CIS 1921

Lecture 3: Algorithms for SAT

Reminders

- Homework 1 due Monday, Sept 23, 11:59PM
- Office Hours will be held online through OHQ.io.
  - Cindy: Wednesday 6-7
  - Ishaan: Saturday 6-7

Grading

- Homework: 60%
- Final Project: 30%
- Attendance: 10%

- Final grades:
  don't worry too much about it.

Recap

Last week

- Using SAT solvers in Python (PycoSAT)
- Encode other problems (graph coloring, stable matching) as SAT

This week

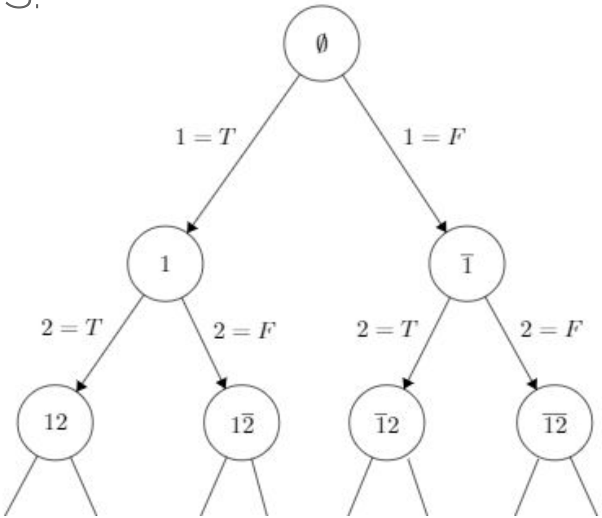
- Build up an algorithm to solve SAT

SAT is Hard!

Naive Search for SAT

- Naive algorithm: try every possible assignment until we find a satisfying assignment or exhaust the search space
- Can interpret this as a DFS:

  (search tree)



Simplify the Search Space

Find a *minimal satisfying assignment* for the following formula:

$$\varphi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_2} \vee x_4) \wedge (\overline{x_1} \vee x_3 \vee x_5) \wedge (\overline{x_2} \vee \overline{x_1}) \wedge (\overline{x_2} \vee x_6 \vee x_7)$$

Find a *minimal satisfying assignment* for the following formula:

$$\varphi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_2} \vee x_4) \wedge (\overline{x_1} \vee x_3 \vee x_5) \wedge (\overline{x_2} \vee \overline{x_1}) \wedge (\overline{x_2} \vee x_6 \vee x_7)$$

$$x_2 = \texttt{FALSE} \quad x_3 = \texttt{TRUE}$$

Trimming the Search Space

- If a formula is satisfiable (has a satisfying assignment to variables), then in the assignment, each clause must individually evaluate to TRUE.

$$\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

Trimming the Search Space

- **When we set $x = T$, what happens to the clauses containing x?**
- **Observation 1:** Any clause containing the positive literal x becomes satisfied, so we no longer need to consider those clauses
    - In logic: $(T \vee 1 \vee 2 \vee \cdots) = T$
    - Significance: we should remove all clauses containing x

Trimming the Search Space

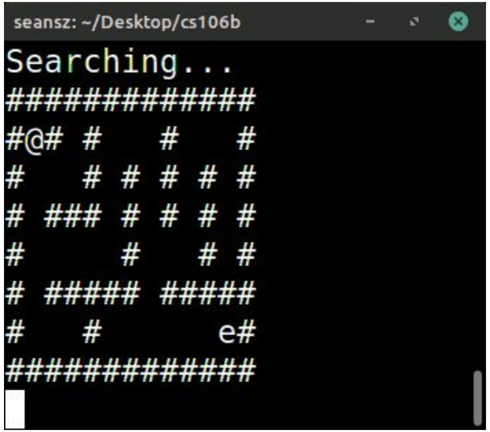
- **When we set $x = T$, what happens to the clauses containing $\overline{x}$?**
- **Observation 2:** Any clause containing the negative literal $\overline{x}$ needs to be satisfied by a different literal, so we can ignore $\overline{x}$ in that clause
  - In logic: $(F \lor 1 \lor 2 \lor \cdots) = (1 \lor 2 \lor \cdots)$
  - Significance: we should remove $\overline{x}$ from all clauses containing it

The Splitting Rule

- The previous observations are called the **splitting rule**
- After repeatedly applying the splitting rule to formula φ:
  - If there are **no clauses left**, then all clauses have been satisfied, so φ is satisfied
    - $\varphi = \emptyset$ denotes that there are no clauses left
  - If φ ever contains an **empty clause**, then all literals in that clause are False, so we made a mistake
    - ϵ denotes the empty clause
    - $\epsilon \in \varphi$ denotes that φ contains an empty clause

