Der Davis-Putnam-Algorithmus

Hauptseminar angewandte Logik bei Prof. Dr. Koepke

Vortrag von Hannah Scholz am 09.11.2023

1. Einleitung

1.1 Zielsetzung

- **Ziel**: Algorithmus, der folgendes leistet:

gesuchter Algorithmus	
Input:	Formel in konjunktiver Normalform
Output:	Erfüllbarkeit der Formel

- **Lösung**: Davis-Putnam-Algorithmus (DP, 1960) oder Davis-Putnam-Logemann-Loveland (DPLL, 1962)
- Zum Testen der Algorithmen: prime(x)

1.2 Grundlegende Definitionen und Konventionen

- Formel p ist **erfüllbar**: Es existiert eine Belegung der atomaren Formeln v sodass eval p v = true
- **Formel** (hier): Formel in konjunktiver Normalform dargestellt als Menge von Mengen
- **Klausel**: Element einer Formel
- **Literal**: Element einer Klausel (also eine atomare Formel oder ihre Negation)
- Formel $\top$: Dargestellt durch eine leere Formel
- Formel $\bot$: Dargestellt durch Formeln, die eine leere Klausel enthalten

2. Davis-Putnam

2.1 Grundidee des Algorithmus

- **Idee**: Die Formel umformen, bis wir ihre Erfüllbarkeit erkennen, wobei in jedem Schritt die Erfüllbarkeit der neuen Formel zu der der ursprünglichen äquivalent sein muss
- Erreichen wir die Formel $\top$, so war die ursprüngliche Formel erfüllbar
- Erreichen wir $\bot$, so war sie es nicht
- Drei verschiedene Regeln zur Umformung der Formel

2.2 Regel II: Pure-Literal-Regel

- **Idee**: Kommt ein Literal entweder nur positiv oder nur negativ vor, so können wir alle Klauseln entfernen, die dieses Literal enthalten

Satz (Äquivalenz der Erfüllbarkeit Regel II)
Sei S unsere ursprüngliche Formel und S' unsere Formel nach der Anwendung der Pure-Literal-Regel. Dann gilt:
S erfüllbar $\iff$ S' erfüllbar

- Regel II verringert die Anzahl der Klauseln und Literale, wir sollten sie also so oft wie möglich anwenden

affirmative_negative_rule	
Input:	Formel als Liste von Listen "clauses"
Output:	Formel, auf die die Pure-Literal-Regel auf jedes pure Literal einmal angewendet wurde (wenn dies nicht möglich: failure)

2.3 Regel III: Regel zur Eliminierung atomarer Formeln

- **Idee**: Sei S eine Menge von Klauseln und x eine atomare Formel, dann können wir S wie folgt aufteilen: $S = \{x \vee C_i \mid 1 \leq i \leq m\} \cup \{\neg x \vee D_j \mid 1 \leq j \leq n\} \cup S_0$
wobei die C_i's und D_j's weder x noch $\neg x$ enthalten und die Klauseln in S_0 entweder x und $\neg x$ oder keins von beiden enthalten. Dann bilden wir $S' = \{C_i \vee D_j \mid 1 \leq i \leq m\,,\, 1 \leq j \leq n\} \cup S_0$.

Satz (Äquivalenz der Erfüllbarkeit Regel III)
Es gilt: S ist erfüllbar $\iff$ S' ist erfüllbar

- Regel III kann die Anzahl der Klauseln stark vergrößern, wir sollten sie also nur selten einsetzen

resolve_on	
Input:	Formel als Liste von Listen "clauses", Literal p
Output:	Regel III ausgeführt für atomare Formel x (für p=x oder p=¬x)

- Zum Aussuchen eines Literals nutzen wir folgende Funktion, die den Zuwachs an Klauseln bei der Anwendung von Regel III auf ein Literal approximiert

resolution_blowup	
Input:	Formel als Liste von Listen "cls" und Literal ℓ
Output:	ungefährer Zuwachs an Klauseln nach Anwendung von Regel III auf cls und ℓ

