CIS 1921

Lecture 3: Algorithms for SAT

Reminders

- Homework 0 was due on Monday
- Homework 1 due Monday, Feb 10, 11:59PM
- OH schedule:
  - Thomas: Sunday 3-4pm
  - Cindy: Tuesday 8-9pm
  - Ishaan: Wednesday 9:30-10:30pm
  - All OH held on OHQ

Grading

- Homework: 44%
- Final Project: 38%
- Quizzes: 10%
- Attendance: 8%

Academic Integrity

- Work on assignments individually (except final project)
  - Discussion encouraged, but work should be yours
- OK: high-level discussions
  - "Can you help me understand the DPLL algorithm?"
- OK: low-level discussions
  - "How do I time my program in OR-Tools?"
- Be careful: mid-level discussions
  - Not OK: "How exactly do I write this constraint?"

Health Logistics

- If you have a reasonable suspicion that you have Covid or sickness, don't come
    - Email me before class and we'll work something out

Recap

Last week

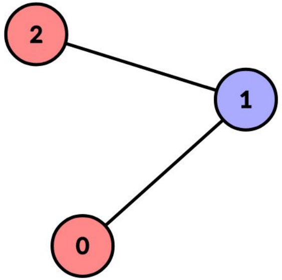
- Using SAT solvers in Python (PycoSAT)
- Encode other problems (graph coloring) as SAT

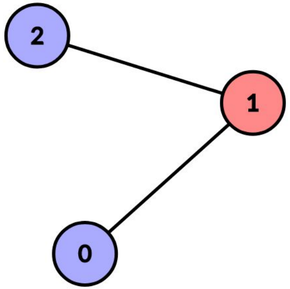
This week

- Build up an algorithm to solve SAT

Symmetry Breaking

- Solving UNSAT graph coloring problems takes a very long time... why?
- Must rule out every symmetric coloring
- Ex: equivalent colorings





Symmetry Breaking

- Key idea: add constraints that rule out equivalent symmetric colorings
- Basic way to do this: pick some vertices (ideally a dense subgraph) and fix their colors

DEMO Part 2

Encoding Stable Matchings

We have n men and n women. Each man and woman submits a preference list ranking everyone of the opposite sex (descending).

Goal: find a **matching** of men to women.

A man and woman who both prefer each other to their matched partners are a **blocking pair**.

A matching is **stable** if it has no blocking pairs.

Encoding Stable Matchings

m_{ip}: if man i is matched to p^{th} woman or later on his list

w_{ip}: if woman i is matched to p^{th} man or later on her list

$[W_1 > W_2]\ M_1 \longleftrightarrow W_1\ [M_1 > M_2]$

$[W_1 > W_2]\ M_2 \longleftrightarrow W_2\ [M_1 > M_2]$

$m_{1,1}$	$m_{1,2}$	$m_{2,1}$	$m_{2,2}$	$w_{1,1}$	$w_{1,2}$	$w_{2,1}$	$w_{2,2}$
T	F	T	T	T	F	T	T

Encoding Stable Matchings

m_{ip}: if man i is matched to p^{th} woman or later on his list

w_{ip}: if woman i is matched to p^{th} man or later on her list

- **C1:** every man is matched

$$\{m_{i1} \mid 1 \leq i \leq n\}$$

(plus symmetric constraints for women for this and the following constraints)

Encoding Stable Matchings

m_{ip}: if man i is matched to p^{th} woman or later on his list

w_{ip}: if woman i is matched to p^{th} man or later on her list

- **C2:** if a man gets his p^{th} or later choice, it's also his $(p-1)^{th}$ or later choice

$$\{m_{ip} \Rightarrow m_{i(p-1)} \mid 1 \le i \le n, 2 \le p \le n\}$$

Encoding Stable Matchings

m_{ip}: if man i is matched to p^{th} woman or later on his list

w_{ip}: if woman i is matched to p^{th} man or later on her list

- **C3:** if man i is matched to woman j, then she is matched to him also

$$\{m_{ip} \wedge \overline{m_{i(p+1)}} \Rightarrow w_{jq} \wedge \overline{w_{j(q+1)}} \mid 1 \leq i,j \leq n\}$$

- p = position of woman j in man i's list
- q = position of man i in woman j's list

Encoding Stable Matchings

m_{ip}: if man i is matched to p^{th} woman or later on his list

w_{ip}: if woman i is matched to p^{th} man or later on her list

- **C4:** if man i is matched to someone worse than woman j, her match must be better than him

$$\{m_{i(p+1)} \Rightarrow \overline{w_{jq}} \mid 1 \le i,j \le n\}$$

- p = position of woman j in man i's list
- q = position of man i in woman j's list

Why Stable Matchings?

