2.1 stellt beide Schreibweisen gegenüber.

$$\alpha = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$$
$$\alpha = \{\{x_1, x_2\}, \{x_1, \neg x_2\}, \{\neg x_1, x_2\}, \{\neg x_1, \neg x_2\}\}$$

Beispiel 2.1: KNF und zugehörige Mengenschreibweise

Wie später gezeigt wird, bietet sich die Mengenschreibweise auch für die Implementierung von Algorithmen für KNF-Formeln an. Die Regeln für die Logikoperatoren $(\wedge, \vee)$ gestatten es auch, Duplikate (von Klauseln oder Literalen innerhalb der selben Klausel) wegzulassen oder die Klauseln nach Belieben umzuordnen. Da jede aussagenlogische Formel als KNF umgeschrieben werden kann (KBL94), werden Formeln in anderer Form hier nicht weiter betrachtet.

2.2 Quantorenlogik

Eine quantifizierte Boole'sche Formel ist eine aussagenlogischen Formel, für die zusätzlich die Atome der Formel quantifiziert sein können. Dies geschieht durch **Quantoren**. Es gibt zwei verschiedene Quantoren: den **Allquantor** $\forall$ und den **Existenzquantor** $\exists$. Wörtlich bedeutet ein Allquantor über x, dass für jede Belegung von x die folgende Formel gilt. Ein Existenzquantor über y bedeutet, dass mindestens eine Belegung für y existiert, sodass die folgende Formel gilt.

$$\forall x_1 \exists y_1 (x_1 \vee y_1)$$

Beispiel 2.2: Eine quantifizierte Boole'sche Formel

Folgende Konvention soll für quantifizierte Boole'sche Formeln in dieser Arbeit gelten:

$$\forall x_i, x_j, i < j : x_i \text{ steht im Präfix vor } x_j$$

Steht x_i im Präfix vor x_j, so heißt „x_i kleiner als x_j". In dieser Arbeit wird x_i für allquantifizierte und y_i für existenzquantifizierte Literale geschrieben, um die Übersichtlichkeit zu erhöhen.

Reduktion einer quantifizierten Boole'schen Formel

An einem Beispiel wird gezeigt, wie eine Quantifizierte Boole'sche Formel aussieht und wie ihr Wahrheitswert bestimmt werden kann. Betrachtet wird die folgende Formel:

$$\forall x \exists y : x \vee y \Leftrightarrow \text{Für jede Belegung von } x \text{ existiert eine Belegung von } y, \text{ sodass } x \vee y \text{ gilt.}$$

Um zu zeigen, dass diese quantifizierte Formel wahr ist, muss also für jede Belegung von x eine Belegung für y gefunden werden, sodass sie erfüllt ist. Durch Ersetzen aller Vorkommen von x mit 0 (bzw. 1) entsteht die Formel Φ_0 (bzw. Φ_1). Φ ist wahr genau dann, wenn Φ_0 und Φ_1 wahr sind. Φ wird also auf $\Phi_0 \wedge \Phi_1$ **reduziert**.

Ersetze alle Vorkommen von x in Φ durch 0 (bzw. 1) um Φ_0 (bzw. Φ_1) zu erhalten:

$$\Phi \Leftrightarrow \Phi_0 \wedge \Phi_1$$
$$\forall x \exists y : x \vee y \Leftrightarrow (\exists y : 0 \vee y) \wedge (\exists y : 1 \vee y)$$

Setze in Φ_0 für y 0 (bzw. 1) ein und erzeuge so $\Phi_{0,0}$ (bzw. $\Phi_{0,1}$):

$$\Phi_0 \Leftrightarrow \Phi_{0,0} \vee \Phi_{0,1}$$
$$(\exists y : 0 \vee y) \Leftrightarrow (0 \vee 0) \vee (0 \vee 1) \Leftrightarrow 0 \vee 1 \Leftrightarrow 1$$

Setze in Φ_1 für y 0 (bzw. 1) ein und erzeuge so $\Phi_{1,0}$ (bzw. $\Phi_{1,1}$):

$$\Phi_1 \Leftrightarrow \Phi_{1,0} \vee \Phi_{1,1}$$
$$(\exists y : 1 \vee y) \Leftrightarrow (1 \vee 0) \vee (1 \vee 1) \Leftrightarrow 1 \vee 1 \Leftrightarrow 1$$

Resultat für Φ:

$$\forall x \exists y : x \vee y \Leftrightarrow (\exists y : 0 \vee y) \wedge (\exists y : 1 \vee y) \Leftrightarrow 1 \wedge 1 \Leftrightarrow 1$$

Beispiel 2.3: Lösung einer quantifizierten Boole'schen Formel durch Reduktion

Nach Beispiel 2.3 existiert für jede Belegung von x eine Belegung von y, sodass $x \vee y$ erfüllt ist und damit ist die quantifizierte Boole'sche Formel wahr. Jede quantifizierte Boole'sche Formel lässt sich auf diese Weise reduzieren, bis sich ihr Wahrheitswert aus der resultierenden aussagenlogischen Formel bestimmen lässt.

Wie im Beispiel gesehen, verhalten sich quantifizierte Atome im Grunde wie Variablen. Daher spricht man auch von aussagenlogischen Variablen (KBL94).

Definition 2.7 (Quantifizierte Boole'sche Formeln, $\mathcal{QBF}$)**.** *Die Menge der quantifizierten Boole'schen Formeln ($\mathcal{QBF}$) ist induktiv wie folgt definiert:*

1. *Jede aussagenlogische Formel ist eine quantifizierte Boole'sche Formel.*
2. *Sei ϕ eine quantifizierte Boole'sche Formel und seien x bzw. y aussagenlogische Variablen, dann sind auch $\exists y \phi$ und $\forall x \phi$ quantifizierte Boole'sche Formeln.*
3. *Nur gemäß 1. und 2. gebildete Formeln sind quantifizierte Boole'sche Formeln.*

Eine Variable wird als **gebunden** bezeichnet, falls sie durch einen Quantor quantifiziert ist. Nicht gebundene Variablen heißen **freie** Variablen. Eine Formel ist **geschlossen**, falls all ihre Variablen gebunden sind.

Eine Formel aus $\mathcal{QBF}$ heißt **erfüllbar** genau dann, wenn für die freien Variablen der Formel mindestens eine Belegung existiert, sodass die Formel wahr ist. Umgekehrt heißt eine Formel aus $\mathcal{QBF}$ **inkonsistent** oder **widersprüchlich** genau dann, wenn die Formel für alle Belegungen falsch ist.

Für Formeln aus $\mathcal{QBF}$, welche über keine freien Variablen verfügen, ist der Wahrheitswert eindeutig bestimmbar. Für jede nicht geschlossene QBF-Formel Φ gilt, dass sie genau dann erfüllbar ist, wenn die geschlossene Formel $\Phi_{gsl} = \exists y_0 \ldots \exists y_n \Phi$ wahr ist, wobei $y_0 \ldots y_n$ die freien Variablen von Φ sind. So lassen sich für die Betrachtung der Erfüllbarkeit von QBF-Formeln nicht geschlossene Formeln vernachlässigen, indem statt der Erfüllbarkeit von Φ der Wahrheitswert von Φ_{gsl} bestimmt wird.

Präfix- und Kernklassen

Quantifizierte Boole'sche Formeln können stets in **pränexer** Schreibweise formuliert werden (KBL94). Pränex bedeutet hier, dass alle Quantifizierungen, als Präfix zusammengefasst, vor der aussagenlogischen Formel stehen.

Da quantifizierte Formeln, die nicht in Pränexform formuliert sind, sich in diese bringen lassen, werden quantifizierte Formeln in nicht-pränexer Form nicht weiter betrachtet. Die Umformung zur Pränexform geschieht durch Umbenennung jeder quantifizierten Variable, sodass sich jeder Quantor auf nur eine Variable bezieht und jede Variable bei jedem Auftreten eindeutig frei oder gebunden auftritt (KBL94). Anschließend werden alle Quantoren vorne an gestellt, als Präfix in Reihenfolge des Auftretens in der Formel (KBL94).

Um Präfixe von quantifizierten Boole'schen Formeln zu klassifizieren, werden die Quantorenwechsel gezählt (KBL94). Durch $\sum_4$ wird ein Präfix mit einem oder mehreren führenden Existenzquantoren und drei Quantorenwechseln bezeichnet. Der Präfixtyp $\prod_{12}$ beschreibt einen Quantor, welcher mit Allquantoren beginnt und elf Wechsel enthält. Eine aussagenlogische Formeln α hat den Präfixtyp $\sum_0 = \prod_0$.

Formel	Präfixtyp
α	$\sum_0 = \prod_0$
$\exists y_1 \exists y_2 \forall x_1 \exists y_3 \forall x_2 \forall x_3 \, \alpha$	$\sum_4$
$\forall x_1 \exists y_1 \forall x_2 \exists y_2 \forall x_3 \exists y_3 \forall x_4 \exists y_4 \forall x_5 \exists y_5 \forall x_6 \exists y_6 \, \alpha$	$\prod_{12}$

Beispiel 2.4: Präfixtypen

Definition 2.8 (Präfixklassen Quantifizierter Boolescher Formeln)**.** *Sei $k \geq 1$, dann definieren wir*

$$\mathcal{QBF}^*_{k,\forall} \coloneqq \{\phi \mid \phi \textit{ hat Präfixtyp } \prod_k\} \quad \mathcal{QBF}^*_k \coloneqq \mathcal{QBF}^*_{k,\exists} \cup \mathcal{QBF}^*_{k,\forall}$$
$$\mathcal{QBF}^*_{k,\exists} \coloneqq \{\phi \mid \phi \textit{ hat Präfixtyp } \sum_k\} \quad \mathcal{QBF}^* \coloneqq \bigcup_k \mathcal{QBF}^*_k$$

Schreibt man statt $\mathcal{QBF}^*$ nur $\mathcal{QBF}$, so sind nur die geschlossenen Formeln aus $\mathcal{QBF}^*$ enthalten (KBL94). In dieser Arbeit werden vor allem geschlossene Formeln betrachtet.

Als **Kern** einer QBF-Formel wird der Teil bezeichnet, der aus einer aussagenlogischen Formel besteht und auf das Präfix folgt (KBL94). Wenn der Kern eine KNF- oder HORN-Formel ist, dann nennt man die zugehörige QBF-Formel auch QKNF- oder QHORN-Formel (KBL94).

Aufgrund der Tatsache, dass aussagenlogische Formeln stets in KNF und quantifizierte Boole'sche Formeln stets in Pränexform gebracht werden können, sind QKNF-Formeln die quantifizierten Boole'schen Formeln, welche hier im Fokus stehen sollen.

Da quantifizierte Boole'sche Formeln aus einem Präfixteil und einem aussagenlogischen Formelteil bestehen, lässt sich offensichtlich zumindest der Formelteil genau wie bei aussagenlogischen Formeln in Mengeschreibweise notieren. Das Präfix kann als Sequenz (von Sequenzen) gefasst werden, wobei für jeden Quantorenwechsel eine neue Sequenzen hinzugefügt wird.

Präfix	Formel
$\forall x_1 \forall x_2 \exists y_1$	$(x_1 \vee y_1) \wedge (x_1 \vee \neg y_1 \vee x_2)$
$[[\forall, x_1, x_2], [\exists, y_1]]$	$\{\{x_1, y_1\}, \{x_1, \neg y_1, x_2\}\}$

Beispiel 2.5: Quantifizierte Boole'sche Formel mit alternativer Schreibweise

2.3 Solver

Im Zentrum dieser Arbeit stehen Algorithmen, welche das Problem `QBF` entscheiden. `QBF` ist das Entscheidungsproblem zur Erfüllbarkeit einer Formel aus $\mathcal{QBF}$ (SB05). Betrachtet wird auch das verwandte Entscheidungsproblem `SAT`. `SAT` ist das Entscheidungsproblem zur Erfüllbarkeit einer Formel aus $\mathcal{AL}$ (VK19).

Definition 2.9 (Erfüllbarkeitsproblem `SAT`, (VK19))**.** *Die Menge aller erfüllbaren Formeln wird bezeichnet mit*

$$\texttt{SAT} = \{\Phi \in \mathcal{AL} \mid \textit{es gibt eine Belegung } \mathfrak{I} \textit{ mit } \mathfrak{I} \models \Phi\}.$$

Definition 2.10 (Erfüllbarkeitsproblem QBF)**.** *Die Menge der erfüllbaren quantifizierten Boole'schen Formeln wird bezeichnet mit*

$$\texttt{QBF} = \{\Phi \in \mathcal{QBF} \mid \textit{es gibt eine Belegung } \mathfrak{I} \textit{ mit } \mathfrak{I} \models \Phi\}.$$

Hierbei bezeichnet $\mathcal{AL}$ die Menge aller aussagenlogischen Formeln (Def. 2.1) und $\mathcal{QBF}$ die Menge aller quantifizierten Boole'schen Formeln (Def. 2.7). Es gilt $\mathcal{AL} \subset \mathcal{QBF}$. QBF generalisiert das Problem SAT (SB05). Jede Formel aus SAT liegt auch in QBF - umgekehrt gilt diese Relation jedoch nicht. Das Entscheidungsproblem besteht bei beiden aus der Frage, ob eine Formel Φ erfüllbar ist. SAT ist auf QBF reduzierbar (SB05).

Das Auswerten von logischen Formeln hat etliche Anwendungsmöglichkeiten, z.B. in Kryptographie, Elektronischer Entwurfautomatisierung, Bioinformatik, Verifizierung von Hardware und Software oder - ein alltäglicheres Beispiel - bei der Signalsteuerung für Zugstrecken (SNC09, Mal10, BGG$^+$15). Algorithmen zum Lösen dieser Erfüllbarkeitsprobleme heißen QBF- bzw. SAT-Solver. Doch der Nutzen wird durch die Effizienz des Lösens logischer Probleme begrenzt.
Das Entscheidungsproblem SAT ist NP-vollständig (MV15). Das bedeutet, es liegt in NP und ist NP-schwer. Das Problem ist effizient überprüfbar, doch ob es möglich ist, es effizient, d.h. in Polynomialzeit, zu lösen, ist nicht bekannt (MV15). Da SAT NP-schwer ist und damit zu den *schwierigsten* Problemen in NP gehört, wäre ein Algorithmus, der es effizient löst, gleichzeitig ein Beweis, dass alle Probleme in NP effizient lösbar sind (MV15). Demnach wäre P = NP. Dies wäre ein überraschendes Ergebnis mit großen Auswirkungen für die Komplexitätstheorie.

Quantifizierte Boole'sche Formeln stellen eine Erweiterung der aussagenlogischen Formeln dar. Die Frage, ob $\Phi \in$ SAT gilt, ist äquivalent zu der, ob $\exists x_1 \dots \exists x_n \Phi \in$ QBF gilt, wobei $x_1 \dots x_n$ die Variablen von Φ sind (VK19). Daraus folgt SAT $\leq_m^P$ QBF (VK19) und damit ist QBF auch NP-schwer. Wie für SAT ist auch für QBF nicht bekannt, ob dieses effizient lösbar ist. Es ist jedoch bewiesen, dass QBF PSPACE-vollständig ist (KBL94).

Bisher ist über die Komplexitätsklassen bekannt, dass P $\subseteq$ NP $\subseteq$ PSPACE gilt (MV15). Wie bereits erwähnt, ist nicht bekannt, ob umgekehrt auch P $\supseteq$ NP und damit P = NP gelten. Dasselbe gilt auch für die Relationen von PSPACE zu P und NP.
Da QBF PSPACE-vollständig ist, würde in dem Fall, dass es einen Algorithmus gäbe, der QBF in Polynomialzeit überprüft, PSPACE = NP gelten. Für den Fall, dass QBF in Polynomialzeit gelöst würde, gälte gar P = NP = PSPACE.

Die Suche nach effizienten Algorithmen, welche SAT oder QBF entscheiden, ist also nicht einfach nur die Suche nach einer effizienten Lösung für dieses eine Problem. Es ist auch die Suche nach einer Antwort auf die große Frage „Ist P = NP?“, welche, seitdem sie vor beinahe 50 Jahren aufkam, bis heute unbeantwortet bleibt (MV15).
Neben dem praktischen Nutzen ist es wohl auch diese Motivation, welche die Aufmerksamkeit für die Suche nach effektiven SAT- und QBF-Solvern in den letzten Jahrzehnten

steigerte. Seit 2002 findet jährlich die „SAT Competition“ statt, ein offener Wettkampf bei dem Teilnehmer ihre SAT-Solver in Benchmarks gegeneinander antreten lassen (Com09). Es gewinnt der schnellste SAT-Solver, wobei falsche Ergebnisse zur Disqualifikation führen und fehlende Ergebnisse bestraft werden (Com09). Analog findet seit 2006 mit der „QBFEVAL“ auch ein Wettkampf für QBF-Solver statt (GNPT05).

Die quantifizierten Boole'schen Formeln stehen bisher im Vergleich zu den aussagenlogischen Formeln eher im Hintergrund. Dabei sind sie scheinbar das *mächtigere* Werkzeug. Quantifizierte Boole'sche Formeln stellen eine Erweiterung der aussagenlogischen Formeln dar. Mit quantifizierten Formeln ergeben sich neue Möglichkeiten.
So lässt sich für jede aussagenlogische Formel eine quantifizierte formulieren, welche genau dann wahr ist, wenn es wahlweise genau eine oder auch mehrere erfüllende Belegungen gibt (KBL94). Auch die logische Äquivalenz und die Erfüllbarkeitsäquivalenz zweier Formeln lassen sich mit quantifizierten Boole'schen Formeln beschreiben (KBL94). Zudem lassen sich Algorithmen für quantifizierte Boole'sche Formeln oft auch auf aussagenlogische Formeln anwenden, da Erstere eine Erweiterung der Letzteren sind.
Es nimmt den quantifizierten Boole'schen Formeln jedoch einiges von ihrer Relevanz, dass sich jede von ihnen in eine aussagenlogische Formel umformen lässt (VK19). Wichtig zu erwähnen ist, dass die so entstehende aussagenlogische Formel exponentiell größer ist als die quantifizierte.

3 Resolution

Ein Ziel der formalen Logik ist, statt freiem Schließen von logischen Konsequenzen, ein System zu schaffen, mit dem nach syntaktischen Regeln **formale Beweise** zu erbringen sind (KBL94). Dies wird durch das **Resolutionskalkül** realisiert. Es werden nur KNF-Formeln betrachtet.

3.1 Resolutionskalkül

Zunächst soll das Resolutionsprinzip ohne Quantifizierung demonstriert werden. Das Resolutionskalkül arbeitet auf KNF-Formeln und beruht auf einer einzigen Schlussregel. Durch Anwendung der Resolutionsregel werden Literale aus einer Formel entfernt und versucht, die **leere Klausel** $\sqcup$ zu erzeugen.

