Bachelorarbeit

Ein Tool für Enumerationsalgorithmen

Hannover, 20.09.2015

Rebecca Veronika Cramer

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

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen genutzt habe.

Hannover, den 20.09.2015

Rebecca Veronika Cramer

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**

2.1 Aussagenlogik . . . 3
2.2 Graphentheorie . . . 4
2.3 Enumerationsprobleme . . . 5
2.4 Komplexitätsklassen . . . 6

3 Enum-KROM-SAT **8**

3.1 Entscheidungsalgorithmus für KROM-SAT . . . 8
3.2 Erweiterung auf Kromformeln mit partiellen Belegungen . . . 11
3.3 Korrektheit . . . 13
3.4 Laufzeitabschätzung . . . 14
3.5 Enumerationsalgorithmus . . . 14
3.6 Korrektheit . . . 16
3.7 Theoretische Laufzeitbetrachtung . . . 17
3.8 Praktische Laufzeitanalyse . . . 22
3.8.1 Systemvoraussetzungen . . . 22
3.8.2 Testinstanzen . . . 23
3.8.3 Ergebnisse . . . 24
3.8.4 Auswertung . . . 27

4 EnumConnComp **29**

4.1 Enumerationsalgorithmus . . . 29
4.2 Korrektheit . . . 29
4.3 Theoretische Laufzeitabschätzung . . . 30
4.4 Praktische Laufzeitanalyse . . . 31
4.4.1 Systemvoraussetzungen . . . 31
4.4.2 Testinstanzen . . . 31
4.4.3 Ergebnisse . . . 34
4.4.4 Auswertung . . . 38

5 Implementierung **39**

5.1 Allgemeines . . . 39

5.2 Zeitmessung 39
5.3 Benutzung des Programms 39
5.3.1 Kommandozeile 40
5.3.2 Graphische Oberfläche 40
5.4 Enum-KROM-SAT 41
5.4.1 Eingabe: DIMACS-Format 41
5.4.2 Ausgabe: csv-Format 42
5.4.3 Zufälliges Erzeugen von Formeln 43
5.5 EnumConnComp 43
5.5.1 Eingabeformat 43
5.5.2 Ausgabe: csv-Format 44
5.5.3 Zufälliges Erzeugen von Graphen 44
5.6 Berechnung der Regressionsfunktion 45
5.7 Einen neuen Algorithmus hinzufügen 45

6 Zusammenfassung **48**

7 Ausblick **49**

Literatur **50**

Abbildungsverzeichnis **52**

Tabellenverzeichnis **53**

Liste der Algorithmen **54**

1 Einleitung

Probleme werden in der theoretischen Informatik oft mit Hilfe von „Sprachen“ definiert. Eine Sprache ist eine Menge an Wörtern (Zeichenketten) über einem Alphabet (einer Menge an Zeichen). Das Problem selbst kann dann „nur noch“ die Entscheidung sein, ob ein gegebenes Wort in der Sprache liegt oder nicht. Das sind die sogenannten Entscheidungsprobleme.

Ein Beispiel hierfür ist die Sprache SAT, die alle erfüllbaren aussagenlogischen Formeln enthält. Um zu entscheiden, ob ein Wort – also eine aussagenlogische Formel – in der Sprache liegt, muss überprüft werden, ob es eine erfüllende Belegung für diese Formel gibt. Eine Lösung für dieses Problem wäre also eine erfüllende Belegung.

Manchmal ist jedoch nicht nur interessant, ob eine Formel erfüllbar ist oder nicht. Man möchte alle erfüllenden Belegungen finden, um beispielsweise auf später auftretende Einschränkungen der Problemstellung reagieren zu können. Dies führt zu den sogenannten Enumerationsproblemen, bei denen es darum geht, *alle* Lösungen (ohne Duplikate) zu finden. Hier reicht es – im Gegensatz zum Entscheidungsproblem – nicht mehr, auszusagen, ob es (irgend)eine Lösung gibt oder nicht.

Diese Arbeit beschäftigt sich damit, ein erweiterbares Framework für Enumerationsalgorithmen in Java zu implementieren und beispielhaft für zwei Probleme einen Vergleich der theoretischen Komplexitätsklasse mit der realen Implementierung durchzuführen. Des Weiteren sollen die implementierten Algorithmen visualisiert werden können.

Als erstes Problem wird ein Enumerationsalgorithmus für 2-SAT, also erfüllbare Formeln in konjunktiver Normalform mit genau zwei Variablen pro Klausel (2-KNF), betrachtet. Formeln in 2-KNF werden nach dem Mathematiker Melven R. Krom auch Kromformeln genannt, da er sich als einer der ersten mit dieser Art von Formeln beschäftigte und 1967 einen effizienten Entscheidungsalgorithmus für dieses Problem veröffentlichte [Kro67]. Kromformeln können unter anderem verwendet werden, um in einem Rundenturnier („jeder gegen jeden“) wie beispielsweise beim Fußball eine Stadienzuteilung zu ermitteln, sodass jede Mannschaft möglichst im Wechsel Heim- und Auswärtsspiele hat. Miyashiro und Matsui veröffentlichten hierzu 2005 einen Artikel, in welchem sie einen auf Kromformeln basierenden Polynomialzeitalgorithmus für dieses Problem angaben [MM05].

Bei dem zweiten Problem geht es darum, alle Zusammenhangskomponenten eines Graphen zu enumerieren. Dies kann zum Beispiel verwendet werden, um Erreichbarkeit in Netzplänen (Bahn, Landkarte) zu prüfen oder Freundeskreise im Graphen eines sozialen Netzwerkes zu finden.

Zu Beginn der Arbeit werden die benötigten Grundlagen und Definitionen erläutert. Es folgen zwei Kapitel zur Erklärung der beiden Enumerationsprobleme. In diesen Kapiteln wird jeweils zunächst der Algorithmus näher beschrieben sowie seine theoretische Komplexitätsklasse bestimmt. Eine Analyse der gemessenen Laufzeiten der Implementierung findet am Ende der jeweiligen Kapitel statt. Abschließend wird auf Details der Implementierung eingegangen.

2 Grundlagen

Zu Beginn der Arbeit ist es notwendig, einige Formalia zu definieren, damit die späteren Aussagen eindeutig zu verstehen sind.

2.1 Aussagenlogik

Für diese Arbeit werden die aussagenlogischen Grundlagen aus dem Buch *Graphentheorie* von W. Rautenberg [Rau09] verwendet, insbesondere werden die Begriffe *wahr*, 1 und $\top$ sowie *falsch*, 0 und $\bot$ jeweils äquivalent verwendet. Ohne Beschränkung der Allgemeinheit werden die Variablen einer aussagenlogischen Formel mit $x_1, \ldots, x_n$ ($n \in \mathbb{N}$) benannt. Es werden die in Tabelle 1 aufgeführten Junktoren verwendet.

Zeichen	Bedeutung
$\vee$	„oder“, Disjunktion
$\wedge$	„und“, Konjunktion
$\overline{x}$	„nicht“, Negation
$\rightarrow$	„wenn... dann“, Implikation

Tabelle 1: Junktoren

Weiterhin werden als Schreibweisen $\bigvee_{i=1}^{n} x_i \coloneqq x_1 \vee \ldots \vee x_n$ und $\bigwedge_{i=1}^{n} x_i \coloneqq x_1 \wedge \ldots \wedge x_n$ verwendet.

Definition 1. Als *Literal* wird eine Variable oder ihre Negation bezeichnet. Sei V die Menge aller Variablen einer aussagenlogischen Formel, dann ist $L \coloneqq \{\overline{x} \mid x \in V\} \cup V$ die Menge aller Literale.

Definition 2. Als *Klausel* wird eine durch Disjunktionen verbundene Menge von Literalen bezeichnet, z.B. $l_1 \vee l_2 \vee l_3$.

Definition 3. Eine aussagenlogische Formel φ ist dann in *konjunktiver Normalform* (KNF), wenn sie aus einer Konjunktion von Klauseln – also einer Konjunktion von Disjunktionen – besteht. Ist die Anzahl der Literale in jeder Disjunktion gleich $m \in \mathbb{N}$, kann man auch sagen, die Formel ist in m-KNF.

Beispiel 1. Eine aussagenlogische Formel φ in 3-KNF könnte zum Beispiel so aussehen:

$$\varphi \coloneqq (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee x_4 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_4). \tag{1}$$

Definition 4. Sei φ eine aussagenlogische Formel und V die Menge ihrer Variablen. Dann ist eine *Belegung* θ eine Abbildung $\theta\colon V \to \{0,1\}$, die jeder Variable einen Wahrheitswert zuordnet.

Belegungen werden wahlweise als Abbildung oder Binärzahl angegeben, abhängig davon, womit sich im Zusammenhang leichter und verständlicher arbeiten lässt. Wird eine aussagenlogische Formel φ von einer Belegung θ erfüllt – das heißt unter Einsetzen von θ zu *wahr* evaluiert –, so schreiben wir $\theta \models \varphi$. Erfüllt θ die Formel nicht, so schreiben wir $\theta \not\models \varphi$.

Beispiel 2. Eine erfüllende Belegung θ_1 für φ (Formel (1)) wäre beispielsweise

$$\theta_1 = \{x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0\}.$$

In Binärschreibweise wäre dies $\theta_1 = 1010$. Es gilt $\theta_1 \models \varphi$.

Beispiel 3. Eine nicht erfüllende Belegung θ_2 für φ (Formel (1)) wäre beispielsweise

$$\theta_2 = \{x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 0\}.$$

In Binärschreibweise wäre dies $\theta_2 = 0000$. Es gilt $\theta_2 \not\models \varphi$.

2.2 Graphentheorie

Neben der Aussagenlogik werden in den folgenden Kapiteln einige Grundlagen der Graphentheorie benötigt. Die im Folgenden angegebenen Definitionen orientieren sich an einem Buch von Reinhard Diestel zu den Grundlagen mathematischer Logik [Die12].

Definition 5. Ein *gerichteter Graph* G ist ein 2-Tupel (V, E), wobei V eine endliche Menge ist (die Menge der *Knoten*) und E eine zweistellige Relation $E \subseteq V \times V$ (die Menge der *Kanten*) ist. Ist $(u, v) \in E$, so gibt es eine Kante von u nach v.

Definition 6. Ein *ungerichteter Graph* G ist ebenfalls ein 2-Tupel (V, E) mit einer endlichen Menge an Knoten V und einer **symmetrischen** Relation $E \subseteq V \times V$, das heißt: $(u, v) \in E \Rightarrow (v, u) \in E$, es ist deswegen auch möglich $\{u, v\} \in E$ zu schreiben.

Definition 7. Ein *Pfad* der Länge n in einem Graphen G ist eine Folge von Knoten $v_1, \ldots, v_n$, in der für alle (v_i, v_{i+1}) mit $1 \leq i \leq n$ gilt: $(v_i, v_{i+1}) \in E$. Man sagt, der Pfad geht von v_1 nach v_n.

Definition 8. Ein *Zyklus* ist ein Pfad, bei dem Start- und Endknoten identisch sind, d.h. $v_1 = v_n$ gilt.

Definition 9. Eine *Zusammenhangskomponente* (abgekürzt ZHK, *engl.* connection component) eines ungerichteten Graphen $G = (V, E)$ ist eine maximale Menge von Knoten $U \subseteq V$ mit der Eigenschaft, dass zwischen je zwei Knoten $u, v \in U$ ein Pfad von u nach v in G existiert.

Beispiel 4. In Abbildung 1 ist der *ungerichtete Graph* $G = (V, E)$ abgebildet mit

$$V = \{i \mid i \in \mathbb{N}, i \leq 8\}$$
$$E = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 1), (6, 7), (7, 8)\}.$$

G besteht aus zwei *Zusammenhangskomponenten* (die erste enthält die Knoten 1 bis 5, die zweite die Knoten 6 bis 8). Ein *Pfad* in G ist zum Beispiel 6, 7, 8. Außerdem enthält G einen *Zyklus*: 1, 2, 3, 4, 5, 1.

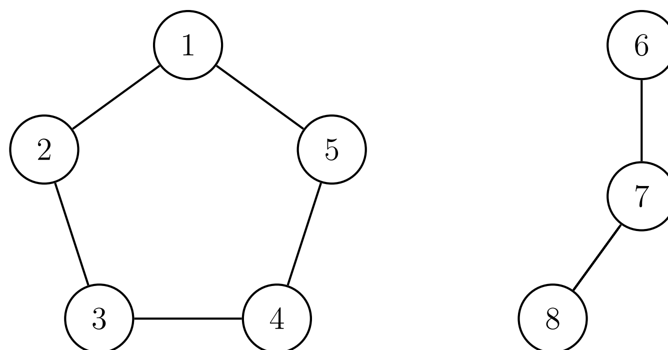

Abbildung 1: Ein ungerichteter Graph mit zwei Zusammenhangskomponenten.

2.3 Enumerationsprobleme

Es werden die Definitionen aus dem Skript zur Vorlesung „SAT-Algorithmen“ der Leibniz Universität Hannover [Mei13] verwendet:

Definition 10. Ein *Enumerationsproblem* ist ein Tupel $E = (I, \mathrm{Sol})$ mit

- I als Menge der Instanzen und
- Sol als Funktion, sodass für alle $x \in I$ $\mathrm{Sol}(x)$ eine endliche Menge der Lösungen von x ist.

Definition 11. Sei $E = (I, \mathrm{Sol})$ ein Enumerationsproblem. Ein Algorithmus $\mathcal{A}$ ist ein *Enumerationsalgorithmus* für E, wenn für alle $x \in I$ gilt:

- $\mathcal{A}$ terminiert nach einer endlichen Anzahl von Schritten.
- $\mathcal{A}$ gibt genau die Elemente von $\mathrm{Sol}(x)$ ohne Duplikate aus.

2.4 Komplexitätsklassen

Für die Grundlagen der Komplexitätsklassen werden das Turingmaschinen-Modell sowie die weiteren grundlegenden Definitionen aus *Komplexität von Algorithmen* von Heribert Vollmer und Arne Meier [MV15] zugrunde gelegt. Um auszusagen, wie gut ein Algorithmus ist, wird seine Laufzeit in Abhängigkeit der Eingabelänge ermittelt und mit Hilfe der Landau-Notation ($\mathcal{O}$-Notation) abgeschätzt. Diese stellt eine obere Schranke der Laufzeit dar und blendet dabei konstante Faktoren aus. Das Vernachlässigen dieser Faktoren ist sinnvoll, da diese oft vom verwendeten Modell bzw. der Soft- und Hardware abhängen, die Laufzeit von Algorithmen zur besseren Vergleichbarkeit untereinander aber unabhängig davon angegeben werden sollte. Im Nachfolgenden werden nun die für diese Arbeit und für Enumerationsprobleme im Allgemeinen interessanten Komplexitätsklassen definiert.

Definition 12. P – Polynomialzeit
Es wird die Definition aus dem bereits erwähnten Werk von Vollmer und Meier [MV15] verwendet:

Demnach ist die Komplexitätsklasse P die Klasse aller Probleme, für die es einen Entscheidungsalgorithmus mit einer Laufzeit von $\mathcal{O}(n^k)$ gibt mit $k \in \mathbb{N}$ und $n =$ Länge der Eingabe.

