Proof Complexity and Solving LAB

DPLL

Dr. Joshua Blinkhorn

Friedrich-Schiller-Universität Jena
https://github.com/JoshuaBlinkhorn/SAT-LAB

Goals

- Implemenatation of SAT solving algorithms

  (a) 2-SAT (polynomial time)

  (b) DPLL (decision tree)

  (c) CDCL
    - clause learning
    - watched literals
    - decision heuristics
    - restart strategy

  (d) QBF expansion..

- Practical programming experience
  - use your favourite language (Python, C, C++, Java, ..)
  - recommended: Python

Pure Literal

- A literal a is pure in a CNF Φ if:
  1. a appears in Φ
  2. $\neg a$ does not appear in Φ
- if Φ is satisfiable, it has a satisfiable assignment that satisfies all pure literals
- so pure literals may as well be assigned immediately

DPLL Psuedocode

function `DPLL-solver`(Φ)
  if `DPLL`(Φ) $=$ true **then return** SAT
  return UNSAT

function `DPLL`(Φ)
  if Φ is the empty formula **then return** true
  if Φ contains the empty clause **then return** false
  $\Phi \leftarrow$ `unit-propagate`(Φ)
  $\Phi \leftarrow$ `eliminate-pure-literals`(Φ)
  $x \leftarrow$ `get-decision-variable`(Φ)
  return `DPLL`($\Phi[x \mapsto 0]$) **or** `DPLL`($\Phi[x \mapsto 1]$)

Practical Guidelines for Implementation (1)

- Resource trade-off: local or global data structures?
- maintaining local data structures for each recursive call costs memory but saves time
- maintaining global data structures saves memory but costs time
- Investigating this trade-off is not our goal

Practical Guidelines for Implementation (2)

- recommendation: use global data structures
  - treat Φ as a global constant data structure
  - maintain a global partial assignment α in sequence
  - determine unit propagations from the state of Φ and α
  - determine pure literals from the state of Φ and α
  - determine decisions variables from the state of α
- Question: why use global data?
  1. because your CDCL solver probably will
  2. because you can easily output a satisying assignment

Practical Guidelines for Implementation (3)

- Psuedocode:

  return $\mathtt{DPLL}(\Phi[x \mapsto 0])$ **or** $\mathtt{DPLL}(\Phi[x \mapsto 1])$

- Real code:

  $\mathtt{assign}(x, 0)$
  if $\mathtt{DPLL}() = \text{true}$ **then return** true
  $\mathtt{unassign}(x)$
  $\mathtt{assign}(x, 1)$
  if $\mathtt{DPLL}() = \text{true}$ **then return** true
  $\mathtt{backtrack}(\text{entry-point})$
  $\mathtt{return}$ false

- if $\mathtt{DPLL}()$ returns false, the assignment at point of return should equal the assignment at point of call

DPLL Task

- implement a DPLL solver
- include a README
- test your solver on random k-SAT formulas
- print statistics, e.g.
  - solving time
  - memory consumption
  - number of decisions
  - number of unit propagations
  - number of pure literal eliminations
  - ..?