The Splitting Rule

- The splitting rule allows us to create a smarter recursive **backtracking** algorithm
- Backtracking: repeatedly make a guess to explore partial solutions, and if we hit "dead end" (contradiction) then undo the last guess



Backtracking Notation

- For a CNF φ and a literal x, define $\varphi|x$ ("φ given x") to be a new CNF produced by:
  - Removing all clauses containing x
  - Removing $\overline{x}$ from all clauses containing it
- Conditioning is "commutative": $\varphi|x_1|x_2 = \varphi|x_2|x_1$

Backtracking (Pseudocode)

```
# check if φ is satisfiable
backtrack(φ):
    if φ = ∅: return True
    if ϵ ∈ φ: return False
    let x = pick_variable(φ)
    return backtrack(φ|x) OR backtrack(φ|x̄)
```

Example: Backtracking

Steps

$(\overline{1} \vee \overline{2})$

$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

1	2	3	4	5

Example: Backtracking

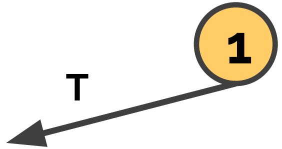
$$(\overline{1} \vee \overline{2})$$

$$(\overline{1} \vee 2 \vee \overline{3})$$

$$(3 \vee \overline{4} \vee \overline{5})$$

$$(3 \vee 4 \vee \overline{5})$$

Steps



1	2	3	4	5
T				

Example: Backtracking

$(\overline{1} \vee \overline{2})$ **Conflict!**

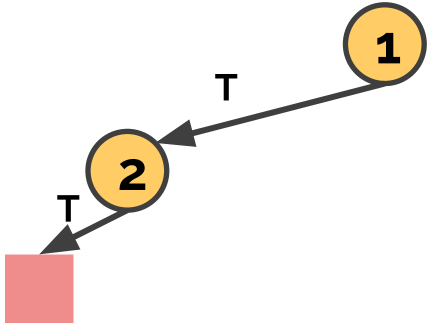
$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