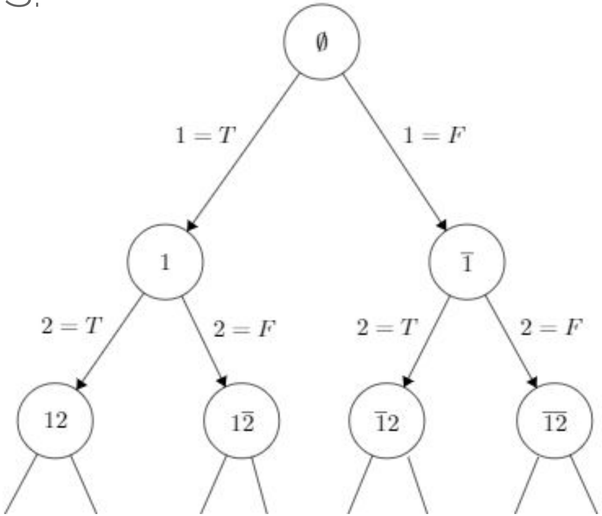
- Gale-Shapley algorithm solves SM problem in linear time. Why use SAT solvers?
- SMTI: stable matching problem where preference lists may be incomplete and contain ties
- SM-C: stable matching problem with couples
- Our encoding easily generalizes to SMTI, SM-C
- Theorem: SMTI and SM-C are NP-complete.

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)



Overarching Class Themes

- Accept the fact that the problems we will look at are very hard and "exponential runtime"
  - Take solace in the fact that for many inputs, the problem won't take exponential time
- Every speed-up counts
  - *Take careful consideration of the balance between runtime and complexity*
- There will never be a "right answer"
  - Often, the best thing to do for a problem depends on the problem itself and its data!

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:

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

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

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

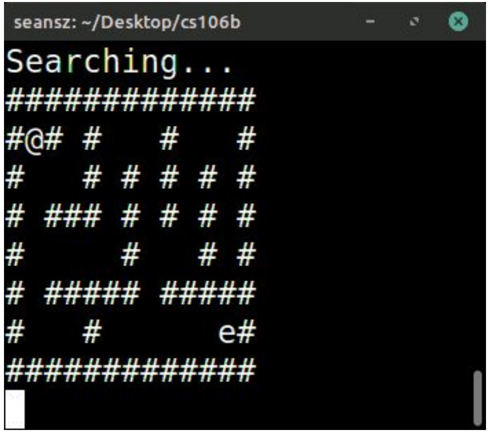
We are honing in on whatever is left that is unassigned and not yet evaluated to *TRUE*.

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

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

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

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

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

Steps

1	2	3	4	5

Example: Backtracking

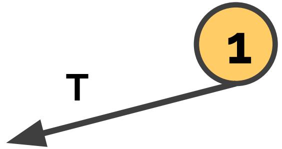
$(\overline{1} \vee \overline{2})$

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

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

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

Steps



1	2	3	4	5
T				

Example: Backtracking

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

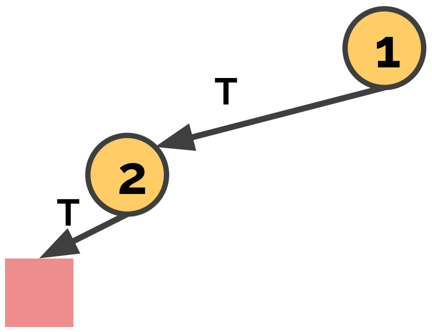
$(\overline{1} \vee 2 \vee \overline{3})$