Definition 3.1 (Resolutionsregel (KBL94))**.** *Sei α eine Klausel mit einem Literal L und β eine Klausel mit dem Literal $\neg L$, dann ist die Resolution auf die beiden Klauseln anwendbar. Wir sagen α und β können über L resolviert werden. Ausgehend von den Klauseln α und β wird so eine neue Klausel $(\alpha \backslash \{L\} \cup (\beta \backslash \{\neg L\})$ erzeugt, die wir als Resolvente bezeichnen. α und β sind die Elternklauseln der Resolvente.*

Die Resolution verringert die Anzahl der (verschiedenen) Literale in einer Formel. Eine Klausel π, die durch Resolutionsschlüsse aus einer Formel α hervorgeht, wird als **herleitbar** aus α bezeichnet.

$$\frac{\alpha \qquad\qquad \beta}{(\alpha \backslash \{L\}) \cup (\beta \backslash \{\neg L\})}$$

Beispiel 3.1: Darstellungsweise des Resolutionskalküls

Ein Resolutionsschritt lässt sich wie im Beispiel 3.1 darstellen. Die Elternklauseln befinden sich links und rechts über dem Strich, die Resolvente darunter.

$$\text{Sei } \alpha = x \vee L \text{ und } \beta = y \vee \neg L.$$
$$\frac{x \vee L \quad y \vee \neg L}{x \vee y}$$

Beispiel 3.2: Resolution mit neuer Klausel als Resolvente

Entsteht eine Resolvente, welche nicht bereits als Klausel in der Formel enthalten ist, so sind weitere Schlüsse mit dieser Klausel erlaubt und die Resolventen daraus gelten immer noch als herleitbar aus der ursprünglichen Formel. Tautologische Resolventen werden verworfen.

$$\text{Sei } \alpha = L \text{ und } \beta = \neg L.$$
$$\frac{L \quad \neg L}{\sqcup}$$

Beispiel 3.3: Resolution mit leerer Klausel als Resolvente

Da in jedem Resolutionsschritt Literale aus den Elternklauseln entfernt werden, ist es möglich, dass eine Resolvente die leere Klausel ist. Dies geschieht, wenn die Resolvente aus zwei Klauseln gebildet wird, die auschließlich ein Literal L bzw. Literal $\neg L$ enthalten (siehe Beispiel 3.3).
Ist eine Formel unerfüllbar, so gilt, dass die leere Klausel aus der Formel herleitbar ist (KBL94). Dies ist die **Widerlegungsvollständigkeit** der Resolution. Die **Korrektheit** der Resolution ist, dass falls die leere Klausel herleitbar ist, die Formel unerfüllbar ist. Diese beiden Eigenschaft sind essenziell für den Resolutionskalkül als **Beweiskalkül**, da dieser somit als Beweismittel zum Widerlegen von aussagenlogischen Formeln qualifiziert ist.

3.2 Q-Resolution

Das Thema dieser Arbeit sind Algorithmen für quantifizierte Boole'sche Formeln. Im Besonderen werden Algorithmen betrachtet, welche ermitteln, ob eine Formel wahr ist und sie widerlegen, falls sie es nicht ist. Für aussagenlogische Formeln wurde bereits der Resolutionskalkül vorgestellt, welcher das Widerlegen oder Lösen von KNF-Formeln erleichtert. Nun soll ein Resolutionskalkül für QKNF-Formeln gezeigt werden, mit dessen Hilfe Algorithmen zur Lösung von QKNF-Formeln implementiert werden können.
Die Idee der Resolution bleibt dieselbe: Zwei Elternklauseln bilden eine Resolvente. In der resultierenden Formel treten die Elternklauseln nicht mehr auf. Betrachtet wird zunächst die Form einer QKNF*-Formel. Das Verfahren der Q-Resolution und ihr Beweis entstammen (KBL94).

Definition 3.2 (Q-Resolution). *Sei $\Phi = \Pi\alpha$ eine Formel in QKNF* mit Kern α und Präfix Π. Seien α_1 und α_2 nicht tautologische Klauseln von α, wobei α_1 das $\exists$-Literal y enthält und α_2 das Literal $\neg y$. Eine Q-Resolvente σ aus α_1 und α_2 erhält man durch Anwendung der Schritte 1 bis 3:*

1. *Eliminiere alle Vorkommen von $\forall$-Literalen in α_i die nicht kleiner sind als ein in α_i vorkommendes $\exists$-Literal (auf beide Elternklauseln anwenden). Die Ergebnisklauseln sind α_1' und α_2'.*

2. *Eliminiere die Vorkommen von y aus α_1' und die von $\neg y$ aus α_2'.*

3. *$\sigma = \alpha_1'' \vee \alpha_2''$.*

Wir schreiben: $\Phi \overset{1}{\underset{Q\text{-}RES}{\vdash\!\!\!-\!\!\!-}} \Pi(\alpha \wedge \sigma)$ oder auch kurz $\alpha \overset{1}{\underset{Q\text{-}RES}{\vdash\!\!\!-\!\!\!-}} \sigma$ bzw. $\Phi \overset{1}{\underset{Q\text{-}RES}{\vdash\!\!\!-\!\!\!-}} \sigma$. Für den auf Basis von $\overset{1}{\underset{Q\text{-}RES}{\vdash\!\!\!-\!\!\!-}}$ definierbaren Herleitungsbegriff der Q-Resolution verwenden wir die Bezeichnung $\underset{Q\text{-}RES}{\vdash\!\!\!-\!\!\!-}$.

3.3 Korrektheit und Widerlegungsvollständigkeit

Es soll gezeigt werden, dass für widersprüchliche quantifizierte Boole'sche Formel per Q-Resolution der Widerspruch gezeigt werden kann. Analog zur Resolution gilt, dass die Herleitung der leeren Klausel die Widersprüchlichkeit der Formel zeigt. Zusätzlich gilt für die Q-Resolution, dass die Erzeugung einer Klausel, die ausschließlich aus allquantifizierten Variablen besteht, als Widerlegungsbeweis ausreicht.

Lemma 3.3. *Für eine nicht tautologische $\forall$-Klausel σ gilt:*
Eine quantifizierte Boole'sche Formel, welche σ enthält, ist widerspruchsvoll.
Wir schreiben: $\sigma \approx \sqcup$. Demnach können wir für $\Phi \underset{Q\text{-}RES}{\vdash\!\!\!-\!\!\!-} \sigma$ auch $\Phi \underset{Q\text{-}RES}{\vdash\!\!\!-\!\!\!-} \sqcup$ schreiben.

Daher wird die Korrektheit und Widerlegungsvollständigkeit der Q-Resolution wie in Theorem 3.4 aufgefasst.

Theorem 3.4 (Korrektheit und Widerlegungsvollständigkeit (KBL94)). *Sei $\Phi \in QKNF^*$, dann gilt: Φ ist widerspruchsvoll $\iff \Phi \underset{Q\text{-}RES}{\vdash\!\!\!-\!\!\!-} \sigma$ für eine nicht tautologische $\forall$-Klausel σ.*

Beweis. Φ kann als geschlossen angenommen werden. Für den Fall, dass es nicht gebundene Literale gäbe, könnten diese existenzquantifiziert werden und die resultierende Formel wäre genau dann inkonsistent, wenn die ursprüngliche Formel widerspruchsvoll ist.

Der Beweis der Widerlegungsvollständigkeit der Q-Resolution wird per Induktion über k realisiert. Hierzu werden verschiedene Fälle anhand des Präfixtyps unterschieden, wobei k für die Anzahl der Quantoren steht.

1. Sei $k = 1$ und $\Phi = \exists y_1(\alpha_1 \wedge ... \wedge \alpha_m)$.

   Der Präfixtyp ist $\sum_1$, d.h. es gibt keinen Quantorenwechsel. Aus „Φ kann als geschlossen angenommen werden“ und $k = 1$ folgt, dass y_1 die einzige auftretende Variable ist. Dieser Fall entspricht der normalen Resolution für aussagenlogische Formeln. Φ ist nur falsch, wenn in Φ y_1 und $\neg y_1$ als Klauseln auftreten. In dem Fall lässt sich durch Q-Resolution die leere Klausel herleiten.

$$\Phi = \exists y_1(y_1) \wedge (\neg y_1) \wedge (y_1 \vee \neg y_1)$$

$$\{y_1\} \quad \{\neg y_1\}$$
$$\diagdown\!\diagup$$
$$\sqcup$$

$$\Phi \overset{\text{Q-RES}}{\vdash\!\!-\!\!-} \sqcup$$

   Beispiel 3.4: Q-Resolution im Fall einer einzelnen $\exists$-Variable

2. Sei $k = 1$ und $\Phi = \forall x_1(\alpha_1 \wedge ... \wedge \alpha_m)$.

   Da x_1 die einzige Variable ist, ist jede Klausel eine $\forall$-Klausel. Um die Annahme „Φ ist falsch“ zu erfüllen, muss lediglich eine der Klauseln keine Tautologie sein. In dem Fall folgt auch nach Satz 3.4, dass Φ widerspruchsvoll ist. Sind alle Klauseln Tautologien, so ist auch kein Q-Resolutionsschritt möglich.

$$\Phi = \forall x_1(x_1) \wedge (\neg x_1)$$

x_1 und $\neg x_1$ sind nicht tautologische $\forall$-Klauseln in $\Phi \Rightarrow \Phi$ widerspruchsvoll

$$x_1 \approx \sqcup \text{ und } \neg x_1 \approx \sqcup$$

   Beispiel 3.5: Q-Resolution im Fall einer einzelnen $\forall$-Variable

3. Sei $k > 1$ und $\Phi = \exists y_1 Q_2 ... Q_k(\alpha_1 \wedge ... \wedge \alpha_m)$, wobei Q_i jeweils für $\forall x_i$ oder $\exists y_i$ steht.

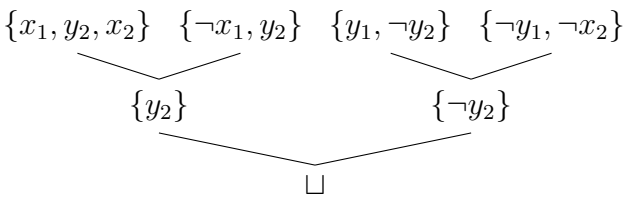
   Φ_0 (bzw. Φ_1) wird (wie bereits in Beispiel 2.3) per Ersetzung aller Vorkommen von y_1 durch 0 (bzw. 1) gebildet. Φ ist genau dann wahr, wenn Φ_0 oder Φ_1 wahr ist.

   $$\Phi \Leftrightarrow \Phi_0 \vee \Phi_1$$

   Alle Resolutionsschritte können auf Φ_1 und Φ_0 genauso angewendet werden wie auf Φ, da das Fehlen des Literals y_1 bzw. $\neg y_1$ keinen Einfluss auf die weitere Resolution hat. Letzteres ist der Fall, da y_1 die „kleinste“ Variable ist und daher nicht entscheidend für die Streichung einer $\forall$-Variable oder für die Tautologieeigenschaft einer Klausel ist. Nach Induktionsvoraussetzung für k-1 gilt:
   $\Phi_0 \underset{\text{Q-RES}}{\vdash\!\!\!-\!\!\!-} \sigma_0$ und $\Phi_1 \underset{\text{Q-RES}}{\vdash\!\!\!-\!\!\!-} \sigma_1$ mit nicht tautologischen Klauseln σ_0 und σ_1.
   Für den ersten Schritt der Q-Resolution gilt, dass es kein $\forall$-Literal geben kann, welches kleiner ist als y_1 und somit keine Elimination einer $\forall$-Variable durch y_1 verhindert werden kann. Für y_1 eingesetzten Einsen bzw. Nullen verhält es sich genauso.

   $$\Phi = \exists y_1 \forall x_1 \exists y_2 \forall x_2 (x_1 \vee y_2 \vee x_2) \wedge (\neg x_1 \vee y_2) \wedge (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee \neg x_2)$$

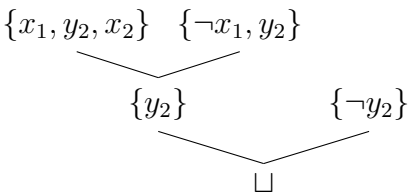
   Q-Resolution für Φ:

   Beispiel 3.6: Q-Resolution der $\mathcal{QBF}$-Formel Φ (Präfixtyp $\sum_4$)

   $$\begin{aligned}\Phi_0 &= \forall x_1 \exists y_2 \forall x_2 (x_1 \vee y_2 \vee x_2) \wedge (\neg x_1 \vee y_2) \wedge (0 \vee \neg y_2) \wedge (1 \vee \neg x_2) \\ &\Leftrightarrow \forall x_1 \exists y_2 \forall x_2 (x_1 \vee y_2 \vee x_2) \wedge (\neg x_1 \vee y_2) \wedge (\neg y_2)\end{aligned}$$

   Q-Resolution für Φ_0:

   Beispiel 3.7: Q-Resolution der $\mathcal{QBF}$-Formel Φ_0 (Präfixtyp $\prod_3$)

$$\Phi_1 = \forall x_1 \exists y_2 \forall x_2 (x_1 \vee y_2 \vee x_2) \wedge (\neg x_1 \vee y_2) \wedge (1 \vee \neg y_2) \wedge (0 \vee \neg x_2)$$
$$\Leftrightarrow \forall x_1 \exists y_2 \forall x_2 (x_1 \vee y_2 \vee x_2) \wedge (\neg x_1 \vee y_2) \wedge (\neg x_2)$$

$$\sigma_1 = \neg x_2 \approx \sqcup$$

σ_1 ist eine nicht tautologische $\forall$-Klausel $\Rightarrow \Phi_1$ ist widerspruchsvoll

Beispiel 3.8: Q-Resolution der $\mathcal{QBF}$-Formel Φ_1 (Präfixtyp $\prod_3$)

Die Substitution von y_1 durch 1 bzw. 0 vereitelt zwar die Q-Resolution über y_1, es wird aber dennoch durch Q-Resolution von Φ_0 bzw. Φ_1 die leere Klausel erzeugt.

4. Sei $k > 1$ und $\Phi = \forall x_1 Q_2 ... Q_k (\alpha_1 \wedge ... \wedge \alpha_m)$.

   Wie bei 3. werden Φ_0 und Φ_1 gebildet. Aus der Annahme „Φ ist falsch“ folgt, dass Φ_0 oder Φ_1 falsch sein muss. Betrachtet wird hier ausschließlich der Fall, dass Φ_0 falsch (Beweis für Φ_1 falsch analog).
   Da Φ_0 falsch, folgt aus der Induktionsvoraussetzung $\Phi_0 \underset{\text{Q-RES}}{\vdash\!\!\!-\!\!\!-} \sigma_0$, wobei σ_0 eine nicht tautologische $\forall$-Klausel ist. Alle Q-Resolutionsschritte, die für die Herleitung von σ_0 erforderlich sind, lassen sich nicht nur in Φ_0, sondern auch in Φ durchführen. Dies ist der Fall, da nur Klauseln, die nicht x_1 enthalten, Elternklauseln für die Resolutionsschritte sein können. Daher wird auch keine tautologische Resolvente durch $x_1 = 1$ erzeugt. Es folgt:

$$\Phi \underset{\text{Q-RES}}{\vdash\!\!\!-\!\!\!-} \sigma_0 \text{ oder } \Phi \underset{\text{Q-RES}}{\vdash\!\!\!-\!\!\!-} \neg x_1 \vee \sigma_0$$

   Somit kann bei der Q-Resolution in Φ entweder die exakt gleiche Klausel σ_0 erzeugt werden, die bei Q-Resolution in Φ_0 entsteht oder die Klausel $\sigma_0 \cup \neg x_1$. Hierbei kann $\sigma_0 \vee \neg x_1$ auch keine Tautologie sein, da $x_1 = 1$. Demnach folgt aus der Widerlegbarkeit von Φ_0 auch die Widerlegbarkeit von Φ.

□

Damit ist bewiesen, dass für jede widerspruchsvolle quantifizierte Boole'sche Formel mit der Q-Resolution die leere Klausel herleitbar ist. Zudem gilt, dass falls die leere Klausel aus einer quantifizierten Boole'schen Formel herleitbar ist, diese Formel widersprüchlich ist. Die Q-Resolution ist widerlegungsvollständig und korrekt (KBL94).

Theorem 3.5 (Widerlegungsvollständigkeit und Korrektheit)**.** *Sei* Φ *eine Formel aus* $\mathcal{QBF}$*. Dann gilt:*

$$\Phi \textit{ widerspruchsvoll} \Leftrightarrow \Phi \underset{\textit{Q-RES}}{\vdash\!\!\!-\!\!\!-} \sqcup$$

Demnach gilt auch für jede Formel aus $\mathcal{QBF}$, dass sie, wenn die leere Klausel nicht herleitbar ist, auch nicht widerspruchsvoll ist und damit wahr. Die Q-Resolution kann somit für jede QBF-Formel (und damit auch für jede QKNF-Formel) den Wahrheitswert bzw. die Erfüllbarkeit bestimmen.

4 Algorithmen

Durch theoretische Betrachtungen wurden in den vorherigen Kapiteln aussagenlogische und quantifizierte Boole'sche Formeln sowie Verfahren zum Lösen oder Widerlegen solcher Formeln kennengelernt. Ziel dieser Arbeit ist es, Algorithmen zur Anwendung dieser Verfahren zu beschreiben und zu präsentieren.
Zunächst wird eine praktische Datenstruktur für logische Formeln gesucht. Diese soll sowohl zur algorithmischen Bearbeitung, als auch zum Schreiben und Auslesen von Dateien geeignet sein. Hierzu werden Standards im Bereich des SAT- und QBF-Solving vorgestellt. Anschließend werden Algorithmen zur Generierung von (KNF und QKNF-) Formeln vorgestellt. Abgesehen von den Algorithmen an sich, sollen vor allem auch Ergebnisse hinsichtlich der Laufzeit erschlossen werden. Diese Ergebnisse sollen möglichst derart gesammelt werden, dass sie sich zueinander in Kontext setzen lassen.

Die richtige Datenstruktur für einen Algorithmus sollte unkompliziert sein, aber alle relevanten Informationen speichern können. Sie sollte gleichzeitig wenig Speicherplatz verlangen und die Laufzeit möglichst nicht beeinträchtigen.
Eine aussagenlogische Formel in KNF lässt sich als Menge von Klauseln auffassen (siehe Beispiel 2.1). Diese Darstellungsweise bietet sich an, da sie formal einfach und übersichtlich ist und sich genauso einfach implementieren lässt. Für eine QKNF-Formel muss zusätzlich zur Klauselmenge das Präfix gespeichert sein. Hier ist die Reihenfolge relevant, daher wird das Präfix als Folge von Quantifizierungen aufgefasst.

Sowohl Klauselmengen als auch das Präfix wurden durch Listen implementiert. Die Implementierung durch Listen ist einfach und lässt sich sortieren. Sortierte Klauseln dienen einerseits der Übersicht, andererseits bei der Q-Resolution praktischen Zwecken.

DIMACS

Das `DIMACS`-Format ist das verbreitete Standardformat für Ein- und Ausgabe von Boole'schen Formeln in KNF (HIK+20). „The International SAT Competition“ setzt für die Probleminstanzen in ihren SAT-Solving-Wettbewerben diesen Formatstandard fest (Com09). Informationen werden folgendermaßen zusammengefasst (vgl. Beispiel 4.1):
Kommentarzeilen zu Beginn der Datei werden durch ein *c* am Anfang der Zeile identifiziert. Darauf folgt eine Headerzeile in der Eigenschaften der Formel genannt werden.