Definition 13. TotalP – Enumeration in polynomieller Abhängigkeit
Es wird sinngemäß die Definition aus einer der ersten Veröffentlichungen zum Thema Enumeration von Johnson, Yannakakis und Papadimitriou [JYP88] verwendet und in ihrer Formulierung präzisiert:

Sei $E = (I, \mathrm{Sol})$ ein Enumerationsproblem, dann ist $E \in \mathsf{TotalP}$ genau dann, wenn es einen Enumerationsalgorithmus $\mathcal{A}$ und ein Polynom p gibt, sodass für alle $x \in I$ gilt:

Die Laufzeit von $\mathcal{A}$ mit Eingabe x ist durch $p(|x|, |\mathrm{Sol}(x)|)$ beschränkt. Die Gesamtlaufzeit des Algorithmus hängt also polynomiell von der Eingabelänge **und** der Anzahl

der Lösungen ab – dabei kann es sich bei exponentieller Anzahl an Lösungen im Vergleich zur Eingabelänge durchaus um eine Exponentialfunktion in Abhängigkeit der Eingabelänge handeln!

Definition 14. Delay
Sei $E = (I, \mathrm{Sol}))$ ein Enumerationsproblem und $\mathcal{A}$ ein Enumerationsalgorithmus für E. Sei $x \in I$ eine Instanz. Dann ist

- der i-te Delay von $\mathcal{A}$ die Zeit zwischen der Ausgabe der i-ten und $(i + 1)$-ten Lösung aus $\mathrm{Sol}(x)$,
- der 0-te Delay die *Vorberechnungszeit*, was die Zeit vor der ersten Ausgabe ist,
- der n-te Delay die *Nachberechnungszeit*, was die Zeit zwischen der letzten Ausgabe und der Terminierung des Algorithmus ist.

Definition 15. DelayP – Enumeration mit polynomiellem Delay
Sei $E = (I, \mathrm{Sol})$ ein Enumerationsproblem, dann ist $E \in \mathsf{DelayP}$ genau dann, wenn es einen Enumerationsalgorithmus $\mathcal{A}$ und ein Polynom p gibt, sodass für alle $x \in I$ und für alle $i \in \{0, \ldots, n\}$ gilt:

Der i-te Delay von $\mathcal{A}$ mit Eingabe x ist durch $p(|x|)$ beschränkt.

Definition 16. EnumerableP – Enumeration in Polynomialzeit
Sei $E = (I, \mathrm{Sol})$ ein Enumerationsproblem, dann ist $E \in \mathsf{EnumerableP}$ genau dann, wenn es einen Enumerationsalgorithmus $\mathcal{A}$ und ein Polynom p gibt, sodass für alle $x \in I$ gilt:

Die Laufzeit von $\mathcal{A}$ mit Eingabe x ist durch $p(|x|)$ beschränkt, das heißt $\mathcal{A}$ gibt in $p(|x|)$ alle Lösungen $\mathrm{Sol}(x)$ ohne Duplikate aus.

3 Enum-KROM-SAT

In diesem Kapitel wird das Enumerationsproblem für Kromformeln näher betrachtet. Zunächst definieren wir das zugehörige Entscheidungsproblem und zeigen, dass dieses in Polynomialzeit lösbar ist. Den Entscheidungsalgorithmus erweitern wir leicht, um mit dessen Hilfe dann einen Enumerationsalgorithmus für Enum-KROM-SAT anzugeben. Nachdem die Komplexitätsklasse zunächst theoretisch ermittelt wird, werden abschließend die gemessenen Ergebnisse ausgewertet.

Definition 17. Das Entscheidungsproblem für Kromformeln sei definiert als:

$$\textsf{2-SAT} = \textsf{KROM-SAT} := \{\varphi \mid \varphi \text{ ist eine erfüllbare Formel in 2-KNF}\}$$

Definition 18. Das Enumerationsproblem für Kromformeln sei definiert als:

$$\begin{aligned} \textsf{Enum-KROM-SAT} &:= (I, \mathrm{Sol}) \\ I &:= \{\varphi \mid \varphi \text{ ist eine Formel in 2-KNF}\} \\ \mathrm{Sol}(\varphi) &:= \{\theta \mid \theta \models \varphi\} \end{aligned}$$

3.1 Entscheidungsalgorithmus für KROM-SAT

Der Algorithmus sowie die Beweise für die Laufzeit und die Korrektheit wurden aus dem Skript zur Vorlesung „SAT-Algorithmen“ der Universität Hannover [Mei13]) entnommen und von mir erweitert.

Sei φ eine Kromformel, dann hat φ die Form:

$$\varphi = \bigwedge_{i=1}^{m} \underbrace{\ell_{i1} \vee \ell_{i2}}_{\text{Klausel}}$$

mit

- $m \in \mathbb{N}$: Anzahl der Klauseln
- ℓ_{ij} mit $1 \leq i \leq m$, $j \in \{1, 2\}$: Literale, das heißt es gilt $\ell_{ij} = x_k$ oder $\ell_{ij} = \overline{x_k}$
- $\overline{\ell_{ij}} = \begin{cases} x, \text{ wenn } \ell_{ij} = \overline{x} \\ \overline{x}, \text{ wenn } \ell_{ij} = x \end{cases}$

Die Idee des Polynomialzeitalgorithmus, der KROM-SAT entscheidet, ist, jede Klausel in die beiden möglichen logischen Implikationen umzuformen und jede Implikation als

Kante eines gerichteten Graphen darzustellen. Die Knotenmenge des Graphen sind sämtliche Variablen sowie ihre Negationen, das entspricht allen möglichen Literalen der Formel.

Betrachten wir beispielsweise die Klausel $u \vee v$: Sie lässt sich in die Implikationen $\overline{u} \rightarrow v$ und $\overline{v} \rightarrow u$ umformen. Im Graphen werden dementsprechend zwei Kanten hinzugefügt: Eine von $\overline{u}$ nach v und eine von $\overline{v}$ nach u. Dies wird für alle Klauseln der Formel gemacht. Daraus entsteht ein Graph, den man formal wie folgt definieren kann:

$$\begin{aligned} G &= (V, E) \\ V &= \{x_i, \overline{x_i} \mid 1 \leq i \leq n\} \\ E &= \left\{ \left(\overline{\ell_{i1}}, \ell_{i2}\right), \left(\overline{\ell_{i2}}, \ell_{i1}\right) \mid 1 \leq i \leq m \right\} \end{aligned}$$

Zwischen diesem Graphen und der Formel besteht nun ein Zusammenhang, der uns einfach feststellen lässt, ob es (k)eine erfüllende Belegung für die Formel gibt: Wenn es einen Zyklus im Graphen gibt, der sowohl x als auch $\overline{x}$ enthält, kann es keine erfüllende Formel geben. Warum nicht? Entweder x oder $\overline{x}$ muss zu *wahr* evaluieren. Da die Kanten des Graphen Implikationen darstellen, muss das Literal eines jeden folgenden Knotens ebenfalls zu *wahr* evaluieren. Da der Weg irgendwann zur Negation der Variable führt, muss diese auch zu *wahr* evaluieren. Damit wäre $x = \overline{x} = \textit{wahr}$, was offensichtlich ein Widerspruch ist. Gibt es nur einen Weg (und keinen Zyklus) von einem Literal zu seiner Negation, ist dies noch kein Widerspruch: Wir können das Literal mit *falsch* belegen – das heißt, die zugehörige Variable so belegen, dass das Literal zu *falsch* evaluiert – und seine Negation mit *wahr* – damit ist die Implikation erfüllt.

In Algorithmus 1 ist diese Idee formaler in Pseudocode formuliert.

```
  Eingabe: Kromformel φ
1 Konstruiere den gerichteten Graphen G = (V, E) wie oben beschrieben.
2 Für alle x ∈ Vars(φ):
3     Wenn es einen Zyklus gibt, der x und x̄ enthält:
4         Lehne ab.
5 Akzeptiere.
```

Algorithmus 1: KromSAT

Beispiel 5.
Betrachten wir als Beispiel die folgende Kromformel:

$$\varphi := (\overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_1 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_4}). \tag{2}$$

Daraus ergeben sich die in Tabelle 2 angegebenen Implikationen.

Klausel:	$(\overline{x_2} \vee x_3)$	$(x_1 \vee \overline{x_3})$	$(\overline{x_1} \vee \overline{x_2})$	$(x_1 \vee x_3)$	$(\overline{x_1} \vee \overline{x_4})$
Implikation 1:	$x_2 \to x_3$	$\overline{x_1} \to \overline{x_3}$	$x_1 \to \overline{x_2}$	$\overline{x_1} \to x_3$	$x_1 \to \overline{x_4}$
Implikation 2:	$\overline{x_3} \to \overline{x_2}$	$x_3 \to x_1$	$x_2 \to \overline{x_1}$	$\overline{x_3} \to x_1$	$x_4 \to \overline{x_1}$

Tabelle 2: Implikationen, die sich aus Formel (2) ergeben.

Aus den in Tabelle 2 angegebenen Implikationen für Formel (2) ergibt sich der Graph in Abbildung 2.

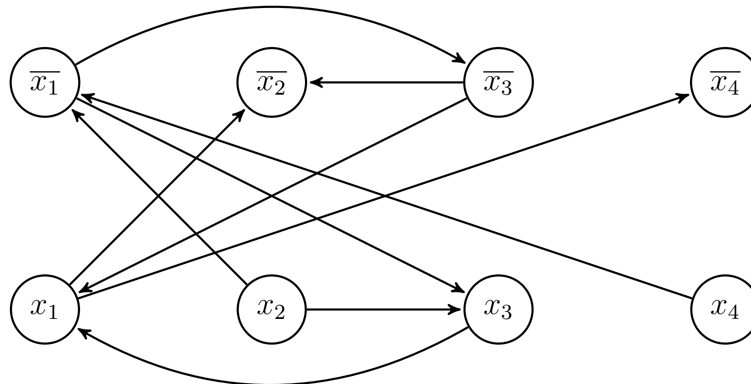

Abbildung 2: Der Graph, der durch den Algorithmus für Formel (2) erstellt wird.

Für jede Variable $x \in \{x_1, x_2, x_3, x_4\}$ wird nun mittels Breitensuche überprüft, ob es einen Weg von x nach $\overline{x}$ und einen Weg von $\overline{x}$ nach x gibt. In diesem Beispiel gibt es keinen solchen Zyklus, das heißt, der Algorithmus akzeptiert – die Formel ist erfüllbar. In Abschnitt 3.3 wird gezeigt, wie man aus diesem Graphen eine erfüllende Belegung konstruieren kann.

3.2 Erweiterung auf Kromformeln mit partiellen Belegungen

Da für den Enumerationsalgorithmus Kromformeln mit gegebenen partiellen Belegungen auf Erfüllbarkeit getestet werden, muss der Entscheidungsalgorithmus etwas angepasst werden. Durch das Einsetzen einer partiellen Belegung können Klauseln ganz oder teilweise aufgelöst werden. Wird eine Klausel zu *wahr* evaluiert, kann sie für den weiteren Algorithmus ignoriert werden. Wird dabei eine Klausel zu *falsch* evaluiert, kann die gesamte Formel nicht mehr erfüllt werden und der Algorithmus kann abbrechen.

Interessanter sind dabei die Klauseln, die zum Teil aufgelöst werden, da sich hierdurch „Klauseln“ ergeben, die nicht mehr der 2-KNF entsprechen. Klauseln der Form $(1 \vee x)$ können hier schon zu *wahr* evaluiert werden und fallen somit weg. Klauseln der Form $(0 \vee x)$ werden zu den Implikationen $\overline{x} \to 0$ und $1 \to x$ umgeformt. Für die Werte $1 \widehat{=}$ *wahr* und $0 \widehat{=}$ *falsch* werden ebenfalls Knoten im Graphen erstellt, die bei der Zyklensuche jedoch anders behandelt werden. Da es sich hierbei nicht um zu belegende Variablen handelt, muss lediglich gelten, dass es keinen Weg von *wahr* zu *falsch* geben darf, da somit die Implikation verletzt würde und es keine erfüllende Belegung mehr geben kann. Der Weg von *falsch* nach *wahr* ist uninteressant, da hierdurch keine Implikation verletzt werden kann.

Eingabe: Krom-Formel φ, partielle Belegung θ
1 Setze θ in φ ein.
2 **Wenn** sich eine leere Klausel ergibt**:**
3   Lehne ab.
4 Lösche alle bereits erfüllten Klauseln.
5 Konstruiere den gerichteten Graphen $G = (V, E)$ wie oben beschrieben.
6 **Für alle** $x \in \text{Vars}(\varphi)$**:**
7   **Wenn** es einen Zyklus gibt, der x und $\overline{x}$ enthält**:**
8   Lehne ab.
9 **Wenn** es einen Weg von *wahr* nach *falsch* gibt**:**
10   Lehne ab.
11 Akzeptiere.

Algorithmus 2: KromSAT

Beispiel 6.
Um dies noch einmal zu verdeutlichen, betrachten wir Formel (2) noch einmal, nun aber mit der partiellen Belegung $\theta = \{x_1 = 0\}$.

$$\begin{aligned}
\varphi &:= (\overline{x_2} \lor x_3) \land (x_1 \lor \overline{x_3}) \land (\overline{x_1} \lor \overline{x_2}) \land (x_1 \lor x_3) \land (\overline{x_1} \lor \overline{x_4}) \quad (2) \\
\varphi[\theta] &= (\overline{x_2} \lor x_3) \land (0 \lor \overline{x_3}) \land (1 \lor \overline{x_2}) \land (0 \lor x_3) \land (1 \lor \overline{x_4}) \\
&= (\overline{x_2} \lor x_3) \land (0 \lor \overline{x_3}) \land \underbrace{\cancel{(1 \lor \overline{x_2})}}_{\text{erfüllt}} \land (0 \lor x_3) \land \underbrace{\cancel{(1 \lor \overline{x_4})}}_{\text{erfüllt}} \\
&= (\overline{x_2} \lor x_3) \land (0 \lor \overline{x_3}) \land (0 \lor x_3) \quad (3)
\end{aligned}$$

Daraus ergeben sich die in Tabelle 3 angegebenen Implikationen sowie der in Abbildung 3 angegebene Graph.

Klausel:	$(\overline{x_2} \lor x_3)$	$(0 \lor \overline{x_3})$	$(0 \lor x_3)$
Implikation 1:	$x_2 \to x_3$	$1 \to \overline{x_3}$	$1 \to x_3$
Implikation 2:	$\overline{x_3} \to \overline{x_2}$	$x_3 \to 0$	$\overline{x_3} \to 0$

Tabelle 3: Implikationen, die sich aus Formel (3) ergeben.