- Unserer endgültiger Algorithmus für Regel III sieht jetzt so aus:

resolution_rule	
Input:	Formel als Liste von Listen "clauses"
Output:	Regel III angewendet auf clauses und ein Literal, das die resolution_blowup Funktion berechnet

2.4 Regel I: 1-Literal-Regel

- **Idee**: Haben wir eine Klausel, die nur aus einem Literal ℓ besteht, so entfernen wir alle Klauseln, die ℓ enthalten (auch unsere einelementige Klausel) und entfernen $\neg\ell$ aus allen verbleibenden Klauseln

Lemma
Sei S eine Formel von der Form $S = S_0 \cup \{\ell\} \cup \{\ell \vee C_i \mid 1 \leq i \leq n\}$ wobei die Klauseln in S_0 nur $\neg\ell$ oder weder ℓ noch $\neg\ell$ enthalten. Sei $S' = S_0 \cup \{\ell\}$. Dann gilt: S erfüllbar $\iff$ S' erfüllbar

Korollar (Äquivalenz der Erfüllbarkeit Regel I)
Die 1-Literal-Regel erhält Erfüllbarkeit.

- Regel I verringert die Anzahl an Literalen und Klauseln, wir sollten sie deshalb so oft wie möglich verwenden

one_literal_rule	
Input:	Formel als Liste von Listen "clauses"
Output:	Regel I einmal angewendet auf "clauses"" (wenn nicht möglich: failure)

2.5 Der Davis-Putnam Algorithmus

- Wir programmieren den DP Algorithmus als rekursiven Algorithmus

dp	
Input:	Formel als Liste von Listen "clauses"
Output:	Wahrheitswert, der angibt, ob clauses erfüllbar ist oder nicht

- Der Algorithmus terminiert, da wir bei jeder Regel die Anzahl an verschiedenen atomaren Formeln verringern
- Wir können unseren Algorithmus verwenden, um neue Funktionen zu schreiben, die Erfüllbarkeit und Tautologien überprüfen:

```
let dpsat fm = dp(defcnfs fm);;
let dptaut fm = not(dpsat(Not fm));;
```

3. Davis-Putnam-Logemann-Loveland

3.1 Grundidee des Algorithmus

- **Problem des DP-Algorithmus**: Hoher Speicheraufwand durch die entstehenden Klauseln bei Regel III
- **Lösung**: Ersetzen von Regel III durch die Aufteilungsregel

3.2 Aufteilungsregel

- **Idee**: Für eine Formel Δ und ein Literal p testen wir $\Delta \cup \{\{p\}\}$ und $\Delta \cup \{\{\neg p\}\}$ separat auf Erfüllbarkeit

Satz (Äquivalenz der Erfüllbarkeit Aufteilungsregel)
Δ ist erfüllbar $\iff$ $\Delta \cup \{\{p\}\}$ oder $\Delta \cup \{\{\neg p\}\}$ ist erfüllbar

- Die entstehende Rekursion terminiert, da wir nach der Aufteilungsregel die Regel I anwenden, wobei das Literal p entfernt wird und wir so die Anzahl an Literalen verringern.
- Wir müssen wieder ein Literal für die Aufteilungsregel aussuchen.
- **Idee**: Wir wählen die atomare Formel, die am häufigsten vorkommt. So werden maximal viele Klauseln und Literale aus Klauseln entfernt.

posneg_count	
Input:	Formel als Liste von Listen “cls” und Literal ℓ
Output:	Summe der Anzahl an Klauseln und Literale aus Klauseln, die bei der folgenden Anwendung von Regel I entfernt werden würden

3.3 Der Davis-Putnam-Logemann-Loveland Algorithmus

dpll	
Input:	Formel als Liste von Listen “clauses”
Output:	Wahrheitswert, der angibt, ob clauses erfüllbar ist oder nicht

