Lecture 3: Solving Boolean Satisfiability

Huan Zhang

huan@huan-zhang.com

Slides adapted from Prof. Sayan Mitra's slides in Fall 2021
Some of the slides for this lecture are adapted from slides by Clark Barrett

Review: Boolean satisfiability problem

Given a *well-formed boolean formula* α, determine whether there exists a satisfying solution

We will assume α to be in *conjunctive normal form (CNF)*

literals: variable or its negation, e.g., x_3, $\neg x_3$

clause: disjunction (or) of literals, e.g., $(x_1 \vee x_2 \vee \neg x_3)$

CNF formula: conjunction (and) of clauses,

e.g., $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_1)$

A variable may appear *positively* or *negatively* in a clause

Review: Boolean satisfiability problem

Restatement: $\exists \boldsymbol{x} \in val(X): \boldsymbol{x} \vDash \alpha$?

If the answer is "No" then α is said to be *unsatisfiable*

SAT problem example:

$$\begin{aligned}\alpha := &(\neg x_1 \vee x_2) \wedge (\neg x_3 \vee x_4) \\ &\wedge (\neg x_2 \vee \neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \\ &\wedge (x_5 \vee x_7) \wedge (x_1 \vee x_6 \vee \neg x_7)\end{aligned}$$

Review: SAT is NP-complete

SAT was the first problem shown to be NP-complete [Cook 71]

1. Essentially we don't know better (in terms of asymptotic complexity) than naïve enumeration
2. A solver for SAT can be used to solve any other problem in the NP class with only polytime slowdown. i.e., makes a lot of sense to build SAT solvers
3. SAT/SMT solving is the cornerstone of *many* verification procedures

Stephen Cook, The complexity of theorem-proving procedures. In Proceedings of the third annual ACM symposium on theory of computing. STOC '71.

A simple greedy algorithm for SAT (GSAT)

Input: Set of clauses *C* over *X*, parameters *max-flips, max-tires*

Output: A satisfying assignment for C, or Ø if none found

for i = 1 to *max-tries*

v := random truth assignment in *val(X)*

for j = 1 to *max-flips*

if $v \vDash C$ then return v

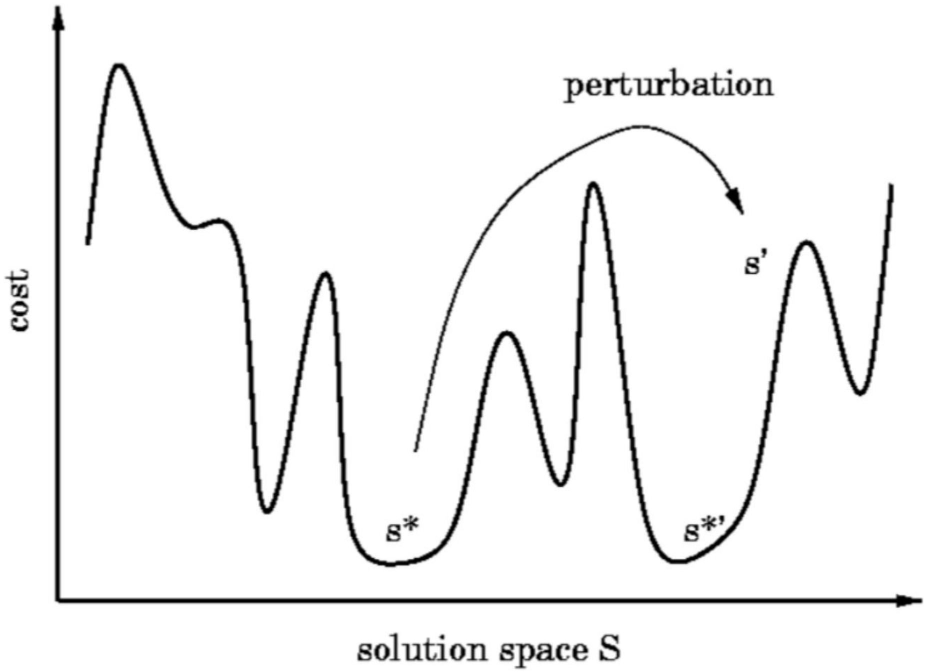
$p \coloneqq$ variable in C such that flipping its value gives the largest increase in the number of clauses of C that are satisfied by v

$v \coloneqq v$ with the assignment to p flipped

return Ø

e.g., $x_1x_2x_3x_4x_5$ = 00100 -> 00110

GSAT is a stochastic local search (SLS) algorithm



Limitation of this approach?