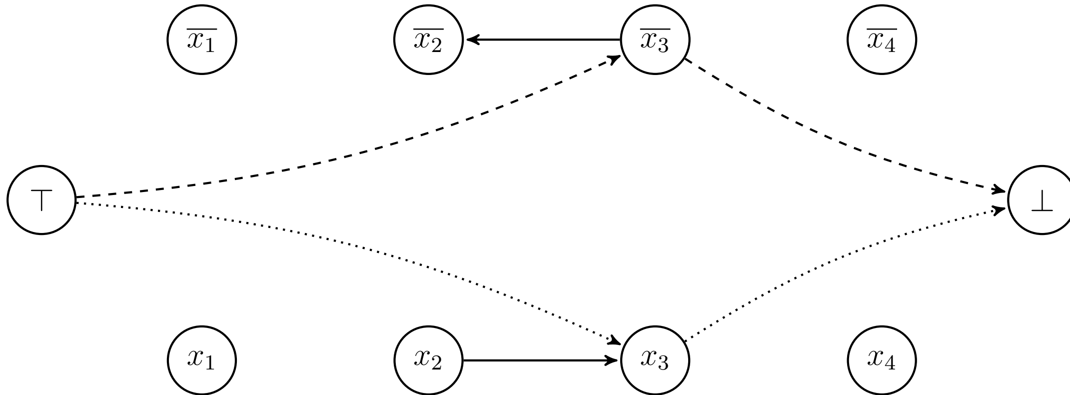

Abbildung 3: Der Graph, der durch den Algorithmus für φ mit der partiellen Belegung θ erstellt wird. Die gepunkteten und gestrichelten Pfeile markieren jeweils einen Pfad, aufgrund dessen der Algorithmus ablehnt.

Da es (mindestens) einen Pfad von *wahr* nach *falsch* gibt (in Abbildung 3 gepunktet bwz. gestrichelt markiert), lehnt der Algorithmus ab – φ ist mit der partiellen Belegung θ unerfüllbar.

3.3 Korrektheit

Um zu beweisen, dass der Algorithmus korrekt arbeitet, wird gezeigt, dass jede akzeptierte Formel eine erfüllende Belegung hat und dass jede abgelehnte Formel keine erfüllende Belegung haben kann.

Jede akzeptierte Formel hat eine erfüllende Belegung

Mit Hilfe des vom Algorithmus erzeugten Graphen lässt sich wie folgt eine Belegung θ zu der eingegebenen Formel φ konstruieren sodass $\theta \models \varphi$:

1. Wähle ein Literal ℓ, zu dem eine Kante von $\top$ ausgeht (das heißt $(\top, \ell) \in E$) oder, wenn es keines mehr gibt, eines, für das es in G keinen Weg zu $\overline{\ell}$ gibt.
2. Belege die zugehörige Variable so, dass das Literal zu *wahr* ausgewertet wird, damit wird das Literal $\overline{\ell}$ automatisch zu *falsch* evaluiert.
3. Folge allen Pfaden in G, die von ℓ ausgehen und belege die entsprechenden Variablen so, dass die Literale zu *wahr* ausgewertet werden.
4. Solange es noch unbelegte Variablen gibt: Beginne wieder bei 1.

Aufgrund der Symmetrie der Implikation gilt:
Gibt es einen Pfad von einem Literal ℓ_1 zu einem Literal ℓ_2, dann gibt es auch einen Pfad von $\overline{\ell_2}$ nach $\overline{\ell_1}$. Daraus folgt, dass es keine Möglichkeit gibt, dass Pfade von ℓ zu einem anderen Literal ℓ' **und** zu seiner Negation $\overline{\ell'}$ existieren, weil es in diesem Fall ebenfalls Pfade von ℓ' und $\overline{\ell'}$ zu $\overline{\ell}$ geben würde, was aber bei der Wahl des Literals in Schritt 1 ausgeschlossen wurde. Da es keine Zyklen gibt, die sowohl ein Literal als auch seine Negation enthalten – in diesem Fall hätte der Algorithmus abgelehnt –, wird eine widerspruchsfreie und erfüllende Belegung erzeugt.

Jede abgelehnte Formel ist unerfüllbar

Eine Formel wird dann abgelehnt, wenn es im Graphen einen Zyklus gibt, der sowohl ein Literal ℓ als auch seine Negation $\overline{\ell}$ enthält oder es einen Weg von *wahr* nach *falsch* gibt. Im ersten Fall muss eines der Literale so belegt werden, dass es zu *wahr* evaluiert wird. Da die Kanten im Graphen Implikationen bedeuten, müssen alle darauf folgenden Literale ebenfalls zu *wahr* evaluiert werden, dementsprechend auch $\overline{\ell}$, was ein Widerspruch ist. Ein Pfad von *wahr* nach *falsch* würde bedeuten, dass es in der Formel eine

Implikation der Form *wahr* $\rightarrow \ldots \rightarrow$ *falsch* gäbe, was unerfüllbar ist.

Daraus folgt, dass der Algorithmus korrekt arbeitet.

3.4 Laufzeitabschätzung

Für die Laufzeitabschätzung wird der später im Enumerationsalgorithmus verwendete, erweiterte Entscheidungsalgorithmus betrachtet. Sei n die Anzahl der Variablen und m die Anzahl der Klauseln der Formel. Für das Einsetzen der Belegung sowie für das Löschen der bereits erfüllten Klauseln wird die Formel einmal durchlaufen. Für den Aufbau des Graphen werden zuerst $2n + 2$ Knoten erstellt und dann pro Klausel zwei Kanten – also $2m$ Kanten – hinzugefügt. Die Schleife wird für jede Variable einmal – also n mal insgesamt – ausgeführt. Das Überprüfen, ob ein Zyklus existiert, wird mittels Breitensuche ausgeführt, welche eine Laufzeit von $\mathcal{O}(|V|+|E|)$ hat (vgl. [Hoc10]), wobei $|V| = 2n + 2$ und $|E| = 2m$. Pro Zyklussuche werden zwei Breitensuchen ausgeführt – eine von x nach $\overline{x}$ und eine von $\overline{x}$ nach x.

Für den Weg von *wahr* nach *falsch* wird ebenfalls eine Breitensuche benötigt. Daraus ergibt sich eine Gesamtlaufzeit von:

$$\underbrace{2 \cdot \mathcal{O}(n)}_{\text{Belegung einsetzen}} + \underbrace{\mathcal{O}(2n + 2 + 2m)}_{\text{Graph erstellen}} + \underbrace{(2n + 1) \cdot \mathcal{O}(2n + 2 + 2m)}_{\text{Breitensuchen}}$$

Dies kann zusammengefasst werden zu einer Laufzeit von:

$$\mathcal{O}(n \cdot (n + m))$$

Es folgt: KROM-SAT $\in$ P.

3.5 Enumerationsalgorithmus

Da uns nun aber nicht nur die Entscheidung interessiert, ob eine Kromformel erfüllbar ist, sondern im Falle ihrer Erfüllbarkeit auch ihre Lösungen, betrachten wir nun einen Enumerationsalgorithmus. Der Algorithmus sowie die Beweise für die Laufzeit und die Korrektheit wurden aus dem Skript zur Vorlesung „SAT-Algorithmen“ der Universität Hannover [Mei13]) entnommen und von mir erweitert.

Die Idee dahinter ist die folgende: Der Belegungsbaum der Formel wird rekursiv in Präordnung durchlaufen (siehe Beispiel 7) und für jeden Knoten wird mit Hilfe von

Algorithmus 2 überprüft, ob die Formel mit dieser Belegung noch erfüllbar ist. Ist sie es nicht, wird die Rekursion an dieser Stelle abgebrochen. Dadurch werden nur Teilbäume betrachtet, in denen sich auch mindestens eine Lösung befindet.

In Algorithmus 3 wird der Enumerationsalgorithmus für Enum-KROM-SAT in Pseudocode dargestellt.

```
  Eingabe: Krom-Formel φ, partielle Belegung θ
1 Wenn φ[θ] ≡ 1 und Vars(φ) = Vars(θ) :
2   Gib θ aus.
3 Wenn φ[θ] erfüllbar ist und Vars(φ) ≠ Vars(θ) :
4   Wähle Variable x ∈ Vars(φ[θ]) .
5   EnumKromSAT(φ, θ ∪ {x = 0}).
6   EnumKromSAT(φ, θ ∪ {x = 1}).
```

Algorithmus 3: EnumKromSAT

Beispiel 7.
Betrachten wir als Beispiel wieder die folgende Formel:

$$\varphi := (\overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_1 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_4}). \tag{2}$$

Der Algorithmus wird mit EnumKromSAT$(\varphi, \emptyset)$, also einer leeren partiellen Belegung, aufgerufen. Da die Formel weder zu *wahr* evaluiert wird noch alle Variablen belegt sind, wird das erste **Wenn** in Zeile 1 nicht ausgeführt. Wie bereits in Beispiel 5 gezeigt, ist diese Formel erfüllbar und es sind noch nicht alle Variablen belegt, das heißt, das zweite **Wenn** in Zeile 3 wird ausgeführt. Es wird eine Variable gewählt – zum Beispiel x_1 – und mit 0 belegt. Mit dieser partiellen Belegung wird die Funktion erneut aufgerufen.

Die Belegung wird eingesetzt und das erste **Wenn** in Zeile 1 wird wieder übersprungen. Auch das zweite **Wenn** in Zeile 3 wird nicht ausgeführt, da die Formel wie in Beispiel 6 gezeigt nicht erfüllbar ist. Die Rekursion wird an dieser Stelle abgebrochen – es gibt keine erfüllende Belegung mit $\theta(x_1) = 0$, dieser Teilbaum kann also übersprungen werden – und wir kehren zum ersten Funktionsaufruf zurück, wo x_1 nun mit 1 belegt und die Funktion erneut mit $\theta = \{x_1 = 1\}$ aufgerufen wird.

Dies wird rekursiv so weitergeführt, bis die erste Lösung $\theta = \{x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 0\}$ gefunden wird. Als nächstes wird die Belegung $\theta = \{x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1\}$ überprüft und zu *falsch* evaluiert, weshalb diese Lösung nicht ausgegeben wird.

In Abbildung 4 wird der Belegungsbaum visualisiert: Die Zahlen in den Knoten stellen stellen die Reihenfolge dar, in der die Belegungen überprüft werden (Präordnung). Blaue Knoten stehen für erfüllbare Teilbelegungen, rot markiert sind Teilbelegungen und Belegungen, mit denen die Formel nicht mehr erfüllbar ist. Grün sind erfüllende Belegungen der Formel, die vom Algorithmus ausgegeben werden. Dies ist auch die Färbung, die für die Visualisierung im entwickelten Programm verwendet wird.

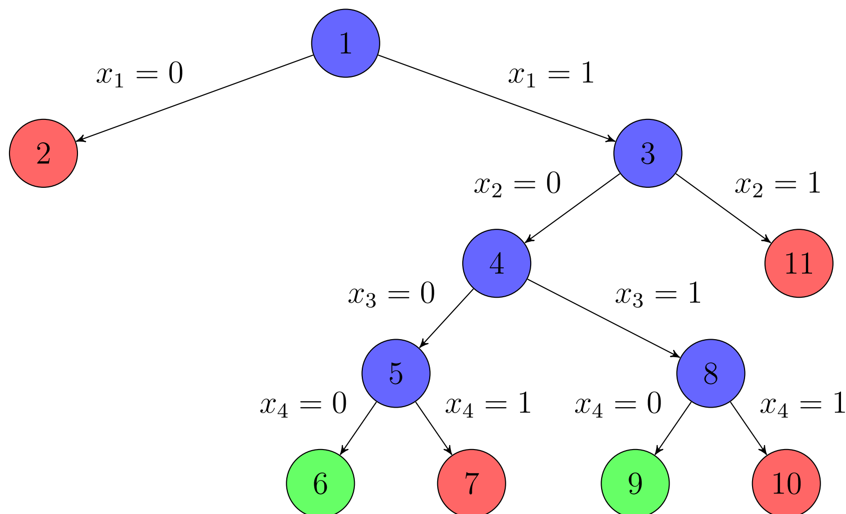

Abbildung 4: Der Belegungsbaum für φ. Blau: φ ist mit θ erfüllbar, θ ist keine vollständige Belegung, rot: φ ist mit θ nicht erfüllbar/erfüllt, grün: θ ist eine erfüllende Belegung für φ.

3.6 Korrektheit

Es ist zu zeigen, dass der Algorithmus nach endlicher Zeit terminiert und alle Lösungen – also alle erfüllenden Belegungen – der Eingabe φ ohne Duplikate ausgibt.

Der Algorithmus terminiert nach endlicher Zeit

Da der Algorithmus im ungünstigsten Fall jede mögliche Belegung der Formel genau einmal überprüft und die Anzahl der möglichen Belegungen (2^n) endlich ist, terminiert der Algorithmus in jedem Fall nach endlicher Zeit.

Alle erfüllenden Belegungen werden ohne Duplikate ausgegeben

Da jede mögliche Belegung maximal einmal betrachtet wird, sind Duplikate ausgeschlossen.

Durch die Überprüfung $\varphi[\theta] \equiv 1$ in Zeile 1 wird sichergestellt, dass nur Belegungen ausgegeben werden, für die $\theta \models \varphi$ gilt. Jetzt bleibt noch zu zeigen, dass alle erfüllenden Belegungen auch ausgegeben werden: Es wird der gesamte Belegungsbaum der Formel φ betrachtet und nur dann ein Teilbaum übersprungen, wenn in selbigem keine erfüllende Belegung mehr möglich ist. Die Korrektheit des hierfür verwendeten Algorithmus wurde bereits in Abschnitt 3.3 gezeigt. Wird in Zeile 4 zudem immer die Variable mit dem kleinsten beziehungsweise größten Index gewählt, werden die Lösungen zusätzlich in lexikographisch aufsteigender beziehungsweise absteigender Reihenfolge angegeben.

Hieraus folgt, dass der angegebene Enumerationsalgorithmus für das Problem Enum-KROM-SAT (Definition 18) korrekt arbeitet.

3.7 Theoretische Laufzeitbetrachtung

Wir wollen nun zeigen, dass Enum-KROM-SAT $\in$ DelayP gilt, das heißt, dass die Zeit, die zwischen der Ausgabe zweier Lösungen benötigt wird, durch ein Polynom in Abhängigkeit der Eingabelänge beschränkt ist. Als Eingabelänge wird hier vereinfachend die Anzahl der Variablen n beziehungsweise die Anzahl der Klauseln m angegeben, da sie unabhängig von der Darstellung der Formel sind.

Ohne Beschränkung der Allgemeinheit kann angenommen werden, dass in Zeile 4 immer die Variable mit dem kleinstmöglichen Index gewählt wird. Der Einfachheit halber werden Belegungen für diesen Beweis als Binärzahlen dargestellt, also $y = y_1 y_2 \dots y_n = \{0,1\}^n$, analog für alle anderen angegebenen Belegungen. Seien p, q die Laufzeiten der Algorithmen für die Evaluierung und das Überprüfen der Erfüllbarkeit einer Formel.