1	2	3	4	5
T	T			

Steps



Example: Backtracking

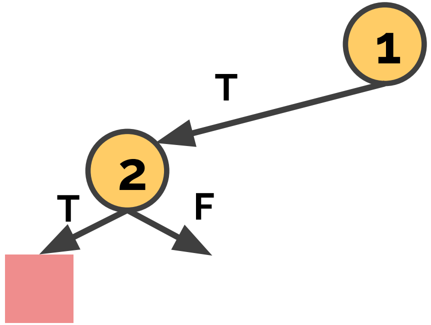
$(\overline{1} \lor \overline{2})$

$(\overline{1} \lor 2 \lor \overline{3})$

$(3 \lor \overline{4} \lor \overline{5})$

$(3 \lor 4 \lor \overline{5})$

Steps



1	2	3	4	5
T	F			

Example: Backtracking

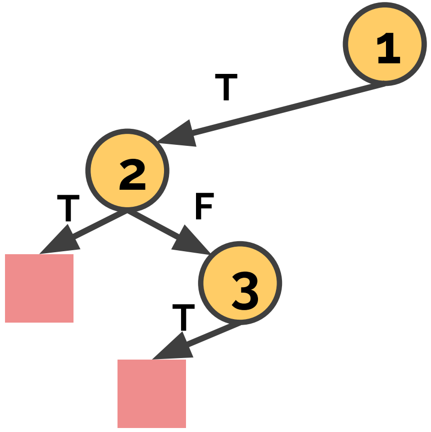
$(\overline{1} \vee \overline{2})$

$(\overline{1} \vee 2 \vee \overline{3})$ **Conflict!**

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

Steps



1	2	3	4	5
T	F	T		

Example: Backtracking

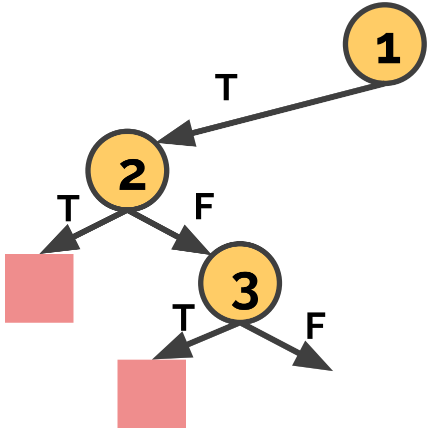
$(\overline{1} \lor \overline{2})$

$(\overline{1} \lor 2 \lor \overline{3})$

$(3 \lor \overline{4} \lor \overline{5})$

$(3 \lor 4 \lor \overline{5})$

Steps



1	2	3	4	5
T	F	F		

Example: Backtracking

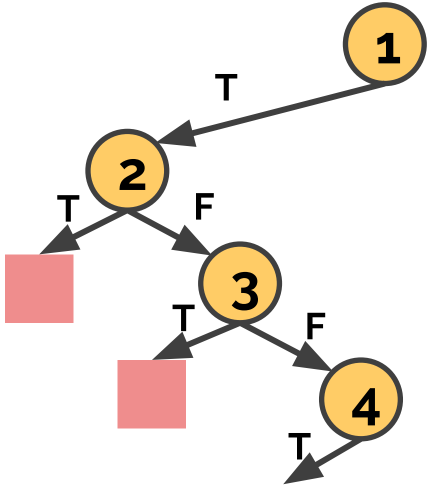
Steps

$(\overline{1} \vee \overline{2})$

$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$



1	2	3	4	5
T	F	F	T	

Example: Backtracking

$(\overline{1} \vee \overline{2})$

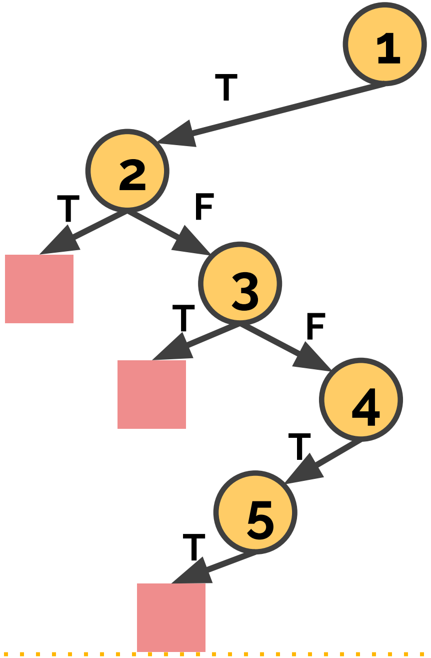
$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$ **Conflict!**

$(3 \vee 4 \vee \overline{5})$

1	2	3	4	5
T	F	F	T	T

Steps



Example: Backtracking

$(\overline{1} \vee \overline{2})$

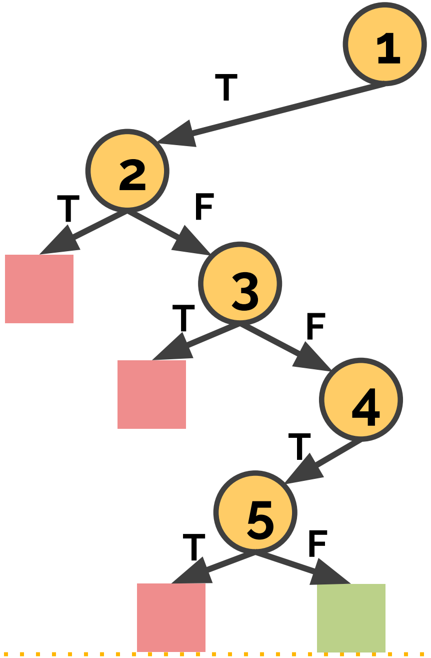
$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

1	2	3	4	5
T	F	F	T	F

Steps



Efficient Splitting

- How do we compute $\varphi|x$?
- Goals:
  - Support fast searching for empty clauses
  - Support fast backtracking
  - Fast to actually compute $\varphi|x$

Naïve Idea 1

- Transform φ into $\varphi|x$ by deleting satisfied clauses and False literals from φ
  - Deletion not too expensive if we use linked lists
  - Can quickly recognize an empty clause (linked list will be empty), but need to check all clauses
  - Big issue: how do we backtrack?

Naïve Idea 2

- Simple fix: instead of modifying φ directly, create a copy first and modify that
  - Easy backtracking – just restore the old formula
  - Big issue: too expensive (time and memory) to copy formula every time we split
    - What if we have hundreds of thousands, even millions of clauses?

Towards a smarter scheme

- Don't modify or copy the formula!
- **Key observation:** We must only backtrack once a clause has become empty *after* the Splitting Rule has been applied!

1 Watched Literal Scheme

- **Observation:** a clause can only become empty if it has just one unassigned literal remaining
  - Ideally, only need to check these clauses

- Each clause "watches" one literal and maintains watching invariant: the watched literal is True or unassigned
  - If the watched literal becomes False, watch another
  - If there are no more True/unassigned literals to watch, then the clause must be empty

