Completeness Theorem und DPP verfeinert

22. März 2016

Inhaltsverzeichnis

Definition

Eine *resolution refutation* zu einer Formel Γ ist eine endliche Kette von Klauseln $A_1, \ldots, A_k$, derart, dass für alle Glieder A_i der Kette gilt: $A_i \in \Gamma$ oder A_i wurde von vorherigen Gliedern der Kette mit Hilfe der Resolutionsregel abgeleitet. Außerdem muss gelten A_k ist die leere Klausel.

Beispiel

$$\Gamma = \{\{A, B, \neg C\}, \{A, D, C\}, \{A, \neg B, D\}, \{\neg D\}, \{\neg A\}$$

Theorem

Vollständigkeitstheorem für die Aussagenlogik

Wenn Γ *eine unerfüllbare Klauselmenge ist, dann existiert eine resolution refutation von* Γ*.*

Theorem

Vollständigkeitstheorem für die Aussagenlogik

Wenn Γ *eine unerfüllbare Klauselmenge ist, dann existiert eine resolution refutation von* Γ*.*

Theorem

Kompaktheitstheorem der Aussagenlogik

Sei Γ *Menge von Formeln.*

(1) Γ *ist erfüllbar gdw. jede endliche Teilmenge von* Γ *erfüllbar ist.*

(2) $\Gamma \models A$ *gdw. es eine endliche Teilmenge* Γ_0 *gibt, sodass* $\Gamma_0 \models A$.

DPLL

Problem bei DPP

Eventuell hoher Speicherbedarf bei DPP durch Resolutionsverfahren!

DPLL

Lösung

DPLL *(Davis-Putnam-Logemann-Loveland)*

- Erweitern der Resolutionsregel durch eine *Splitting Regel*
- Überprüfung der Klauselmengen

$$S \cup \{p\} \text{ und } S \cup \{\neg p\}$$

wobei S der Resolvent im Bezug auf p ist.

DPLL

```
let rec dpll clauses =
if clauses = [ ] then true else if mem [ ] clauses then false else
try dpll(one_literal_rule clauses) with Failure _ →
try dpll(affirmative_negative_rule clauses) with Failure _ →
let pvs = filter positive (unions clauses) in
let p = maximize (posneg_count clauses) pvs in dpll (insert [p]
clauses) or dpll (insert [negate p] clauses);;
```

DPLL

```
let rec dpll clauses =
if clauses = [] then true else if mem [] clauses then false else
try dpll(one_literal_rule clauses) with Failure _ →
try dpll(affirmative_negative_rule clauses) with Failure _ →
let pvs = filter positive (unions clauses) in
let p = maximize (posneg_count clauses) pvs in dpll (insert [p]
clauses) or dpll (insert [negate p] clauses);;
```

Aber:

Gegebenenfalls hohe Stacks durch Rekursion

Iteratives DPLL

- In modernen Implementierungen: iterative Kontrollstruktur zur Reduzierung redundanter Informationen im Speicher
- In der Kontrollstruktur: Speichern der bisher angenommenen/berechneten Variablen samt Herkunft
- Momentan betrachtete Variable: oben (LIFO)

Iteratives DPLL

Neuer Datentyp zum Markieren von Variablen:

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

Auflisten potentieller Variablen zum Case-Split:

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

Iteratives DPLL

Anpassung von unit_propagate an die neue Situation:

```
let rec unit_subpropagate (cls,fn,trail) =
let cls' = map (filter ((not) ** defined fn ** negate)) cls in
let uu = function [c] when not(defined fn c) → [c] |_ → failwith  in
let newunits = unions(mapfilter uu cls') in
if newunits = [ ] then (cls',fn,trail) else
let trail' = itlist (fun p t → (p,Deduced)::t) newunits trail
and fn' = itlist (fun u → (u |→ ())) newunits fn in
unit_subpropagate (cls',fn',trail');;
```

```
let unit_propagate (cls,trail) =
let fn = itlist (fun (x,_) → (x |→ ())) trail undefined in
let cls',fn',trail' = unit_subpropagate (cls,fn,trail) in cls',trail';;
```

Iteratives DPLL

```
let rec backtrack trail = match trail with
(p,Deduced)::tt → backtrack tt
| _ → trail;;
```

```
let rec dpli cls trail =
let cls',trail' = unit_propagate (cls,trail) in
if mem [ ] cls' then
match backtrack trail with
(p,Guessed)::tt → dpli cls ((negate p,Deduced)::tt)
|_ → false else
match unassigned cls trail' with
→ true | ps → let p = maximize (posneg_count cls')
```

Backjumping und Lernen

Problem

In der Praxis: Etwas langsamer als klassisches DPLL

Lösung

Backjumping und Lernen

Backjumping und Lernen

- Reaktion auf Widersprüche in Abhängigkeit von einer echten Teilmenge der angenommenen Variablen
- Strategien zur Vermeidung der Überprüfung unnötiger Paths vor allem bei nicht erfüllbaren Klauselmengen
- Beim **Backjumping**: *nicht-chronologisches* Annehmen von Variablen
- Beim **Learning**: Hinzufügen sogenannter *Konfliktterme* zur Klauselmenge

Backjumping und Lernen

```
let rec dplb cls trail =
let cls',trail' = unit_propagate (cls,trail) in
if mem [ ] cls' then
match backtrack trail with
(p,Guessed)::tt →
let trail' = backjump cls p tt in
let declits = filter (fun (_,d) → d = Guessed) trail' in
let conflict = insert (negate p) (image (negate ** fst) declits) in
dplb (conflict::cls) ((negate p,Deduced)::trail')
| _ → false
else
match unassigned cls trail' with
[ ] → true
| ps → let p =
maximize (posneg_count cls') ps in
dplb cls ((p,Guessed)::trail');;
```

Weitere Optimierungsmöglichkeiten

- Optimierung der bisherigen Techniken
- Periodische Resets mit gelernten Variablen
- Geschicktere Wahl der Variablen zum Spliten