Um die Delays zu berechnen, werden zwei Fälle unterschieden:

- Die beiden Belegungen folgen aufeinander
- Die beiden Belegungen folgen nicht direkt aufeinander

Fall 1: Die beiden Belegungen folgen aufeinander

Seien y, z zwei aufeinanderfolgende erfüllende Belegungen. Der Delay wird maximal für $y = 01\ldots1$ und $z = 10\ldots0$, da hier die gesamte Tiefe des Baumes durchlaufen und auf jeder Ebene (bis auf der obersten Ebene mit der leeren Belegung) einmal die Erfüllbarkeit der partiellen Belegung getestet werden muss.

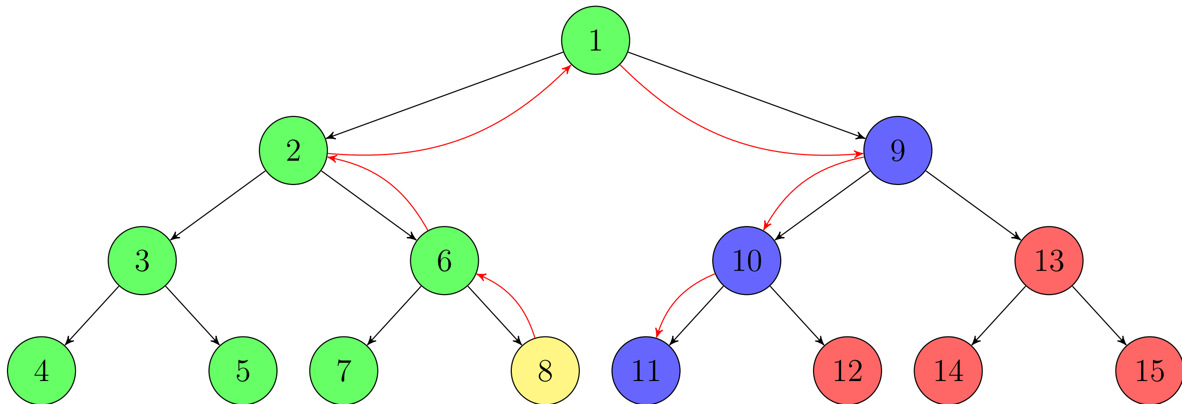

Abbildung 5: Belegungsbaum für eine Formel mit 4 Variablen:
grün: Belegung wurde bereits betrachtet,
gelb: Aktuell betrachtete Belegung,
blau: Belegung wird auf dem Weg zur nächsten Lösung überprüft,
rot: Belegung wird nach der nächsten Belegung betrachtet.

In Abbildung 5 wird beispielhaft der Belegungsbaum für eine Formel mit 4 Variablen dargestellt und der eben beschrieben Fall visualisiert. Die Knoten sind durchnummeriert in der Reihenfolge, in der die zugehörigen Belegungen durchlaufen werden, wobei eine Abzweigung in der i-ten Ebene nach links bedeutet, dass hier x_i auf 0 gesetzt wird. Eine Abzweigung nach rechts bedeutet, dass x_i auf 1 gesetzt wird. Der Weg durch die rekursiv erstellten Inkarnationen[1] der Funktion wird durch den roten Weg symbolisiert.

Der Delay für diesen Fall setzt sich wie folgt zusammen:

$$\underbrace{(n-1)}_{\text{für jede Ebene}} \cdot (\underbrace{p(|\varphi|)}_{\text{Evaluierung}} + \underbrace{q(|\varphi|)}_{\text{Erfüllbarkeitstest}})$$
$$\in \mathcal{O}(n \cdot (p(|\varphi|) + q(|\varphi|)))$$

Dabei handelt es sich offensichtlich um ein Polynom in Abhängigkeit der Eingablänge.

[1] Eine Inkarnation meint einen Unteraufruf einer Funktion.

Fall 2: Die beiden Belegungen folgen nicht aufeinander

Seien y, z zwei direkt nacheinander ausgegebene, lexikographisch aber nicht direkt aufeinander folgende erfüllende Belegungen, ohne Beschränkung der Allgemeinheit sei $y < z$. Das heißt, dass es Belegungen $\theta_1, \ldots, \theta_k$ gibt mit $k \in \{1, \ldots, 2^n - 2\}$, die lexikographisch zwischen y und z liegen, die aber φ nicht erfüllen ($\forall\ i \in \mathbb{N}, i \leq k : \theta_i \not\models \varphi$). Diese werden jedoch nicht alle einzeln betrachtet, sondern durch abgebrochene Rekursionsaufrufe gruppenweise ausgeschlossen. Um den Delay zu bestimmen, müssen wir zuerst herausfinden, wie viele Belegungsgruppen – und damit (abgebrochene) Rekursionsaufrufe – es zwischen y und z allgemein gibt.

Die Idee der Gruppierungen

Es wird der Weg von y nach z des Algorithmus durch den Baum verfolgt. Das gemeinsame Präfix der Länge r der beiden Belegungen wird ignoriert – es stellt einen gemeinsamen Weg von der Wurzel im Baum bis zu einem bestimmten Knoten dar. Ab dem Punkt, an dem sich die Wege trennen, werden nun die Gruppen ermittelt. Da $y < z$, nimmt y im Baum den linken und z den rechten Weg, das heißt, eine Variable wird bei y auf 0 gesetzt ($y_{r+1} = 0$), die bei z auf 1 gesetzt wird ($z_{r+1} = 1$). Geht man die Belegung von y nun weiter durch, wurden bei jeder Variable y_i mit $\{y_i = 1\} \in y$ bereits alle Belegungen mit $y_i = 0$ vorher betrachtet – d.h. überall, wo y den rechten Zweig nimmt, wurde der linke bereits überprüft. Für alle $i \geq r + 2$ mit $\{x_i = 0\} \in y$ liegt der Fall $\{y_i = 1\}$ zwischen y und z und wird hier betrachtet. Da es zwischen y und z keine Lösung geben kann (vgl. Annahme), wird hier die Rekursion abgebrochen und somit 2^{n-i} Belegungen ausgeschlossen, die zu einer Gruppe zusammengefasst werden können. Das bedeutet, für jede 0, die in y nach dem gemeinsamen Präfix und der ersten 0 folgt, wird eine Gruppe von Belegungen ausgeschlossen.

Für z wird dies analog gemacht, jedoch werden hier die Gruppen betrachtet, die vor z selbst geprüft werden, das heißt für jede 1, die in z nach dem gemeinsamen Präfix und der ersten 1 auftritt, – also jede rechte Abzweigung auf dem Weg zu z – liegt eine Gruppe zwischen y und z, nämlich die, wo der linke Zweig genommen wird.

Der Algorithmus für die Gruppierungen

Sei $p_1 \cdots p_r$ mit $r \in \{0, \ldots, n-2\}$ das gemeinsame Präfix der beiden nacheinander ausgegebenen, erfüllenden Belegungen y und z. $r = 0$ bedeutet, dass es kein gemeinsames Präfix gibt und $r = n - 1$ bedeutet, dass y und z lexikographisch auf einander folgen,

was in Fall 1 abgedeckt wurde und hier somit ausgeschlossen wird. Es gilt also:

$$y = p_1 \cdots p_r 0 y_{r+2} \cdots y_n \qquad = y_1 \cdots y_r 0 y_{r+2} \cdots y_n$$
$$z = p_1 \cdots p_r 1 z_{r+2} \cdots z_n \qquad = z_1 \cdots z_r 1 z_{r+2} \cdots z_n$$

1. Wiederhole die folgenden Schritte für $i \in \{n, \dots, r+2\}$:
   a) Wenn $y_i = 0$: Gruppiere 2^{n-i} Belegungen der Form $y_1 \dots y_{i-1} 1 e_1 \dots e_{n-i}$ mit $e_j \in \{0,1\}$
   b) Wenn $y_i = 1$: Tue nichts
2. Wiederhole die folgenden Schritte für $i \in \{r+2, \dots, n\}$:
   a) Wenn $z_i = 0$: Tue nichts
   b) Wenn $z_i = 1$: Gruppiere 2^{n-i} Belegungen der Form $z_1 \dots z_{i-1} 0 e_1 \dots e_{n-i}$ mit $e_j \in \{0,1\}$

Pro Gruppe wird nur einmal überprüft, ob die Formel mit dieser partiellen Belegung erfüllbar ist. Zudem wird noch für jede partielle Belegung auf dem Weg zu z überprüft, ob φ damit erfüllbar ist (in Beispiel 8 blau). Die Anzahl der Gruppen wird maximal für $r = 0$ und wenn im ersten Schritt nie a) und im zweiten Schritt nie b) eintrifft, das heißt in jedem Schritt eine Gruppe hinzugefügt wird. Es ergibt sich eine maximale Anzahl an Gruppen von $2 \cdot (n-2)$ (in Schritt 1 und 2 jeweils $n-2$ neue Gruppen).

Das heißt der Delay setzt sich zusammen aus:

$$\begin{aligned}&(\#\text{Gruppen} + \underbrace{(n-r)}_{\text{Knoten auf dem Weg zu } z}) \cdot (\underbrace{p(|\varphi|) + q(|\varphi|)}_{\text{Evaluierung und Erfüllbarkeitstest}})\\ =&(2 \cdot (n-2) + (n-r)) \cdot (p(|\varphi|) + q(|\varphi|))\\ \in& \mathcal{O}(n \cdot (p(|\varphi|) + q(|\varphi|)))\end{aligned}$$

Daraus folgt: Enum-KROM-SAT $\in$ DelayP.

Beispiel 8.
Als Beispiel für den Gruppenfindungsalgorithmus betrachten wir nun eine Formel mit drei Variablen und den zwei nacheinander ausgegebenen erfüllenden Belegungen $y = 001$ und $z = 101$.

Die beiden Belegungen haben kein gemeinsames Präfix, das heißt $r = 0$.

Nun gehen wir die Belegung für y durch von $y_n = y_3$ bis $y_{r+2} = y_2$:

- $y_3 = 1$: Es wird nichts getan.
- $y_2 = 0$: Es wird die Gruppe der Belegungen $01*$ gezählt, wobei * dafür steht, dass hier beliebig 0 oder 1 folgen kann.

Bis hierhin haben wir nun also eine Gruppe gezählt.

Für z gehen wir nun analog vor und gehen die Belegung von $z_{r+2} = z_2$ bis $z_n = z_3$ durch:

- $z_2 = 0$: Es wird nichts getan
- $z_3 = 1$: Es wird eine Gruppe mit der Belegung 100 hinzugefügt.

Wir zählen insgesamt also zwei Gruppen. In Abbildung 6 wird dieses Beispiel noch einmal visualisiert: Die roten Pfeile stellen den Weg des Algorithmus durch die Inkarnationen der Funktion und somit die Reihenfolge der Betrachtung zwischen y und z dar. Die Zahlen in den Knoten symbolisieren die Reihenfolge, in welcher der Baum durchlaufen wird. Die grünen Knoten wurden bereits betrachtet, der gelbe Knoten ist y. Die roten Knoten stellen jeweils die Wurzel eines übersprungenen Teilbaumes dar – also quasi den „Kopf“ einer Gruppe. Die grauen Knoten sind die, die durch das Überspringen gar nicht erst betrachtet werden. Die blauen Knoten sind erfüllbar und müssen für den Delay ebenfalls betrachtet werden, da die Erfüllbarkeit vor dem Erreichen von z geprüft wird. Die weißen Knoten werden erst nach z betrachtet.

Es ist zu sehen: Die Belegungen θ_1 und θ_2 werden zu einer Gruppe zusammengefasst, ebenso bildet θ_3 eine Gruppe – es gibt also 2 Gruppen von Belegungen (in der Abbildung grau hinterlegt) zwischen y und z.

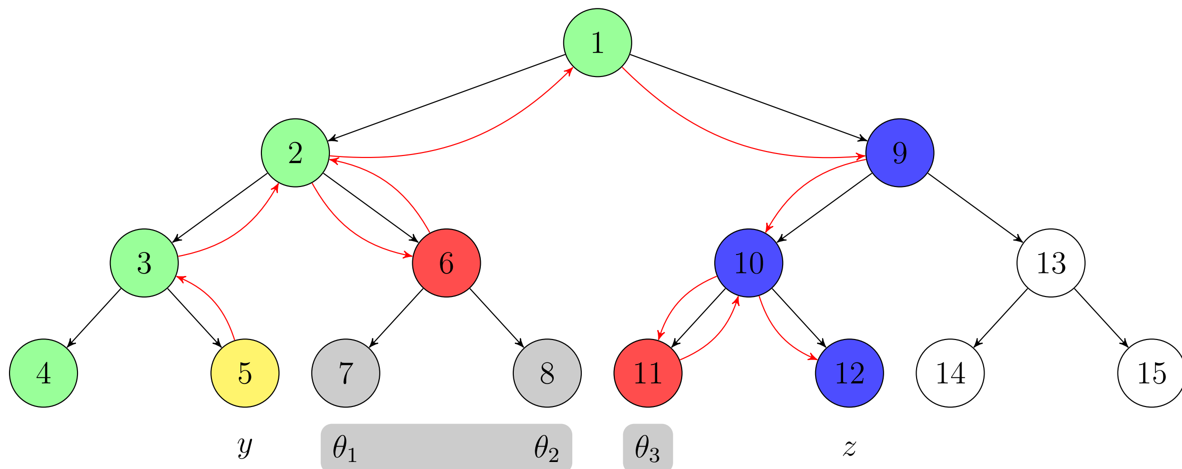

Abbildung 6: Belegungsbaum für eine Formel mit 3 Variablen.
Visualisierung für Fall 2 der Laufzeitabschätzung.

3.8 Praktische Laufzeitanalyse

Mit dem im Rahmen dieser Bachelorarbeit entwickelten Tool für Enumerationsalgorithmen wurde der Algorithmus für Enum-KROM-SAT nun praktisch ausgeführt und die Delays gemessen. Es soll nun untersucht werden, inwiefern die theoretisch ermittelte Laufzeit mit der gemessenen übereinstimmt. Hierzu wird aus den Messdaten wie in Abschnitt 5.6 erklärt eine polynomielle Regressionsfunktion erstellt.

3.8.1 Systemvoraussetzungen

Alle Tests wurden unter den folgenden Voraussetzungen auf dem gleichen System durchgeführt.

CPU: Intel i5-4590, 4 × 3,3GHz

RAM: 2 × 4GB (DDR3-1600)

Betriebssystem: Arch Linux, Kernel 3.14

Java: openjdk 1.8.0_51, 64-Bit

Umgebung: Zum Zeitpunkt der Tests lief außer den Standardsystemanwendungen (wie Windowmanager u.Ä.) nichts. Das Programm arbeitet 4 Instanzen parallel ab.

3.8.2 Testinstanzen

Es wurden zwei Experimente durchgeführt, die im Folgenden näher beschrieben werden. Unerfüllbare Formeln werden hierbei vernachlässigt, da sich hier kein aussagekräftiger Delay oder ähnliches messen lässt. Das bedeutet, alle getesteten Instanzen haben mindestens eine erfüllende Belegung.
Alle Testinstanzen wurden mit dem Tool selbst wie in Abschnitt 5.4.3 beschrieben erstellt.