Sie wird begonnen mit einem p wie „problem“ oder s wie „solution“. Dann steht in ihr cnf für Formeln in KNF, sowie die Anzahl der Variablen und Klauseln des Problems. Schließlich folgen Klauselzeilen, in denen Zahlen als Repräsentation für Atome stehen. Eine 1 steht für x_1, eine -1 für $\neg\, x_1$. Am Ende jeder Klauselzeile wird eine 0 geschrieben.

```
c Kommentar
p cnf 5 3
1 3 -4 -2 0
-3 2 5 0
2 3 0
```

Zugehörige Formel:
$(x_1 \vee x_3 \vee \neg x_4 \vee \neg x_2) \wedge (\neg x_3 \vee x_2 \vee x_5) \vee (x_2 \vee x_3)$

Beispiel 4.1: `DIMACS` Dateiformat

Dieses Format lässt sich auf eine Datenstruktur für die Algorithmen übertragen. Kommentarzeilen werden nicht weiter beachtet.

class `Dimacs`:
int *nbvar*, *nbclauses*
list *clauses*

QDIMACS

Das `QDIMACS`-Format erweitert das Prinzip des `DIMACS`-Formats für quantifizierte KNF-Formeln. Zusätzliche Zeilen enthalten die Quantoren. Jede `DIMACS`-Formel ist auch eine `QDIMACS`-Formel, dessen Quantorenpräfix leer ist. „The Quantified Boolean Formulas Satisfiability Library“, kurz QBFLIB, verwendet für ihre jährlichen QBF-Solver Wettbewerbe auch den `QDIMACS`-Standard (GNPT05).
Eine Quantorenzeile beginnt entweder mit einem a für $\forall$ oder einem e für $\exists$. Freie Variablen werden als in der ersten (äußersten) Quantorenmenge existenziell quantifiziert aufgefasst.

```
p cnf 5 3
a 1 2 0
e 3 4 5 0
1 3 -4 -2 0
-3 2 5 0
2 3 0
```

Zugehörige Formel:
$\forall x_1, x_2 \exists x_3, x_4, x_5 (x_1 \vee x_3 \vee \neg x_4 \vee \neg x_2) \wedge (\neg x_3 \vee x_2 \vee x_5) \wedge (x_2 \vee x_3)$

Beispiel 4.2: `QDIMACS` Dateiformat

class `Qdimacs`:
int $nbvar$, $nbclauses$, $nbquant$
list $quantifiers$, $clauses$

4.1 Generierung von logischen Problemen

Die verschiedenen Algorithmen für die logischen Probleme sollen vergleichbar gemacht werden. Um dies zu realisieren, sollen die logischen Probleme den Anforderungen entsprechend generiert werden. So können ihre Maße angepasst und die Auswirkungen der Anpassungen verglichen werden.
Im Fall von aussagenlogischen Formeln, die im `DIMACS`-Format gefasst werden sollen, sind diese Maße die Anzahl der Variablen und die Anzahl sowie Länge der Klauseln. Bei `QDIMACS`-Problemen lässt sich zudem die Anzahl der quantifizierten Variablen verändern. Im Rahmen dieser Arbeit wurde nur ein grundlegender, auf Zufallszahlgeneratorprinzip basierender Algorithmus zur Erzeugung von logischen Problemen verwendet. Es sei an dieser Stelle jedoch darauf hingewiesen, dass das Finden von schwierigen Formeln eine Herausforderung für sich ist, welche eigens darauf konzentrierte Veröffentlichungen inspiriert wie zum Beispiel (ART20) und (HJKN06). Ziel ist es, schwierigere Probleme zu finden, welche hilfreich zum Verbessern von Solvern sind, und Testmengen zu bilden, die Algorithmen in einem fairen Wettbewerb untereinander messbar machen (ART20, RH01, HJKN06).

Zunächst soll ein Algorithmus zur Erzeugung von zufälligen Klauseln gefunden werden. Die Klausel wird in Abhängigkeit von der Anzahl verfügbarer Variablen generiert. Es wurden zwei verschiedene Ansätze verfolgt. Der erste Ansatz (Algorithmus 4.1) iteriert über alle Atome und wählt zufällig, ob das Atom positiv, negativ oder gar nicht in der Klausel vorkommt. So erzeugte Klauseln sind automatisch nach „aufsteigenden Atomen“ geordnet (d.h. erst 1 oder -1, dann 2 oder -2, ..). Allerdings ist die Länge so erzeugter Klausel häufig nahe dem Durchschnitt von $\frac{2}{3} * nbvar$, da für jeden Iterationsschritt die gleichen Wahrscheinlichkeiten gelten. Für so erzeugte Formeln bedeutet dies eine sehr geringe Varianz der Klausellänge. Für eine Formel mit $nbvar = 10$ beträgt die Wahrscheinlichkeit eine Klausel der Länge 1 zu erzeugen 0,03% ($\frac{1}{3}^9 * \frac{2}{3}^1 * 10 \approx 0,000339$).

Algorithm 4.1 Generiere Klausel 1

Input: *nbvar* Anzahl der Variablen
Output: C Klausel

```
1: function MakeClause(nbvar)
2:     C ← {}
3:     for i ← 1 ... nbvar do
4:         Wähle x aus {i, 0, −i}
5:         if x ≠ 0 then
6:             Füge x zu C hinzu
7:     return C
```

Der zweite Ansatz (Algorithm 4.2) wählt zuerst zufällig die Länge der Klausel aus dem Intervall von 1 bis *nbvar*. So ist eine größere Varianz bezüglich der Länge der Klauseln möglich. Um gleichzeitig die Wahrscheinlichkeit des Auftretens für alle Variablen gleich groß zu halten, wird zunächst eine Variable nach der anderen zufällig ausgewählt und die Klausel dann im Anschluss aufsteigend sortiert.
Außerdem soll es möglich sein, Klauseln bestimmter Länge zu erzeugen. Ist eine bestimmte Länge gewünscht, wird diese mit dem Parameter *length* übergeben.

Algorithm 4.2 Generiere Klausel 2

Input: *nbvar* Anzahl der Variablen
Output: C Klausel

```
1: function MakeClause(nbvar, length)
2:     C ← {}
3:     available ← 1 ... nbvar
4:     if length ≤ 0 then
5:         Wähle length aus {1 ... nbvar}
6:     for i ← 0 ... length do
7:         Wähle x aus available
8:         Entferne x aus available
9:         Wähle ob x positiv oder negiert vorkommt und füge es zu C hinzu
10:    Sortiere C                                          ▷ Optional
11:    return C
```

Um QBF-Formeln zu erzeugen, muss auch ein Präfix generiert werden. Erzeugte Formeln sollen den in Abschnitt 2.2 beschriebenen Konventionen folgen, wobei die kleinsten Variablen auch durch die kleinsten Zahlen repräsentiert werden sollen. Daher sind die Quantifizierungen eine aufsteigende Zahlenreihe (siehe 4.2). Der folgende Algorithmus lässt auch freie Variablen zu. Hierzu werden sie zu Beginn ausgewählt und aus der Menge zu quantifizierender Variablen entfernt. Anschließend werden die Variablen in dieser Menge zufällig mit $\forall$ oder $\exists$ quantifiziert.

Algorithm 4.3 Generiere Präfix

Input: *nbvar* Anz. Variablen, *nbquant* Anz. Quantifizierungen
Output: Π Liste aus Quantifizierungen

```
1: Π ← {}
2: available ← 1 ... nbvar
3: nFreeVars ← nbvar − nbquant
4: for i ← 1 ... nFreeVars do                ▷ Wähle freie Variablen aus
5:     Wähle x aus available
6:     Entferne x aus available
7: for x ∈ available do                      ▷ Quantifiziere übrige Variablen
8:     Wähle Q aus {∀, ∃}
9:     Füge Qx zu Π hinzu
10: return Π
```

4.2 Q-Reduktion

Das Prinzip der Lösung von quantifizierten Boole'schen Formeln per Reduktion wurde in Beispiel 2.3 im Abschnitt 2.2 gezeigt. Die Quantifizierung wird Schritt für Schritt aufgelöst und die Formel erweitert. Betrachtet wird nun ein Algorithmus, der dies für geschlossene QKNF-Formeln tut.

Die `SOLVE`-Funktion wird mit folgenden Parametern aufgerufen:

- Akkumulator (zu Beginn mit Wert 1)
- Anzahl der Variablen (*nbvar*)
- Quantoren (Formelpräfix)
- Klauselmenge (Formelkern)

In der Funktion wird, für das Atom dessen Index dem Wert des Akkumulators entspricht (x_1 bzw. y_1 für Akkumulator 1), jedes Vorkommen ein Mal durch $True$ und ein Mal durch $False$ ersetzt. So werden zwei neue Klauselmengen erstellt. Für die entstandenen Klauselmengen wird `SOLVE` rekursiv aufgerufen. In jedem Funktionsaufruf werden zwei neue Aufrufe getätigt. Dadurch wächst der Aufwand exponentiell in Abhängigkeit von der Anzahl der Variablen. Mit jedem Rekursionsschritt wird eine Variable aus dem Präfix entfernt und in der Formel ersetzt.

Algorithm 4.4 QBF-Reduktion

Input: QKNF-Formel $\Pi\varphi$ mit Klauselmenge φ und Präfix Π
Output: Boole'scher Wert, der die Erfüllbarkeit von φ angibt

```
1: function SOLVE(x, nbvar, Π, φ)
2:     if x > nbvar then                          ▷ Abbruchbedingung, alle Literale bearbeitet
3:         return SOLVE-CLAUSES(φ)                                  ▷ Werte Klauseln aus
4:     C_0 ← {}  ▷ Klauseln in denen x durch False und ¬x durch True ersetzt wurde
5:     C_1 ← {}  ▷ Klauseln in denen x durch True und ¬x durch False ersetzt wurde
6:     for clause ∈ φ do
7:         if x ∈ clause then                          ▷ Fall 1: Klausel enthält x
8:             temp ← clause \ {x}                                  ▷ Entferne x
9:             C_1 ← C_1 ∪ {temp ∪ {True}}          ▷ Füge Klausel + True zu C_1 hinzu
10:            C_0 ← C_0 ∪ {temp ∪ {False}}         ▷ Füge Klausel + False zu C_0 hinzu
11:        else if ¬x ∈ clause then        ▷ Fall 2: Klausel enthält ¬x; Spiegelt Fall 1
12:            temp ← clause \ {¬x}
13:            C_1 ← C_1 ∪ {temp ∪ {False}}
14:            C_0 ← C_0 ∪ {temp ∪ {True}}
15:        else                          ▷ Fall 3: Klausel enthält weder x noch ¬x
16:            C_1 ← C_1 ∪ {clause}                 ▷ Füge unveränderte Klausel hinzu
17:            C_0 ← C_0 ∪ {clause}
18:    if ∀x ∈ Π then                                ▷ Fall 1: x ist allquantifiziert
19:        Π ← Π \ {∀x}                                  ▷ Entferne ∀x aus dem Präfix
20:        return SOLVE(x + 1, nbvar, Π, C_0) and SOLVE(x + 1, nbvar, Π, C_1)   ▷ Gebe
   True zurück, falls beide rekursiven Aufrufe True zurückgeben
21:    else if ∃x ∈ Π then                           ▷ Fall 2: x ist existenzquantifiziert
22:        Π ← Π \ {∃x}
23:        return SOLVE(x + 1, nbvar, Π, C_0) or SOLVE(x + 1, nbvar, Π, C_1)      ▷ Gebe
   True zurück, außer beide rekursiven Aufrufe geben False zurück
24: function SOLVE-CLAUSES(φ)
25:    for clause ∈ φ do
26:        if True ∉ clause then
27:            return False
28:    return True
29: nbvar ← len(var(φ))
30: return SOLVE(1, nbvar, Π, φ)
```

Sei $n = len(\Pi\varphi)$ die Länge der Eingabe. n bleibt im Verlauf der Ausführung konstant, auch wenn der Formelkern wächst. Dieses Wachstum wird durch $len(\varphi_x)$ beschrieben, wobei x die Rekursionstiefe bezeichnet. Betrachtet wird zunächst die Funktion `Solve`. Die Funktion wird für jede Variable zwei mal rekursiv aufgerufen. Ein Aufruf iteriert über jede Klausel an denen einige Mengenoperationen durchgeführt werden, bevor das Präfix gekürzt wird und die rekursiven Aufrufe getätigt werden.

Funktion	`Solve`
Anzahl Aufrufe	2^{nbvar}
Laufzeit pro Aufruf	$\mathcal{O}(1) + \mathcal{O}(len(\varphi_x)) + \mathcal{O}(n)$

Funktion	`Solve-clauses`
Aufrufe	2^{nbvar}
Laufzeit	$\mathcal{O}(len(\varphi_{nbvar}))$

Länge des Formelkerns	
$len(\varphi_x)$	$2 * len(\varphi_{x-1}) = 2^{x-1} * len(\varphi_0)$

QBF-Reduktion	
Laufzeit	$(\sum_{x=0}^{nbvar} 2^x * \mathcal{O}(len(\varphi_x))) + 2^{nbvar} * \mathcal{O}(len(\varphi_{nbvar}))$

4.3 Umformung in eine aussagenlogische Formel

Ähnlich zum Prinzip der QBF-Reduktion, lässt sich eine quantifizierte Boole'sche Formel in eine logisch äquivalente aussagenlogische umformen. Wenn bei der Reduktion die Variablen durch 1 bzw. 0 ersetzt werden, werden hier die Variablen stehen gelassen bzw. negiert.

$$\forall x_1 \forall x_2 (x_1 \vee x_2)$$
$$\forall x_2 (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2)$$
$$(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2)$$

Beispiel 4.3: Umformung QBF $\rightarrow$ KNF

In Beispiel 4.3 wird im ersten Schritt die erste Quantifizierung bearbeitet. $\forall x_1$ wird aus dem Präfix gestrichen. Dafür wird die Klauselmenge dupliziert und im Duplikat wird x_1 durch $\neg x_1$ ersetzt. Die ursprüngliche Formel wird mit dem modifizierten Duplikat durch $\wedge$ verbunden. Im Fall eines Existenzquantors werden die Formelteile durch ein $\vee$ verbunden.

Die Funktionsweise der Umformung ist also beinahe identisch zur QBF-Reduktion, aber das Ergebnis unterscheidet sich. Das Resultat der QBF-Reduktion ist eine aussagenlogische Formel, die keine Literale enthält, sondern ausschließlich Einsen (oder $True$) und Nullen ($False$). Diese Formel besitzt einen eindeutigen Wahrheitswert, welcher sich bestimmen lässt.

Das Resultat der Umformung ist eine aussagenlogische Formel, welche dieselben Literale enthält, wie die vorangegangene quantifizierte Boole'sche Formel. Diese Formel hat 2^{nbvar} Belegungsmöglichkeiten und ist genau dann erfüllbar, wenn das Resultat der Reduktion wahr ist.

Algorithm 4.5 Umformung QBF → Aussagenlogische Formel

Input: QKNF-Formel $\Pi\varphi$ mit Klauselmenge φ und Präfix Π
Output: Aussagenlogische Formel φ

```
1:  function RemoveQuantifier(QKNF-Formel Πφ)
2:      if Keine Quantifizierung in Π then
3:          return φ
4:      Wähle Quantifizierung Qx aus Π
5:      C_0 ← {}                ▷ Sammle Klauseln in denen x durch ¬x ersetzt wird
6:      C_1 ← {}                                          ▷ Unveränderte Klauseln
7:      for clause ∈ φ do
8:          if x ∈ clause then                            ▷ Fall 1: Klausel enthält x
9:              temp ← clause \ {x}                                  ▷ Entferne x
10:             C_0 ← C_0 ∧ {temp ∪ {¬x}}          ▷ Füge Klausel + ¬x zu C_0 hinzu
11:         else if ¬x ∈ clause then      ▷ Fall 2: Klausel enthält ¬x; Spiegelt Fall 1
12:             temp ← clause \ {¬x}
13:             C_0 ← C_0 ∧ {temp ∪ {x}}
14:         else                          ▷ Fall 3: Klausel enthält weder x noch ¬x
15:             C_1 ← C_1 ∧ {clause}                  ▷ Füge unveränderte Klausel hinzu
16:             C_0 ← C_0 ∧ {clause}
17:     if Q = ∀ then                                  ▷ Fall 1: x ist allquantifiziert
18:         return RemoveQuantifier(C_0) ∧ RemoveQuantifier(C_1)
19:     else                                ▷ Q = ∃; Fall 2: x ist existenzquantifiziert
20:         return RemoveQuantifier(C_0) ∨ RemoveQuantifier(C_1)
```

Die Laufzeit sowie die Vorgehensweise, entsprechen beinahe der der QBF-Reduktion.

Funktion	`RemoveQuantifier`
Anzahl Aufrufe	2^{nbvar}
Laufzeit pro Aufruf	$\mathcal{O}(1) + \mathcal{O}(len(\varphi_x)) + \mathcal{O}(n)$

Länge des Formelkerns	
$len(\varphi_x)$	$2 * len(\varphi_{x-1}) = 2^{x-1} * len(\varphi_0)$

Umformung QBF → Aussagenlogische Formel	
Laufzeit	$\sum_{x=0}^{nbvar} 2^x * \mathcal{O}(len(\varphi_x))$

4.4 Resolution

Der folgende Algorithmus zur Resolution ist (VK19) entnommen und wurde nur leicht abgeändert. Er bestimmt die Erfüllbarkeit von KNF-Formeln.

Algorithm 4.6 Davis-Putnam-Resolution

Input: KNF-Formel φ
Output: Boole'scher Wert, der die Erfüllbarkeit von φ angibt

1: **function** DP-Resolution(KNF-Formel φ)
2: **if** Keine Variable kommt positiv und negativ vor **then**
3: **return** $True$
4: Wähle Variable p aus φ, die positiv und negativ vorkommt
5: $resolventen \leftarrow \{C_1 \cup C_2 \mid C_1 \cup \{p\}, C_2 \cup \{\neg p\}$ sind Klauseln in $\varphi\}$
6: **if** $\sqcup \in resolventen$ **then** ▷ Falls durch Resolution die leere Klausel entsteht ist die Formel unerfüllbar
7: **return** $False$
8: $\varphi \leftarrow \varphi \cup resolventen$ ▷ Resolventen zur Klauselmenge hinzufügen
9: $\varphi \leftarrow \{C \in \varphi \mid p \notin C, \neg p \notin C\}$ ▷ Klauseln, die p oder $\neg p$ enthalten, aus der Klauselmenge entfernen
10: **return** DP-Resolution(φ)

Funktion	`DP-Resolution`
Anzahl Aufrufe	$k \leq nbvar$
Laufzeit pro Aufruf	$\mathcal{O}(len(\varphi_x)^3)$

Länge des Formelkerns	
$len(\varphi_x)$	$len(\varphi_{x-1}) \leq len(\varphi_x) \leq (\frac{1}{2} * len(\varphi_{x-1}))^2$

Davis-Putnam-Resolution	
Laufzeit	$\sum_{x=0}^{k} \mathcal{O}(len(\varphi_x)^3)$

Auch hier wächst der Formelkern unter Umständen exponentiell und steigert damit auch die Laufzeit in exponentiellem Maße.

4.5 CryptoMiniSat

Der SAT-Solver CryptoMiniSat ist ein OpenSource-Projekt und befindet sich seit mehr als 10 Jahren in aktiver Entwicklung (Soo). Dabei gehen etwa 98% der Commits auf den

Hauptentwickler Mate Soos zurück (Soo).
Im Zusammenhang zum Solver entstand das Paper „Extending SAT Solvers to Cryptographic Problems“. Ansatz des Projekts war es, SAT-Solver auf kryptographische Probleme anzuwenden um die Schwierigkeit des Problems zu bewerten (SNC09). Dem Team gelang es so auch, ein schnelleres Verfahren zum Entschlüsseln eines Stream Ciphers zu entwickeln (SNC09).
CryptoMiniSat bietet einige gute Schnittstellen zur Verwendung und bietet ausführliche Informationen zur Laufzeit, Speichernutzung und Lösungsverlauf. Aufgrund dessen wird CryptoMiniSat im Rahmen dieser Arbeit als Referenz-Solver verwendet.