Local search algorithms are usually **incomplete**: they cannot show unsatisfiability!

Image Source: Alan Mackworth

SAT Solving with backtracking

Sysmetically enumerating all possibilities!

$$\alpha := (\cancel{\neg x_1} \lor x_2) \land (\neg x_3 \lor x_4)$$
$$\land (\neg x_2 \lor \neg x_5 \lor \neg x_6) \land (\neg x_5 \lor x_6)$$
$$\land (x_5 \lor \neg x_6 \lor \neg x_7) \land \cancel{(x_1 \lor x_6 \lor \neg x_7)}$$

First, assume x_1 is True, and substitute

Because (True ∨ A) = True, (False ∨ A) = A, we can simplify α and it becomes:

$$x_2 \land (\neg x_3 \lor x_4) \land (\neg x_2 \lor \neg x_5 \lor \neg x_6) \land (\neg x_5 \lor x_6) \land (x_5 \lor x_6 \lor \neg x_7)$$

But we still don't know if it is satisfiable!

SAT Solving with backtracking

After assuming x_1 is True and we get:

$$x_2 \wedge (\neg x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$$

Search tree

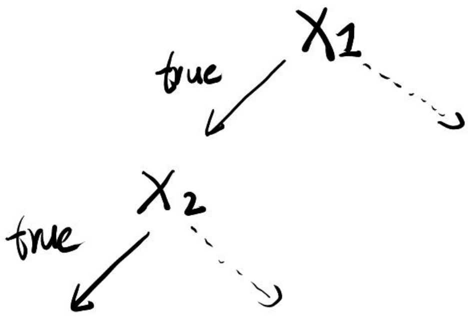
SAT Solving with backtracking

Then, let's substitute x_2 = True in

$$\cancel{x_2} \wedge (\neg x_3 \vee x_4) \wedge (\cancel{\neg x_2} \vee \neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$$

α is still unresolved:

$$(\neg x_3 \vee x_4) \wedge (\neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$$

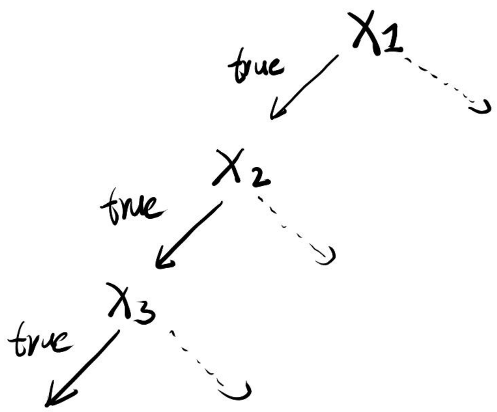


SAT Solving with backtracking

Keep setting and substituting variables

$$(\neg x_3 \vee x_4) \wedge (\neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$$

Set x_3 = True $\quad x_4 \wedge (\neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$



