Propositional Logic

Julian Parsert

November 4, 2016

Table of contents

1. Recap

2. SAT

   Davis-Putnam

   Davis-Putnam-Logemann-Loveland

3. Binary Decision Diagrams

4. Compactness

Recap

Satisfiablility

Definitions [2]

- The formula ϕ is *satisfiable* if it has a valuation in which ϕ evaluates to $\top$.
- Formula ϕ is satisfiable iff $\neg\phi$ is not valid.
- CNF $\equiv \top \iff$ every clause contains an atom p and its negation.
- DNF is satisfiable $\iff$ no clause contains an atom and its negation.

Satisfiablility

Definitions [2]

- The formula ϕ is *satisfiable* if it has a valuation in which ϕ evaluates to $\top$.
- Formula ϕ is satisfiable iff $\neg\phi$ is not valid.
- CNF $\equiv \top \iff$ every clause contains an atom p and its negation.
- DNF is satisfiable $\iff$ no clause contains an atom and its negation.

Satisfiablility

Definitions [2]

- The formula ϕ is *satisfiable* if it has a valuation in which ϕ evaluates to $\top$.
- Formula ϕ is satisfiable iff $\neg\phi$ is not valid.
- CNF $\equiv \top \iff$ every clause contains an atom p and its negation.
- DNF is satisfiable $\iff$ no clause contains an atom and its negation.

Use in SAT-Solving

This leads to...

- We only need to set a valuation for one DNF clause with no contradiction, if there is one.
- But transforming a formula ϕ into its DNF can lead to an exponential blowup of a formula.

Use in SAT-Solving

This leads to...

- We only need to set a valuation for one DNF clause with no contradiction, if there is one.
- But transforming a formula ϕ into its DNF can lead to an exponential blowup of a formula.

Tseitin Transformation

Algorithm that transforms an arbitrary formula ϕ into an equisatisfiable CNF with a *linear* blowup.

SAT

Davis-Putnam

One-literal rule

A clause containing one element can only be satisfied if that element evaluates to true.

Davis-Putnam

One-literal rule

A clause containing one element can only be satisfied if that element evaluates to true.

Affirmative-negative rule

If a literal p occurs only positively or negatively. $v(p)$ can be set to true without changing the satisfiablility of the formula.

Davis-Putnam

One-literal rule

A clause containing one element can only be satisfied if that element evaluates to true.

Affirmative-negative rule

If a literal p occurs only positively or negatively. $v(p)$ can be set to true without changing the satisfiablility of the formula.

rule for eliminating atomic formulas

- Performs resolution on set of clauses.
- Can lead to a significant increase in size of CNF.

Code

Davis-Putnam procedure from lecture book

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


let resolve_on p clauses =
  let p' = negate p and pos,notpos = partition (mem p)
                        clauses in
  let neg,other = partition (mem p') notpos in
  let pos' = image (filter (fun l -> l <> p)) pos
  and neg' = image (filter (fun l -> l <> p')) neg in
  let res0 = allpairs union pos' neg' in
  union other (filter (non trivial) res0)
```

But why?

$$\frac{\{p \vee C_1 \ldots\} \qquad \{\neg p \vee D_1 \ldots\}}{C_1 \vee D_1}$$

But why?

$$\frac{\{p \vee C_1 \ldots\} \qquad \{\neg p \vee D_1 \ldots\}}{C_1 \vee D_1}$$

Proof [5]

Isabelle/HOL(`https://isabelle.in.tum.de/`)

Propositional Resolution

`https://www.isa-afp.org/entries/PropResPI.shtml`

Davis–Putnam–Logemann–Loveland

One-Literal rule

Remove all clauses only containing one literal

AKA Unit Propagation

Affirmative-negative rule

Evaluate all literals to that occur only negatively or only positively to true

AKA Pure Literal Rule

Davis–Putnam–Logemann–Loveland

One-Literal rule

Remove all clauses only containing one literal

AKA Unit Propagation

Affirmative-negative rule

Evaluate all literals to that occur only negatively or only positively to true

AKA Pure Literal Rule

Decide or Case-split

- Decide on a valuation of a literal and continue procedure. In case of failure a case splits will be the point to we backtrack.

DPLL Implementation

```
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) ||
        dpll (insert [negate p] clauses)
```

Improving DPLL

Iterative DPLL

- *Tailrecursion*

  Accumulate a list that contains information about “chosen literals”.

  Use of *backtracking*.

Improving DPLL

Iterative DPLL

- *Tailrecursion*

  Accumulate a list that contains information about “chosen literals”.

  Use of *backtracking*.

Backjumping

- When a conflict is derived utilizing data structures one can *backjump* further than the last decision.

Improving DPLL

Iterative DPLL

- *Tailrecursion*

  Accumulate a list that contains information about "chosen literals".

  Use of *backtracking*.

Backjumping

- When a conflict is derived utilizing data structures one can *backjump* further than the last decision.

Learning

Given two clauses $\{\neg p_1, \neg p_{10}, \neg p_{21}\}$ and $\{\neg p_1, \neg p_{10}, p_{21}\}$, one can add the clause $\{\neg p_1, \neg p_{10}\}$

Improved DPLL

Improved DPLL Implementation