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

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

1	2	3	4	5
T	T			

Steps



Example: Backtracking

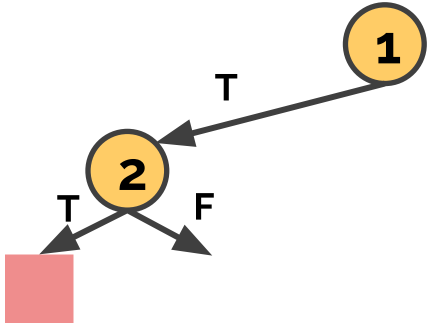
$(\overline{1} \vee \overline{2})$

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

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

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

Steps



1	2	3	4	5
T	F			

Example: Backtracking

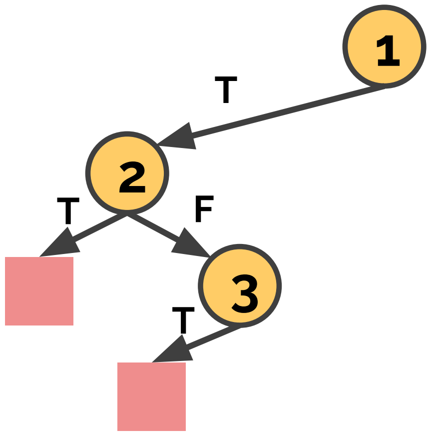
$(\overline{1} \vee \overline{2})$

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

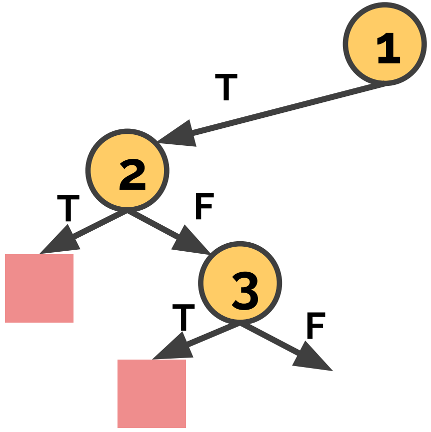
$(\overline{1} \lor \overline{2})$

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

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

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

Steps



1	2	3	4	5
T	F	F		

Example: Backtracking

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

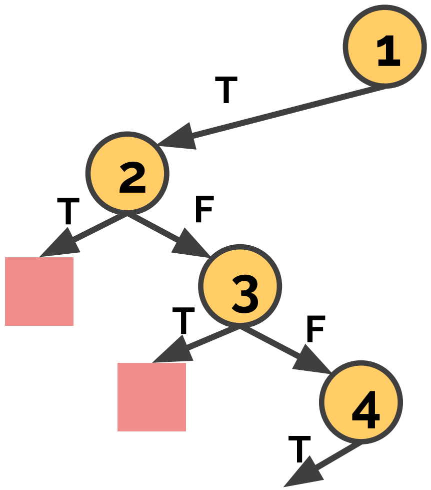
$(\overline{1} \vee 2 \vee \overline{3})$

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

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

1	2	3	4	5
T	F	F	T	

Steps



Example: Backtracking

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

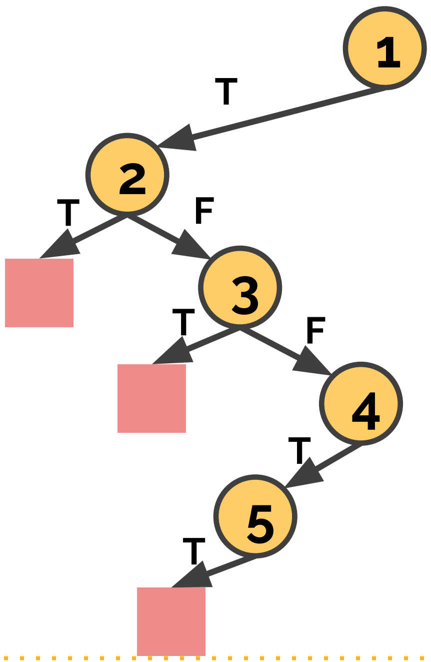
$(\overline{1} \vee 2 \vee \overline{3})$

