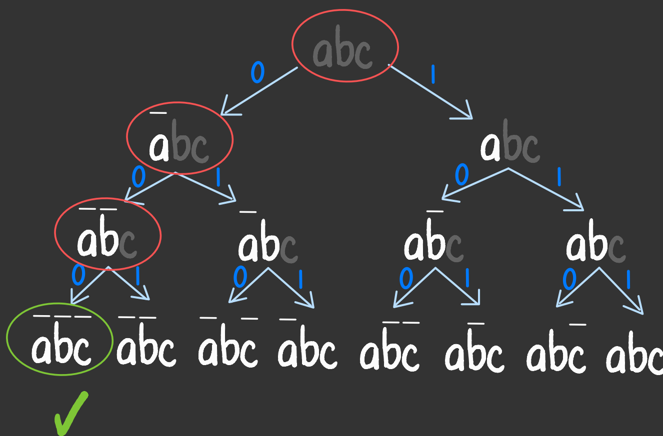
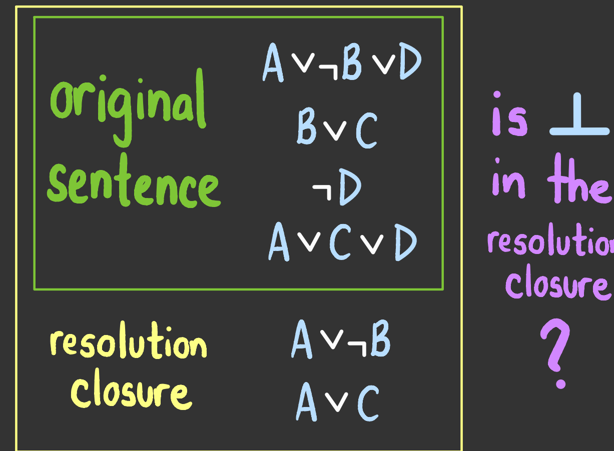
dpll

CSCI 373

search-based

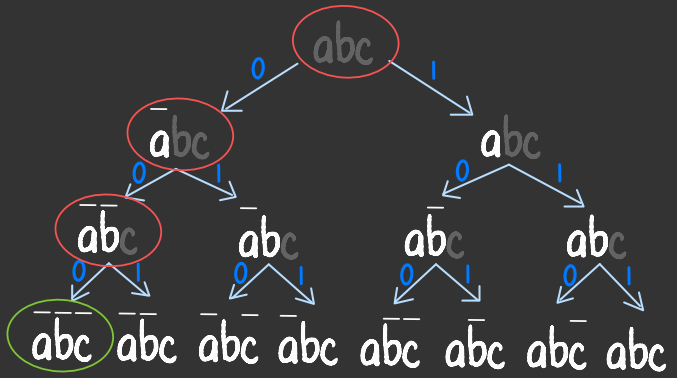


resolution-based



two satisfiability solvers

search-based



✓

terminates early if it finds a satisfying model

(only for satisfiable sentences)

resolution-based

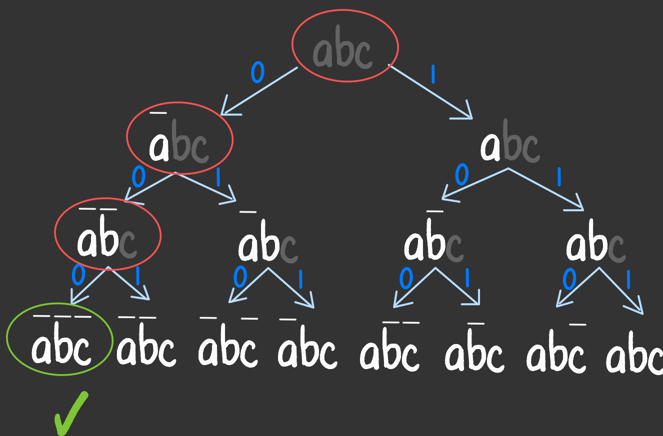
original sentence	$A \vee \neg B \vee D$
	$B \vee C$
	$\neg D$
	$A \vee C \vee D$
resolution closure	$A \vee \neg B$
	$A \vee C$

is $\bot$ in the resolution closure ?

terminates early if it entails $\bot$

(only for unsatisfiable sentences)

search-based



resolution-based

original sentence	$A \lor \neg B \lor D$ $B \lor C$ $\neg D$ $A \lor C \lor D$
resolution closure	$A \lor \neg B$ $A \lor C$

is $\perp$ in the resolution closure ?

can we combine these approaches?

could we use resolution here?

```
findSatisfyingModel(α):
  container = new stack()
  container.put({})
  repeat:
    if container.empty() then return "unsatisfiable"
    m = container.get()
    if m ∈ I(α) then return m
    if α ∧ term(m) is unsatisfiable:
      do nothing
    else:
      for m' ∈ successors(m):
        container.put(m')
```

could we use resolution here?

```
findSatisfyingModel(α):
  container = new stack()
  container.put({})
  repeat:
    if container.empty() then return "unsatisfiable"
    m = container.get()
    if m ∈ I(α) then return m
    if ⊥ ∈ RC(α ∧ term(m)):
      do nothing
    else:
      for m' ∈ successors(m):
        container.put(m')
```

what is the resolution closure?

