Der Davis-Putnam-Algorithmus

Hannah Scholz

Hauptseminar Angewandte Logik

09.11.2023

Gliederung

Einleitung

- **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)

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

Davis-Putnam (1)

- **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

Davis-Putnam (2)

- Drei verschiedene Regeln zur Umformung der Formel:

3 Umformungsregeln des Davis-Putnam-Algorithmus

- Regel I: 1-Literal-Regel
- Regel II: Pure-Literal-Regel
- Regel III: Regel zur Eliminierung einer atomaren Formel

Regel II: Pure-Literal-Regel (1)

- **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

Regel II: Pure-Literal-Regel (2)

OCaml-Code (affirmative_negative_rule)

```
0  let affirmative_negative_rule clauses =
1      let neg',pos = partition negative (unions clauses) in
2      let neg = image negate neg' in
3      let pos_only = subtract pos neg and neg_only = subtract neg pos in
4      let pure = union pos_only (image negate neg_only) in
5      if pure = [] then failwith "affirmative_negative_rule" else
6      filter (fun cl -> intersect cl pure = []) clauses;;
```

Pseudocode (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)

1 Sei neg' $=$ [Literal $\ell \mid \exists x : \neg x = \ell \wedge \ell \in \bigcup$ clauses] und pos$=$ [Literal $\ell \mid \exists x : x = \ell \wedge \ell \in \bigcup$ clauses]
2 Sei neg $= \{\neg\ell \mid \ell \in$ neg'$\}$
3 Sei pos_only $=$ pos$\backslash$neg und neg_only $=$ neg$\backslash$pos
4 Sei pure $=$ pos_only $\cup \{\neg\ell \mid \ell \in$ neg_only$\}$
5 Falls pure $= \varnothing$ dann failure "affirmative_negative_rule" sonst
6 Gib [cl $\in$ clauses $\mid$ cl $\cap$ pure $= \varnothing$] aus

Regel III: Regel zur Eliminierung atomarer Formeln (1)

- **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

Regel III: Regel zur Eliminierung atomarer Formeln (2)

OCaml-Code (resolve_on)

```
0  let resolve_on p clauses =
1      let p' = negate p and pos,notpos = partition (mem p) clauses in
2      let neg,other = partition (mem p') notpos in
3      let pos' = image (filter (fun l -> l <> p)) pos
4      and neg' = image (filter (fun l -> l <> p')) neg in
5      let res0 = allpairs union pos' neg' in
6      union other (filter (non trivial) res0);;
```

Pseudocode (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)

1 Sei $p' = \neg p$, pos = $[\text{cl} \in \text{clauses} \mid p \in \text{cl}]$ und notpos = $[\text{cl} \in \text{clauses} \mid p \notin \text{cl}]$
2 Sei neg = $[\text{cl} \in \text{notpos} \mid p' \in \text{cl}]$ und other = $[\text{cl} \in \text{notpos} \mid p' \notin \text{cl}]$
3 Sei pos' = $\{\text{cl} \setminus \{p\} \mid \text{cl} \in \text{pos}\}$
4 neg' = $\{\text{cl} \setminus \{\neg p\} \mid \text{cl} \in \text{neg}\}$
5 Sei res0 = $[\text{pcl} \cup \text{ncl} \mid \text{pcl} \in \text{pos}', \text{ncl} \in \text{neg}']$
6 Gib other $\cup$ $[\text{cl} \in \text{res0} \mid \neg(\text{trivial cl})]$ aus

Regel III: Regel zur Eliminierung atomarer Formeln (3)

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

OCaml-Code (resolution_blowup)

```
0  let resolution_blowup cls l =
1      let m = length(filter (mem l) cls)
2      and n = length(filter (mem (negate l)) cls) in
3      m * n - m - n;;
```

Pseudocode (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

1 Sei $m = |\,[\text{cl} \in \text{cls} \mid \ell \in \text{cl}]\,|$
2 Sei $n = |\,[\text{cl} \in \text{cls} \mid \neg\ell \in \text{cl}]\,|$
3 Gib m·n-m-n aus

Regel III: Regel zur Eliminierung atomarer Formeln (4)

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

OCaml-Code (resolution_rule)

```
0  let resolution_rule clauses =
1      let pvs = filter positive (unions clauses) in
2      let p = minimize (resolution_blowup clauses) pvs in
3      resolve_on p clauses;;
```

Pseudocode (resolution_rule)

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

1 Sei pvs $= \{\ell \in \bigcup \text{clauses} \mid \ell = x$ für x atomare Formel$\}$
2 Sei $p \in$ pvs so, dass resolution_blowup(clauses, p) minimal ist
3 Wende Regel III auf clauses und p an und gebe das Ergebnis aus

Regel I: 1-Literal-Regel (1)

- **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

Regel I: 1-Literal-Regel (2)