4.6 Q-Resolution

Ziel des nächsten Algorithmus ist das Lösen einer beliebigen QKNF per Q-Resolution. Das Verfahren folgt dem Prinzip von Definition 3.2 und überträgt das Vorgehen des Davis-Putnam-Algorithmus' (siehe Algorithmus 4.6) für KNF-Formeln auf QKNF-Formeln.

Algorithm 4.7 Q-Resolution

Input: QKNF-Formel $\Pi\varphi$ mit Klauselmenge φ und Präfix Π
Output: Boole'scher Wert, der die Erfüllbarkeit von $\Pi\varphi$ angibt

1: **function** Q-Resolution(QBF-Formel $\Pi\varphi$)
2: **if** Keine $\exists$-Variable kommt positiv und negativ vor **then**
3: **return** $True$
4: Wähle Variable p aus φ, die positiv und negativ vorkommt
5: $pos \leftarrow \{c \in \varphi \mid p \in c\}$ ▷ Sammle Klauseln die p enthalten
6: $neg \leftarrow \{c \in \varphi \mid \neg p \in c\}$ ▷ Sammle Klauseln die $\neg p$ enthalten
7: **for** $c \in pos \cup neg$ **do** ▷ Eliminiere Vorkommen von größten $\forall$-Variablen
8: $c \leftarrow c \setminus \{p \mid p \in c, p$ ist $\forall$-Variable und es existiert keine $\exists$-Variable in c, welche kleiner ist als $p\}$ ▷ Erklärung zu „kleiner“, siehe 2.2
9: $resolventen \leftarrow \{C_1 \cup C_2 \mid C_1 \cup \{p\}, C_2 \cup \{\neg p\}$ sind Klauseln in $\varphi\}$
10: **for** $c \in resolventen$ **do**
11: **if** c ist $\forall$-Klausel **then**
12: **return** $False$
13: $\varphi \leftarrow \varphi \cup resolventen$
14: $\varphi \leftarrow \{c \in \varphi \mid p \notin c, \neg p \notin c\}$
15: **return** Q-Resolution($\Pi\varphi$)

Die Q-Resolution fällt in dieselbe Laufzeitkomplexitätsklasse wie die Davis-Putnam-Resolution. Jedoch tritt hier zusätzlicher Aufwand zum Verwalten der Quantifizierungen sowie Eliminieren von $\forall$-Variablen auf.

5 Analyse

Es sollen Algorithmen anhand von gesammelten Daten zu Resultat, Laufzeit und Speichernutzung verglichen werden. Hierzu werden KNF- und QKNF-Formeln im `DIMACS`- bzw. `QDIMACS`-Format verwendet. Diese logischen Probleme werden durch verschiedene Algorithmen verarbeitet. Dabei werden die benötigte Zeit und der benötigte Speicherplatz gemessen. Algorithmen, welche identische Formeln mit demselben Ziel verarbeiten, sollen durch Analyse der Messdaten miteinander verglichen werden.

Messmethode

Die Algorithmen wurden, abgesehen von dem SAT-Solver CryptoMiniSat, eigens in der Programmiersprache Python implementiert. Laufzeit- und Speichermessungen konnten daher im Programm selbst vorgenommen werden (Pythonmodule *time, tracemalloc*). Die Hardwareumgebung der Tests war ein Heimcomputer (siehe 6.7). Da die Laufzeit für die Algorithmen mitunter exponentiell zunimmt, wurde stets ein Subprozess gestartet (Pythonmodul *multiprocessing*), der im Falle eines Timeouts frühzeitig terminiert wurde. Die Timeoutgrenze wurde in den meisten Fällen bei 20 Sekunden angesetzt.
Bei einer derartigen Zeitmessung ist eine geringe konstante Verzögerung enthalten, die durch das Starten des Subprozesses zustande kommt. Messdaten zu Laufzeit und Speicherbedarf zu CryptoMiniSat sind der Ausgabe des aufgerufenen Algorithmus entnommen und enthalten dadurch nicht dieselbe Verzögerung.

Zeitmessungen zu einem Algorithmus und einer Formel wurde stets 20 mal wiederholt. Aus den 20 Messwerten wurde der Median ausgewählt. So sollte eine Beeinflussung der Werte durch parallel laufende Prozesse minimiert werden.
Zu bestimmten Parametern wurde je eine Vielzahl an Formeln generiert und Zeitmessungen für den Algorithmus und die Formeln wie oben beschrieben durchgeführt. Durch die Betrachtung einer Vielzahl von Formeln soll eine möglichst gute Voraussage für alle Formeln mit diesen Parametern getroffen werden.

Probleminstanzen

Es werden zwei verschiedene Arten von Probleminstanzen betrachtet: Quantifizierte Boole'sche Formeln in QKNF als `QDIMACS`-Datei (siehe 4.2) und aussagenlogische Formeln in KNF als `DIMACS`-Datei (siehe 4.1). Als Lösung zu diesen Problemen wird ihre Erfüllbarkeit oder im Fall einer geschlossen quantifizierten Boole'schen Formel ihr Wahrheitswert gesucht. Damit ergeben sich zunächst zwei Wettbewerbskategorien in welchen Algorithmen untereinander verglichen werden können. Diese sind im Grunde der Bereich des SAT- und des QBF-Solvings. Für die Anpassung der Probleminstanzen gibt es eine Reihe von Parametern, die im Folgenden erläutert werden.

1. Anzahl der Variablen:
   Die Anzahl der verschiedenen Variablen, die bei der Erstellung der Formel im Pool sind. Für den Algorithmus zum Erzeugen von logischen Problemen, der in dieser Arbeit beschrieben (siehe Algorithmus 4.2) und für folgende Messungen verwendet wurde, beschreibt diese Zahl nur die *maximale Anzahl* verschiedener Variablen. Die tatsächliche Zahl kann davon abweichen.

2. Anzahl der Klauseln:
   Die genaue Zahl der Klauseln, die in der Formel enthalten sind.

3. Länge der Klauseln:
   Wenn die Länge der Klauseln konstant sein sollen, setzt dieser Wert sie fest. Ansonsten werden Klauseln zufälliger Länge zwischen 1 und „Anzahl der Variablen“ erzeugt.

4. Anzahl der Quantifizierungen:
   Die Anzahl von Variablen, die zufällig existenz- oder allquantifiziert werden. Für die Ausführung der Algorithmen werden freie Variablen als existenzquantifiziert betrachtet um die Erfüllbarkeit der Formel zu erschließen.

Es muss in Betracht gezogen werden, dass zwei Probleme mit der gleichen Anzahl an Variablen, Klauseln, Quantifizierungen und sogar der selben Klauselllänge, in beinahe allen Fällen unterschiedlich schwer zu lösen sind. Um ein Beispiel zu liefern, werden folgende Formeln betrachtet.

$$\varphi = \forall x_1 \exists y_2 \{x_1\}, \{x_1, \neg y_2\}$$
$$\psi = \exists y_1 \exists y_2 \{y_1\}, \{x_1, \neg y_2\}$$

Beispiel 5.1: Unterschiede in der Schwierigkeit von Problemen

Die Formel φ enthält mit $\{x_1\}$ eine $\forall$-Klausel. Diese ist offensichtlich unerfüllbar. Eine solche Formel lässt sich schnell als widersprüchlich erkennen. Resolutionsschritte werden

gar nicht benötigt. Für Formel ψ, die über identische Parameter verfügt, aber nicht über eine $\forall$-Klausel, lassen sich nicht die selben Schlüsse ziehen. Geht man so vor, dass man zunächst alle Klauseln auf Unerfüllbarkeit prüft, so ist für φ die Lösung leichter zu finden. Eine Formel mit identischen Parametern, welche keine derartige unerfüllbare Klausel enthält wie ψ, ist in dem Fall schwerer zu lösen. Selbst wenn φ zusätzliche Klauseln enthielte, den Parametern nach komplexer und scheinbar schwieriger wäre, könnte die Lösung noch leichter sein als für φ.

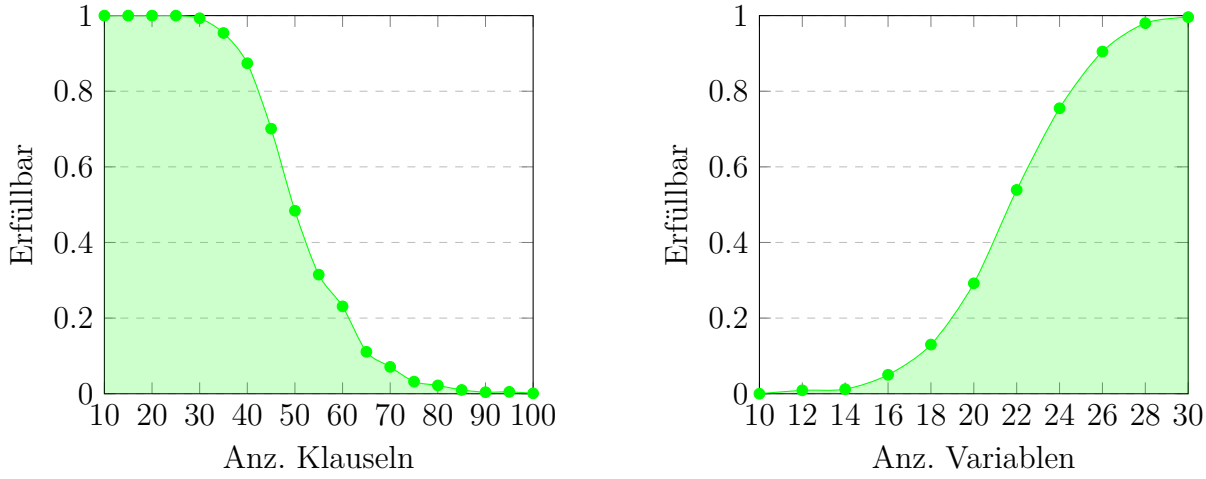
Um diese Argumentation zu unterstützen, wird hier noch ein solcher Fall näher untersucht. Es werden zwei QKNF-Formeln (siehe Anhang 6.1, 6.2) mit den gleichen Parametern und Klauseln betrachtet. Die Erste enthält triviale $\forall$-Klauseln ($\{x_{25}\}$ und $\{x_{15}\}$) und ist daher widersprüchlich. Bei der Zweiten sind zwei Variablen existenz- statt allquantifiziert, wodurch diese beiden Klauseln keine $\forall$-Klauseln mehr sind. Nach der vorangegangenen Argumentation lässt sich vermuten, dass die erste Formel einfacher und schneller lösbar ist. Für beide Formeln wurde der Q-Resolution Algorithmus durchgeführt und Messdaten zur Ausführung gesammelt (Anhang 6.3, 6.4). Der Median der Zeit, die der Algorithmus zum Lösen der Formeln brauchte, betrug bei der ersten 0.155525 Sekunden und bei der zweiten 0.204494 Sekunden. Der Wert für die zweite Formel ist etwas über 30% höher als für die erste. Der Algorithmus löste die als leichter zu Lösen vermutete Formel also tatsächlich messbar schneller.
Dies bestätigt, dass Algorithmen für verschiedene Formeln mit identischen Parametern unterschiedlich gute Leistungen erbringen. Außerdem liegt die Vermutung nahe, dass verschiedene Algorithmen für erfüllbare bzw. unerfüllbare Formeln mit identischen Parametern unterschiedlich gute Leistungen erbringen.

5.1 Erfüllbarkeit generierter Formeln

Algorithmen müssen also über genügend verschiedene Formeln mit gleichen Parametern hinweg verglichen werden, um ein faires Urteil für diese Parameter zu ermöglichen.
Zudem muss bedacht werden, dass für erfüllbare bzw. wahre und unerfüllbare bzw. widersprüchliche Formeln verschiedene Algorithmen unterschiedlich effizient arbeiten können. Es muss also gewährleistet sein, dass die beiden Gruppen (erfüllbarer und unerfüllbarer Formeln) in fairem Häufigkeitsverhältnis zueinander auftreten oder differenziert betrachtet werden.

Doch welches Häufigkeitsverhältnis ist fair? Es wäre naheliegend gleich viele erfüllbare und unerfüllbare Formeln zu vergleichen. Allerdings variiert die Wahrscheinlichkeit, ob eine generierte Formel erfüllbar ist, in Abhängigkeit der Parameter. Demnach wären identische Häufigkeiten in den meisten Fällen eine Über- oder Untergewichtung der einen oder der anderen Gruppe - zumindest in Hinblick auf die im Rahmen dieser Arbeit generierten Formeln.

Dies spricht dafür erfüllbare und unerfüllbare Formeln getrennt zu untersuchen. Doch zunächst soll untersucht werden, wie Parameter Einfluss auf die Erfüllbarkeit generierter Formeln nehmen. Die folgenden Graphen betrachten `Unique 3SAT`.
`Unique 3SAT` ist das Erfüllbarkeitsproblem zu 3-KNF-Formeln; das sind KNF-Formeln deren Klauseln ausschließlich über 3 Literale verfügen. Es werden der Einfluss einer Änderung der Klausel- bzw. Variablenzahl untersucht.



(a) Anz. Variablen = 10, Klausellänge = 3

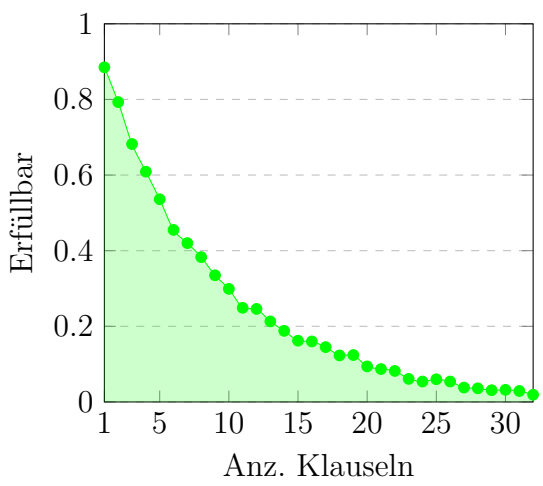
(b) Anz. Klauseln = 100, Klausellänge = 3

Graph 5.1: Anteil erfüllbarer Formeln bei generierten 3KNF-Formeln

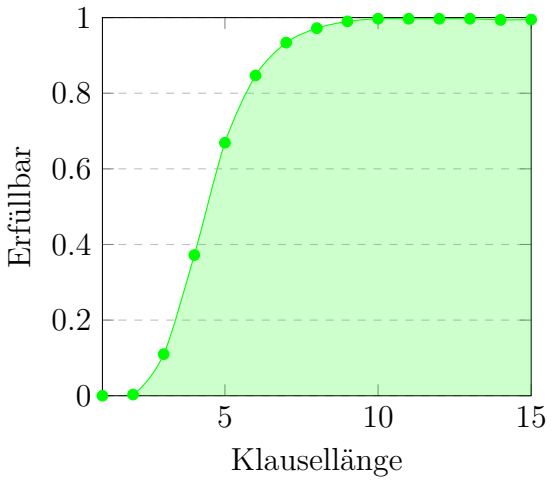
Der Verlauf der Kurve in Graph 5.1 (a) zeigt, dass mit steigender Klauselzahl weniger erfüllbare Formeln entstehen. Graph 5.1 (b) zeigt das gegensätzliche Bild. Bei steigender Zahl der Variablen erhöht sich die Wahrscheinlichkeit für erfüllbare Formeln.
Beides lässt sich einfach erklären. Je mehr Klauseln eine Formel besitzt, umso mehr Aussagen müssen wahr sein, sollte die Formel wahr sein. Die Wahrscheinlichkeit, dass widersprüchliche Aussagen enthalten sind, steigt.
Mit erhöhter Variablenzahl „verteilen“ sich die Aussagen besser. Variablen sind Teil von weniger Klauseln. Die Wahrscheinlichkeit, dass widersprüchliche Aussagen entstehen, sinkt, da sich nur Aussagen widersprechen können, welche die selbe Variable betreffen.

Betrachtet wird in Graph 5.2 (a) nun zunächst der Einfluss der Klauselzahl auf die Erfüllbarkeit generierter QBF-Formeln. Ähnlich zu Graph 5.1 (a) sinkt der Anteil erfüllbarer Formeln mit steigender Klauselzahl. Bei gleicher Zahl der Variablen sinkt die Erfüllbarkeitsrate für Q3-KNF-Formeln gegenüber `Unique 3SAT` erheblich schneller. Q3-KNF umfasst geschlossene QBF-Formeln mit Kern aus 3-KNF (Bie09). So sind bereits bei 19 Klauseln nur noch 5 von 50 erzeugten Formeln erfüllbar. Es stellt sich als schwierig heraus, komplexe QBF-Formeln, speziell in Q3-KNF, zu erzeugen, welche erfüllbar sind.

Graph 5.2 (b) zeigt, dass ein starker Zusammenhang zwischen Klausellänge und Anzahl erfüllbarer Formeln besteht. Bei Klausellänge 2 ist noch keine erfüllbare Formel dabei,



(a) Anz. Variablen = 10, Klausellänge = 3, Anz. Quantifizierungen = 10



(b) Anz. Variablen = 10, Anz. Klauseln = 20, Anz. Quantifizierungen = 10

Graph 5.2: Erfüllbarkeitsanteil bei generierten geschlossenen QBF-Formeln

doch schon für Länge 5 steigt die Erfüllbarkeitsrate auf fast 70%. Erklären lässt sich dies folgendermaßen: Kürzere Klauseln begünstigen das Auftreten von ∀-Klauseln, welche Formeln unerfüllbar machen.