Experiment 1

Im ersten Experiment soll getestet werden, wie sich die Delays verhalten, wenn sich das Variablen-Klausel-Verhältnis nahe 2 bewegt, also insgesamt konstant ist. Hierzu werden erfüllbare Kromformeln erstellt und mit dem bereits vorgestellten Enumerationsalgorithmus ausgewertet. In Tabelle 4 wird die Verteilung der verwendeten Testinstanzen dargestellt.

#Variablen	#Klauseln	#Instanzen
10	20	200
20	40	200
30	60	200
40	80	200
50	100	200
60	120	200
70	140	169
80	160	121

Tabelle 4: Testinstanzen für Experiment 1 (Variablen-Klausel-Verhältnis nahe 2).

Experiment 2

Im zweiten Experiment soll getestet werden, wie sich die Delays verhalten, wenn sich das Variablen-Klausel-Verhältnis nahe $|V|$ (Anzahl der Variablen) bewegt, also insgesamt linear ist. Hierzu werden erfüllbare Kromformeln erstellt und mit dem bereits vorgestellten Enumerationsalgorithmus ausgewertet. In Tabelle 5 wird die Verteilung der verwendeten Testinstanzen dargestellt.

#Variablen	#Klauseln	#Instanzen
10	100	200
20	400	200
30	600	200
40	800	200
50	1000	150
60	1500	100
70	3000	50
80	5000	26

Tabelle 5: Testinstanzen für Experiment 2 (Variablen-Klausel-Verhältnis nahe $|V|$).

Erwartung

Da jede Klausel eine Einschränkung an eine erfüllende Belegung für die Formel ausdrückt, erwarte ich, dass im zweiten Experiment die durchschnittliche Anzahl an erfüllenden Belegungen deutlich geringer ist als im ersten Experiment. Da der Delay zwischen den Lösungen theoretisch etwas größer wird, je weiter die Lösungen auseinander liegen – was bei weniger Lösungen häufiger der Fall sein wird – erwarte ich, dass die Delays in Experiment 2 minimal größer sein werden als in Experiment 1.

3.8.3 Ergebnisse

Experiment 1

Die Ergebnisse für Experiment 1 werden auf drei Nachkommastellen gerundet in Tabelle 6 angegeben und in Abbildung 7 als Diagramm abgebildet.

#Variablen	#Klauseln	durchschn. #Lösungen	durchschn. Delay (ms)
10	20	3,685	0,542
20	40	14,52	5,149
30	60	66,43	11,717
40	80	225,18	22,727
50	100	775,25	30,817
60	120	3190,56	34,522
70	140	9241,16	43,506
80	160	11418,91	61,485

Tabelle 6: Testergebnisse für Experiment 1 (Variablen-Klausel-Verhältnis nahe 2).

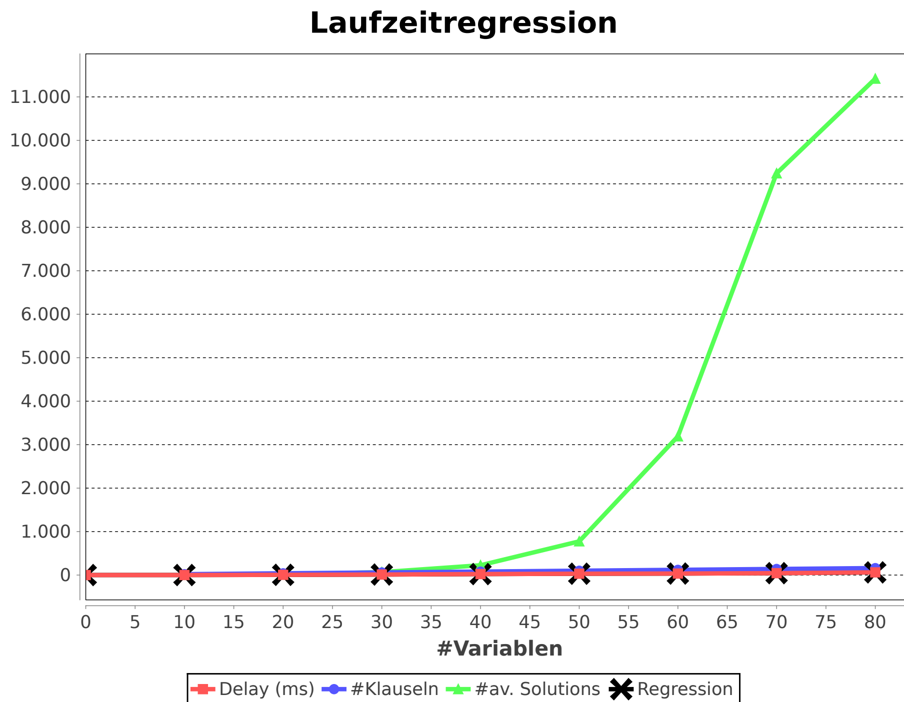

Abbildung 7: Ergebnisse von Experiment 1 in einem Diagramm.

Als Regressionsfunktion (auf drei Nachkommastellen gerundet) ergibt sich anhand dieser Werte:

$$
\begin{aligned}
&-0{,}002 \cdot x^0 \\
&-1{,}282 \cdot x^1 \\
&+0{,}273 \cdot x^2 \\
&-0{,}020 \cdot x^3 \\
&+7{,}696 \cdot 10^{-4} \cdot x^4 \\
&-1{,}495 \cdot 10^{-5} \cdot x^5 \\
&+1{,}418 \cdot 10^{-7} \cdot x^6 \\
&-5{,}204 \cdot 10^{-10} \cdot x^7
\end{aligned}
$$

Experiment 2

Die Ergebnisse für Experiment 1 werden auf drei Nachkommastellen gerundet in Tabelle 7 angegeben und in Abbildung 8 als Diagramm abgebildet.

#Variablen	#Klauseln	durchschn. #Lösungen	durchschn. Delay (ms)
10	100	1,0	3,27
20	400	1,0	44,725
30	600	1,0	142,92
40	800	1,0	316,795
50	1000	1,0	617,267
60	1500	1,0	1256,14
70	3000	1,0	4044,2
80	5000	1,0	9496,269

Tabelle 7: Testergebnisse für Experiment 2 (Variablen-Klausel-Verhältnis nahe $|V|$).

Als Regressionsfunktion (auf drei Nachkommastellen gerundet) ergibt sich anhand dieser Werte:

$$
\begin{aligned}
+0{,}533 \cdot x^0 \\
-85{,}169 \cdot x^1 \\
+18{,}801 \cdot x^2 \\
-1{,}512 \cdot x^3 \\
+0{,}059 \cdot x^4 \\
-0{,}001 \cdot x^5 \\
+1{,}159 \cdot 10^{-5} \cdot x^6 \\
-4{,}395 \cdot 10^{-8} \cdot x^7
\end{aligned}
$$

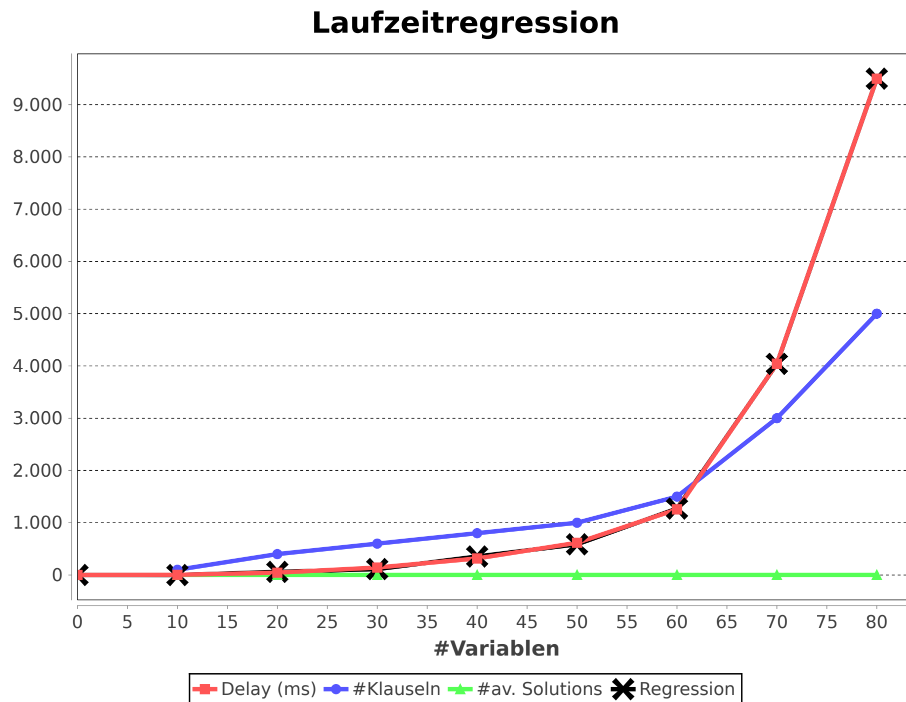

Abbildung 8: Ergebnisse von Experiment 2 in einem Diagramm.

3.8.4 Auswertung

Der ermittelte theoretische Delay für Enum-KROM-SAT liegt bei

$$\mathcal{O}(n \cdot (p(|\varphi|) + q(|\varphi|))).$$

p und q sind hierbei die Polynome, die angeben, wie viel Zeit für die Evaluierung und den Erfüllbarkeitstest für eine aussagenlogische Formel benötigt wird.

Die Anzahl der Lösungen liegt wie erwartet bei Experiment 2, unabhängig von der Anzahl der Variablen, im Durchschnitt bei 1. Auch die Delays sind wie erwartet bei Experiment 2 etwas größer als bei Experiment 1. Bei Experiment 1 mit linearer Klauselanzahl liegen die Delays zwischen 0,5ms bei 10 Variablen und 1s bei 80 Variablen. Bei Experiment 2 mit quadratischer Klauselanzahl liegen diese Delays bei 3ms bzw. 9s. Beide Delay-Abbildungen lassen sich jedoch gut durch eine polynomielle Regressionsfunktion annähern. In beiden Fällen werden die Vorfaktoren der Funktion ab der vierten Potenz (bei linearer Klauselanzahl bereits ab der dritten Potenz) verschwindend klein, was darauf hindeutet, dass eine Regressionsfunktion vierten Grades die Delays bereits hinreichend genau annähern könnte.

Dass bei größerer Klauselanzahl die Laufzeit steigt, ergibt Sinn, da bei der Erfüllbarkeitsüberprüfung der Graph größer wird und bei der Breitensuche somit mehr Kanten betrachtet werden müssen. Damit kommen wir zu dem Ergebnis, dass die gemessene Laufzeit gut mit der bereits theoretisch berechneten übereinstimmt.

4 EnumConnComp

Dieses Kapitel befasst sich mit dem Enumerationsalgorithmus für Zusammenhangskomponenten in Graphen.

Definition 19. Sei das Enumerationsproblem EnumConnComp definiert als:

$$\begin{aligned}\mathsf{EnumConnComp} &:= (I, \mathrm{Sol})\\ I &:= \{G \mid G = (V, E) \text{ ist ein ungerichteter Graph}\}\\ \mathrm{Sol}(G) &:= \{V' \mid V' \subseteq V,\ V \text{ ist eine Zusammenhangskomponente von G}\}\end{aligned}$$

4.1 Enumerationsalgorithmus

Wir wollen nun alle Zusammenhangskomponenten eines Graphen enumerieren. Die Idee ist, einen unmarkierten Knoten zu suchen, und dann mittels iterativer Breitensuche alle mit ihm verbundenen Knoten auch zu markieren. Ist die Breitensuche abgeschlossen, wurde eine ZHK gefunden und die markierten Knoten werden ausgegeben. Dann wird von vorne begonnen und mit einer neuen Markierung wird die nächste ZHK markiert. Dies wird solange wiederholt bis es keine unmarkierten Knoten mehr gibt.

In Algorithmus 4 wird diese Idee formal als Pseudocode formuliert.

Eingabe: ungerichteter Graph $G = (V, E)$

1. **Solange** es einen unmarkierten Knoten $x \in V$ gibt**:**
2. Markiere x.
3. $V' \leftarrow \{x\}$.
4. **Solange** es Kanten $(u, v) \in E$ gibt mit u markiert und v unmarkiert**:**
5. Markiere v.
6. Füge v zu V' hinzu.
7. Gib V' aus.

Algorithmus 4: EnumConnComp

4.2 Korrektheit

Um zu zeigen, dass Algorithmus 4 ein korrekt arbeitender Enumerationsalgorithmus für EnumConnComp ist, muss gezeigt werden, dass er nach einer endlichen Anzahl von Schritten terminiert und genau die ZHK der Eingabe G ohne Duplikate ausgibt.

Der Algorithmus terminiert nach endlicher Zeit

Die äußere Schleife läuft, solange es unmarkierte Knoten gibt. Pro Durchlauf wird mindestens ein Knoten markiert. Da es endlich viele Knoten gibt, hat die äußere Schleife eine endliche Laufzeit. Analog in der inneren Schleife: Um die Bedingung der inneren Schleife zu prüfen, müssen im ungünstigsten Fall alle Kanten durchsucht werden. In der Schleife wird immer ein Knoten markiert. Da es endlich viele Kanten und Knoten gibt, hat auch diese Schleife eine endliche Laufzeit.

Der Algorithmus terminiert also bei jeder gültigen Eingabe nach endlicher Zeit.

Alle ZHK werden ohne Duplikate ausgegeben

Der Algorithmus sucht eine unmarkierten Knoten und markiert nacheinander alle anderen Knoten, die von diesem aus erreichbar sind. Ist kein Knoten mehr erreichbar, das heißt es gibt keinen markierten Knoten mehr, der mit einem unmarkierten verbunden ist, wurde die ZHK vollständig markiert. Die Liste mit allen zu dieser ZHK gehörenden Knoten wird dann ausgegeben.

Da markierte Knoten im weiteren Verlauf nicht mehr zu der Liste hinzugefügt werden, können keine Knoten – und somit auch keine ZHK – doppelt auftreten. Des Weiteren ist durch die äußere Schleife garantiert, dass am Ende alle Knoten markiert sind und somit auch ausgegeben wurden. Die innere Schleife stellt sicher, dass eine ZHK erst dann ausgegeben wird, wenn alle ihr zugehörigen Knoten hinzugefügt wurden.

Es folgt, dass Algorithmus 4 ein korrekt arbeitender Enumerationsalgorithmus für EnumConnComp ist.

4.3 Theoretische Laufzeitabschätzung

Wir wollen nun zeigen, dass EnumConnComp $\in$ EnumerableP gilt, das heißt, dass der Algorithmus in Polynomialzeit alle ZHK des eingegebenen Graphen enumeriert.

Die äußere Schleife wird maximal $|V|$ mal ausgeführt, da in jedem Durchlauf mindestens ein Knoten markiert wird. Um die Bedingung dieser Schleife zu überprüfen, müssen im ungünstigsten Fall alle Knoten einmal betrachtet werden (Laufzeit $|V|$).