OCaml-Code (one_literal_rule)

```
0  let one_literal_rule clauses =
1      let u = hd (find (fun cl -> length cl = 1) clauses) in
2      let u' = negate u in
3      let clauses1 = filter (fun cl -> not (mem u cl)) clauses in
4      image (fun cl -> subtract cl [u']) clauses1;;
```

Pseudocode (one_literal_rule)

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

1 Sei u das Element der ersten einelementigen Klausel in clauses
(wenn das nicht existiert: failure)
2 Sei $u' = \neg u$
3 Sei clauses1 $= [\text{cl} \in \text{clauses} \mid u \notin \text{clauses}]$
4 Gib $\{\text{cl} \backslash \{u'\} \mid \text{cl} \in \text{clauses1}\}$ aus

Der Davis-Putnam Algorithmus (1)

- Wir programmieren den DP Algorithmus als rekursiven Algorithmus

OCaml-Code (dp)

```
0  let rec dp clauses =
1      if clauses = [] then true else if mem [] clauses then false else
2      try dp (one_literal_rule clauses) with Failure _ ->
3      try dp (affirmative_negative_rule clauses) with Failure _ ->
4      dp(resolution_rule clauses);;
```

Pseudocode (dp)

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

1 Falls clauses $= \varnothing$, gib true zurück. Falls $\varnothing \in$ clauses, gib false zurück.
2 Falls keins von beiden zutrifft, versuche erst Regel I anzuwenden und danach den Algorithmus rekursiv wieder aufzurufen
3 Wird dabei failure erhalten, so versuche erst Regel II anzuwenden und danach den Algorithmus rekursiv wieder aufzurufen
4 Wird dabei failure erhalten, so wende Regel III an und rufe danach den Algorithmus rekursiv wieder auf

Der Davis-Putnam Algorithmus (2)

- 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:

OCaml-Code (dpsat, dptaut)

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

Davis-Putnam-Logemann-Loveland

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

Aufteilungsregel (1)

- **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

Aufteilungsregel (2)

- Wir müssen wieder ein Literal für die Auteilungsregel aussuchen
- **Idee**: Wir wählen die atomare Formel, die am häufigsten vorkommt. So werden maximal viele Klauseln und Literale aus Klauseln entfernt.

OCaml-Code (posneg_count)

```
0  let posneg_count cls l =
1      let m = length(filter (mem l) cls)
2      and n = length(filter (mem (negate l)) cls) in
3      m + n;;
```

Pseudocode (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

1 Sei $m = |\,[\text{cl} \in \text{cls} \mid \ell \in \text{cl}]\,|$
2 Sei $n = |\,[\text{cl} \in \text{cls} \mid \neg\ell \in \text{cl}]\,|$
3 Gib m+n aus

Der Davis-Putnam-Logemann-Loveland Algorithmus (1)

OCaml-Code (dpll)

```
0  let rec dpll clauses =
1      if clauses = [] then true else if mem [] clauses then false else
2      try dpll(one_literal_rule clauses) with Failure _ ->
3      try dpll(affirmative_negative_rule clauses) with Failure _ ->
4      let pvs = filter positive (unions clauses) in
5      let p = maximize (posneg_count clauses) pvs in
6      dpll (insert [p] clauses) or dpll (insert [negate p] clauses);;
```

Pseudocode (dpll)

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

1 Falls clauses $= \varnothing$, gib true zurück. Falls $\varnothing \in$ clauses, gib false zurück.
2 Falls keins von beiden zutrifft, versuche erst Regel I anzuwenden und danach den Algorithmus rekursiv wieder aufzurufen
3 Wird dabei failure erhalten, so versuche erst Regel II anzuwenden und danach den Algorithmus rekursiv wieder aufzurufen
4 Wird dabei failure erhalten, so setze pvs $= \{\ell \in \bigcup \text{clauses} \mid \ell = x$ für x atomare Formel $\}$
5 Sei $p \in$ pvs so, dass posneg_count(clauses, p) maximal ist
6 Gib dpll(clauses$\cup\{\{p\}\}$)$\vee$dpll(clauses$\cup\{\{\neg p\}\}$) aus

Der Davis-Putnam-Logemann-Loveland Algorithmus (2)

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

OCaml-Code (dpllsat, dplltaut)

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

Iterativer DPLL (1)

- **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

Iterativer DPLL (2)

- 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

OCaml-Code (unassigned)

```
0  let unassigned =
1      let litabs p = match p with Not q -> q | _ -> p in
2      fun cls trail -> subtract (unions(image (image litabs) cls))
3      (image (litabs ** fst) trail);;
```

Pseudocode (unassigned)

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

1 Definiere für ein Literal p litlabs(p) $= q$ falls $p = \neg q$ für eine atomare Formel q sonst litlabs(p)$=p$

2 Gib $(\bigcup\{\{\text{litlabs}(\ell) \mid \ell \in \text{cl}\} \mid \text{cl} \in \text{clauses}\}) \setminus \{\text{litlabs}(\ell) \mid \ell \text{ ist Literal im trail}\}$ aus