```
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') ps in
              dpli cls ((p,Guessed)::trail')
```

Complexity

k-CNF

CNF of form: $(p_1 \vee p_2 \vee \cdots \vee p_k) \wedge (q_1 \vee q_2 \vee \cdots \vee q_k) \wedge \ldots$

- Incomplete $\forall k < 3$

Complexity

k-CNF

CNF of form: $(p_1 \vee p_2 \vee \cdots \vee p_k) \wedge (q_1 \vee q_2 \vee \cdots \vee q_k) \wedge \ldots$

- Incomplete $\forall k < 3$

Complexity

- 2-CNF can be solved in Polynomial time.
- Complexities for k-CNF $O(2^{(1-\epsilon_k)n})$ [3]
- Improved algorithm based on DPLL with $O(2^{0.386n})$ [4]

MiniSAT example

MiniSAT

`http://minisat.se/`

Sudoku

`http://cl-informatik.uibk.ac.at/teaching/ws16/lics/material/sudoku.ml`

Binary Decision Diagrams

BDD Representation

Node

Every node has an index >1.

$+1$ is index for true and -1 for false

BDD Representation

Node

Every node has an index >1.

$+1$ is index for true and -1 for false

Code

```
type bddnode = prop * int * int
```

BDD Representation

Node

Every node has an index >1.

+1 is index for true and -1 for false

Code

```
type bddnode = prop * int * int
```

BDD/Tree

```
type bdd = Bdd of ((bddnode,int)func *
    (int,bddnode)func * int) *
    (prop->prop->bool)
```

Selected BDD function

Create BDD from Formula

```
let rec mkbdd (bdd,comp as bddcomp) fm =
  match fm with
    False -> bddcomp,-1
  | True -> bddcomp,1
  | Atom(s) -> let bdd',n = mk_node bdd (s,1,-1)
    in (bdd',comp),n
  | Not(p) -> let bddcomp',n = mkbdd bddcomp p
    in bddcomp',-n
  | And(p,q) -> thread bddcomp bdd_and (mkbdd,p) (mkbdd,q)
  | Or(p,q) -> thread bddcomp bdd_or (mkbdd,p) (mkbdd,q)
  | Imp(p,q) -> thread bddcomp bdd_imp (mkbdd,p) (mkbdd,q)
  | Iff(p,q) -> thread bddcomp bdd_iff (mkbdd,p) (mkbdd,q);;
```

Compactness

Compactness

Satisfiability

A set(Γ) of formulas, $\{\phi_1, \phi_2, \ldots, \phi_n\}$ is said to be satisfiable if there is a valuation that satisfies all formulas simultaneously [1].

Compactness

Satisfiability

A set(Γ) of formulas, $\{\phi_1, \phi_2, \ldots, \phi_n\}$ is said to be satisfiable if there is a valuation that satisfies all formulas simultaneously [1].

Compactness theorem:

Let Γ be a *countable* set of formulas.

If each finite $\Delta \subseteq \Gamma$ is satisfiable, then Γ is satisfiable.

Questions?

Stålmarck's method [6]

General Outline

- Based on triples (p, q, r) with a certain interpretation(example: $p \iff (q \implies r)$).
- We assume that the formula is unsatisfiable, and try to derive a contradiction.
- Based on the chosen interpretation, one can derive a set of *simple rules* and a *dilemma rule*.
- Using those rules we try to derive a *terminal* triple(example: $(0, 0, x)$ which is a contradiction since when 0/false is an antecedent the implication cannot be false).

Stålmarck's method

Simple rules

Assuming the aforementioned interpretation we can derive multiple simple rules.

Two examples of those rules would be:

$$\frac{(x, y, 1)}{(1, y, 1)}\,[1/x]$$ Since $y \implies 1$ means that x is true.

$$\frac{(x, 1, z)}{(z, 1, z)}\,[z/x]$$ Since $1 \implies z$ means that $x \iff z$

Stålmarck's method

Dilemma rule

Let Γ be a set of triples, we get the Dilemma rule:

$$\frac{\Gamma}{\Gamma[1/x] \quad \Gamma[0/x]}$$

Using the simple rules we can derive two sets of triples Δ_1 and Δ_2 with the substitutions S_1 and S_2.
If we are able to derive a contradiction in one of those *splits*, then this is the result of the *other* split and we "proved" Γ to be satisfiable. Otherwise we continue:

$$\frac{\Delta_1[S_1] \qquad \Delta_2[S_2]}{\Gamma[S_1 \cap S_2]}$$

References

John Harrison. *Handbook of Practical Logic and Automated Reasoning*. 1st. New York, NY, USA: Cambridge University Press, 2009. ISBN: 9780521899574.

Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. New York, NY, USA: Cambridge University Press, 2004. ISBN: 052154310X.

References II

B. Monien and E. Speckenmeyer. “Solving satisfiability in less than 2n steps”. In: *Discrete Applied Mathematics* 10.3 (1985), pp. 287–295. ISSN: 0166-218X. DOI: http://dx.doi.org/10.1016/0166-218X(85)90050-2. URL: http://www.sciencedirect.com/science/article/pii/0166218X85900502.

Ramamohan Paturi et al. “An Improved Exponential-time Algorithm for k-SAT”. In: *J. ACM* 52.3 (May 2005), pp. 337–364. ISSN: 0004-5411. DOI: 10.1145/1066100.1066101. URL: http://doi.acm.org/10.1145/1066100.1066101.

References III

Nicolas Peltier. “Propositional Resolution and Prime Implicates Generation”. In: *Archive of Formal Proofs* (Mar. 2016). http://isa-afp.org/entries/PropResPI.shtml, Formal proof development. ISSN: 2150-914x.

Mary Sheeran and Gunnar Stålmarck. “A Tutorial on Stålmarck’s Proof Procedure for Propositional Logic”. In: *Formal Methods in System Design* 16.1 (2000), pp. 23–58. ISSN: 1572-8102. DOI: 10.1023/A:1008725524946. URL: http://dx.doi.org/10.1023/A:1008725524946.