Die innere Schleife wird maximal $|V|$ mal ausgeführt, da pro Schleifendurchlauf ein Knoten markiert wird und es $|V|$ Knoten gibt. Zum Überprüfen der Schleifenbedingung müssen im ungünstigsten Fall alle Kanten durchsucht werden. Die Auswertung dauert also maximal $|E|$ Schritte.

Es wird davon ausgegangen, dass das Markieren sowie die genutzten Mengenoperationen in $\mathcal{O}(1)$ möglich sind.

Es ergibt sich also eine nach oben abgeschätzte Gesamtlaufzeit von:

$$|V| \cdot \left(\underbrace{|V|}_{\text{äußere Bedingung}} + \overbrace{|V| \cdot \underbrace{|E|}_{\text{innere Bedingung}}}^{\text{innere Schleife}} \right)$$

Ein vollständiger Graph mit n Knoten hat $\frac{1}{2} \cdot (n^2 + n)$ Kanten. Mit der Abschätzung, dass sich die maximale Kantenanzahl quadratisch zur Knotenanzahl verhält, lässt sich diese Laufzeit wie folgt angeben:

$$\begin{aligned} &\mathcal{O}(|V| \cdot \big(|V| + |V| \cdot |V|^2\big)) \\ =\ &\mathcal{O}(|V|^2 + |V|^4) \\ =\ &\mathcal{O}(|V|^4) \end{aligned}$$

Daraus folgt: EnumConnComp $\in$ EnumerableP.

4.4 Praktische Laufzeitanalyse

Mit dem im Rahmen dieser Bachelorarbeit entwickelten Tool für Enumerationsalgorithmen wurde der Algorithmus für EnumConnComp nun praktisch ausgeführt und die Delays gemessen. Es soll nun untersucht werden, inwiefern die theoretisch ermittelte Laufzeit mit der praktisch gemessenen übereinstimmt.

4.4.1 Systemvoraussetzungen

Es gelten dieselben Systemvoraussetzungen wie in Abschnitt 3.8.1 bereits beschrieben.

4.4.2 Testinstanzen

Es wurden drei Experimente durchgeführt, die im Folgenden näher beschrieben werden. Alle Testinstanzen wurden mit dem Tool wie in Abschnitt 5.5.3 beschrieben erstellt.

Experiment 3

In diesem Experiment soll getestet werden, wie sich die Laufzeit verhält, wenn sich das Kanten-Knoten-Verhältnis nahe 2 bewegt, also insgesamt konstant ist. Die Größe der Instanzen wird ähnlich wie in Abschnitt 3.8.2 gewählt, um den Vergleich der

Größenordnung zwischen DelayP- und EnumerableP-Algorithmen zu erhalten. Hierzu werden Graphen erstellt und mit dem bereits vorgestellten Enumerationsalgorithmus ausgewertet. In Tabelle 8 wird die Verteilung der verwendeten Testinstanzen dargestellt.

#Knoten	#Kanten	#Instanzen
100	50	20
200	100	20
300	150	20
400	200	20
500	250	20
600	300	20
700	350	20
800	400	20
900	450	20

Tabelle 8: Testinstanzen für Experiment 3.

Experiment 4

In diesem Experiment soll getestet werden, wie sich die Laufzeit verhält, wenn sich das Kanten-Knoten-Verhältnis nahe 2 bewegt, also insgesamt konstant ist. Die Größe der Instanzen wird hierfür größer als im ersten Experiment gewählt, um einen weiteren Vergleich zur Größenordnung zwischen DelayP- und EnumerableP-Algorithmen zu erhalten. Hierzu werden Graphen erstellt und mit dem bereits vorgestellten Enumerationsalgorithmus ausgewertet. In Tabelle 9 wird die Verteilung der verwendeten Testinstanzen dargestellt.

#Knoten	#Kanten	#Instanzen
2500	1250	20
5000	2500	20
7500	3750	20
10000	5000	20
12500	6250	20
15000	7500	20
17500	8750	20
20000	10000	20
22500	11250	20

Tabelle 9: Testinstanzen für Experiment 4.

Experiment 5

In diesem Experiment soll getestet werden, wie sich die Laufzeit verhält, wenn sich das Kanten-Knoten-Verhältnis nahe $|V|$ bewegt, also insgesamt linear ist. Die Größe der Instanzen wird hierfür ähnlich wie im ersten Experiment gewählt, um einen weiteren Vergleich zur Größenordnung zwischen DelayP- und EnumerableP-Algorithmen zu erhalten. Hierzu werden Graphen erstellt und mit dem bereits vorgestellten Enumerationsalgorithmus ausgewertet. In Tabelle 10 wird die Verteilung der verwendeten Testinstanzen dargestellt.

#Knoten	#Kanten	#Instanzen
100	2500	50
200	5000	50
300	15000	50
400	40000	50
500	55000	50
600	65000	40
700	100000	21
800	150000	16

Tabelle 10: Testinstanzen für Experiment 5.

4.4.3 Ergebnisse

Experiment 3

Die Ergebnisse für Experiment 3 werden auf drei Nachkommastellen gerundet in Tabelle 11 angegeben und in Abbildung 9 als Diagramm abgebildet.

#Knoten	#Kanten	durchschn. #Lösungen	durchschn. Laufzeit (ms)
100	50	50,4	0,05
200	100	100,8	0,1
300	150	150,65	1,7
400	200	200,45	2,65
500	250	250,5	1,95
600	300	300,5	5,85
700	350	350,8	6,55
800	400	400,7	10,15
900	450	451,1	14,35

Tabelle 11: Testinstanzen für Experiment 3 (Knoten-Kanten-Verhältnis fast konstant).

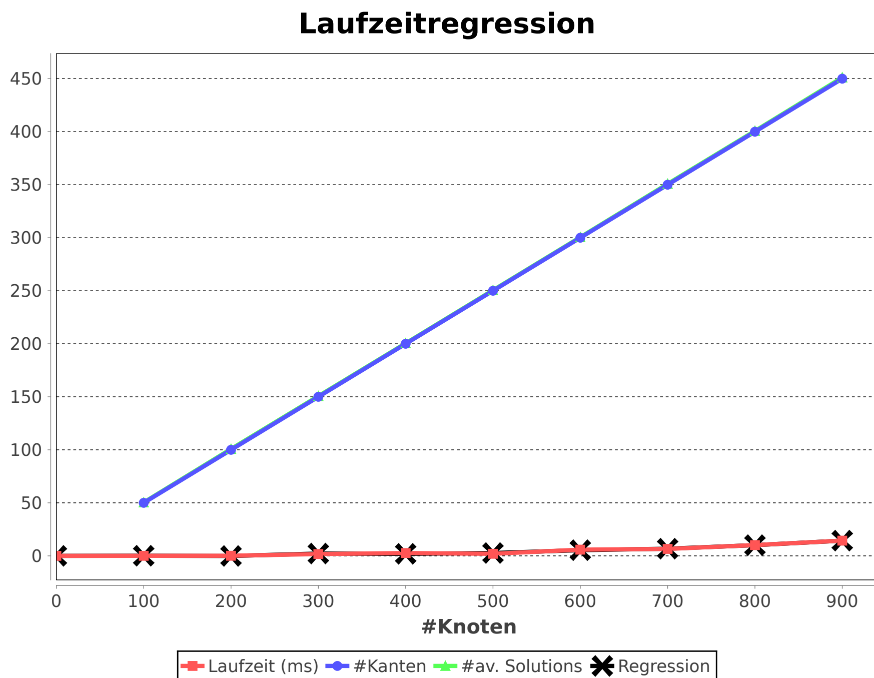

Abbildung 9: Ergebnisse von Experiment 3 in einem Diagramm.

Als Regressionsfunktion ergibt sich anhand dieser Werte:

$$
\begin{aligned}
&-0{,}006 \cdot x^0 \\
&+0{,}210 \cdot x^1 \\
&-0{,}005 \cdot x^2 \\
&+3{,}968 \cdot 10^{-5} \cdot x^3 \\
&-1{,}645 \cdot 10^{-7} \cdot x^4 \\
&+3{,}754 \cdot 10^{-10} \cdot x^5 \\
&-4{,}795 \cdot 10^{-13} \cdot x^6 \\
&+3{,}215 \cdot 10^{-16} \cdot x^7 \\
&-8{,}811 \cdot 10^{-20} \cdot x^8
\end{aligned}
$$

Experiment 4

Die Ergebnisse für Experiment 4 werden auf drei Nachkommastellen gerundet in Tabelle 12 angegeben und in Abbildung 10 als Diagramm abgebildet.

#Knoten	#Kanten	durchschn. #Lösungen	durchschn. Laufzeit (ms)
1000	500	500,75	0,15
2500	1250	1251,25	6,4
5000	2500	2501,4	37,3
7500	3750	3751,4	104,85
10000	5000	5001,4	98,4
12500	6250	6251,25	535,15
15000	7500	7501,35	441,15
17500	8750	8750,95	1079,45
20000	10000	10001,15	1772,15

Tabelle 12: Testinstanzen für Experiment 4 (Knoten-Kanten-Verhältnis fast konstant).

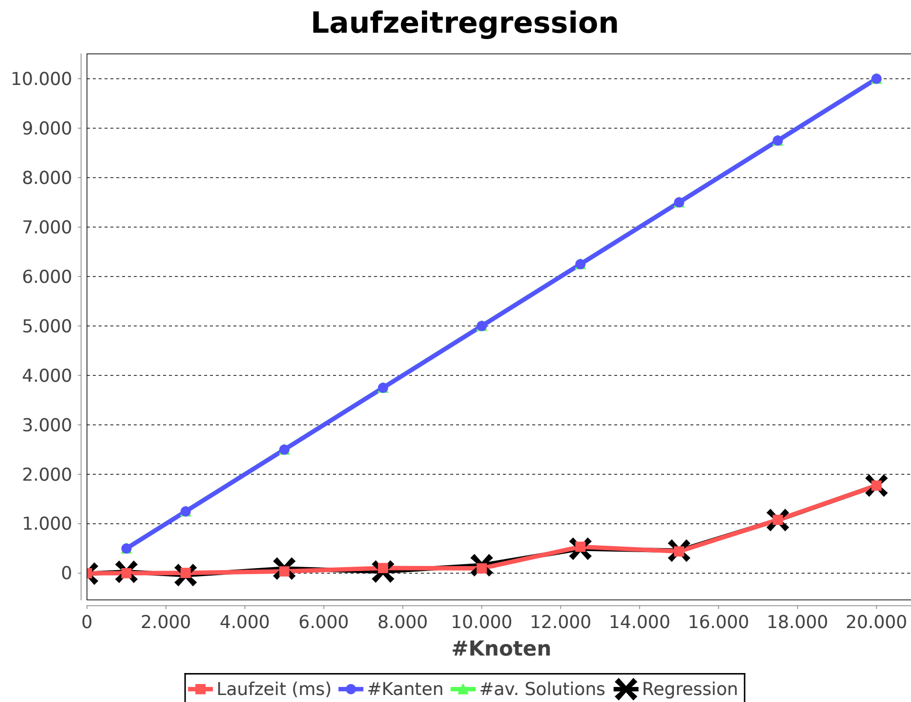

Abbildung 10: Ergebnisse von Experiment 4 in einem Diagramm.

Als Regressionsfunktion ergibt sich anhand dieser Werte:

$$\begin{aligned}
&-8{,}1534 \cdot x^0 \\
&+0{,}2560 \cdot x^1 \\
&-3{,}6025 \cdot 10^{-4} \cdot x^2 \\
&+1{,}7700 \cdot 10^{-7} \cdot x^3 \\
&-4{,}0876 \cdot 10^{-11} \cdot x^4 \\
&+4{,}9944 \cdot 10^{-15} \cdot x^5 \\
&-3{,}3136 \cdot 10^{-19} \cdot x^6 \\
&+1{,}1266 \cdot 10^{-23} \cdot x^7 \\
&-1{,}5355 \cdot 10^{-28} \cdot x^8
\end{aligned}$$

Experiment 5

Die Ergebnisse für Experiment 5 werden auf drei Nachkommastellen gerundet in Tabelle 13 angegeben und in Abbildung 11 als Diagramm abgebildet.

#Knoten	#Kanten	durchschn. #Lösungen	durchschn. Laufzeit (ms)
100	2500	1,0	20,56
200	5000	1,0	68,98
300	15000	1,0	604,92
400	40000	1,0	11569,78
500	55000	1,0	29385,14
600	65000	1,0	57019,275
700	100000	1,0	181958,238
800	150000	1,0	513695,938

Tabelle 13: Testinstanzen für Experiment 5 (Knoten-Kanten-Verhältnis linear).

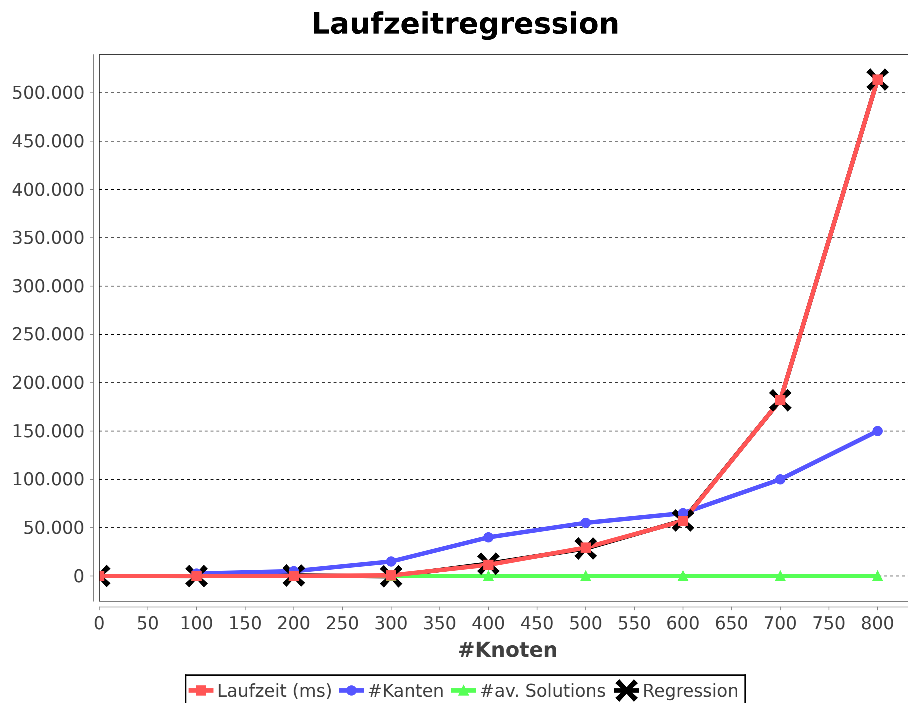

Abbildung 11: Ergebnisse von Experiment 5 in einem Diagramm.