Example: 1 Watched Literal

Steps

$(\overline{1} \vee \overline{2})$

$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

1	2	3	4	5

Example: 1 Watched Literal

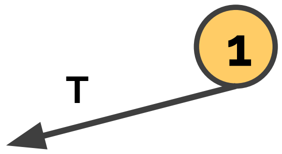
$(\overline{1} \lor \overline{2})$

$(\overline{1} \lor 2 \lor \overline{3})$

$(3 \lor \overline{4} \lor \overline{5})$

$(3 \lor 4 \lor \overline{5})$

Steps



1	2	3	4	5
T				

Example: 1 Watched Literal

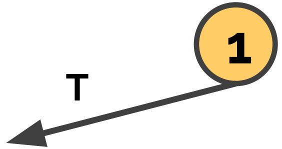
$(\overline{1} \vee \overline{2})$

$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

Steps



1	2	3	4	5
T				

Example: 1 Watched Literal

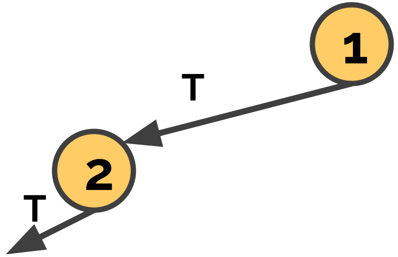
$(\overline{1} \vee \overline{2})$

$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

Steps



1	2	3	4	5
T	T			

Example: 1 Watched Literal

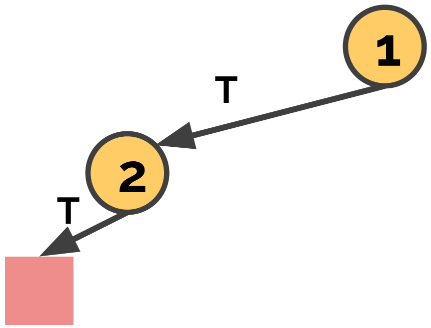
$(\overline{1} \lor \overline{2})$ **Conflict!**