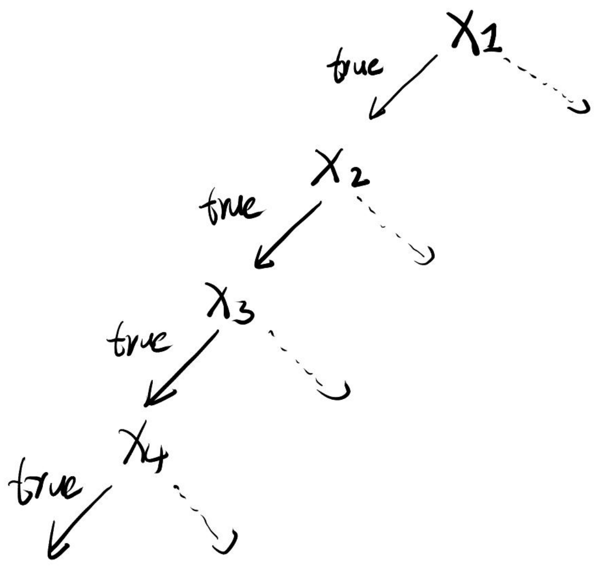
SAT Solving with backtracking

Keep setting variables

$$x_4 \land (\neg x_5 \lor \neg x_6) \land (\neg x_5 \lor x_6) \land (x_5 \lor \neg x_6 \lor \neg x_7)$$

Set x_4 = True

$$(\neg x_5 \lor \neg x_6) \land (\neg x_5 \lor x_6) \land (x_5 \lor \neg x_6 \lor \neg x_7)$$



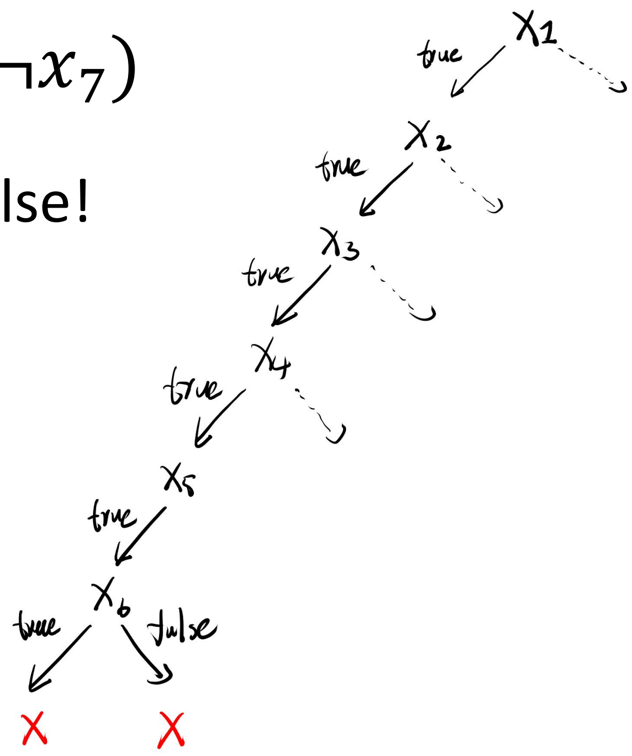
SAT Solving with backtracking

Keep setting variables

Set x_5 = True $(\neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$

Set x_6 = True $\neg x_6 \wedge x_6 \wedge (\neg x_6 \vee \neg x_7)$

Conflict, α evaluates to False!



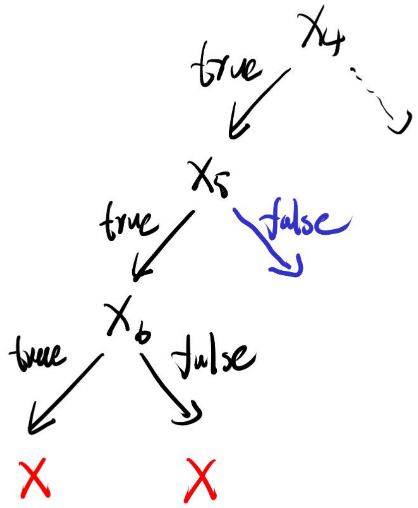
SAT Solving with backtracking

$$\neg x_6 \land x_6 \land (x_6 \lor \neg x_7)$$

Set x_6 = True does not work. Backtrack and try a different x_6.