Als Regressionsfunktion ergibt sich anhand dieser Werte:

$$
\begin{aligned}
&+16{,}567 \cdot x^0 \\
&-431{,}393 \cdot x^1 \\
&+9{,}554 \cdot x^2 \\
&-0{,}076 \cdot x^3 \\
&+2{,}877 \cdot 10^{-4} \cdot x^4 \\
&-5{,}49 \cdot 10^{-7} \cdot x^5 \\
&+5{,}113 \cdot 10^{-10} \cdot x^6 \\
&-1{,}821 \cdot 10^{-13} \cdot x^7
\end{aligned}
$$

4.4.4 Auswertung

Die ermittelte theoretische Laufzeit für EnumConnComp liegt bei

$$\mathcal{O}(|V| \cdot (|V| + |V| \cdot |E|)).$$

Die Größenordnung der Laufzeiten bewegt sich bei 1s (Experiment 3) bei 100 bis 800 Knoten und linearer Kantenanzahl. Bei größerer Knotenanzahl (1000 bis 20.000) bewegt sich die Laufzeit zwischen 0,1ms und etwa 2s. Bei quadratischer Kantenanzahl mit 100 bis 800 Knoten steigt die Laufzeit stark auf 20ms bis ca. 8min an.

Auch diese Einschätzung trifft die Regressionsfunktion sehr gut – die Vorfaktoren werden bei Experiment 3 (lineare Kantenanzahl) ab der zweiten Potenz und bei Experiment 5 (quadratische Kantenanzahl) ab der dritten Potenz verschwindend klein. Dies deutet darauf hin, dass auch ein Polynom vierten Grades (wie in der theoretischen Abschätzung) die Laufzeiten bereits hinreichend genau annähern könnte.

Weiterhin fällt auf, dass bei quadratischer Kantenanzahl (also tendenziell eher vollständigen Graphen) die Anzahl der Lösungen (ZHK) bei 1 liegt.

5 Implementierung

Ziel dieser Arbeit war es, ein erweiterbares Framework für Enumerationsalgorithmen anhand der beiden vorgestellten Algorithmen zu implementieren. Diese sollten dabei visualisiert sowie ihre Laufzeiten beziehungsweise Delays gemessen werden.

5.1 Allgemeines

Die Umsetzung erfolge in der Sprache Java mit dem `openjdk 1.8.0_51`, 64-Bit-Version unter dem Betriebssystem Arch Linux, Kernel 3.14 [ope]. Die Visualisierung der Belegungsbäume und Graphen sowie die interne Repräsentation der Graphen erfolgt mit Hilfe der Bibliothek `JUNG` [jun]. Aus der Bibliothek `Apache-Commons` werden das `math`-Packet und das `batik`-Packet verwendet, ersteres für die Berechnung der Regressionsfunktionen aus den Messdaten, letzteres zusammen mit der Bibliothek `JFreeChart` zusammen, um die Diagramme zu exportieren [apab, apaa]. Für ein komfortables Parsen der Eingabeparameter der Kommandozeile wird das Framework `JCommander` benutzt [jco].

5.2 Zeitmessung

Für die Zeitmessung wurde die bereits von Arne Meier implementierte Klasse verwendet, welche eine Genauigkeit von einer Mikrosekunde aufweist [Mei05]. Es werden die jeweiligen Delays gemessen, die vor der Ausgabe einer Lösung auftreten. Die Gesamtlaufzeit ist die Summe aller Delays. Damit werden Zeiten, die für Ein- und Ausgabe benötigt werden (zum Beispiel für das Speichern der Lösung in einer Datei), von der Zeitmessung ausgeschlossen.

5.3 Benutzung des Programms

Das Programm liegt als ausführbare *.jar-Datei vor, welche sich mit

```
java -jar programm.jar [Parameter]
```

starten lässt. Es ist darauf zu achten, die richtige Java-Version zu verwenden. Die möglichen Parameter werden in Tabelle 14 aufgelistet und erklärt.

Parameter	Erklärung
`-x`	Startet die graphische Oberfläche. Wenn dieser Parameter gesetzt ist, werden alle anderen ignoriert.
`-a`	Gibt an, welcher Algorithmus ausgeführt werden soll. `EKS` steht hierbei für Enum-KROM-SAT, `ECC` für EnumConnComp.
`-i`	Auf diesen Parameter folgt der absolute Pfad zum Verzeichnis mit den Instanzen, für die der Algorithmus ausgeführt werden soll.
`-o`	Auf diesen Parameter folgt der absolute Pfad zum Verzeichnis, in dem die Lösungsdateien gespeichert werden sollen.

Tabelle 14: Parameter des Programms.

5.3.1 Kommandozeile

Ein beispielhafter Kommandozeilenaufruf, der die *.cnf-Dateien aus dem Verzeichnis `/home/nutzer/instanzen` mit dem Algorithmus für Enum-KROM-SAT testen und in `/home/nutzer/loesungen` speichern soll, sieht so aus:

```
java -jar programm.jar -a EKS -i /home/nutzer/instanzen -o
/home/nutzer/loesungen
```

5.3.2 Graphische Oberfläche

Startet man das Programm mit dem Parameter `-x`, so öffnet sich ein Fenster mit dem Hauptmenü. Hier kann der Algorithmus ausgewählt werden. Auf Buttonklick öffnet sich dann ein für diesen Algorithmus angepasstes neues Fenster. Jedes dieser Fenster enthält 4 Tabs, deren Funktionen im Folgenden am Beispiel von Enum-KROM-SAT vorgestellt werden.

Neue Instanzen erstellen: Im ersten Tab mit dem Titel „Neue Instanzen“ können neue Kromformeln erstellt werden. Hierfür muss die Anzahl der Variablen, die Anzahl der Klauseln sowie ein Pfad zum Speichern selbiger angegeben werden. Ein Haken in der Checkbox „Alle Variablen verwenden“ sorgt dafür, dass beim Erstellen der Formel das Vorkommen aller Variablen in der Formel erzwungen wird. Es werden nur erfüllbare Formeln generiert. Für eine genauere Beschreibung siehe Abschnitt 5.4.3.

Den Algorithmus ausführen: Im zweiten Tab mit dem Titel „Testen“ kann ein Pfad zu einem Verzeichnis mit Testinstanzen sowie ein Ausgabepfad zu einem Verzeichnis, in dem die Lösungen gespeichert werden sollen, angegeben werden. Als Instanzen werden alle Dateien des Eingabeverzeichnisses mit der richtigen Dateiendung ausgelesen und verwendet. Der Algorithmus läuft *threaded*, das heißt, es werden immer bis zu 4 Instanzen parallel abgearbeitet. Die Ergebnisse werden als *.`*-sol` im in Abschnitt 5.4.2 angegebenen Format abgespeichert. Sind alle Testinstanzen abgearbeitet, wird automatisiert ein Polynom maximal möglichen Grades erstellt, welches die Ergebnisse approximiert. Für eine genauere Erklärung dieser Funktion siehe Abschnitt 5.6.

Den Algorithmus für eine Instanz visualisieren: Im dritten Tab mit dem Titel „Visualisieren“ kann ein Durchlauf für eine bestimmte Instanz visualisiert werden. Es gibt die Möglichkeiten eine Instanz aus einer Datei zu laden oder zufällig eine neue Instanz zu erstellen. Um die Abarbeitung besser beobachten zu können, kann hier ein Delay angegeben werden, das zwischen den einzelnen Schritten eingebaut wird. Bei Enum-ConnComp wurde auf die Textdarstellung der Instanz verzichtet, da diese aus dem angezeigten Graphen ersichtlich wird.

Bereits vorhandene Ergebnisse anzeigen: Im vierten Tab mit dem Titel „Ergebnisplot“ können Ergebnisse visualisiert und eine Regressionsfunktion zur Annäherung der Delays/Laufzeiten erstellt werden. Als Pfad zum Verzeichnis mit den Lösungen ist hier automatisch der Ausgabepfad aus dem Tab „Testen“ voreingestellt. Die im Diagramm dargestellten Ergebnisse können als *`.svg`-Datei exportiert werden.

5.4 Enum-KROM-SAT

5.4.1 Eingabe: DIMACS-Format

Die Eingabe erfolgt im sogenannten DIMACS-Format, welches vom *Center for Discrete Mathematics and Theoretical Computer Science (DIMACS)* für Wettbewerbe mit SAT-Algorithmen vorgeschlagen wurde [SAT93]. Dieses Format hat sich als de-facto Standard für SAT-Instanzen etabliert und wird auch bei der jährlichen SAT-Competition verwendet [SAT]. Die Dateien werden mit der Dateiendung *`.cnf` gespeichert und sind wie folgt aufgebaut:

Kommentare Kommentarzeilen beginnen mit dem Zeichen `c`.

Problemzeile Die erste Zeile, die keinen Kommentar enthält, ist die Problemspezifikation und sieht so aus:

```
p cnf #Var #Klauseln
```

Hierbei gibt `cnf` an, dass es sich um eine Formel in konjunktiver Normalform handelt, `#Var` die Anzahl der Variablen und `#Klauseln` die Anzahl der Klauseln, die folgen werden.

Klauseln Die Klauseln werden zeilenweise angegeben, die Variablen dabei durch ihre Indexzahlen referenziert und durch Leerzeichen getrennt. Eine negative Zahl bedeutet, dass die Variable negiert verwendet wird. Jede Klausel wird mit einer `0` beendet, was mehrzeilige Klauseln ermöglicht. Das Parsen solcher Klauseln wurde jedoch nicht implementiert.

Beispiel 9. Eine gültige *.`cnf`-Datei könnte zum Beispiel wie in Listing 1 aussehen.

```
1 c Kommentar
2 c Hier könnte man schreiben, ob die Formel erfüllbar ist
3 c oder weitere Eigenschaften erfüllt
4 p cnf 4 5
5 -2 3 0
6 1 -3 0
7 -1 -2 0
8 1 3 0
9 -1 -4 0
```

Listing 1: Darstellung von Formel (2) als Datei im DIMACS-Format.

5.4.2 Ausgabe: csv-Format

Die Ausgabe des Programmes erfolgt über *comma-separated-value*-Dateien (abgekürzt csv-Datei). Hierbei wird für jede Lösung zunächst der Delay in Millisekunden (ms) angegeben, worauf ein Semikolon und die Lösung in einem formatierten String folgen, der – um das spätere Verarbeiten der Lösungsdateien zu erleichtern – selbst kein Semikolon enthalten sollte. Diese werden mit der Dateiendung *.`cnf-sol` abgespeichert.

Beispiel 10. Eine gültige Lösungsdatei könnte zum Beispiel wie in Listing 2 aussehen.

```
42;{1=true, 2=false, 3=false, 4=false}
0;{1=true, 2=false, 3=true, 4=false}
```

Listing 2: Darstellung der Lösung von Formel (2) als Datei im csv-Format.

5.4.3 Zufälliges Erzeugen von Formeln

Um eine Kromformel mit n Variablen und m Klauseln zu erzeugen, werden zufällig m verschiedene Klauseln erstellt. Eine Klausel wird erstellt, indem zwei Zahlen $x, y \in \mathbb{N}$ mit $1 \leq x, y \leq n$ mit zufälligem Vorzeichen gewählt werden. Mit jeder neuen Klausel wird überprüft, ob die Formel erfüllbar ist. Ist sie es, wird die Klausel hinzugefügt und die nächste generiert. Ist sie es nicht, wird sie nicht hinzugefügt und eine neue erzeugt. Dies garantiert jedoch nicht, dass alle n verschiedenen Variablen auch tatsächlich in der Formel vorkommen.

Sollen alle Variablen erzwungen werden, so wird in den ersten n Klauseln jeweils x auf die Zahl der Klausel mit zufälligem Vorzeichen gesetzt sowie am Ende geprüft, ob alle Variablen vorkommen. Ist dies nicht der Fall, wird die Formel verworfen und eine neue erzeugt. Dies wird solange wiederholt, bis eine gültige, alle Variablen enthaltende Formel generiert wurde.

5.5 EnumConnComp

5.5.1 Eingabeformat

Das Eingabeformat ähnelt dem DIMACS-Format aus Abschnitt 5.4.1. Kommentare beginnen analog mit einem `c`. Es gibt eine Spezifikationszeile `p edge #Knoten #Kanten`. Der Dateityp `edge` bedeutet, dass danach alle Kanten im Format `e Knoten1 Knoten2` angegeben werden.

Beispiel 11. Eine gültige Eingabedatei könnte zum Beispiel wie in Listing 3 aussehen.

```
1 c Kommentar
2 p edge 4 5
3 e 2 3
4 e 1 3
5 e 1 2
6 e 2 5
7 e 1 4
```

Listing 3: Darstellung des Graphen aus Abbildung 12 im Eingabeformat.

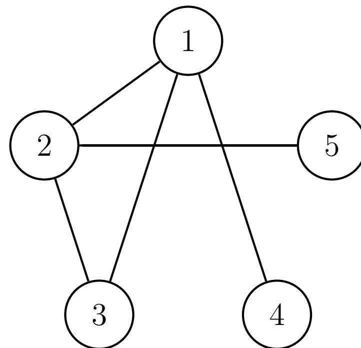

Abbildung 12: Der Graph aus Listing 3.

5.5.2 Ausgabe: csv-Format

Die Ausgabe erfolgt analog zu der in Abschnitt 5.4.2 beschriebenen.

5.5.3 Zufälliges Erzeugen von Graphen

Ein Graph mit n Knoten und m Kanten wird erzeugt, indem ein Graph mit n Knoten und keiner Kante erstellt wird und dann zufällig m verschiedene Kanten hinzugefügt werden.

5.6 Berechnung der Regressionsfunktion

Enum-KROM-SAT Die vom Programm berechnete Regressionsfunktion, welche die Delays mit einem Polynom in Abhängigkeit der Variablenanzahl approximieren soll, wurde wie folgt implementiert:

Zunächst wird für jede Instanz der durchschnittliche Delay berechnet (arithmetisches Mittel aller Delays). Für alle Instanzen mit gleicher Anzahl an Variablen wird dann das arithmetische Mittel über die durchschnittlichen Delays berechnet und als Wert für diese Anzahl an Variablen gesetzt. Für diese Wertepaare ist nun ein Polynom gesucht, das sie möglichst genau wiedergibt. Da die Werte fehlerbehaftet oder leichten Ungenauigkeiten (Messfehler) ausgesetzt sein können, ist es nicht sinnvoll, eine Interpolation genau durch die Punkte durchzuführen. Aus diesem Grund wird ein Polynom maximal möglichen Grades mit der Methode der kleinsten Quadrate angenähert (siehe hierzu [Spä94]).

EnumConnComp Die vom Programm berechnete Regressionsfunktion, welche die Laufzeiten mit einem Polynom in Abhängigkeit der Knotenanzahl approximieren soll, wurde wie folgt implementiert:

Für alle Instanzen mit gleicher Anzahl an Variablen wird das arithmetische Mittel über die gemessene Laufzeit berechnet und als Wert für diese Anzahl an Variablen gesetzt. Für diese Wertepaare wird nun wie oben bereits beschrieben ein Polynom mittels der Methode der kleinsten Quadrate erstellt.