Sei die Wahrscheinlichkeit, dass eine Variable allquantifiziert ist, 50%. Sei außerdem l die Klausellänge und n die Anzahl der Klauseln einer geschlossenen QBF-Formel, so gilt für diese Formel:

Ereignis	Wahrscheinlichkeit
Klausel c ist eine ∀-Klausel	$0,5^l$
Klausel c ist keine ∀-Klausel	$1 - 0,5^l$
Formel ohne ∀-Klauseln	$(1 - 0,5^l)^n$
Formel mit mind. einer ∀-Klausel	$1 - (1 - 0,5^l)^n$

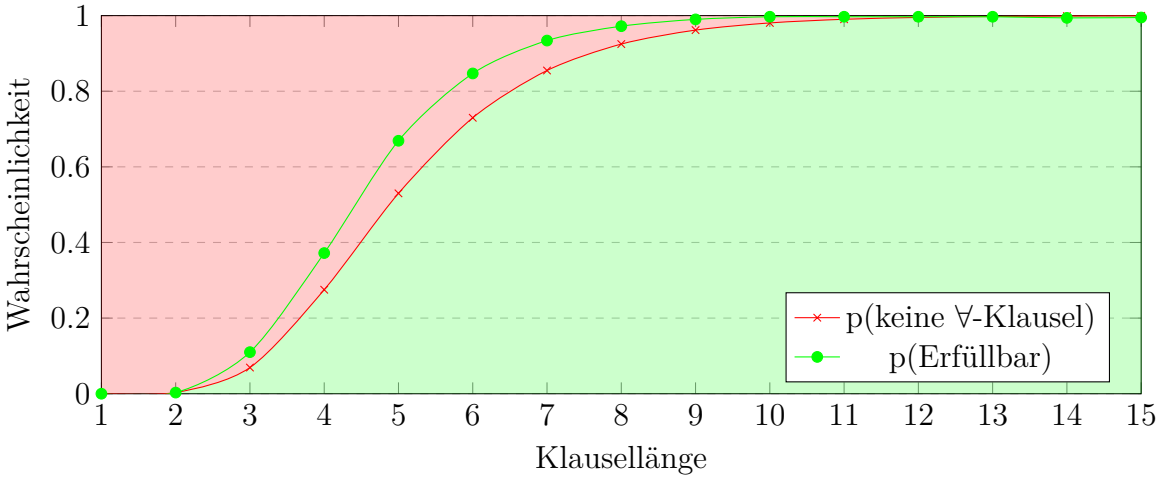
Bei 20 Klauseln lässt sich die Wahrscheinlichkeit auf eine ∀-Klausel so vorhersagen:

Klausellänge	1	2	3	4	5	6	7
p(∀-Klausel)	0,99999	0,99682	0,93079	0,72494	0,47005	0,27018	0,14517

Beispiel 5.2: Vorhersagemodell für ∀-Klauseln

Überträgt man dieses Vorhersagemodell in den Graphen 5.2 (b), ergibt sich folgendes Bild (siehe Graph 5.3): Die Kurven der Erfüllbarkeit und der Vorhersage für ∀-Klauseln liegen nah beieinander. Dies indiziert, dass ein großer Teil der unerfüllbaren Formeln direkt durch ∀-Klauseln unerfüllbar wird. Es gibt jedoch Zweifel an der Aussagekraft des

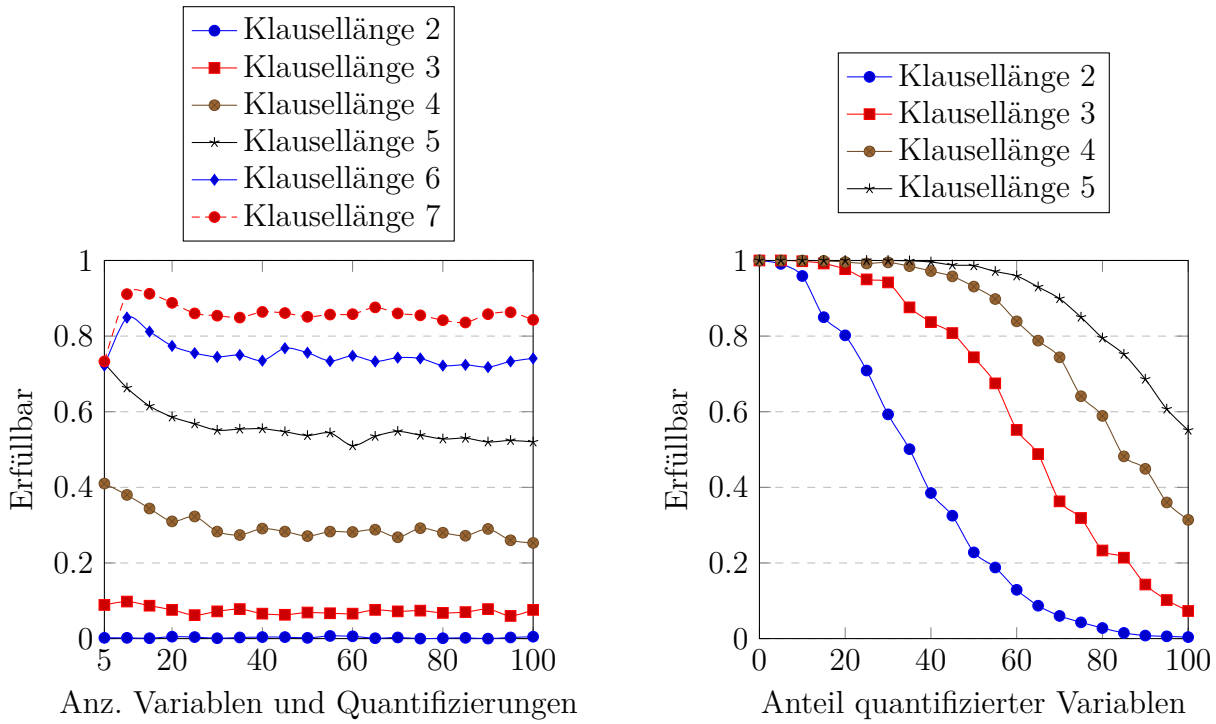
Graphen, denn die Wahrscheinlichkeit für Erfüllbarkeit liegt konsequent höher als die auf keine ∀-Klauseln. Da jede Formel, die eine ∀-Klausel enthält, unerfüllbar ist, dürfte die Wahrscheinlichkeit auf Erfüllbarkeit nicht größer sein, als die auf keine ∀-Klausel. Diese Unstimmigkeit könnte von einer zu kleinen Testmenge herrühren, doch rechtfertigt die Konsistenz, mit der die Erfüllbarkeitsrate höher ausfiel, daran gewisse Zweifel. Ob ein Fehler im Modell oder ein anderer Fehler vorliegt, ist hier ungeklärt.



Graph 5.3: Vergleich des Vorhersagemodells für ∀-Klauseln und Erfüllbarkeit

Trotz der offenen Fragen zeigt die Analyse, dass für quantifizierte Boole'sche Formeln die Suche nach einer ∀-Klausel häufig ein schnelles Ergebnis ermöglicht. Dies betrifft allerdings vor allem Formeln mit kurzen Klauseln. Ein Optimierungsansatz für QBF-Solver könnte also das Untersuchen aller oder nur kurzer Klauseln sein, um festzustellen, ob diese ∀-Klauseln sind oder durch Resolution hervorbringen könnten.

In Graph 5.4 (a) zeigt sich, dass die alleinige Veränderung der Variablenzahl bei geschlossenen QBF-Formeln geringe bis gar keine Auswirkung auf den Anteil der erfüllbaren Formeln hat. Graph 5.4 (b) offenbart, dass der Anteil der quantifizierten Variablen starken Einfluss auf die Erfüllbarkeit generierter Formeln nimmt.
Dies lässt sich einfach erklären. Nicht quantifizierte - also freie Variablen - sind im Sinne der Erfüllbarkeit existenzquantifiziert. Gebundene Variablen sind zufällig und mit gleicher Wahrscheinlichkeit entweder existenz- oder allquantifiziert. Ein höherer Anteil zufällig quantifizerter Variablen bedeutet einen höheren Anteil allquantifizierter Variablen. Allquantifizierte Variablen machen Formeln erheblich „schneller" unwahr als existenzquantifizierte. Dies folgt offensichtlich aus Tatsachen wie z.B., dass eine ∀-Klausel unerfüllbar ist. Deshalb sinkt mit einem höheren Anteil zufällig quantifizierter Variablen die Wahrscheinlichkeit erfüllbarer Formeln.



(a) Anz. Klauseln = 20, Klausellänge = 3

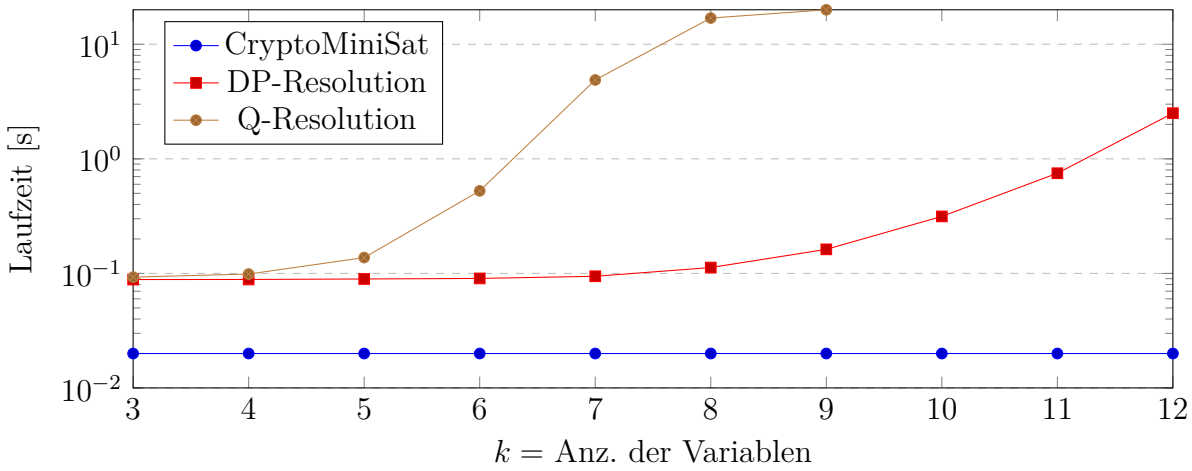
(b) Anz. Variablen = 100, Anz. Klauseln = 20, Klausellänge = 3

Graph 5.4: Erfüllbarkeitsanteil bei generierten QBF-Formeln

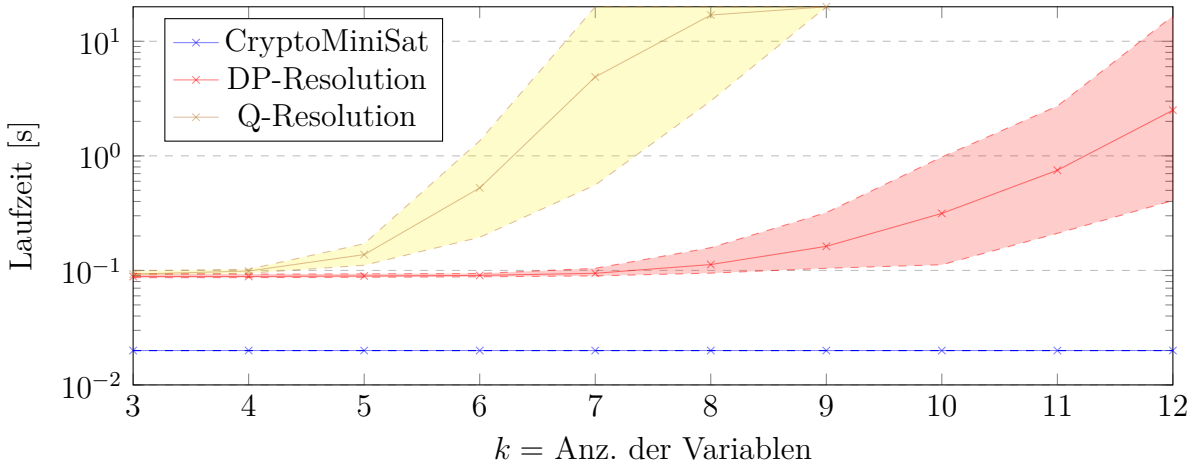
5.2 SAT

Algorithmen für `SAT` sollen hinsichtlich ihrer Laufzeit und dem Speicherbedarf untereinander verglichen werden. Als Testmenge dienen zufällig erzeugte Formeln mit k Variablen, $5 * k$ Klauseln und Klausellänge 3. KNF-Formeln deren Klauseln ausschließlich 3 Literale enthalten, gehören der Formelklasse Unique 3KNF an (MV15, Wik). Das spezielle Erfüllbarkeitsproblem zu Unique 3KNF wird `Unique 3SAT` genannt (MV15, Wik).
Es werden je 50 Formeln von den Algorithmen bearbeitet. Also zuerst 50 Formeln mit 3 Variablen und 15 Klauseln der Länge 3, dann 50 Formeln mit 4 Variablen und 20 Klauseln der Länge 3, ... und so weiter. Die Zahl der Variablen und Klauseln wurde basierend auf den Beobachtungen aus Graph 5.1 gewählt. Bei einem Verhältnis von Klausel- zu Variablenzahl von 5:1 entstanden ungefähr gleich viele erfüllbare wie unerfüllbare Formeln. Es werden die Algorithmen CryptoMiniSat, Davis-Putnam-Resolution (DP-Resolution) und Q-Resolution verglichen. Da `SAT` ein spezieller Fall der `QBF` ist, lässt sich der Algorithmus der Q-Resolution hier anwenden, wobei alle Variablen als existenzquantifiziert betrachtet werden.

Laufzeit



Graph 5.5: Durchschnittliche Laufzeit bei `Unique 3SAT`

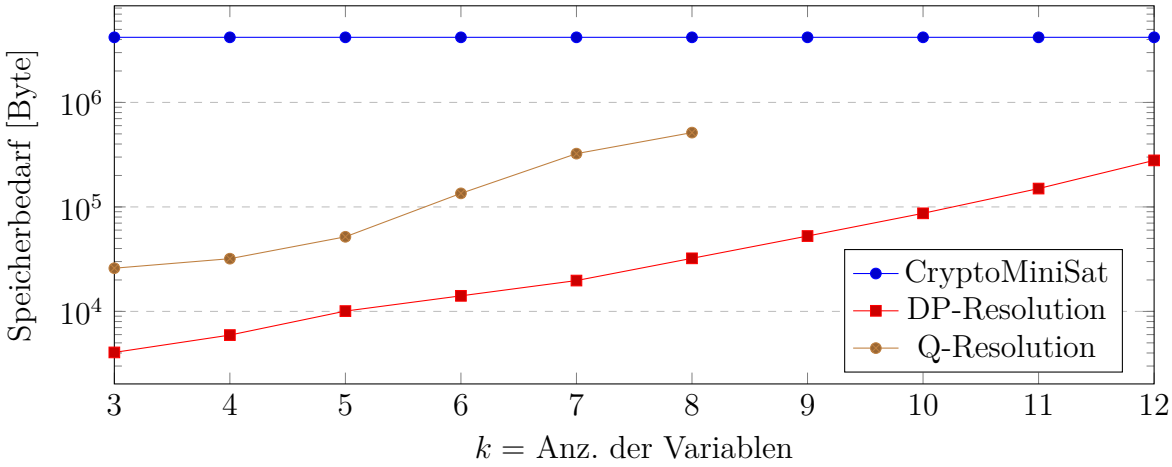


Graph 5.6: Maximale und minimale Laufzeit bei `Unique 3SAT`

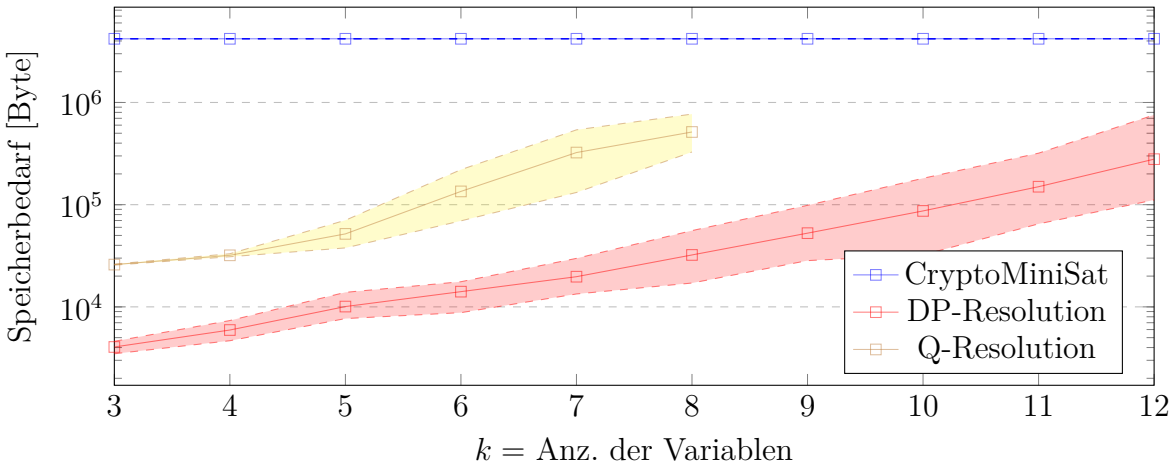
An der horizontalen Achse des Graph 5.5 steht der Parameterfaktor k. Die vertikale Achse ist logarithmisch und zeigt die Laufzeit in Sekunden. Etwa 55% der Formeln der Testmenge waren erfüllbar.
Der CryptoMiniSat-Algorithmus gibt konstant eine Laufzeit von 0.02 Sekunden an und löst damit alle Formeln am schnellsten. Die DP-Resolution zeigt bis $k = 7$ (Klauselzahl 35) zuverlässig Laufzeiten unter 0.1 Sekunden. Er arbeitet von den drei Algorithmen am zweitschnellsten und hat auch bei 60 Klauseln ($k = 12$) keinen Timeout. Die Laufzeit der Q-Resolution nimmt schnell exponentiell zu. Bei $k = 7$ treten vereinzelt erste Timeouts auf und bei $k = 9$ löst er im Zeitfenster von 20 Sekunden keine einzige Formel mehr. Sowohl die Laufzeit der DP- als auch die der Q-Resolution weisen hohe Disparität auf. Diese Beobachtungen decken sich mit den Erwartungen. Der langjährig in C# entwickelte SAT-Solver CryptoMiniSat arbeitet am schnellsten und wird durch die Zunahme der

Formelgröße in diesen Größenordnungen noch nicht verlangsamt. Die Resolution für aussagenlogische Formel wird von zunehmender Problemgröße nicht so schnell überwältigt wie die Q-Resolution. Da die Q-Resolution aus dem `SAT` Problem ein komplexeres `QBF` Problem macht, war auch das zu erwarten.

Speicher



Graph 5.7: Durchschnittlicher Speicherbedarf bei `Unique 3SAT`



Graph 5.8: Maximaler und minimaler Speicherbedarf bei `Unique 3SAT`

An der horizontalen Achse des Graph 5.7 steht erneut der Parameterfaktor k. Die vertikale Achse ist logarithmisch und zeigt den Speicherbedarf in Byte. Die Testmenge ist dieselbe wie bei Graph 5.5.
Interessanterweise ist der Speicherbedarf, den der CryptoMiniSat Algorithmus angibt, höher, als der von den anderen Algorithmen. Doch wie auch seine Laufzeit ändert sich der angegebene Speicherbedarf nicht mit Veränderung der Formelgröße. Er liegt stetig

bei 4,2MB und weicht nur vereinzelt minimal ab. Der Speicherbedarf liegt zunächst für die DP-Resolution am niedrigsten.

Der CryptoMiniSat-Algorithmus ist eindeutig der beste zum Lösen von KNF-Formeln. Bei erfüllbaren Formeln mit 10.000 Variablen und 50.000 Klauseln der Länge 5 gab er immer noch nur eine durchschnittliche Laufzeit von 0.113 Sekunden an.