Setting x_6 = False also does not work. Backtrack one-level up and try x_5

$$(\neg x_5 \lor \neg x_6) \land (\neg x_5 \lor x_6) \land (x_5 \lor x_6 \lor \neg x_7)$$



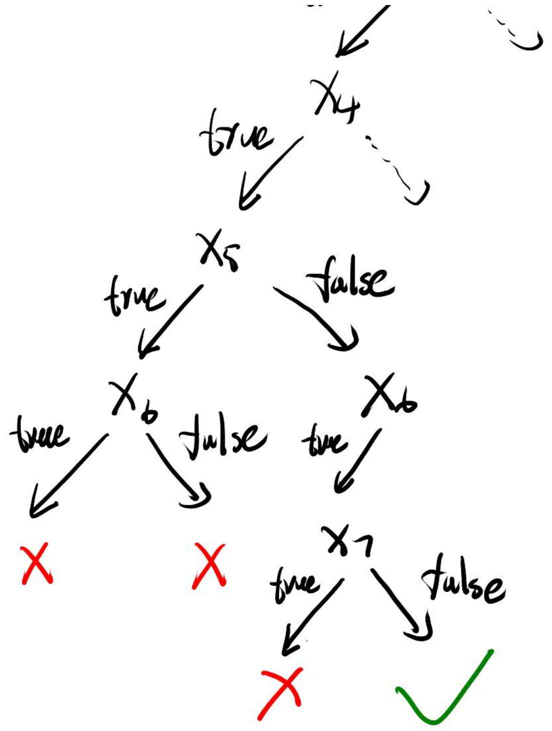
SAT Solving with backtracking

$$(\neg x_5 \lor \neg x_6) \land (\neg x_5 \lor x_6) \land (x_5 \lor x_6 \lor \neg x_7)$$

Now set x_5 = False $\quad \neg x_6 \lor \neg x_7$

Now set x_6 = True $\quad \neg x_7$

Now set x_7 = True, does not work.
Set x_7= False, α is now true!



SAT Solving with backtracking

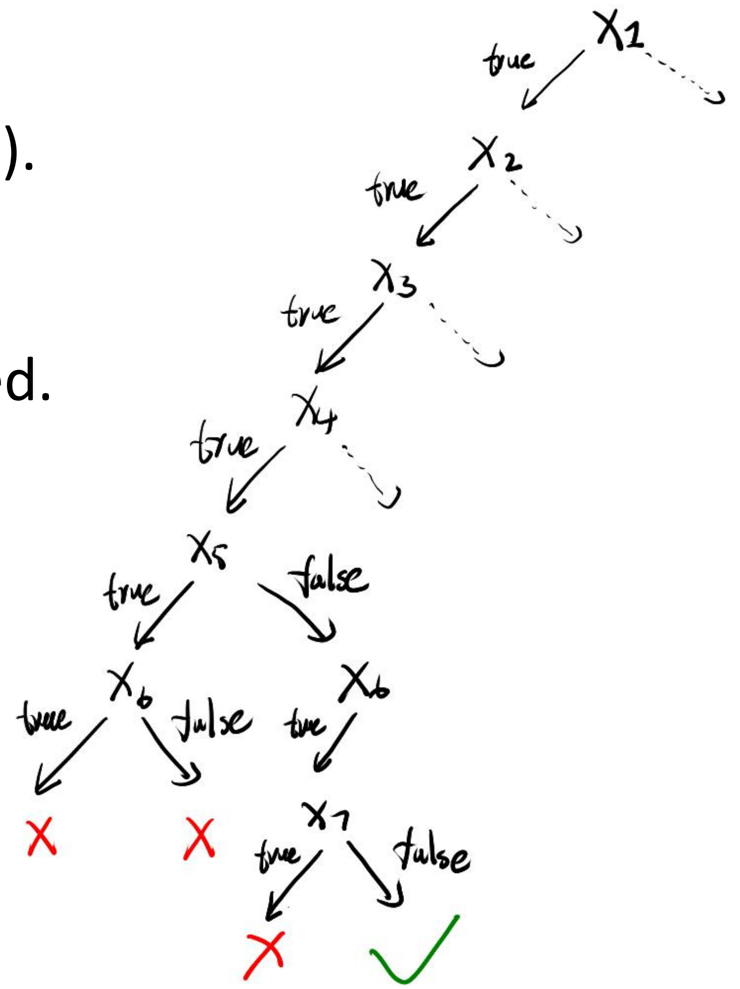
```
function BackTracking(α)
    if α is true then return true;
    if α is false then return false;
    // α is unresolved, need to decide on a literal
    l ← choose-literal(α);
    return (BackTracking(substitute l in α with true) or
            BackTracking(substitute l in α with false));
```

Search tree can be large!