$(\overline{1} \lor 2 \lor \overline{3})$

$(3 \lor \overline{4} \lor \overline{5})$

$(3 \lor 4 \lor \overline{5})$

Steps



1	2	3	4	5
T	T			

Example: 1 Watched Literal

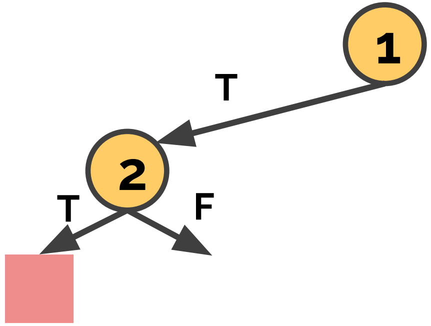
$(\overline{1} \lor \overline{2})$

$(\overline{1} \lor 2 \lor \overline{3})$

$(3 \lor \overline{4} \lor \overline{5})$

$(3 \lor 4 \lor \overline{5})$

Steps



1	2	3	4	5
T	F			

Example: 1 Watched Literal

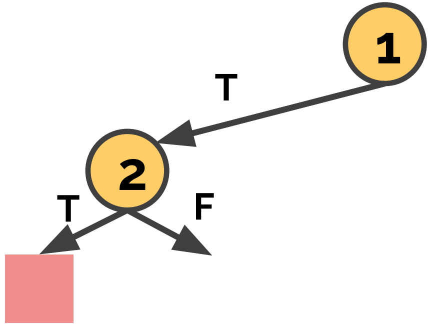
$(\overline{1} \vee \overline{2})$

$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

Steps



1	2	3	4	5
T	F			

Example: 1 Watched Literal

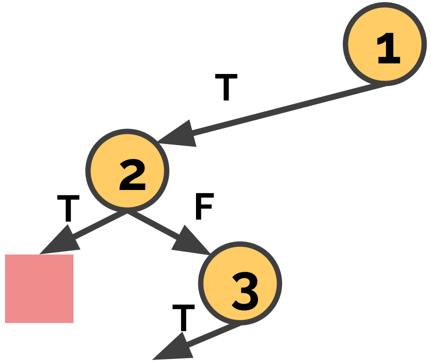
$(\overline{1} \vee \overline{2})$

$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

Steps



1	2	3	4	5
T	F	T		

Example: 1 Watched Literal

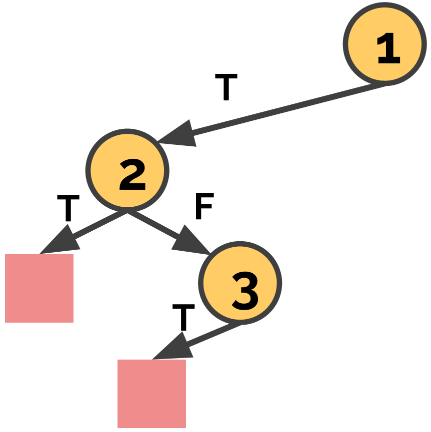
$(\overline{1} \lor \overline{2})$

$(\overline{1} \lor 2 \lor \overline{3})$ **Conflict!**

$(3 \lor \overline{4} \lor \overline{5})$

$(3 \lor 4 \lor \overline{5})$

Steps



1	2	3	4	5
T	F	T		

Example: 1 Watched Literal

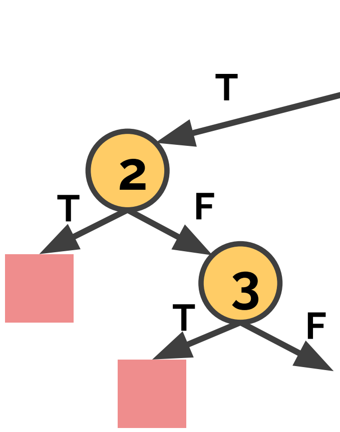
$(\overline{1} \vee \overline{2})$

$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

Steps



1	2	3	4	5
T	F	F		

Example: 1 Watched Literal

$(\overline{1} \vee \overline{2})$

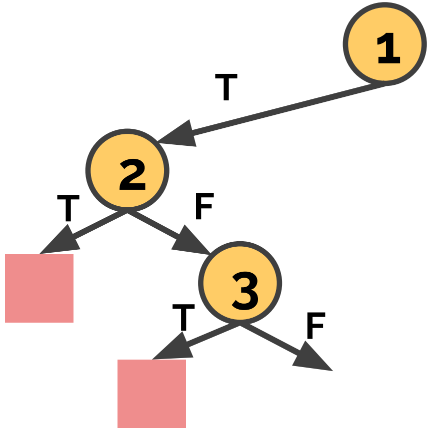
$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

1	2	3	4	5
T	F	F		

Steps



Example: 1 Watched Literal

$(\overline{1} \vee \overline{2})$

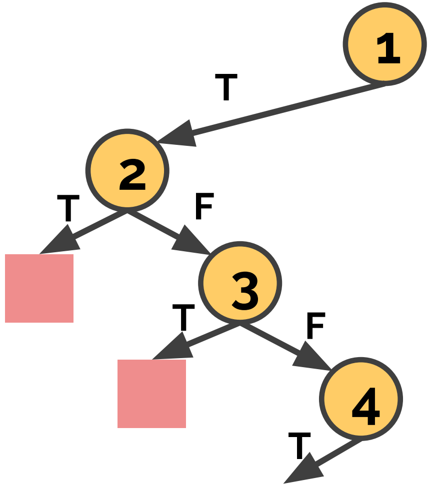
$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

1	2	3	4	5
T	F	F	T	

Steps



Example: 1 Watched Literal

$(\overline{1} \vee \overline{2})$

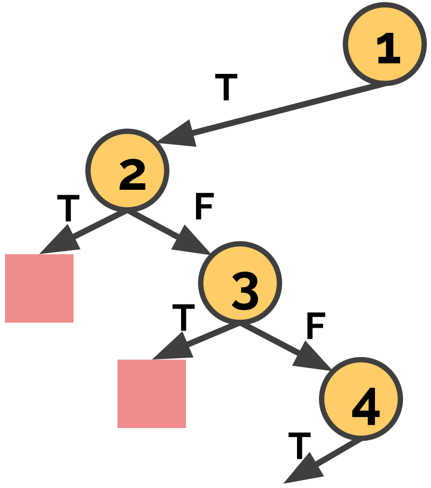
$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$

$(3 \vee 4 \vee \overline{5})$

1	2	3	4	5
T	F	F	T	

Steps



Example: 1 Watched Literal

$(\overline{1} \lor \overline{2})$

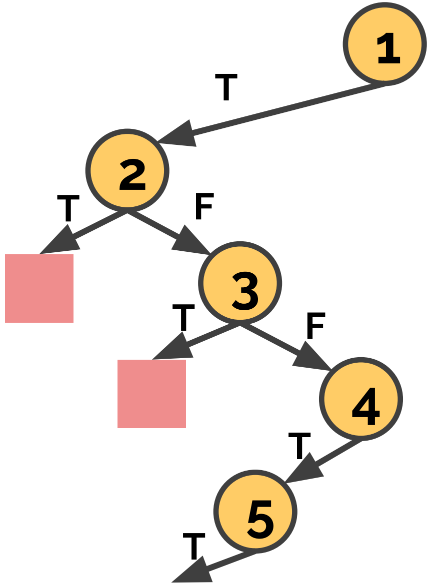
$(\overline{1} \lor 2 \lor \overline{3})$