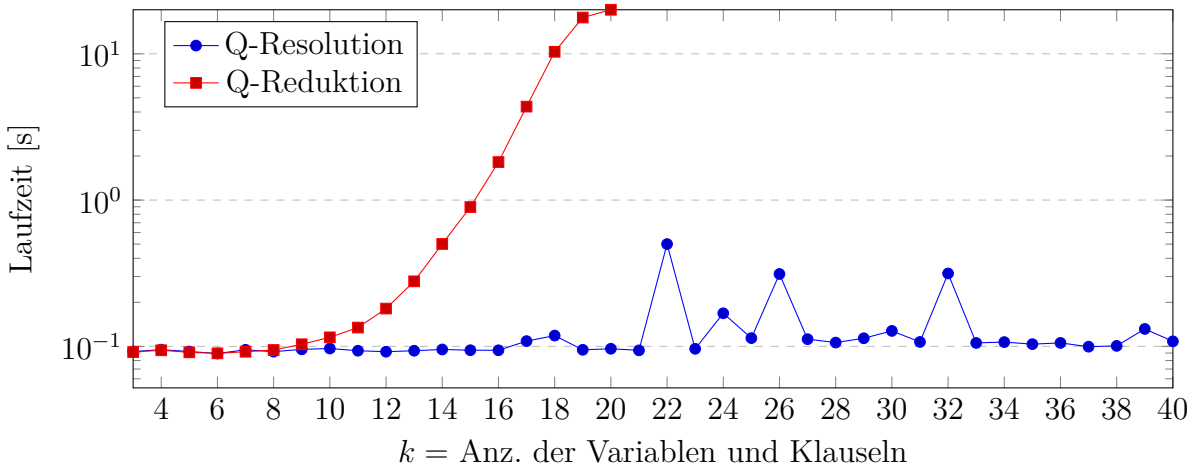
5.3 QBF

Messdaten zu Laufzeit und Speicherbedarf verschiedener Algorithmen für `QBF` sollen präsentiert und verglichen werden. Algorithmen für `QBF` sind das Hauptaugenmerk dieser Arbeit. Daher sind diese Vergleiche die wichtigsten Ergebnisse im Rahmen dieser Analyse. Die richtige Testmenge zum Vergleich zu finden, stellt eine große Herausforderung dar. Wie in Kapitel 5.1 festgestellt, wurde es mit den Algorithmen zur Erzeugung von QBF-Formeln auf Basis von Zufallszahlengenerierung schwierig, größere Formeln zu erzeugen, welche erfüllbar sind. Die große Mehrheit der komplexeren QBF-Formeln wurde unerfüllbar.
Dies stellt eine Verzerrung der Ergebnisse dar. Für größere Formeln nimmt die Präferenz für Algorithmen, welche unerfüllbare Probleme schneller lösen, zu. Wie die Häufigkeit erfüllbarer und unerfüllbarer QBF-Formeln in praktischen Anwendungsbereichen sich verhält, ist hier leider nicht bekannt. Daher soll an dieser Stelle nur stets im Blick behalten werden, wie die Testmenge sich verhält, und ob bestimmte Algorithmen möglicherweise bevorteilt sind. Zudem werden mehrere Testläufe mit verschiedenen Parametern durchgeführt.

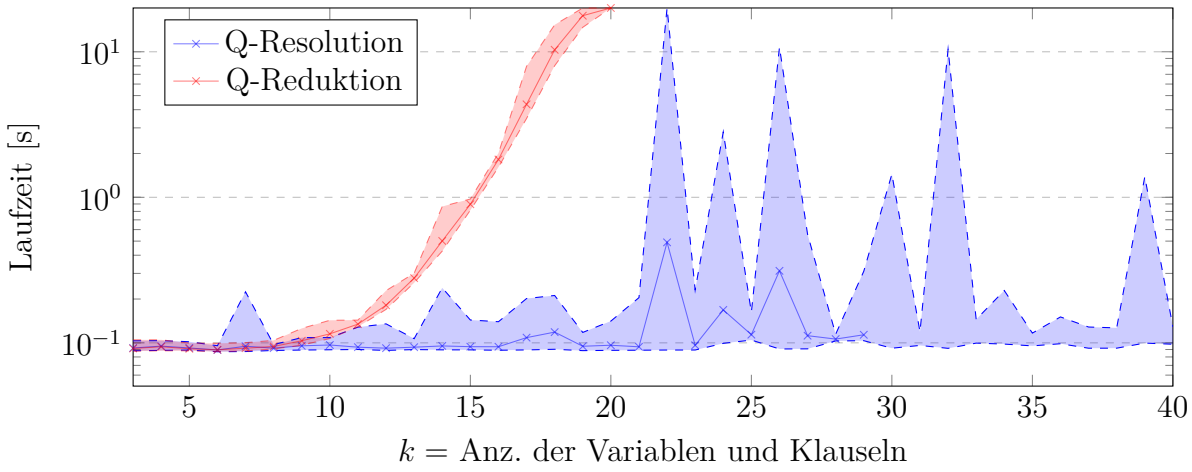
Testlauf 1

Parameter	Wert
Anz. Variablen	k
Anz. Quantifizierungen	k
Anz. Klauseln	k
Klausellänge	3

Laufzeit



Graph 5.9: Durchschnittliche Laufzeit bei Q3-KNF

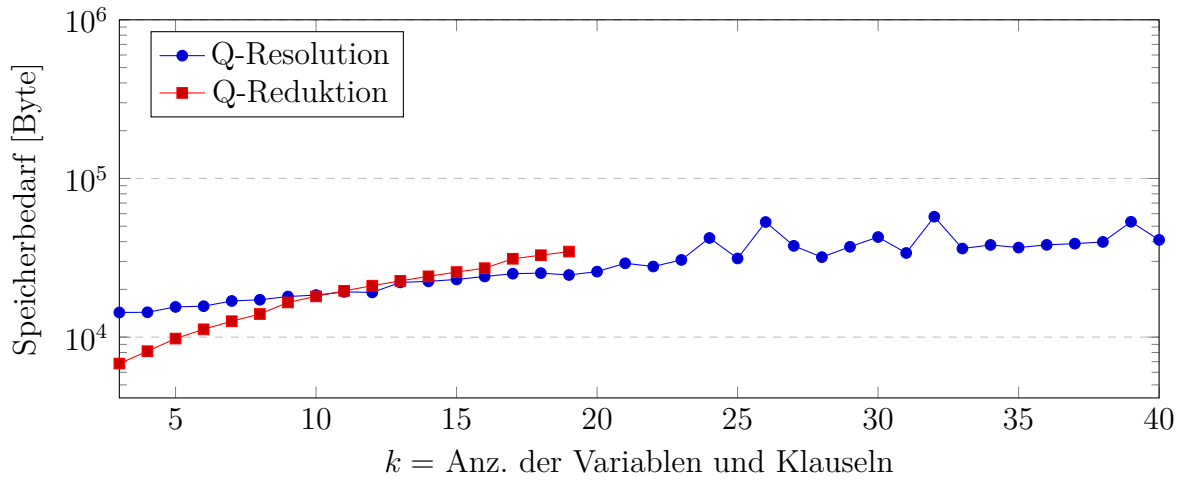


Graph 5.10: Maximale und minimale Laufzeit bei Q3-KNF

Die horizontale Achse zeigt die Variablen- und Klauselzahl, die vertikale die Laufzeit in Sekunden. Der Anteil erfüllbarer Klauseln lag bei der Testmenge bei ziemlich genau 20%. Wobei dieser Anteil bei 3 Variablen und Klauseln bei 76% liegt und im Verlauf immer geringer wird. Für Formeln mit 37-40 Variablen und Klauseln liegt er bei 0%. Erneut zeigt sich, dass steigende Klauselzahlen die Wahrscheinlichkeit auf Erfüllbarkeit stark verringern.
Die Laufzeit des Algorithmus zur Q-Resolution ist im Allgemeinen geringer, als die der Q-Resolution. Die durchschnittliche Laufzeit bleibt unter einer Sekunde und während die Q-Reduktion ab 19 Variablen und Klauseln stets einen Timeout verursacht, gab es mit der Q-Resolution auch bei höherer Komplexität nur vereinzelte Timeouts. Doch das liegt wohl daran, dass mit steigender Komplexität kaum noch erfüllbare Formeln auftraten

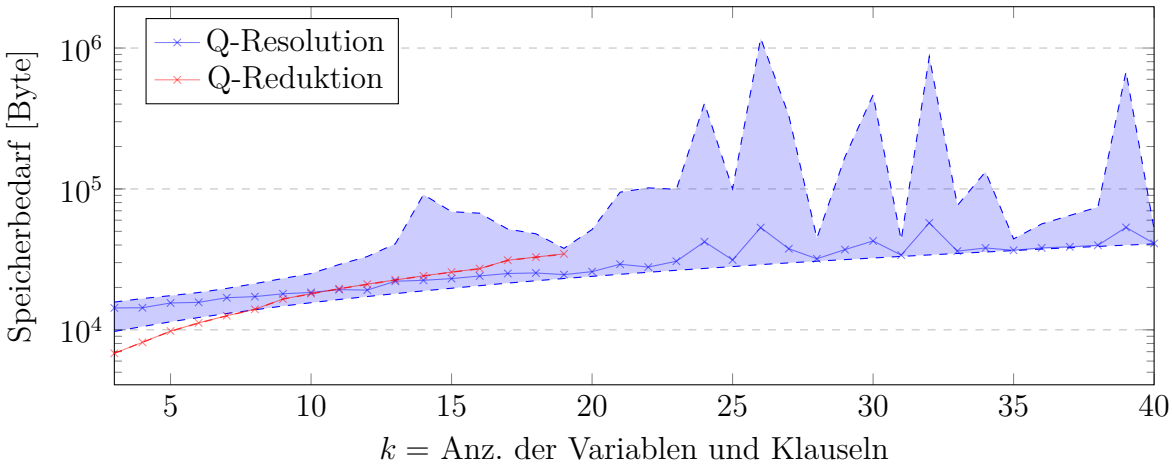
und viele Formeln direkt durch ∀-Klauseln widerlegt werden können.
Die Varianz der Laufzeit ist für die Q-Resolution erheblich größer. Bei 22 Variablen und Klauseln wurden von der Q-Resolution aus 50 Formeln nur eine als erfüllbar identifiziert und eine in 20 Sekunden nicht gelöst. Die Laufzeit zur erfüllbaren Formel betrug 0.1951 Sekunden. Das ist immerhin doppelt so lang, wie bei den 48 unerfüllbaren Formeln. Die ungelöste Formel findet sich im Anhang 6.5. Ein gesonderter Testlauf für diese Formel ergab, dass sie erfüllbar ist. Die Laufzeit betrug etwa 3 Minuten (Median: 183 Sekunden).
Die Laufzeit bei der Q-Reduktion hängt direkt mit der Größe des Problems zusammen und weist auch nur geringe Varianz für Formeln gleicher Größe auf. Die Laufzeit der Q-Resolution verhält sich komplett anders.
Die minimale Laufzeit bleibt bei wachsenden Formeln beinahe konstant. Maximalwerte wachsen früh und führen zu vereinzelten Spitzen. Anscheinend treten nicht genügend Formeln auf, welche diese Spitzen verursachen. Ein Zusammenhang zwischen hohen Laufzeiten und erfüllbaren Formeln konnte hergestellt werden.

Speicher



Graph 5.11: Durchschnittlicher Speicherbedarf bei Q3-KNF

Die horizontale Achse zeigt die Variablen- und Klauselzahl. Die vertikale den Speicherbedarf in Byte. Die Testmenge ist die von Graph 5.9.
Beim Speicherbedarf scheint die Q-Reduktion erstmal besser abzuschneiden, doch nimmt ihr durchschnittlicher Speicherbedarf schneller zu als bei der Q-Resolution. Bei 11 Klauseln und Variablen überschreitet er den der Q-Resolution.
Allerdings liegt der maximale Speicherbedarf der Q-Resolution stets höher. Auch hier zeigt die Q-Reduktion beinahe keine und die Q-Resolution große Varianz. Dieselben Spitzen treten auf mit Ausnahme der bei $x = 22$, da kein Messwert zum Speicherbedarf für die ungelöste Formel vorliegt.
Die erfüllbaren Formeln scheinen im Durchschnitt einen höheren Speicherbedarf zu



Graph 5.12: Maximaler und minimaler Speicherbedarf bei Q3-KNF

verursachen. Da für eine erfüllbare Formel zunächst alle möglichen Resolutionsschritte vollzogen werden, macht es Sinn, dass hierzu höherer Zeit- und Speicheraufwand vonnöten ist. Der Speicheraufwand der Q-Resolution ist in so einem Worst Case und im Rahmen der Beobachtungen tatsächlich höher als bei der Q-Reduktion.

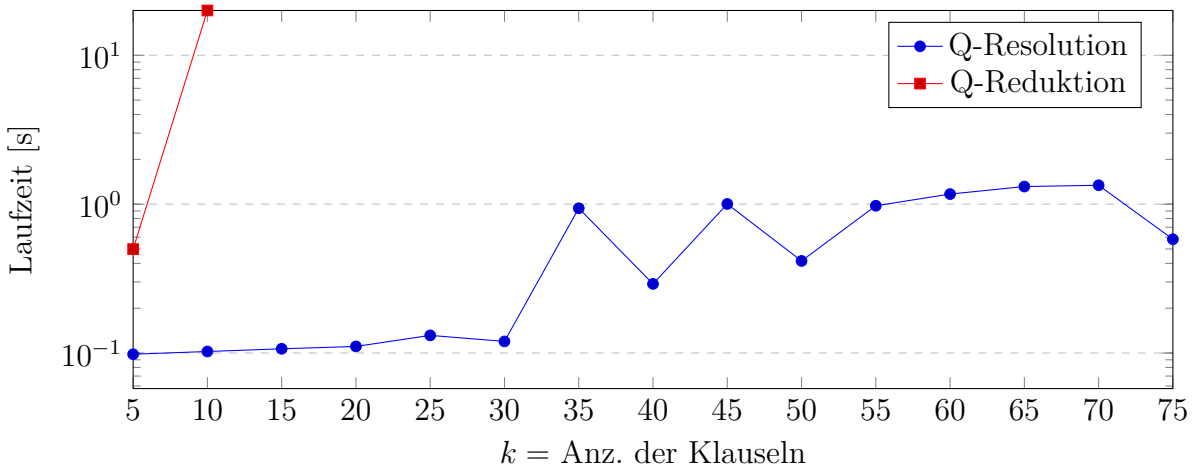
Da in Testlauf 1 die Zahl der unerfüllbaren Formeln erheblich höher war als die der erfüllbaren, wird in weiteren Testläufen versucht, die Parameter anzupassen, sodass mehr erfüllbare Formeln entstehen. Für den zweiten Testlauf wird die Anzahl der Variablen und die Länge der Klauseln erhöht. Beobachtungen aus Graph 5.2 zeigten, dass bei höherer Klauselläng mehr erfüllbare Formeln entstehen. Zudem soll untersucht werden, ob und welchen Einfluss die Anzahl der Variablen auf die Messdaten hat.

Testlauf 2

Parameter	Wert
Anz. Variablen	$3 * k$
Anz. Quantifizierungen	$3 * k$
Anz. Klauseln	k
Klauselläng	5

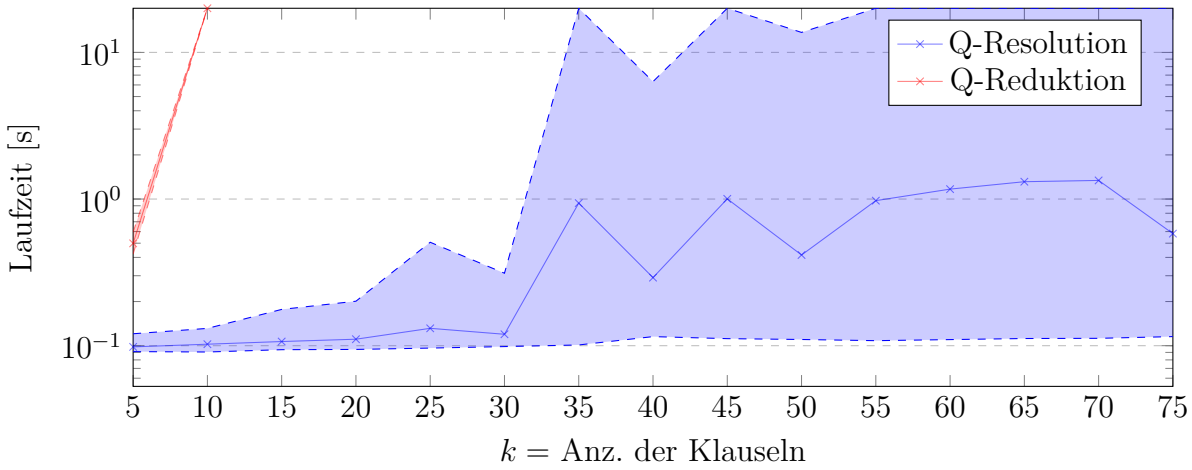
Laufzeit

Auffällig ist bei Graph 5.13, dass die Q-Reduktion bereits bei 10 Klauseln jedes Mal zum Timeout führt. Der Formelumfang ist mit 10 Klauseln der Länge 5 (50 Literale) nicht größer, als beim vorherigen Testlauf bei 18 Klauseln der Länge 3 (54 Literale).



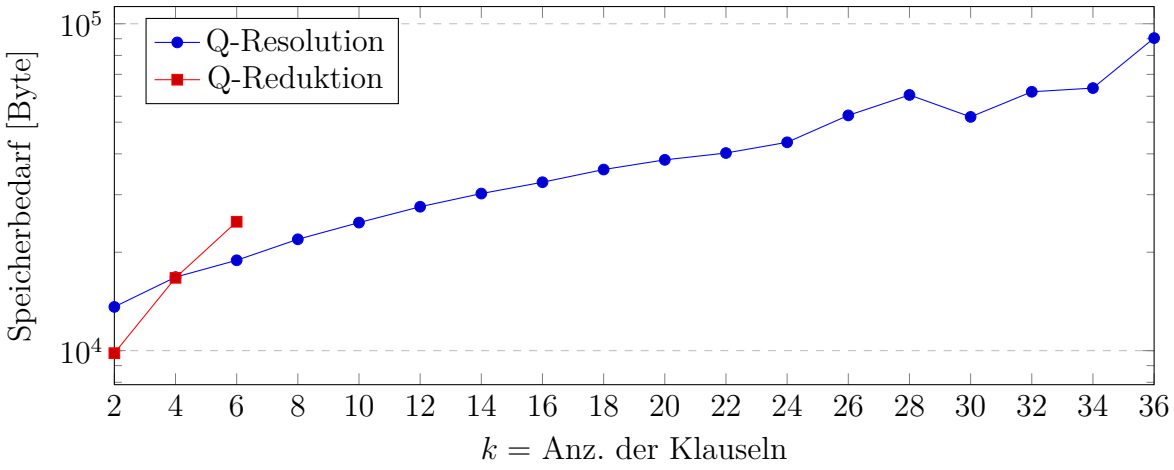
Graph 5.13: Durchschnittliche Laufzeit bei Q5-KNF

Dies liegt an der größeren Anzahl an Variablen. Die Q-Reduktion macht 2^{nbvar} rekursive Aufrufe. Auch wenn die Dauer der Aufrufe nicht zunimmt, entsteht durch die höhere Variablenzahl eine erhebliche Zunahme der Laufzeit.