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

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

1	2	3	4	5
T	F	F	T	T

Steps



Example: Backtracking

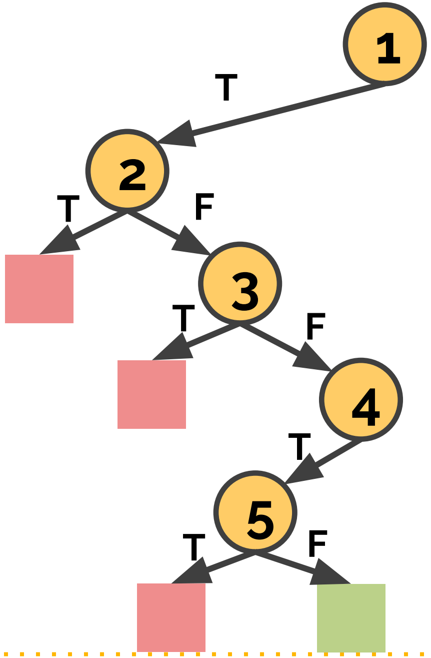
$(\overline{1} \lor \overline{2})$

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

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

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

Steps



1	2	3	4	5
T	F	F	T	F

Towards Implementation: 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

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

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

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

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

1	2	3	4	5

Example: 1 Watched Literal

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

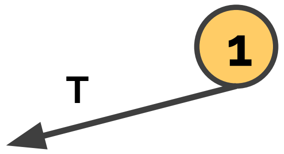
$(\overline{1} \vee 2 \vee \overline{3})$

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

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

1	2	3	4	5
T				

Steps



Example: 1 Watched Literal

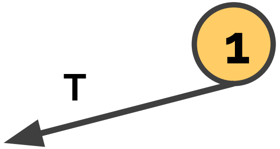
$(\overline{1} \lor \overline{2})$

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

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

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

Steps



1	2	3	4	5
T				

Example: 1 Watched Literal

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

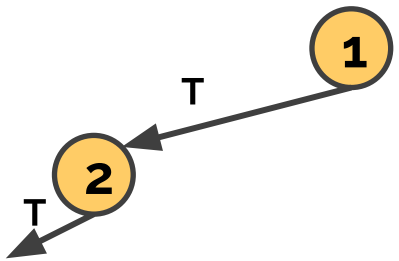
$(\bar{1} \lor 2 \lor \bar{3})$

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

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

1	2	3	4	5
T	T			

Steps



Example: 1 Watched Literal

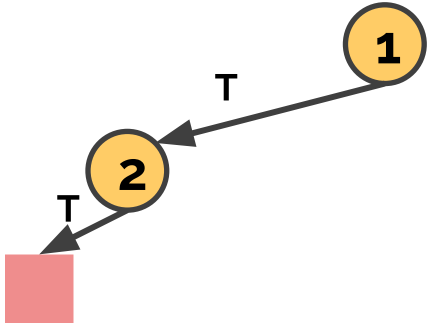
$(\bar{1} \vee \bar{2})$ **Conflict!**

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

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

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

Steps



1	2	3	4	5
T	T			

Example: 1 Watched Literal

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

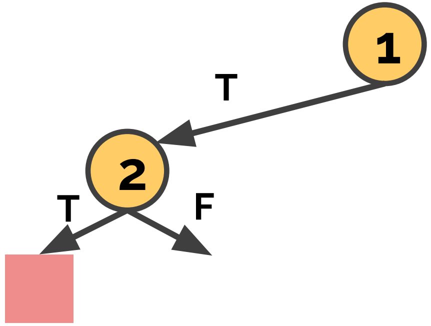
$(\overline{1} \vee 2 \vee \overline{3})$

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

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

1	2	3	4	5
T	F			

Steps



Example: 1 Watched Literal

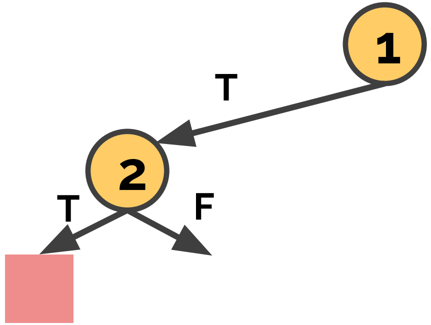
$(\overline{1} \vee \overline{2})$

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

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

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

Steps



1	2	3	4	5
T	F			

Example: 1 Watched Literal

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

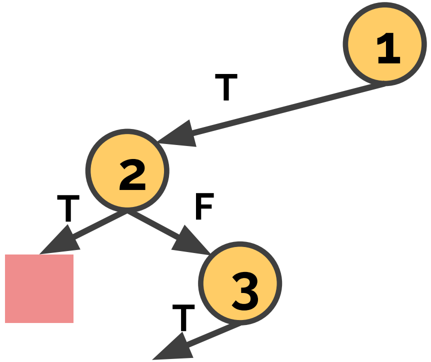
$(\overline{1} \vee 2 \vee \overline{3})$