$(3 \lor \overline{4} \lor \overline{5})$

$(3 \lor 4 \lor \overline{5})$

1	2	3	4	5
T	F	F	T	T

Steps



Example: 1 Watched Literal

$(\overline{1} \vee \overline{2})$

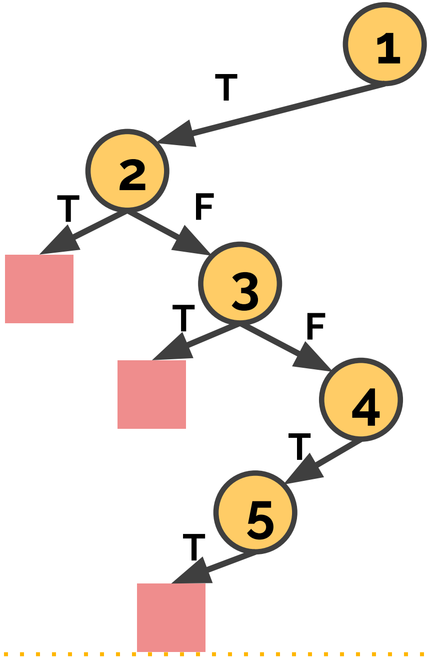
$(\overline{1} \vee 2 \vee \overline{3})$

$(3 \vee \overline{4} \vee \overline{5})$ **Conflict!**

$(3 \vee 4 \vee \overline{5})$

1	2	3	4	5
T	F	F	T	T

Steps



Example: 1 Watched Literal

$(\overline{1} \lor 2)$

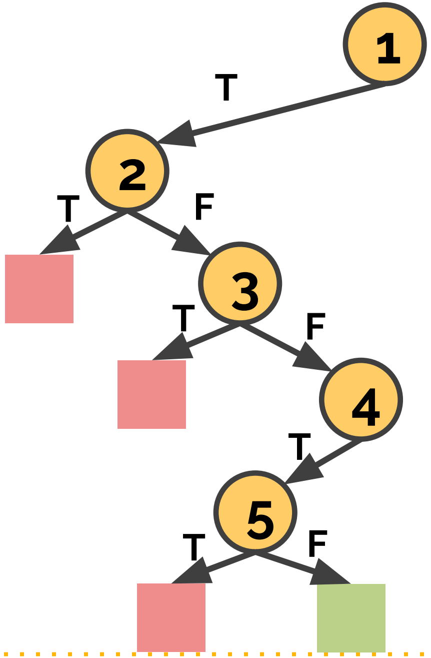
$(\overline{1} \lor 2 \lor \overline{3})$

$(3 \lor \overline{4} \lor \overline{5})$

$(3 \lor 4 \lor \overline{5})$

1	2	3	4	5
T	F	F	T	F

Steps



Find a *satisfying assignment* for the following formula:

$$\varphi = (x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_3) \wedge (x_4 \vee \overline{x_5} \vee \overline{x_7}) \wedge (x_3 \vee x_5 \vee x_6 \vee \overline{x_7}) \wedge (\overline{x_5} \vee \overline{x_6})$$

Find a *satisfying assignment* for the following formula:

$$\varphi = (x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_3) \wedge (x_4 \vee \overline{x_5} \vee \overline{x_7}) \wedge (x_3 \vee x_5 \vee x_6 \vee \overline{x_7}) \wedge (\overline{x_5} \vee \overline{x_6})$$

$x_1 = \texttt{FALSE}$ $x_2 = \texttt{FALSE}$ $x_3 = \texttt{TRUE}$

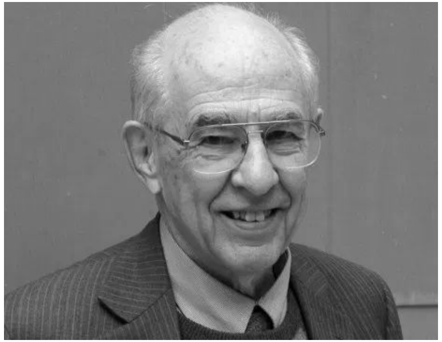
$x_4 = \texttt{TRUE}$ $x_5 = \texttt{FALSE}$ $x_6 = \texttt{TRUE}$ $x_7 = \texttt{TRUE}$

Unit Propagation (UP)

- A **unit clause** is a clause containing only one literal
- **Unit propagation rule:** for any unit clause $\{\ell\}$, we must set $\ell = T$
- Applying unit propagation can massively speed up the backtracking algorithm in practice
  - Combining with the splitting rule can lead to a "domino effect" of cascading unit propagation

The DPLL Algorithm

- Davis-Putnam-Logemann-Loveland (1962)
- Improved upon naive backtracking (search) with unit propagation (inference)
- Still the basic algorithm behind most state-of-the-art SAT solvers today!