Graph 5.14: Minimale und maximale Laufzeit bei Q5-KNF

Speicher

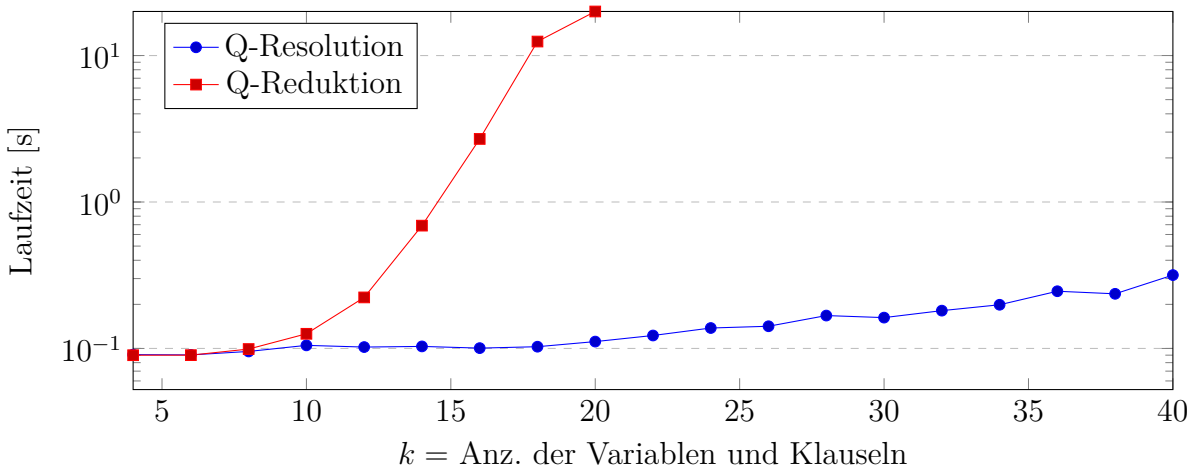


Graph 5.15: Durchschnittlicher Speicherbedarf QBF-Resolution und -Reduktion

Nach Beobachtungen aus Graphen 5.2 und 5.4 erhöhen längere Klauseln und freie Variablen die Wahrscheinlichkeit auf erfüllbare Formeln, während sie mit steigender Klauselzahl abnimmt.
Um die Komplexität der Formeln zunehmend zu steigern, wurden bisher Klausel- und Variablenzahl erhöht, wobei die Klauselzahl den größeren Einfluss auf die Komplexität hat. Es stellte sich als Herausforderung dar, mit Steigerung der Komplexität einen ähnlich bleibenden Anteil an erfüllbaren Formeln zu erzeugen. Für den nächsten Testlauf wird die Klausellänge daher nicht festgelegt, sondern wie in Algorithmus 4.2 beschrieben für jede Klausel zufällig aus $\{1 \dots nbVar\}$ gewählt.

Testlauf 3

Parameter	Wert
Anz. Variablen	k
Anz. Quantifizierungen	k
Anz. Klauseln	k
Klausellänge	Zufällig $\leq k$ (für jede Klausel)

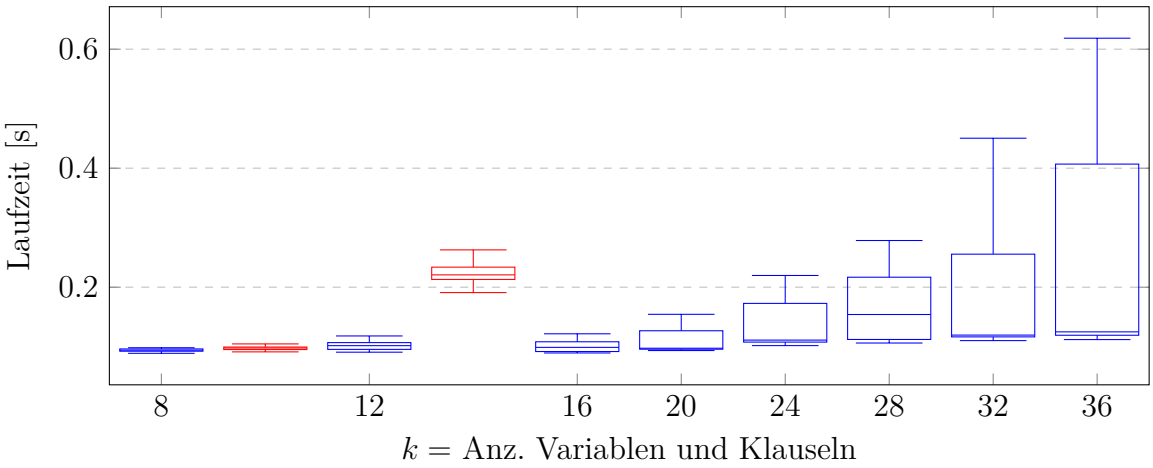


Graph 5.16: Laufzeitvergleich von QBF-Resolution und -Reduktion für Formeln mit zufälliger Klauselläng (begrenzt durch Variablenzahl)

Betrachtungen von Durchschnittswerten vernachlässigen Aspekte, die hier nicht außer Acht gelassen werden sollen. So könnte z.B. ein Algorithmus, der extrem geringe Laufzeiten bei unerfüllbaren Formeln aufweist, eventuell höhere Laufzeiten bei erfüllbaren kaschieren. Um einen guten Überblick über die Streuung der Werte zu gewinnen, werden BoxPlots gezeichnet. Es werden Messdaten aus demselben Testlauf verwendet wie bereits in Graph 5.16. Von den Formeln in dieser Testmenge waren etwa 35% erfüllbar.

Der Graph 5.17 wurde bewusst nicht mit einer logarithmischen vertikalen Achse versehen, um die Streuung der Werte im tatsächlichen Maßstab zu zeigen. Für die Q-Reduktion sind in rot Werte für $k = 8$ und $k = 12$ repräsentiert. Für die Q-Resolution sind in blau Werte von $k = 8$ bis $k = 36$ in Viererschritten vertreten. Jede Boxdarstellung zeigt mit den Antennen Maximal- und Minimalwert und mit den Boxen oberes und unteres Quartil sowie dazwischen den Median. Diese begrenzen vier Abschnitte, in welche jeweils 25% der Messwerte fallen. Diese Grenzen wurden aus 50 Werten zu diesen Parametern bestimmt.

Zunächst zeigt sich für die Q-Reduktion, wie bereits in vorherigen Betrachtungen zu Minimal- und Maximalwerten, dass die Werte eine geringe Streuung aufweisen. Dabei liegen das obere und untere Quartil besonders eng zusammen.
Wie lässt sich die - nun in mehreren Messungen festgestellte - geringe Spannweite erklären? Der Q-Reduktion Algorithmus folgt für jede Formel mit den gleichen Parametern demselben Ablaufmuster. Für jede Variable gibt es zwei rekursive Aufrufe und der Ablauf der Aufrufe weist nur geringe Unterschiede auf.
Der Algorithmus zur Q-Resolution hingegen kann für zwei Formeln mit den gleichen Parametern komplett verschieden funktionieren (siehe Abschnitt 5). Auswirkungen auf die Laufzeit und den Speicherbedarf wurden bereits nachgewiesen (siehe Anhang 6.3 und 6.4). Eine große Spannweite war daher zu erwarten. Interessant ist aber vor allem auch, dass der Median häufig sehr nah am unteren Quartil und zudem das untere Quartil sehr



Graph 5.17: Laufzeitverteilung von Q-Resolution (blau) und Q-Reduktion (rot)

nah am Minimum liegt. Die oberen 50% sind sehr weit gestreut und der Abstand von Median zu Maximum nimmt mit der Zeit stark zu. Diese großen Abstände entstehen dadurch, dass für viele unerfüllbare Formeln schnelle Widerlegungen gefunden werden können und so die minimale Laufzeit bei steigender Laufzeit beinahe konstant bleibt. Für erfüllbare Formeln und schwer widerlegbare Formeln nimmt die Laufzeit mit steigender Komplexität jedoch erheblich zu.

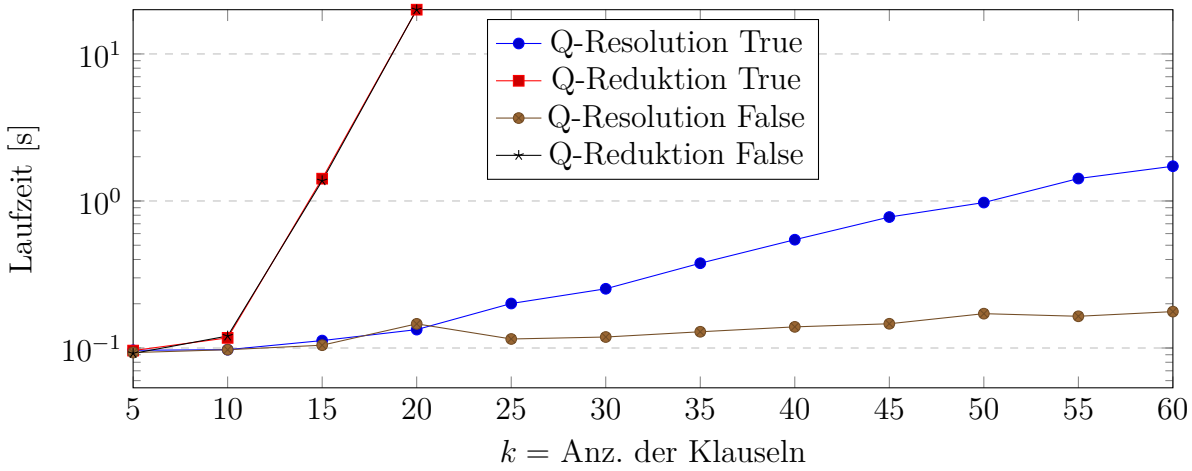
Schwierigkeit erfüllbarer und unerfüllbarer Formeln

Viele Erklärungen der bisherigen Ergebnisse stützen sich auf folgende Argumentation: Es existieren verschiedene Formeln mit gleichen Parametern, die für Algorithmen, speziell die Q-Resolution, schwerer bzw. leichter zu lösen sind. **Erfüllbare Formeln sind im Allgemeinen schwerer zu lösen als unerfüllbare.** Auch wenn es leichtere und schwerere unerfüllbare Formeln gibt, steht die Hypothese, dass die erfüllbaren generell zu den schwereren gehören.
Nun soll dies mit empirischen Daten untermauert werden. Hierzu werden zwei gefilterte Testmengen gegenüber gestellt und mit den Algorithmen zum Lösen von `QBF` bearbeitet. Der durchschnittliche Laufzeit- und Speicherbedarf der Algorithmen für die Testmengen wird verglichen. Eine Testmenge besteht ausschließlich aus erfüllbaren und eine aus unerfüllbaren Formeln. Beide wurden mit den selben Formelparametern erzeugt.
Das Augenmerk liegt auf möglichen Unterschieden bei den Messwerten für einen Algorithmus bei verschiedenen Testmengen. Nach bisherigen Erwartungen sollte die Q-Resolution für unerfüllbare Formeln schneller arbeiten und weniger Speicher benötigen.

Testlauf 4

Parameter	Wert
Anz. Variablen	k
Anz. Quantifizierungen	k
Anz. Klauseln	k
Klausellänge	Zufällig $\leq k$ (für jede Klausel)

Laufzeit

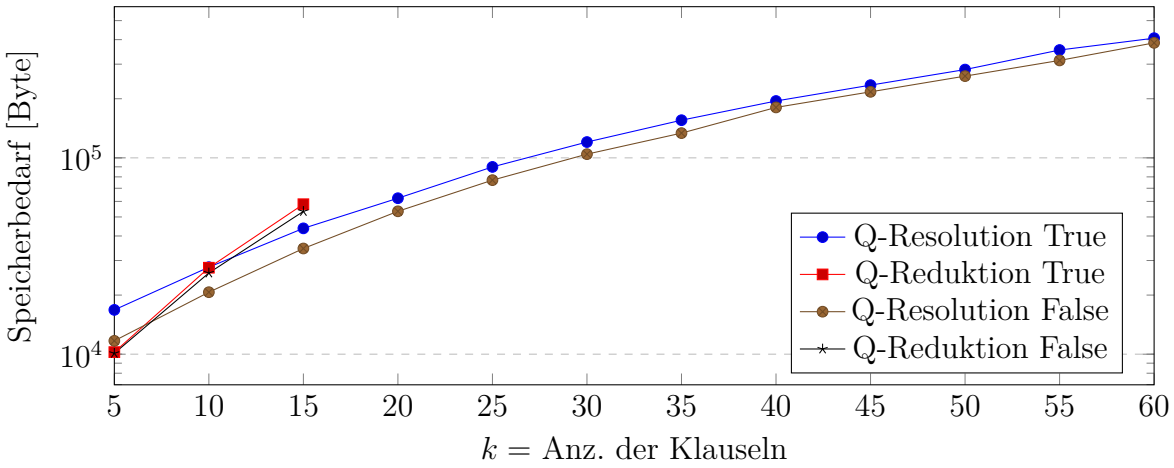


Graph 5.18: Durchschnittliche Laufzeit erfüllbarer und unerfüllbarer QBF-Formeln

Graph 5.18 bestätigt die aufgestellte Hypothese. Die Q-Resolution arbeitet für unerfüllbare Formeln erheblich schneller. Die Laufzeit für diese Formeln nimmt auch bei zunehmend größeren Formeln kaum zu. Für erfüllbare Formeln nimmt sie stetig zu.
Für die Q-Reduktion ist die Laufzeit für erfüllbare und unerfüllbare Formeln nicht unterschiedlich groß.

Speicher

Betrachtet wird nun Graph 5.19. Beim Speicherbedarf der Q-Resolution fällt der Unterschied zwischen erfüllbaren und unerfüllbaren Formeln nicht so groß aus. Doch er ist vorhanden. Aufgrund der Implementierung, bei der beim Einlesen der Formeln sämtliche Datenstrukturen erzeugt werden - selbst wenn der Widerlegungsbeweis trivial ist - fällt der Unterschied nicht so groß aus. Sind die Datenstrukturen allokiert, wird im Weiteren nicht mehr so viel neuer benötigt.



Graph 5.19: Durchschnittlicher Speicherbedarf erfüllbarer und unerfüllbarer QBF-Formeln

Heuristik

Der Algorithmus zur Q-Resolution muss entscheiden, über welchem Literal wann der Resolutionsschritt ausgeführt wird (siehe Algorithmus 4.7, Zeile 4). Die Heuristikmethode, die in der Implementation zu dieser Arbeit angewandt wurde, soll im Folgenden erläutert werden. Außerdem werden Alternativen in Betracht gezogen.

Bevor der Algorithmus das nächste Literal wählen kann, muss er zunächst Informationen zur Formel sammeln. Die Formel wird durchlaufen und die Häufigkeit, mit der alle existenzquantifizierten Atome auftreten, gezählt. Dabei werden positives und negatives Auftreten getrennt gezählt. Tritt ein Atom sowohl positiv als auch negativ auf, wird es mit der addierten Häufigkeit vermerkt. Es wird eine Liste gebildet, von größter Häufigkeit zur kleinsten Häufigkeit, welche noch einen Resolutionsschritt ermöglicht. Diese Liste wird Schritt für Schritt abgearbeitet und jeweils der Resolutionsschritt über dem Atom ausgeführt. Dies geschieht bis die Formel widerlegt ist oder keine Resolutionsschritte mehr möglich sind.

Es sollen einige Möglichkeiten zum alternativen Vorgehen genannt werden:

1. *Das Produkt positiver und negativer Vorkommen eines Atoms betrachten*, statt der Summe. Die Summe repräsentiert die Anzahl der Elternklauseln für den zugehörigen Resolutionsschritt. Das Produkt stellt die Anzahl möglicher Resolventen dar und repräsentiert damit besser die Komplexität des Resolutionsschritt sowie möglicherweise auch die Chance, dass in dem Resolutionsschritt ein Widerspruch herleitbar ist.

2. *Das Atom mit der geringsten Häufigkeit zuerst bearbeiten* oder ein zufälliges Atom zuerst bearbeiten. Anstatt zunächst den Resolutionsschritt für das Atom mit der größten Häufigkeit durchzuführen, könnte man mit dem Atom mit der geringsten beginnen. Dies bedeutet zunächst weniger viele Resolventen. Also schnellere Resolutionsschritte, jedoch könnte es weniger wahrscheinlich sein, in diesen Schritten einen Widerspruch herzuleiten.

3. *Weitere Informationen sammeln* und die Auswahl des nächsten Resolutionsschrittes auf diese stützen. Beispielsweise könnten kürzere Klauseln oder Klauseln mit vielen allquantifizierten Variablen über ein größeres Potenzial als Elternklauseln für Resolutionswidersprüche verfügen. Dementsprechend könnte man nach Variablen suchen, die in solchen Klauseln enthalten und für einen Resolutionsschritt qualifiziert sind.

Dies sind natürlich nur einige wenige Beispiele aus einer Vielzahl von Möglichkeiten und viele werden hier nicht erwähnt. Doch sollen diese Beispiele andeuten, wie viele Möglichkeiten zur Optimierung von Solvern vorhanden sind. An dieser Stelle soll zumindest eine Veränderung implementiert und verglichen werden.

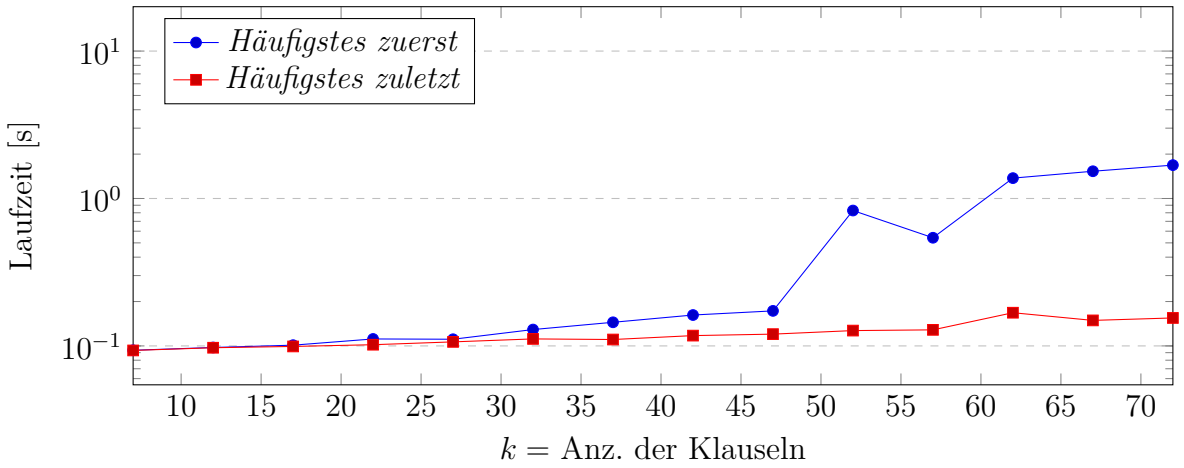
Testlauf 5

Parameter	Wert
Anz. Variablen	$3 * k$
Anz. Quantifizierungen	$3 * k$
Anz. Klauseln	k
Klausellänge	5

Dieses Mal sollen zwei Varianten der Q-Resolution ausgeführt werden. Eine läuft mit der bisherigen und eine mit einer neuen Heuristik. Die Erste bearbeitet stets zuerst das Atom mit der größten Häufigkeit („Häufigstes zuerst“), während bei der Neuen zuerst das Atom mit der geringsten Häufigkeit bearbeitet wird („Häufigstes zuletzt“). Anschließend sollen Laufzeiten und Speicherbedarf beider Varianten verglichen werden. Die Testmenge enthält knapp 40% erfüllbare Formeln.

Laufzeit

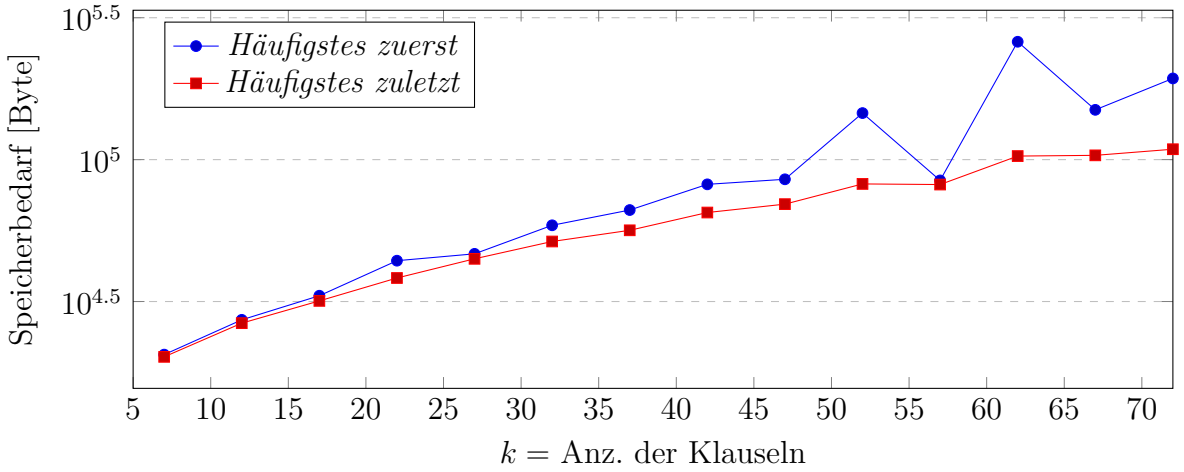
Bei Betrachtung von Graph 5.20 scheint diese kleine Änderung tatsächlich bereits einen deutlichen Unterschied für die Laufzeit zu verursachen. Die Laufzeit für die Variante *Häufigstes zuerst* nimmt stärker zu, als für die Variante *Häufigstes zuletzt*.