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

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

1	2	3	4	5
T	F	T		

Steps



Example: 1 Watched Literal

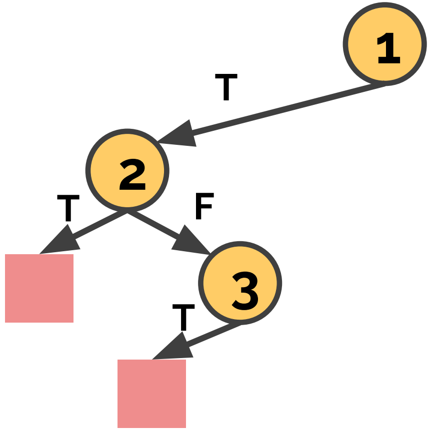
$(\overline{1} \lor \overline{2})$

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

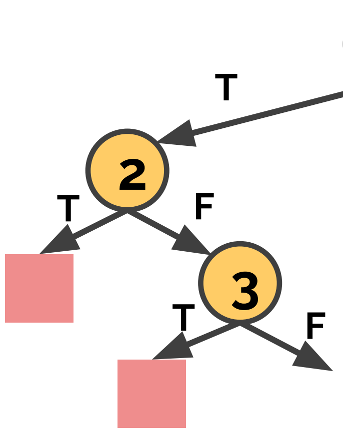
$(\overline{1} \vee \overline{2})$

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

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

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

Steps



1	2	3	4	5
T	F	F		

Example: 1 Watched Literal

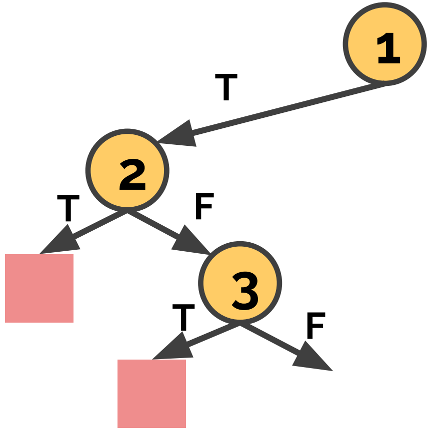
$(\overline{1} \vee \overline{2})$

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

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

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

Steps



1	2	3	4	5
T	F	F		

Example: 1 Watched Literal

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

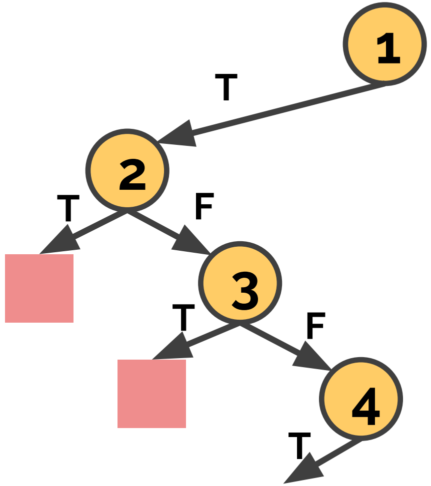
$(\overline{1} \vee 2 \vee \overline{3})$

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

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

1	2	3	4	5
T	F	F	T	

Steps



Example: 1 Watched Literal

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

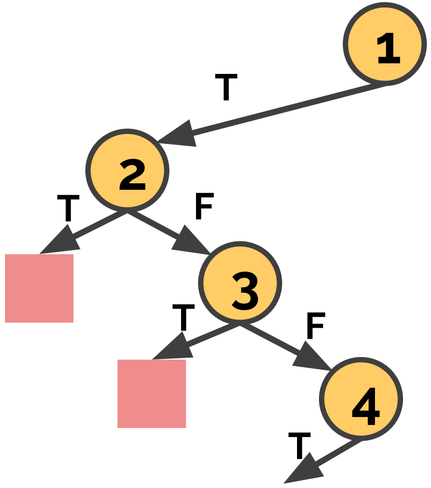
$(\overline{1} \vee 2 \vee \overline{3})$

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

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

1	2	3	4	5
T	F	F	T	

Steps



Example: 1 Watched Literal

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

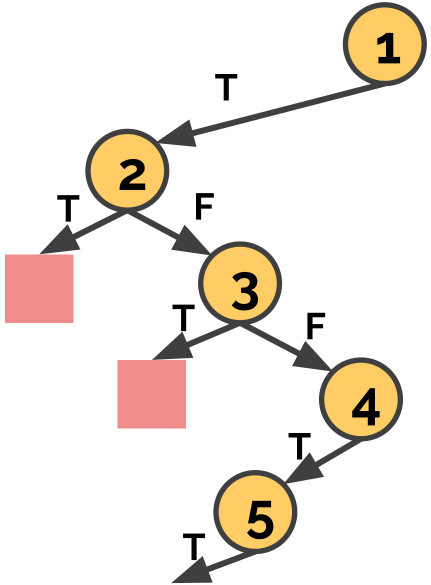
$(\overline{1} \vee 2 \vee \overline{3})$