Each variable is tested with two cases (true and false).
Complexity exponential to the number of variables.

To prove unsatifiability, the entire tree must be visited.

We need to reduce the number of variables that requires **decision** (try both true and false cases).



Davis Putnam Logemann Loveland Algorithm (DPLL) 1962

Backtracking with a few transformation rules to improve efficiency (reduce decision variables and search tree depth)

Transform the given formula α by applying a **sequence of satisfiability preserving rules**

If final result has an empty clause then *unsatisfiable*

if final result has no clauses then the formula is *satisfiable*

Transform 1: Unit propagation

A clause has a single literal

$$\alpha \equiv \ldots \wedge \ldots \wedge p \wedge \ldots \wedge \ldots$$

What choice do we really have?

$$\alpha \equiv \ldots \wedge (x_1 \vee \cancel{\neg p} \vee x_2) \wedge \cancel{p} \wedge \ldots \wedge (\neg x_3 \vee \cancel{\neg p} \vee x_1) \ldots$$

Transform 1: Unit propagation

A clause has a single literal

$$\alpha \equiv \ldots \wedge \ldots \wedge p \wedge \ldots \wedge \ldots$$

All clauses mentioning $\neg p$ have this literal deleted

All clauses mentioning p are deleted

$$\alpha' \equiv \ldots \wedge (x_1 \vee x_2) \wedge \ldots \wedge (\neg x_3 \vee x_1) \ldots$$

α and α' are **equisatisfiable**

Transform 1: Unit propagation

How about

$$\alpha \equiv \ldots \wedge \ldots \wedge p \wedge \ldots \wedge (\neg p) \wedge \ldots$$

By deleting $\neg p$, we have an "**empty clause**" which means α is unsatisifiable

$$\alpha \equiv \ldots \wedge \ldots \wedge \ldots \wedge () \wedge \ldots$$

Transform 2: Pure literal

A literal appears only positively (or negatively) in α

$$\alpha \equiv \ldots \wedge (x_1 \vee \neg p \vee x_2) \wedge (x_4 \vee \neg p) \wedge \ldots \wedge (\neg x_3 \vee \neg p \vee x_1)\ldots$$

p does not appear anywhere

Makes sense to set $p = 0$ and remove all occurrences of $\neg p$

Transform 2: Pure literal

A literal appears only positively (or negatively) in α

$$\alpha \equiv \ldots \wedge (x_1 \vee \neg p \vee x_2) \wedge (x_4 \vee \neg p) \wedge \ldots \wedge (\neg x_3 \vee \neg p \vee x_1) \wedge (\neg x_3 \vee x_4)\ldots$$

p does not appear anywhere

Makes sense to set $p = 0$ and remove all occurrences of $\neg p$

$$\alpha' \equiv \ldots \wedge \ldots \wedge \ldots \wedge (\neg x_3 \vee x_4)\ldots [p = 0]$$

α and α' are **equisatisfiable**

DPLL Algorithm: backtracking with unit-propagation and pure-literal assignment

```
function DPLL(α)
    α ← unit-propagate(α);
    α ← pure-literal-assign(α);
    // stopping conditions:
    if α is empty then return true;
    if α contains an empty clause then return false;
    // DPLL procedure:
    l ← choose-literal(α);
    return DPLL(α ∧ {l}) or DPLL(α ∧ {¬l});
```

DPLL Algorithm example

$\alpha := (\neg x_1 \vee x_2) \wedge \cancel{(\neg x_3 \vee x_4)}$
$\wedge (\neg x_2 \vee \neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6)$
$\wedge (x_5 \vee \neg x_6 \vee \neg x_7) \wedge (x_1 \vee x_6 \vee \neg x_7)$

```
function DPLL(α)
    unit-propagate
    pure-literal-assign
    check-stopping-conditions
    l ← choose-literal(α);
    return (DPLL(α ∧ {l}) or
            DPLL(α ∧ {¬l}));
```

Possible to apply unit propagation?

Possible to apply pure literal assignment?