DPLL (Pseudocode)

```
dpll(φ):
    if φ = ∅: return TRUE
    if ϵ ∈ φ: return FALSE
    if φ contains unit clause {ℓ}:
        return dpll(φ|ℓ)
    let x = pick_variable(φ)
    return dpll(φ|x) OR dpll(φ|x̄)
```

Example: DPLL

$(\bar{1} \vee \bar{2})$

$(\bar{1} \vee 2)$

$(1 \vee \bar{2} \vee 3)$

$(1 \vee 2 \vee \bar{4})$

1	2	3	4

Steps

Example: DPLL

$(\overline{1} \vee \overline{2})$

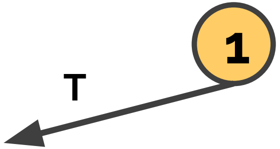
$(\overline{1} \vee 2)$

$(1 \vee \overline{2} \vee 3)$

$(1 \vee 2 \vee \overline{4})$

Unit!

Steps



1	2	3	4
T			

Example: DPLL

$(\overline{1} \vee \overline{2})$

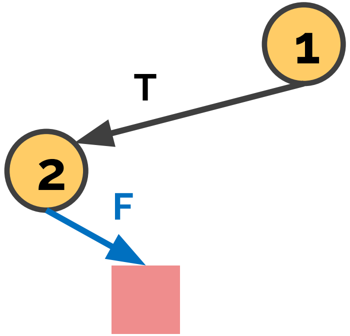
$(\overline{1} \vee 2)$

Conflict!

$(1 \vee \overline{2} \vee 3)$

$(1 \vee 2 \vee \overline{4})$

Steps



1	2	3	4
T	F		

Example: DPLL

$(\overline{1} \vee \overline{2})$

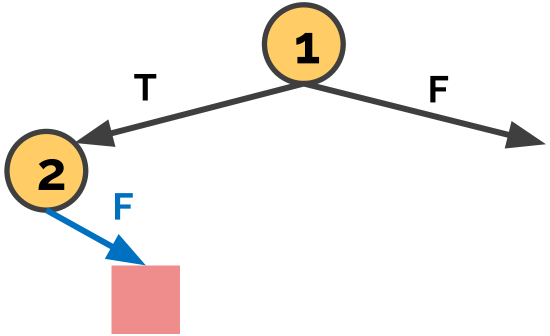
$(\overline{1} \vee 2)$

$(1 \vee \overline{2} \vee 3)$

$(1 \vee 2 \vee \overline{4})$

1	2	3	4
F			

Steps



Example: DPLL

$(\overline{1} \vee \overline{2})$

$(\overline{1} \vee 2)$

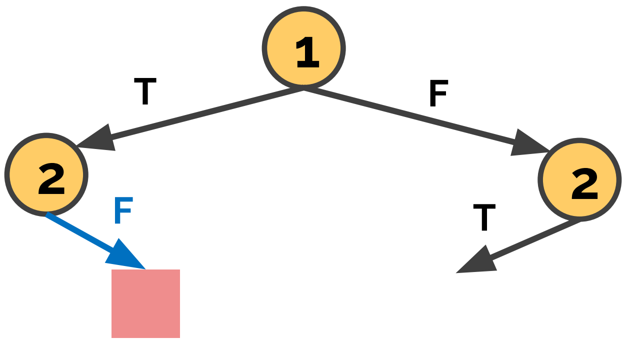
$(1 \vee \overline{2} \vee 3)$

$(1 \vee 2 \vee \overline{4})$

Unit!

1	2	3	4
F	T		

Steps



Example: DPLL

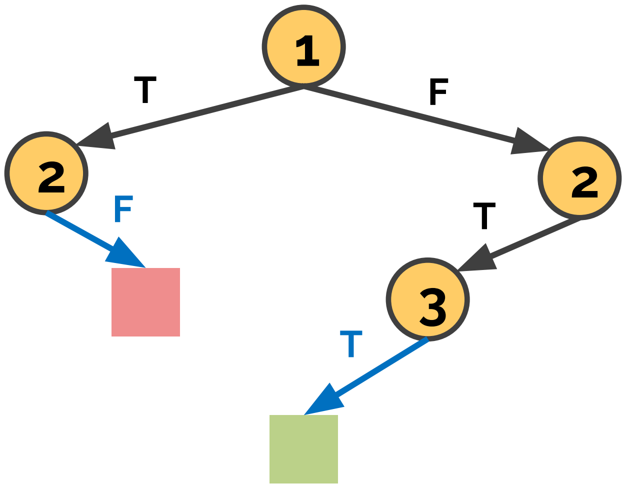
$(\overline{1} \vee \overline{2})$

$(\overline{1} \vee 2)$

$(1 \vee \overline{2} \vee 3)$

$(1 \vee 2 \vee \overline{4})$

Steps



1	2	3	4
F	T	T	

Engineering Matters

- Since the main DPLL subroutine might run exponentially many times, every speedup counts
- DPLL spends by far the most time on UP
  - How can we speed this up?
- Although DPLL has a natural recursive formulation, recursion is slow — lots of overhead
  - We can make DPLL **iterative** using a stack