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

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

1	2	3	4	5
T	F	F	T	T

Steps



Example: 1 Watched Literal

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

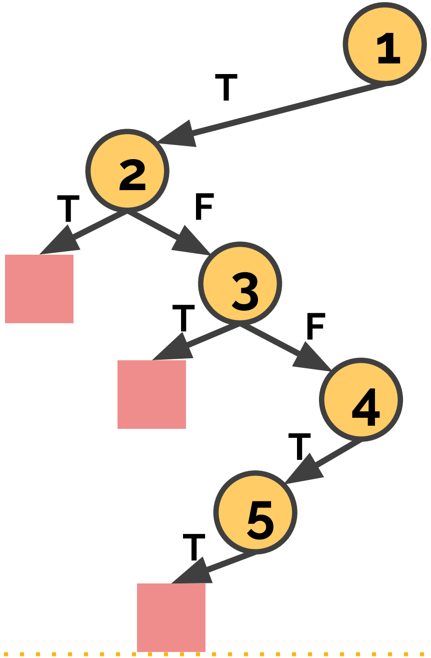
$(\overline{1} \lor 2 \lor \overline{3})$

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

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

1	2	3	4	5
T	F	F	T	T

Steps



Example: 1 Watched Literal

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

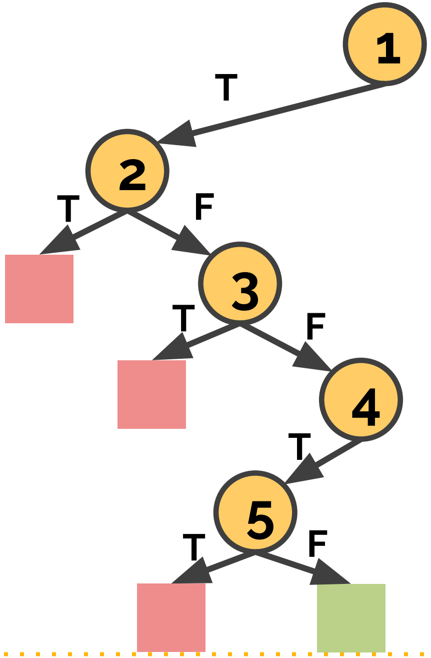
$(\overline{1} \vee 2 \vee \overline{3})$

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

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

1	2	3	4	5
T	F	F	T	F

Steps



Find a *satisfying assignment* for the following formula:

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

Find a *satisfying assignment* for the following formula:

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

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

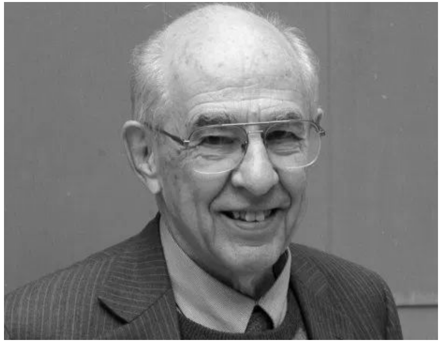
$x_4 = \texttt{TRUE}$ $x_5 = \texttt{FALSE}$ $x_6 = \texttt{TRUE}$ $x_7 = \texttt{TRUE}$