- Wir können unseren Algorithmus wieder verwenden, um neue Funktionen zu schreiben, die Erfüllbarkeit und Tautologien überprüfen:

```
let dpllsat fm = dpll(defcnfs fm);;
let dplltaut fm = not(dpllsat(Not fm));;
```

4. Iterativer Davis-Putnam-Logemann-Loveland

4.1 Grundidee des Algorithmus

- **Problem des DPLL-Algorithmus**: Auch unsere neue Aufteilungsregel verbraucht sehr viel Speicherplatz
- **Lösung**: Wir programmieren den DPLL-Algorithmus iterativ statt rekursiv
- Dazu speichern wir die Literale, die wir auf true setzten, in einem Pfad und merken uns auch, wie wir auf dieses Belegung gekommen sind
- Datentyp für Zustände der Literale im Pfad:

```
type trailmix = Guessed | Deduced;;
```

- Der Pfad ist eine Liste von Paare aus einem Literal und einem Zustand; die neuste Information steht ganz vorne
- Wir bearbeiten unsere Formel jetzt nicht mehr direkt sondern speichern die Zuweisungen im Pfad
- Zunächst schreiben wir nun eine Funktion, die uns die noch nicht zugewiesenen atomaren Formeln ausgibt

unassigned	
Input:	Formel als Liste von Listen “cls” und Pfad “trail”
Output:	Menge aller atomaren Formeln, denen im Pfad noch kein Wert zugewiesen wurde

4.2 Überarbeitung der Regel I

- Bei Regel I würden wir gerne trotzdem noch intern die Formel direkt bearbeiten
- Außerdem erstellen wir eine partielle Funktion fn, die uns für ein Literal aus dem Pfad “()” zurück gibt

unit_subpropagate	
Input:	Formel als Liste von Listen “cls”, partielle Belegungsfunktion “fn” und Pfad “trail”
Output:	cls’, fn’, trail’ nachdem Regel I so oft wie möglich durchgeführt wurde

- Aus unit_subpropagate erstelle wir jetzt die eigentliche Funktion für Regel I:

unit_propagate	
Input:	Formel als Liste von Listen “cls” und Pfad “trail”
Output:	Veränderte Formel cls’ und Pfad trail’ nachdem Regel I so oft wie möglich angewendet wurde

4.3 Backtracking

- Falls wir aus einem geratenen Literal einen Widerspruch erhalten, so müssen wir zu diesem Punkt zurück und die andere Möglichkeit überprüfen

backtrack	
Input:	Pfad “trail”
Output:	Pfad “trail” bis zur letzten geratenen Variable

4.4 Der iterative DPPL-Algorithmus

- Regel II implementieren wir der Einfachheit halber nicht

dpli	
Input:	Formel als Liste von Listen “cls” und Pfad “trail”
Output:	Wahrheitswert der angibt, ob cls erfüllbar ist oder nicht

- Daraus können wir jetzt wieder Funktionen schreiben, die Formeln auf Erfüllbarkeit und Tautologie überprüfen:

```
let dplisat fm = dpli (defcnfs fm) [];;
let dplitaut fm = not(dplisat(Not fm));;
```

4.5 Mögliche Verbesserungen

- Optimisierung der Datenstruktur
- **Backjumping** statt Backtracking: Nicht zur letzten Fallunterscheidung zurück sondern zur frühsten, die immer noch mit der letzten Entscheidung einen Widerspruch produziert
- **Learning**: Wenn wir wissen, dass die Literale a und b zusammen einen Widerspruch verursachen, können wir eine neue Klasuel $\neg a \vee \neg b$ hinzufügen
- Wir können dann den Algorithmus “dpli” zu “dplb” verändern und erhalten wieder:

```
let dplbsat fm = dplb (defcnfs fm) [];;
let dplbtaut fm = not(dplbsat(Not fm));;
```