Iterativer DPLL - Überarbeitung Regel I (1)

- 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

Iterativer DPLL - Überarbeitung Regel I (2)

OCaml-Code (unit_subpropagate)

```
0  let rec unit_subpropagate (cls,fn,trail) =
1      let cls' = map (filter ((not) ** defined fn ** negate)) cls in
2      let uu = function [c] when not(defined fn c) -> [c] |_ -> failwith "" in
3      let newunits = unions(mapfilter uu cls') in
4      if newunits = [] then (cls',fn,trail) else
5      let trail' = itlist (fun p t -> (p,Deduced)::t) newunits trail
6      and fn' = itlist (fun u -> (u |-> ())) newunits fn in
7      unit_subpropagate (cls',fn',trail');;
```

Pseudocode (unit_subpropagate)

Input: Formel als Liste von Listen "cls", Belegungsfunktion "fn" und Pfad "trail"
Output: cls', fn', trail' nachdem Regel I so oft wie möglich durchgeführt wurde

1 Sei "defined_fn" die Funktion, die uns angibt, ob fn für ein Literal definiert ist. Sei dann cls' $=$ [[$\ell \in$ cl | defined_fn($\neg\ell$) $=$ false] | cl $\in$ cls]
2 Definiere die Funktion uu([c])=[c], wenn defined_fn(c)=false, sonst failure
3 Sei newunits $= \bigcup$\{cl $\in$ cls' | uu(cl) kein failure\}
4 Falls newunits $= \varnothing$ gib (cls',fn,trail) aus
5 Wir bilden trail' aus trail, indem wir für jedes p in newunits (p, Deduced) vorne hinzufügen
6 Wir erweitern fn zu fn' indem wir für jedes u in newunits fn'(u) auf () setzen
7 Rufe dann rekursiv unit_subpropogate(cls',fn',trail') auf

Iterativer DPLL - Überarbeitung Regel I (3)

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

OCaml-Code (unit_propagate)

```
0  let unit_propagate (cls,trail) =
1      let fn = itlist (fun (x,_) -> (x |-> ())) trail undefined in
2      let cls',fn',trail' = unit_subpropagate (cls,fn,trail) in cls',trail';;
```

Pseudocode (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

1 Sei fn die partielle Funktion, die jedes Literal im trail auf () schickt
2 Sei (cls',fn',trail') = unit_subpropagate(cls,fn,trail) und gebe cls' und trail' zurück

Backtracking

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

OCaml-Code (backtrack)

```
0  let rec backtrack trail =
1      match trail with
2      (p,Deduced)::tt -> backtrack tt
3      | _ -> trail;;
```

Pseudocode (backtrack)

Input: Pfad "trail"
Output: Pfad "trail" bis zur letzten geratenen Variable

1 Falls trail $=$
2 [(p,deduced)] $\cup$ tt, rufe rekursiv backtrack(tt) auf und gebe das Ergebnis aus
3 Gebe sonst trail zurück

- Regel II implementieren wir der Einfachheit halber nicht

Der iterative Davis-Putnam-Logemann-Loveland (1)

OCaml-Code (dpli)

```
0   let rec dpli cls trail =
1       let cls',trail' = unit_propagate (cls,trail) in
2       if mem [] cls' then match backtrack trail with
3           (p,Guessed)::tt -> dpli cls ((negate p,Deduced)::tt)
4           | _ -> false
5       else match unassigned cls trail' with
6           [] -> true
7           | ps -> let p = maximize (posneg_count cls') ps in
8                   dpli cls ((p,Guessed)::trail');;
```

Pseudocode (dpli)

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

1 Sei cls',trail' = unit_propagate(cls,trail)
2-4 Falls $\varnothing \in$ cls' überprüfe ob backtrack(trail) = [(p,Guessed)] $\cup$ tt. Falls ja gebe dpli(cls, [($\neg$p, Deduced)]$\cup$ tt) zurück. Falls nein gebe false zurück.
5-6 Sonst falls unassigned(cls, trail') = $\varnothing$ gib true zurück.
7-8 Falls dies nicht so ist, sei ps = unassigned(cls, trail') und sei p das Literal aus ps sodass posneg_count maximal ist gebe dann dpli(cls, [(p,Guessed)]$\cup$trail') zurück

Der iterative Davis-Putnam-Logemann-Loveland (2)

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

OCaml-Code (dplisat, dplitaut)

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

Mögliche Verbesserungen

- Optimierung 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:

OCaml-Code (dplisat, dplitaut)

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

Aufgabe

Aufgabe: Oft möchte man nicht nur wissen, ob eine Formel erfüllbar ist, sondern auch bei Erfüllbarkeit ein Beispiel kennen. Bearbeite den (nicht-iterativen) DPLL-Algorithmus deshalb so, dass bei Erfüllbarkeit auch eine mögliche Belegung der atomaren Formeln gut lesbar ausgegeben wird.

Danke für die Aufmerksamkeit!