5.7 Einen neuen Algorithmus hinzufügen

Der Programmcode ist modular aufgebaut. Für jede Problemart gibt es ein *package*, welches in die *subpackages* `data` und `algorithms` aufgeteilt ist, um die Datenstrukturen von den Algorithmen zu trennen. Dies erleichtert das Einfügen, Erweitern und/oder Überschreiben neuer Algorithmen zu bereits vorhandenen Datenstrukturen. Die Struktur des Programms ist in Abbildung 13 als Klassenübersicht dargestellt. Die packages werden hierbei durch gestrichelte Linien voneinander abgegrenzt.

Um einen neuen Enumerationsalgorithmus hinzuzufügen, sollte von der abstrakten Klasse `EnumerationAlgorithm` geerbt werden. Diese Klasse enthält bereits Methoden zum Speichern und Ausgeben von Lösungen sowie zum Messen der Zeiten, was den Vorteil einheitlicher Aufrufe bringt.

Um den neuen Algorithmus von der Kommandozeile aus aufzurufen, muss in der `if`-Kaskade der `main`-Methode ein weiterer Fall für den neuen Algorithmus hinzugefügt und dieser dort entsprechend aufgerufen werden.

Die Graphik ist mit JavaFX erstellt. Für jede `Scene` bzw. `Stage` – also im Prinzip jedes Fenster – gibt es eine *.`fxml`-Datei, die gut von den bereits bestehenden Dateien kopiert und angepasst werden kann. Zu jeder *.`fxml`-Datei gibt es einen Controller, in dem die entsprechenden Reaktionen auf Ereignisse der graphischen Oberfläche implementiert sind.

Um eine neu erstellte Scene für einen neuen Algorithmus aufzurufen, kann das Grid-Layout des Hauptmenüs in der Datei `ProblemOverview.fxml` um eine Zeile mit einem Button erweitert werden und das Klick-Event analog zu den anderen in der Datei `ProblemOverviewController.java` bereits vorhandenen behandelt werden.

Beispiel 12. Um beispielsweise einen Enumerationsalgorithmus für Hornformeln hinzuzufügen, ist Folgendes zu tun:

1. Überlegen, was für eine Datenstruktur verwendet werden soll. Es kann beispielsweise die Klasse `CNF` verwendet werden, oder aber eine ganz eigene Klasse im package `logic.data` angelegt werden.

2. Den Algorithmus schreiben: Eine Klasse `EnumHorn` anlegen, die von `EnumerationAlgorithm` erbt und die entsprechenden, vererbten Methoden ausfüllen. Zum Ausgeben der Lösungen wird die Methode `printSolution` verwendet.

3. Den Algorithmus aufrufbar machen:
   - In der Kommandozeile: In der `main`-Methode in der Klasse `EnumerationSolver` die gewünschten Parameter in der bereits vorhandenen **if**-Kaskade ergänzen und den Algorithmus wie gewünscht starten.
   - In der graphischen Oberfläche: In der Datei `ProblemOverview.fxml` im GridLayout einen weiteren Button hinzufügen und in der Klasse `ProblemOverviewController` das entsprechende Event behandeln (analog zu den bereits vorhandenen Buttons). Hier kann beispielsweise eine eigene *.`fxml`-Datei geladen werden. Der Controller hierfür muss selbst implementiert werden, kann sich aber gut an den beiden bereits vorhandenen orientieren.

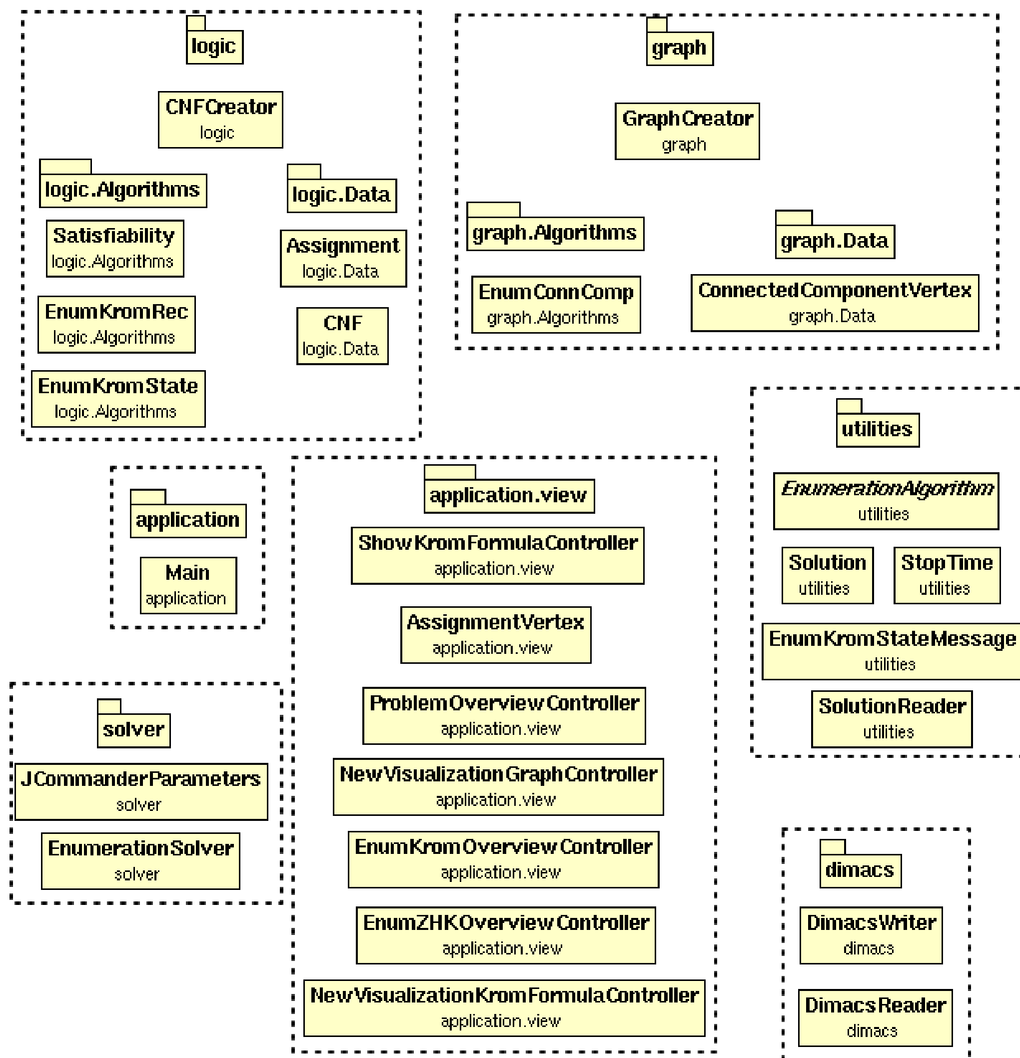

Abbildung 13: Ein Überblick über die Klassen des Programms.

6 Zusammenfassung

Zu Beginn der Arbeit wurde gezeigt, dass KromSAT $\in$ P gilt. Darauf basierend wurde ein Enumerationsalgorithmus für Enum-KROM-SAT angegeben, dessen Korrektheit bewiesen und Laufzeit berechnet wurde. Im weiteren Verlauf wurde ein Enumerationsalgorithmus für EnumConnComp angegeben und ebenso seine Korrektheit bewiesen sowie die Laufzeit berechnet.

Weiter wurde ein Tool für Enumerationsalgorithmen entwickelt, welches sich auf einfache Weise um neue Algorithmen erweitern lässt. Der Quellcode ist modular und übersichtlich gestaltet, ein neuer Algorithmus kann leicht mit Hilfe von Vererbung hinzugefügt werden.

Die bereits vorhandenen Algorithmen können im Programm visualisiert werden. Bei Enum-KROM-SAT kann der Benutzer beispielsweise sehen, wie sich der Belegungsbaum aufbaut und in welchem Zustand sich die einzelnen Knoten befinden. Außerdem können größere Mengen an Testdateien von den Algorithmen verarbeitet und die Ergebnisse vom Tool ausgewertet werden. Dabei wird aus den Daten eine polynomielle Regressionsfunktion berechnet.

Implementiert sind bereits der angegebene EnumerableP-Algorithmus für das Problem EnumConnComp sowie der DelayP-Algorithmus für das Problem Enum-KROM-SAT. Mit Hilfe des entwickelten Tools haben wir die Delays bzw. Laufzeiten für Instanzen dieser Probleme mit verschiedenen Problemgrößen gemessen und mit der Methode der kleinsten Quadrate ein approximierendes Polynom gefunden. Die gemessenen Werte blieben hierbei in allen Fällen unter der theoretischen Schranke. Die bestimmten Polynome enthielten bis zur vierten Potenz Vorfaktoren deutlich größer als 10^{-4}, was von der Größenordnung genau zur theoretisch berechneten Laufzeit passt. Es wurde also gezeigt, dass die theoretischen Zeiten auch bei einer realen Implementierung erreicht werden können.

7 Ausblick

In Zukunft können weitere Enumerationsalgorithmen entwickelt, in das Framework eingebunden und visualisiert werden. Die graphische Gestaltung könnte hier noch etwas modularer gestaltet werden, indem gemeinsame Komponenten aller möglichen Algorithmen analysiert werden und die *.`fxml`-Dateien stufenweise nach Problemart aufgebaut werden. Auch das parallele Abarbeiten mehrerer Instanzen kann weiter optimiert werden, beispielsweise durch dynamisches Anpassen der maximalen Threadanzahl an die Kernanzahl der CPU oder eine Benutzereinstellung hierfür. Des Weiteren kann der Export des erstellten Diagramms um diverse Dateiformate erweitert werden.

Bezüglich der durchgeführten Experimente wäre es interessant, verschiedene andere Variablen-Klausel- beziehungsweise Kanten-Knoten-Verhältnisse zu betrachten und auszuwerten. Auch die Erstellung der Regression kann noch optimiert werden, indem beispielsweise nicht immer die maximal mögliche Potenz gewählt wird, sondern nach der ersten eine zweite Regression mit einem passenderen Wert durchgeführt wird. Dafür wäre es sinnvoll, einen Schwellenwert für die Vorfaktoren zu wählen, ab welchem ein Faktor als „verschwindend gering“ und für die zweite Regression als nicht mehr nötig gewertet wird, sodass hier ein von der Größenordnung besser passendes Polynom gewählt wird.

Literatur

[apaa] Apache batik. https://xmlgraphics.apache.org/batik/. Aufgerufen am 09.09.2015.

[apab] Apache-commons math. https://commons.apache.org/proper/commons-math/. Aufgerufen am 09.09.2015.

[Die12] Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[Hoc10] Winfried Hochstättler. *Algorithmische Mathematik*. Springer-Verlag, 2010.

[jco] Jcommander. http://jcommander.org/. Aufgerufen am 09.09.2015.

[jun] Jung. http://jung.sourceforge.net/. Aufgerufen am 09.09.2015.

[JYP88] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119 – 123, 1988.

[Kro67] M. R. Krom. The decision problem for a class of first-order formulas in which all disjunctions are binary. *Mathematical Logic Quarterly*, 13(1-2):15–20, 1967.

[Mei05] Arne Meier. *SAT-Algorithmen*. Leibniz Universität Hannover, Bachelorarbeit, 2005.

[Mei13] Arne Meier. *Skript zur Vorlesung SAT-Algorithmen*. Leibniz Universität Hannover, 2013.

[MM05] Ryuhei Miyashiro and Tomomi Matsui. A polynomial-time algorithm to find an equitable home–away assignment. *Operations Research Letters*, 33(3):235 – 241, 2005.

[MV15] Arne Meier and Heribert Vollmer. Komplexität von Algorithmen. In *Komplexität von Algorithmen*, pages 20 – 21. lehmanns media, 2015.

[ope] Openjdk. http://openjdk.java.net/projects/jdk8/. Aufgerufen am 09.09.2015.

[Rau09] W. Rautenberg. *Einführung in die Mathematische Logik: Ein Lehrbuch*. Vieweg+Teubner Verlag, 2009.

[SAT] Sat-competition. http://www.satcompetition.org/. Aufgerufen am 06.09.2015.

[SAT93] Satisfiability suggested format. www.satlib.org/Benchmarks/SAT/satformat.ps, 1993. Aufgerufen am 06.09.2015.

[Spä94] Helmuth Späth. *Numerik*. Vieweg, 1994.

Abbildungsverzeichnis

1 Ein ungerichteter Graph mit zwei Zusammenhangskomponenten. 5
2 Der Graph, der durch den Algorithmus für Formel (2) erstellt wird. . . 10
3 Der Graph, der durch den Algorithmus für φ mit der partiellen Belegung θ erstellt wird. Die gepunkteten und gestrichelten Pfeile markieren jeweils einen Pfad, aufgrund dessen der Algorithmus ablehnt. 12
4 Der Belegungsbaum für φ. Blau: φ ist mit θ erfüllbar, θ ist keine vollständige Belegung, rot: φ ist mit θ nicht erfüllbar/erfüllt, grün: θ ist eine erfüllende Belegung für φ. 16
5 Belegungsbaum für eine Formel mit 4 Variablen: grün: Belegung wurde bereits betrachtet, gelb: Aktuell betrachtete Belegung, blau: Belegung wird auf dem Weg zur nächsten Lösung überprüft, rot: Belegung wird nach der nächsten Belegung betrachtet. 18
6 Belegungsbaum für eine Formel mit 3 Variablen. Visualisierung für Fall 2 der Laufzeitabschätzung. 22
7 Ergebnisse von Experiment 1 in einem Diagramm. 25
8 Ergebnisse von Experiment 2 in einem Diagramm. 27
9 Ergebnisse von Experiment 3 in einem Diagramm. 34
10 Ergebnisse von Experiment 4 in einem Diagramm. 36
11 Ergebnisse von Experiment 5 in einem Diagramm. 37
12 Der Graph aus Listing 3. 44
13 Ein Überblick über die Klassen des Programms. 47

Tabellenverzeichnis

1 Junktoren . . . 3
2 Implikationen, die sich aus Formel (2) ergeben. . . . 10
3 Implikationen, die sich aus Formel (3) ergeben. . . . 12
4 Testinstanzen für Experiment 1 (Variablen-Klausel-Verhältnis nahe 2). 23
5 Testinstanzen für Experiment 2 (Variablen-Klausel-Verhältnis nahe $|V|$). 24
6 Testergebnisse für Experiment 1 (Variablen-Klausel-Verhältnis nahe 2). 24
7 Testergebnisse für Experiment 2 (Variablen-Klausel-Verhältnis nahe $|V|$). 26
8 Testinstanzen für Experiment 3. . . . 32
9 Testinstanzen für Experiment 4. . . . 33
10 Testinstanzen für Experiment 5. . . . 33
11 Testinstanzen für Experiment 3 (Knoten-Kanten-Verhältnis fast konstant). 34
12 Testinstanzen für Experiment 4 (Knoten-Kanten-Verhältnis fast konstant). 35
13 Testinstanzen für Experiment 5 (Knoten-Kanten-Verhältnis linear). . . 37
14 Parameter des Programms. . . . 40

Liste der Algorithmen

1 KromSAT 9
2 KromSAT 11
3 EnumKromSAT 15
4 EnumConnComp 29