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

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

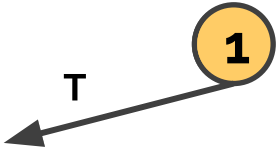
$(\overline{1} \vee 2)$

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

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

1	2	3	4
T			

Steps



Example: DPLL

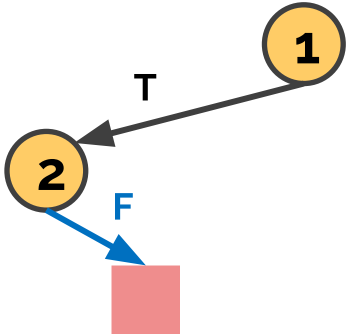
$(\overline{1} \vee \overline{2})$

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

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

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

Steps



1	2	3	4
T	F		

Example: DPLL

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

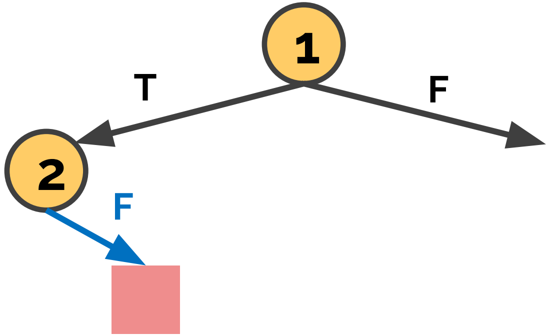
$(\overline{1} \vee 2)$

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

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

1	2	3	4
F			

Steps



Example: DPLL

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

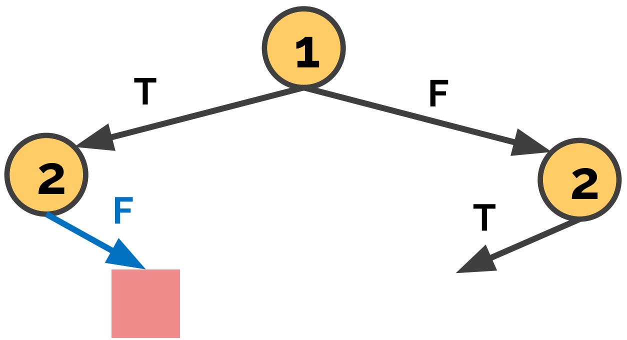
$(\overline{1} \vee 2)$

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

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

Unit!

Steps



1	2	3	4
F	T		

Example: DPLL

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

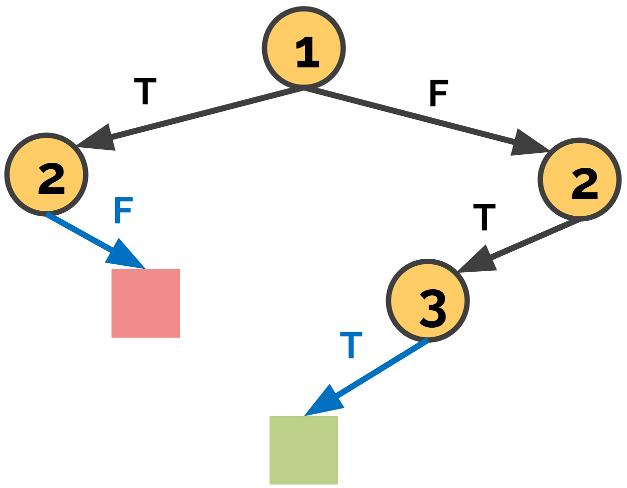
$(\bar{1} \vee 2)$

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

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

1	2	3	4
F	T	T	

Steps



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

$$\left(\overline{1} \vee 2 \vee \overline{3}\right) \xrightarrow{\text{Set } 1 = T} \left(\overline{1} \vee 2 \vee \overline{3}\right) \xrightarrow{\text{Set } 2 = F} \left(\overline{1} \vee 2 \vee \overline{3}\right)$$

Unit!

How should we branch?

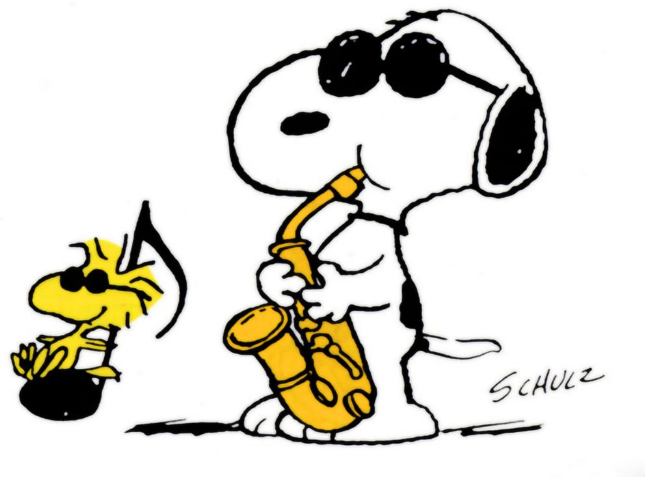
- Order of assigning variables greatly affects runtime
- Want to find a satisfying assignment quicker and find conflicts (rule out bad assignments) quicker

- **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.