We can essentially remove x_3 and x_4 from the search tree!

DPLL Algorithm example

$\alpha := (\neg x_1 \vee x_2) \wedge \cancel{(\neg x_3 \vee x_4)}$
$\wedge (\neg x_2 \vee \neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6)$
$\wedge (x_5 \vee \neg x_6 \vee \neg x_7) \wedge (x_1 \vee x_6 \vee x_7)$

We decide to choose x_1 and search the two cases

```
function DPLL(α)
    unit-propagate
    pure-literal-assign
    check-stopping-conditions
    l ← choose-literal(α);
    return (DPLL(α ∧ {l}) or
            DPLL(α ∧ {¬l}));
```

DPLL Algorithm example

$(\neg x_1 \lor x_2) \land (\neg x_2 \lor \neg x_5 \lor \neg x_6) \land (\neg x_5 \lor x_6) \land (x_5 \lor \neg x_6 \lor \neg x_7)$
$\land (x_1 \lor x_6 \lor x_7) \land x_1$

Possible to apply unit propagation? Always Yes!

DPLL Algorithm example

$(\cancel{\neg x_1} \vee x_2) \wedge (\neg x_2 \vee \neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$
$\wedge \cancel{(x_1 \vee x_6 \vee x_7)} \wedge \cancel{x_1}$

Possible to apply unit propagation? Always Yes!

$$x_2 \wedge (\neg x_2 \vee \neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$$

Possible to apply unit propagation again?

DPLL Algorithm example

$(\cancel{\neg x_1} \vee x_2) \wedge (\neg x_2 \vee \neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$
$\wedge\ \cancel{(x_1 \vee x_6 \vee x_7)}\ \wedge \cancel{x_1}$

Possible to apply unit propagation? Always Yes!

$$\cancel{x_2} \wedge (\cancel{\neg x_2} \vee \neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$$

Possible to apply unit propagation again?

$$(\neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$$

Possible to apply unit propagation again?

DPLL Algorithm example

$$(\neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$$

Possible to apply pure-literal assigment?

DPLL Algorithm example

$$\cancel{(\neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)}$$

Possible to apply pure-literal assigment?

$$(\neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6)$$

Possible to apply pure-literal assigment again?

DPLL Algorithm example

$$(\neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee x_6) \wedge (x_5 \vee \neg x_6 \vee \neg x_7)$$

Possible to apply pure-literal assignment?

$$\cancel{(\neg x_5 \vee \neg x_6)} \wedge \cancel{(\neg x_5 \vee x_6)}$$

Possible to apply pure-literal assignment again?

Empty clause left, we are done (return true).

With unit-propagation and pure-literal assignment, search process is much facster!

DPLL Algorithm

```
function DPLL(α)
    α ← unit-propagate(α);
    α ← pure-literal-assign(α);
    // stopping conditions:
    if α is empty then return true;
    if α contains an empty clause then return false;
    l ← choose-literal(α); // We decided to choose x1
    return DPLL(α ∧ {l}) or DPLL(α ∧ {¬l});
```

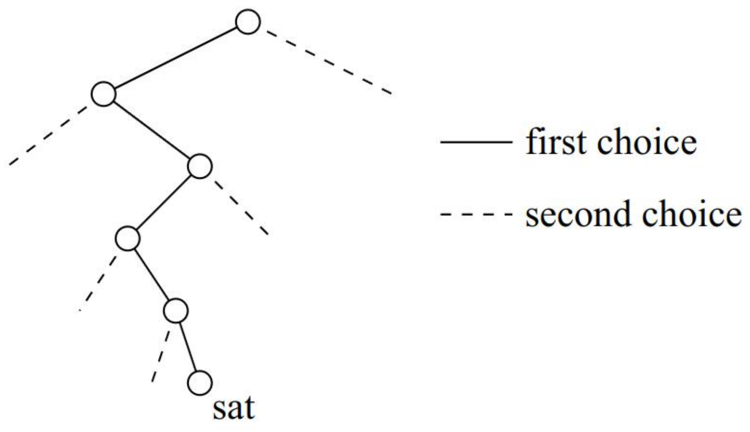
First condition returns true. No need to execute the second DPLL call.