Graph 5.20: Durchschnittliche Laufzeit von zwei Varianten der Q-Resolution

Die kleineren, schnelleren Resolutionsschritte erwiesen sich in diesem Testlauf als effektiver. Zudem löste die *Häufigstes zuletzt*-Variante sogar Formeln, bei denen *Häufigstes zuerst* durch Timeout abgebrochen wurde.

Speicher



Graph 5.21: Durchschnittlicher Speicherbedarf von zwei Varianten der Q-Resolution

Graph 5.21 zeigt ein ähnliches Bild. Die Variante *Häufigstes zuletzt* scheint die bessere Variante hinsichtlich sowohl Laufzeit als auch Speicherbedarf zu sein.

Ergebnisse

Abschließend sollen eine Reihe von Ergebnissen dieser Arbeit zusammengefasst werden. Diese Ergebnisse sind nicht außergewöhnlich oder überraschend, doch werden sie von einer Reihe an Testläufen gestützt. Einige der Erkenntnisse sind für eine Fortsetzung dieser Untersuchungen höchst nützlich.

1. *Formelparameter beeinflussen Erfüllbarkeit.* In Abschnitt 5.1 wurden Auswirkungen von Änderungen verschiedener Formelparameter, auf den Anteil generierter erfüllbarer Formeln, untersucht.

2. *Erfüllbarkeit beeinflusst Laufzeit und Speicherbedarf mancher Algorithmen.* Graph 5.18 und 5.19 zeigen, dass sowohl die Laufzeit als auch der Speicherbedarf von Algorithmen zum Lösen von erfüllbaren und unerfüllbaren Formeln eindeutige Unterschiede aufweisen können. Es wurde auch beobachtet, dass es Algorithmen gibt, die kein derartiges Verhalten aufweisen.

3. *QBF-Formeln mit kurzen Klauseln werden oft durch ∀-Klauseln unerfüllbar.* Untersuchungen zu ∀-Klauseln in Graphen 5.2 und 5.3 wiesen daraufhin, dass ∀-Klauseln ein großer Faktor, hinsichtlich der Erfüllbarkeit von QBF-Formeln, sind.

4. *Unerfüllbare QBF-Formeln mit vielen Klauseln sind schwer zu finden.* Da mit hoher Klauselzahl die Chance auf ∀-Klauseln steigt, führt der vorige Punkt dazu, dass Formeln mit vielen Klauseln selten erfüllbar sind.

5. *CryptoMiniSat ist besser als DP-Resolution ist besser als Q-Resolution für* `SAT` (siehe Abschnitt 5.2). CryptoMiniSat weist mit Abstand die geringsten Laufzeiten auf und stieß in den Testläufen gar nicht an seine Grenzen. Die Davis-Putnam-Resolution schnitt besser ab als die Q-Resolution.

6. *Q-Resolution ist besser als QBF-Reduktion für* `QBF` (siehe Abschnitt 5.3). Der Algorithmus zur Q-Resolution war vor allem mit steigender Formelgröße deutlich schneller, als der zur Q-Reduktion. Für sehr kleine Formeln war der Speicherbedarf der Q-Reduktion geringer, ansonsten lag der durchschnittliche Speicherbedarf der Q-Resolution jedoch auch niedriger.

7. *Die durchschnittliche Laufzeit von Resolutionsalgorithmen steigt unter Umständen nicht mit zunehmender Formelgröße, falls parallel die Erfüllbarkeit abnimmt.* In Graph 5.9 blieb die Laufzeit der Q-Resolution beinahe konstant, da die Testmenge mit steigender Formelgröße immer weniger nicht trivial unerfüllbare und erfüllbare Formeln enthielt.

8. *Laufzeit und Speicherbedarf von Resolutionsverfahren sind relativ stark gestreut.* Im Vergleich zu der Q-Reduktion, wies die Q-Resolution eine hohe Spannweite an Werten auf. Die Differenz von Maximal- und Minimalwerten war erheblich größer (siehe z.B. Graph 5.13).

9. *Erfüllbare Formeln sind für die Q-Resolution im Durchschnitt aufwendiger zu Lösen als unerfüllbare.* In Graph 5.18 und 5.19 zeigte sich, dass der durchschnittliche Zeit- und Speicheraufwand zum Lösen von erfüllbaren Formeln größer war. Dies bekräftigte Schlüsse aus vorausgegangenen Testläufen sowie näheren Betrachtungen von Einzelfällen (siehe Anhang 6.3 und 6.4).

10. *Eine kleine Änderung der Heuristik kann einen großen Effekt haben - „Häufigstes zuletzt“ ist besser als „Häufigstes zuerst“.* Im letzten Testlauf zu `QBF` zeigte sich, dass von den zwei getesteten Heuristiken eine messbar besser abschnitt. Die Variante *Häufigstes zuletzt* erwies sich als schneller und sparsamer an Speicherbedarf als die Variante *Häufigstes zuerst*.

6 Fazit

Der Bereich der SAT- und QBF-Solver ist ein großes Forschungs- und Anwendungsfeld, welches voller Gelegenheiten und Herausforderungen steckt. Diese Arbeit hat nicht mehr als an der Oberfläche gekratzt. Die Frage, ob `P` = `NP` ist, gewinnt im 21. Jahrhundert nach wie vor an Aufmerksamkeit und Relevanz.
Aktuell existiert ein Algorithmus zu `Unique 3-SAT` mit oberer Laufzeitschranke $\mathcal{O} * 1,306995^n$ (HKZZ19). Algorithmen nähern sich also einer effizienten Lösung zu `SAT` an, aber ob diese tatsächlich erreichbar ist, bleibt unklar. Bis heute existiert kein im Sinne der Theorie effizienter Algorithmus zu `SAT` oder `QBF`.
Die Fortschritte der letzten Jahrzehnte sind groß und lassen den praktischen Nutzen von SAT- und QBF-Solvern wachsen. Man kann wohl zumindest ein wenig davon träumen, dass auf diesem Wege eine Antwort auf die große Frage gefunden werden könnte.

Von Logikoperatoren und Normalformen bis zur Q-Resolution haben wir in dieser Arbeit grundlegende Verfahren zum Lösen von `SAT` und `QBF` kennengelernt. Mit den Algorithmen zur Generierung von Formeln, der Davis-Putnam-Resolution, QBF-Reduktion und der Q-Resolution, besteht ein komplettes System zum Erzeugen und Lösen von KNF- und QBF-Problemen.
Dabei können Laufzeit und Speicherbedarf der Algorithmen betrachtet werden. Die Ergebnisse werden in ausführlichen Logdateien exportiert. Durch die so durchgeführten Messungen konnten eine Reihe von Beobachtungen angestellt und durch Daten gestützt werden.
Die Implementierung der eigenen Algorithmen ist leider für mich noch am wenigsten befriedigend. Benutzte Datenstrukturen und Verfahren sind suboptimal. Jedoch besteht ein Gerüst, in dem ich neue Algorithmen implementieren und direkt in Vergleich zu den bisherigen Implementationen stellen kann. So konnte gut - aber spät - festgestellt werden, dass die Heuristik zur Q-Resolution leicht verbesserbar war.

Die meisten der Ergebnisse waren im Vorhinein zu erwarten; trotzdem war es für mich eine willkommene Herausforderung, meine Vermutungen in Frage zu stellen, zu untersuchen und zu belegen. Die Algorithmen in Python zu implementieren war im Vergleich zum Durchführen, Sammeln und Auswerten der Vielzahl an Testläufen definitiv der einfache Teil. Vor allem die hohe Zeitanforderung der Testläufe war erdrückend. Nicht wenige Tests liefen mehrere Stunden und erwiesen sich schließlich als fehlerhaft - menschliches Versagen.
Am meisten verwundert hat mich die Herausforderung schwere bzw. große erfüllbare

Formeln zu erzeugen. Interessant sind in diesem Zusammenhang Verfahren zum systematischen Erzeugen von Formeln z.B. durch Nutzung mathematischer Methoden. Aus diesem Grund war es auch nicht leicht, die Ergebnisse der Testläufe richtig und schlüssig zu erklären.
Sehr spannend für mich war die Entdeckung einer Formel, deren Parameter keine große Komplexität vermuten ließen, die jedoch der Q-Resolution satte drei Minuten Rechenarbeit abverlangte und sogar durch die QBF-Reduktion schneller gelöst wurde. Ich finde es faszinierend, wie unterschiedlich die Schwierigkeit von Formeln ausfallen können und welche Anpassungsmaßnahmen von Solvern dies nach sich zieht. Ich hoffe, dieses Projekt mit der Zeit weiter zu entwickeln, weitere Algorithmen hinzuzufügen und zu verbessern.

Anhang

```
p cnf 25 20
a 1 2 3 4 5 0
e 6 7 0
a 8 0
e 9 0
a 10 11 12 0
e 13 14 0
a 15 16 0
e 17 18 19 20 21 22 0
a 23 24 25 0
3 7 -8 9 -10 -13 -15 16 -17 -18 -19 -20 21 22 23 -24 0
-1 -2 -3 -4 5 -6 7 8 -9 10 -11 -13 -15 16 -17 -18 19 -20 21 22 23 -24 25 0
3 -10 -13 14 15 17 19 21 23 24 0
1 -2 -3 -7 10 -11 12 13 -14 15 16 -20 -21 22 24 0
-4 5 -6 -10 -15 -19 24 -25 0
-7 13 0
-9 14 15 0
25 0
15 0
-2 3 -4 -5 -8 -9 -13 -15 -16 -19 20 -23 0
1 -4 -6 -7 -11 -12 -14 -16 19 -23 24 0
-1 -2 -3 -5 -6 -7 -9 11 12 13 -14 15 -16 17 -18 -19 -21 -22 24 25 0
1 4 -6 -7 -9 10 -12 -14 16 17 19 -25 0
-3 -4 5 -6 -8 9 14 -17 -18 20 22 23 0
-14 15 17 -20 -22 0
1 -5 8 -9 -13 -16 19 0
2 -7 -12 -13 22 0
1 -4 6 7 8 9 -10 -11 -12 -13 -18 19 20 21 -23 24 0
-1 -2 3 -4 -5 6 -7 -8 10 -11 12 -14 -15 -16 -17 -18 19 21 -22 -23 -24 -25 0
1 -2 -4 5 -6 8 -9 10 11 -12 -13 -14 15 16 17 -18 -19 -20 21 22 -23 24 25 0
```

Anhang 6.1: QKNF-Formel mit ∀-Klausel

```
p cnf 25 20
a 1 2 3 4 5 0
e 6 7 0
a 8 0
e 9 0
a 10 11 12 0
e 13 14 15 0
a 16 0
e 17 18 19 20 21 22 0
a 23 24 0
e 25 0
3 7 -8 9 -10 -13 -15 16 -17 -18 -19 -20 21 22 23 -24 0
-1 -2 -3 -4 5 -6 7 8 -9 10 -11 -13 -15 16 -17 -18 19 -20 21 22 23 -24 25 0
3 -10 -13 14 15 17 19 21 23 24 0
1 -2 -3 -7 10 -11 12 13 -14 15 16 -20 -21 22 24 0
-4 5 -6 -10 -15 -19 24 -25 0
-7 13 0
-9 14 15 0
25 0
15 0
-2 3 -4 -5 -8 -9 -13 -15 -16 -19 20 -23 0
1 -4 -6 -7 -11 -12 -14 -16 19 -23 24 0
-1 -2 -3 -5 -6 -7 -9 11 12 13 -14 15 -16 17 -18 -19 -21 -22 24 25 0
1 4 -6 -7 -9 10 -12 -14 16 17 19 -25 0
-3 -4 5 -6 -8 9 14 -17 -18 20 22 23 0
-14 15 17 -20 -22 0
1 -5 8 -9 -13 -16 19 0
2 -7 -12 -13 22 0
1 -4 6 7 8 9 -10 -11 -12 -13 -18 19 20 21 -23 24 0
-1 -2 3 -4 -5 6 -7 -8 10 -11 12 -14 -15 -16 -17 -18 19 21 -22 -23 -24 -25 0
1 -2 -4 5 -6 8 -9 10 11 -12 -13 -14 15 16 17 -18 -19 -20 21 22 -23 24 25 0
```

Anhang 6.2: QKNF-Formeln mit geänderter Quantifizierung ohne ∀-Klausel

Algorithm	**Q-Resolution**
Results	
All Results equal	True
Result	False
Time [s]	
Min	0.14613700000000007
Max	0.1708031000000001
Median	0.15552500000000002
Memory [Byte]	
Min	56173
Max	56173

Anhang 6.3: Laufzeit und Speicherbedarf der Q-Resolution für 6.1

Algorithm	**Q-Resolution**
Results	
All Results equal	True
Result	True
Time [s]	
Min	0.19506599999999974
Max	0.20974009999999943
Median	0.20449404999999965
Memory [Byte]	
Min	67620
Max	67620

Anhang 6.4: Laufzeit und Speicherbedarf der Q-Resolution für 6.2

```
p cnf 22 22
a 1 0
e 2 3 0
a 4 0
e 5 0
a 6 0
e 7 8 9 10 11 12 13 14 0
a 15 0
e 16 17 0
a 18 0
e 19 0
a 20 0
e 21 0
a 22 0
-7 14 16 0
-7 13 22 0
 11 15 21 0
 8 -13 -17 0
-3 5 12 0
 9 14 16 0
 6 11 -18 0
-8 9 10 0
 3 -12 21 0
 4 -9 -11 0
-2 7 -10 0
-8 -13 22 0
-8 18 -20 0
-12 18 -19 0
-1 15 -19 0
-5 8 -10 0
-1 -12 19 0
-13 -14 -16 0
 7 12 -14 0
-5 -14 -15 0
-10 -12 13 0
-7 13 -21 0
```

Anhang 6.5: Außergewöhnliche schwer lösbare QKNF-Formel

Algorithm	**Q-Resolution**
Results	
All Results equal	True
Result	True
Time [s]	
Min	169.1624951
Max	192.13102890000005
Median	183.48329515
Memory [Byte]	
Min	2273255
Max	2273255

Algorithm	**Q-Reduktion**
Results	
All Results equal	True
Result	True
Time [s]	
Min	141.09467940000013
Max	150.85510569999997
Median	143.79161965000003
Memory [Byte]	
Min	39592
Max	39592

Anhang 6.6: Laufzeit und Speicherbedarf der Q-Resolution und -Reduktion für 6.5

Betriebssystem	Windows 10
Programmiersprache	Python 3.7.2
Prozessor	Quad Core @ 3,5GHz
Arbeitsspeicher	16GB DDR3-1600MHz RAM
Festplatte	SSD Lese-/Schreibgeschw.: ca. 500 MB/s

Anhang 6.7: Testumgebung

Literaturverzeichnis

[ART20] Amendola, Giovanni ; Ricca, Francesco ; Truszczynski, Miroslaw: New models for generating hard random boolean formulas and disjunctive logic programs. In: *Artificial Intelligence* 279 (2020), S. 103185. `http://dx.doi.org/https://doi.org/10.1016/j.artint.2019.103185`. – DOI https://doi.org/10.1016/j.artint.2019.103185. – ISSN 0004–3702

[BGG+15] Biere, Armin ; Ganesh, Vijay ; Grohe, Martin ; Nordström, Jakob ; Williams, Ryan: Theory and Practice of SAT Solving (Dagstuhl Seminar 15171). In: *Dagstuhl Reports* 5 (2015), Nr. 4, 98–122. `http://dx.doi.org/10.4230/DagRep.5.4.98`. – DOI 10.4230/DagRep.5.4.98

[Bie09] Biere, A.: *Handbook of Satisfiability*. IOS Press, 2009 (Frontiers in artificial intelligence and applications). `https://books.google.de/books?id=YVSM3sxhBhcC`. – ISBN 9781586039295

[Com09] Competition, SAT: *SAT Competition 2009: Benchmark Submission Guidelines*. `http://www.satcompetition.org/2009/format-benchmarks2009.html`. Version: 2009

[GNPT05] Giunchiglia, E. ; Narizzano, M. ; Pulina, L. ; Tacchella, A.: *Quantified Boolean Formulas satisfiability library (QBFLIB)*. 2005. – www.qbflib.org

[HIK+20] Hirsch, Edward ; Itsykson, Dmitry ; Kojevnikov, Arist ; Kulikov, Alexander ; Nikolenko, Sergey: *Report on the Mixed Boolean-Algebraic Solver*. 2020. – https://logic.pdmi.ras.ru/~basolver/dimacs.html

[HJKN06] Haanpää, Harri ; Järvisalo, Matti ; Kaski, Petteri ; Niemelä, Ilkka: Hard Satisfiable Clause Sets for Benchmarking Equivalence Reasoning Techniques. In: *JSAT* 2 (2006), 03, S. 27–46. `http://dx.doi.org/10.3233/SAT190015`. – DOI 10.3233/SAT190015

[HKZZ19] Hansen, Thomas D. ; Kaplan, Haim ; Zamir, Or ; Zwick, Uri: Faster K-SAT Algorithms Using Biased-PPSZ. In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. New York, NY, USA : Association for Computing Machinery, 2019 (STOC 2019). – ISBN 9781450367059, 578–589

[KBL94] Kleine Büning, Hans ; Lettmann, Theodor: *Aussagenlogik: Deduktion und Algorithmen*. Teubner Stuttgart, 1994. – ISBN 3–519–02133–1

[Mal10] Malik, Sharad: *SAT Solvers:A Condensed History*. 2010

[MV15] Meier, Arne ; Vollmer, Heribert ; Schöning, Uwe (Hrsg.): *Komplexität von Algorithmen: Mathematik für Anwendungen Band 4*. Lehmanns, 2015 (Mathematik für Anwendungen). – ISBN 9783865417664

[RH01] Rijke, Maarten de ; Heguiabehere, Juan: The random modal qbf test set, 2001

[SB05] Samulowitz, Horst ; Bacchus, Fahiem: Using SAT in QBF. In: *CP*, 2005

[SNC09] Soos, Mate ; Nohl, Karsten ; Castelluccia, Claude: Extending SAT Solvers to Cryptographic Problems. In: Kullmann, Oliver (Hrsg.): *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings* Bd. 5584, Springer, 2009 (Lecture Notes in Computer Science), 244–257

[Soo] Soos, Mate: *CryptoMiniSat*. `https://github.com/msoos/cryptominisat`

[VK19] Vollmer, Heribert ; Kluge, Thorsten: *Logik und formale Systeme. Vorlesungsskript*. 2019

[Wik] Wikipedia: *3-SAT*. `https://de.wikipedia.org/wiki/3-SAT`