

Proof Complexity and Solving LAB

DPLL

Dr. Joshua Blinkhorn

Friedrich-Schiller-Universität Jena

<https://github.com/JoshuaBlinkhorn/SAT-LAB>

Goals

- Implementation 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 Literals

$$(x \vee \neg y) \wedge (x \vee y \vee \neg z) \wedge (z)$$

- 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( $\Phi$ )  
  if DPLL( $\Phi$ ) = true then return SAT  
  return UNSAT
```

```
function DPLL( $\Phi$ )  
  if  $\Phi$  is the empty formula then return true  
  if  $\Phi$  contains the empty clause then return false  
   $\Phi \leftarrow$  unit-propagate( $\Phi$ )  
   $\Phi \leftarrow$  eliminate-pure-literals( $\Phi$ )  
   $x \leftarrow$  get-decision-variable( $\Phi$ )  
  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 satisfying assignment

Practical Guidelines for Implementation (3)

- Psuedocode:

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

- Real code:

```
assign(x, 0)
if DPLL() = true then return true
unassign(x)
assign(x, 1)
if DPLL() = true then return true
backtrack(entry-point)
return false
```

- if 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
 - ..?