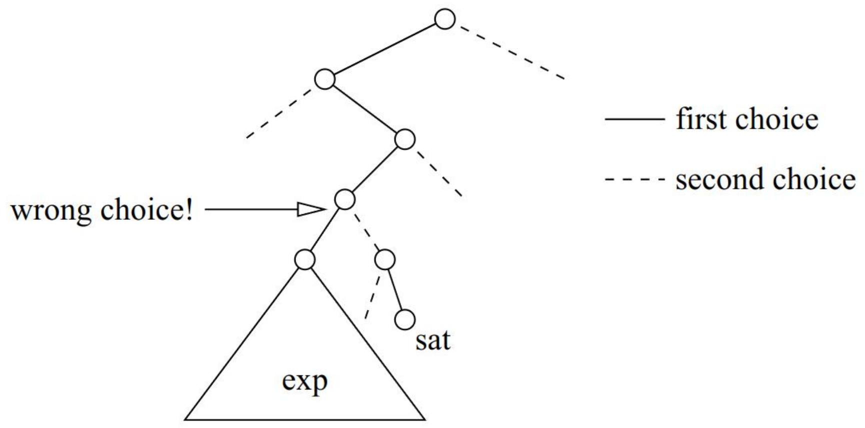
DPLL Algorithm

```
function DPLL(α)
    α ← unit-propagate(α);
    α ← pure-literal-assign(α);
    // stopping conditions:
    if α is empty then return true;
    if α contains an empty clause then return false;
    l ← choose-literal(α); // We decided to choose x1
    return DPLL(α ∧ {l}) or DPLL(α ∧ {¬l});
```

The order of choosing literals is important - it usually defines the size of the search tree!

DPLL Algorithm: choosing literals

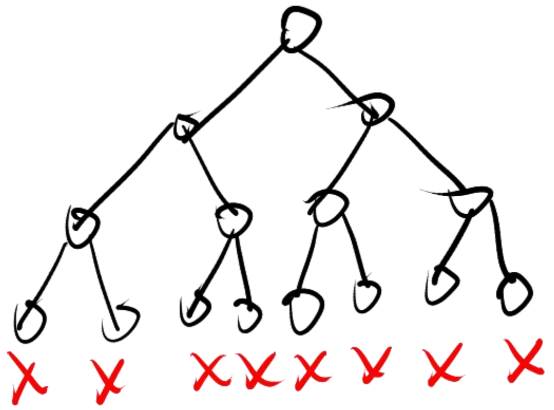


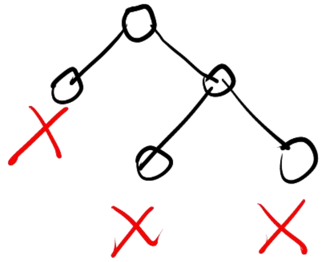


The order of choosing literals is important - it usually defines the size of the search tree!

Image from https://www.diag.uniroma1.it/~liberato/ar/dpll/dpll.html

DPLL Algorithm: choosing literals





Proving unsatifiability is even harder. To explore the search tree faster, we want to find conflicts earlier. Roughly speaking, more clauses lead to more conflicts.

Modern DPLL with Conflict-driven Clause Learning

What can we do if we find a conflict in DPLL?

Let's say we set x_1=1, x_3=0, x_5=1, leading to a conflict in α

Then we know that $(x_1 \land \neg x_3 \land x_5)$ => conflict

No conflict => $\neg(x_1 \land \neg x_3 \land x_5) \ = \neg x_1 \lor x_3 \lor \neg x_5$

if A => B, then
(not B) => (not A)

Then we know $\alpha \ \land (\neg x_1 \lor x_3 \lor \neg x_5 \ )$ is equisatisfiable. The added clause helps to reduce search tree size.

Checkout "Handbook of Satisfiability" for more details

Assignments

- HW1 (due Feb 11th)
  - Install Z3

- Keep thinking about class projects! Form teams (max 2 people).

- Next lecture: SMT solving