2 Watched Literals (2WL)

- **Key observation:** a clause can only be unsatisfied or unit if it has at most one non-False literal
  - Optimize unit propagation: only visit those clauses
- Each clause "watches" two literals and maintains watching **invariant:** the watched literals are not False, unless the clause is satisfied
  - If a watched literal becomes False, watch another
- If can't maintain invariant, clause is unit (can propagate)

2 Watched Literals (2WL)

- Still use watchlists (list of all clauses watching each lit)
- Best part: since backtracking only unassigns variables, it can never break the 2WL invariant
  - Don't need to update watchlists

$$(\bar{1} \lor 2 \lor \bar{3}) \xrightarrow{\text{Set } 1 = T} (\bar{1} \lor 2 \lor \bar{3}) \xrightarrow{\text{Set } 2 = F} (\bar{1} \lor 2 \lor \bar{3})$$

Unit!

Iterative DPLL

- A **decision** refers to any time our algorithm *arbitrarily* assigns a variable (without being forced to do so)
    - Selecting a literal and assigning it True is a decision
    - Unit propagation & reassigning selected literal after backtracking are not decisions
- All assignments implied by the i^{th} decision are said to be on the i^{th} **decision level**
    - Can assignments ever be on the zeroth decision level?

Iterative DPLL

- Maintain an **assignment stack** with the assignments from each decision level
  - Whenever we make a new decision, copy the current assignment onto the top of the stack
- To backtrack: pop the current assignment off the stack, restoring the previous one
- Keep a **propagation queue** of literals that are set to False
  - Take literals from the queue and check if their watching clauses are empty/unit

Assignment Stack

T	T	F	T	T
T	T	F		
T				
1	2	3	4	5

Set $2 = T$. Propagate $3 = F$.

Set $1 = T$

Assignment Stack

Pop!

T	T	F	T	T

Backtrack!

T	T	F		
T				
1	2	3	4	5

Set $2 = T$. Propagate $3 = F$.

Set $1 = T$

Iterative DPLL (Pseudocode)

```
dpll(φ):
    if unit_propagate() = CONFLICT: return UNSAT
    while not all variables have been set:
        let x = pick_variable()
        create new decision level
        set x = T
        while unit_propagate() = CONFLICT:
            if decision_level = 0: return UNSAT
            backtrack()
            set x = F
   return SAT
```

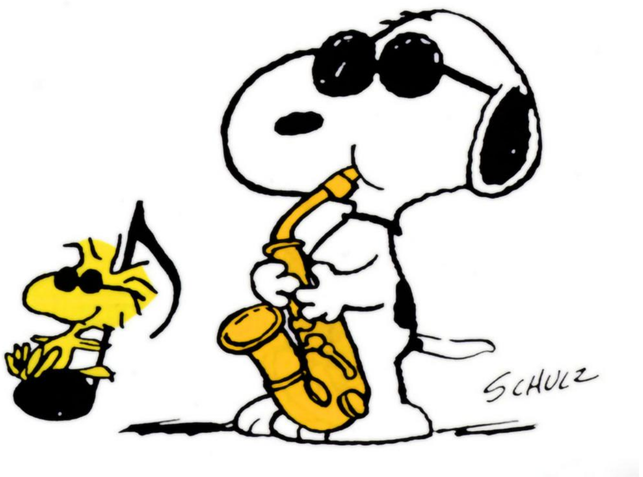
How should we branch?

- Order of assigning variables greatly affects runtime
- Want to find a satisfying assignment quicker and find conflicts (rule out bad assignments) quicker
- **Ex:** $\left\{1\bar{2}34, \overline{12}3, 12\bar{3}5, 23\bar{5}, 3\overline{45}, \ldots, 67, \bar{6}7, 6\bar{7}, \overline{67}\right\}$
  - If we assign 6 first, then we can find conflicts right away

Decision Heuristics

- **Static heuristics:** variable ordering fixed at the start
- **Dynamic heuristics:** variable ordering is updated as the solver runs
  - More effective, but also more expensive
- Basic examples of decision heuristics:
  - Random ordering
  - Most-frequent static ordering
  - Most-frequent dynamic ordering

Stay Wise



"Intelligence is knowing it is a one-way street, wisdom is still looking both ways before crossing."

References

A. Biere, *Handbook of satisfiability*. Amsterdam: IOS Press, 2009.

N. Eén and N. Sörensson, "An Extensible SAT-solver," *Theory and Applications of Satisfiability Testing Lecture Notes in Computer Science*, pp. 502–518, 2004.