$\neg A \vee B \vee C$

$A \vee D \vee E$

$\neg A \vee C \vee \neg F$

$A \vee \neg D \vee G$

$B \vee C \vee D \vee E$

$C \vee D \vee E \vee \neg F$

$A \vee E \vee G$

$C \vee \neg D \vee \neg F \cdot \vee G$

what is the resolution closure?

$\neg A \vee B \vee C$

$A \vee D \vee E$

$\neg A \vee C \vee \neg F$

$A \vee \neg D \vee G$

$B \vee C \vee D \vee E$

$B \vee C \vee \neg D \vee G$

$C \vee D \vee E \vee \neg F$

$C \vee \neg D \vee \neg F \vee G$

$B \vee C \vee E \vee \neg F \vee G$

$A \vee B \vee C \vee E \vee G$

etc.

even starting with only 4 clauses of length 3,
the resolution closure can grow large

$\neg A \vee B \vee C$

$A \vee D \vee E$

$\neg A \vee C \vee \neg F$

$A \vee \neg D \vee G$

$B \vee C \vee D \vee E$

$B \vee C \vee \neg D \vee G$

$C \vee D \vee E \vee \neg F$

$C \vee \neg D \vee \neg F \vee G$

$B \vee C \vee E \vee \neg F \vee G$

$A \vee B \vee C \vee E \vee G$

etc.

in practice,
this check is
too expensive

```
findSatisfyingModel(α):
  container = new stack()
  container.put({})
  repeat:
    if container.empty() then return "unsatisfiable"
    m = container.get()
    if m ∈ I(α) then return m
    if ⊥ ∈ RC(α ∧ term(m)):
      do nothing
    else:
      for m′ ∈ successors(m):
        container.put(m′)
```

recall:

the unsatisfiability test needs to be sound, but it does not have to be complete

```
findSatisfyingModel(α):
  container = new stack()
  container.put({})
  repeat:
    if container.empty() then return "unsatisfiable"
    m = container.get()
    if m ∈ I(α) then return m
    if α ∧ term(m) is unsatisfiable:
      do nothing
    else:
      for m' ∈ successors(m):
        container.put(m')
```

one way to save time:
only perform resolution
if one of the clauses
has length 1

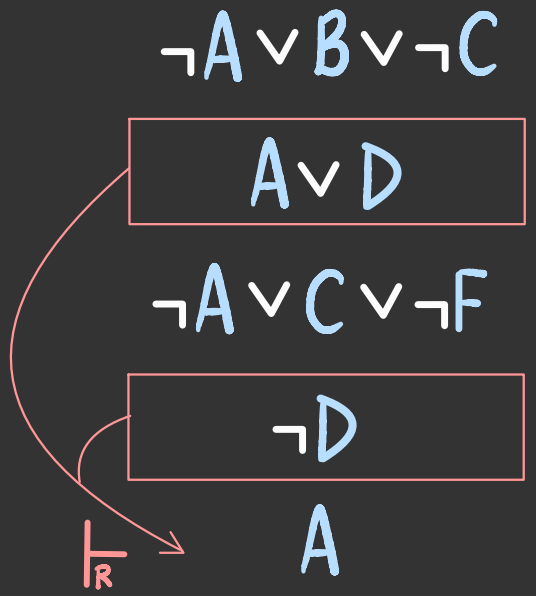
$\neg A \vee B \vee \neg C$

$A \vee D$

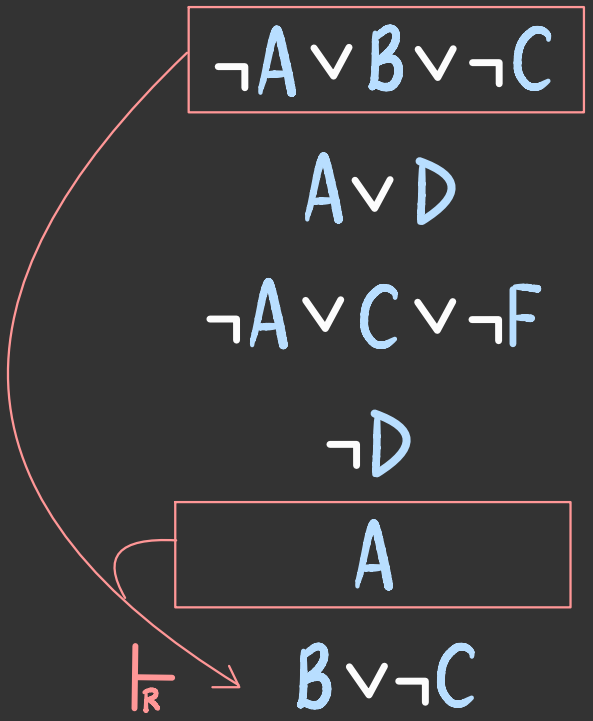
$\neg A \vee C \vee \neg F$

$\neg D$

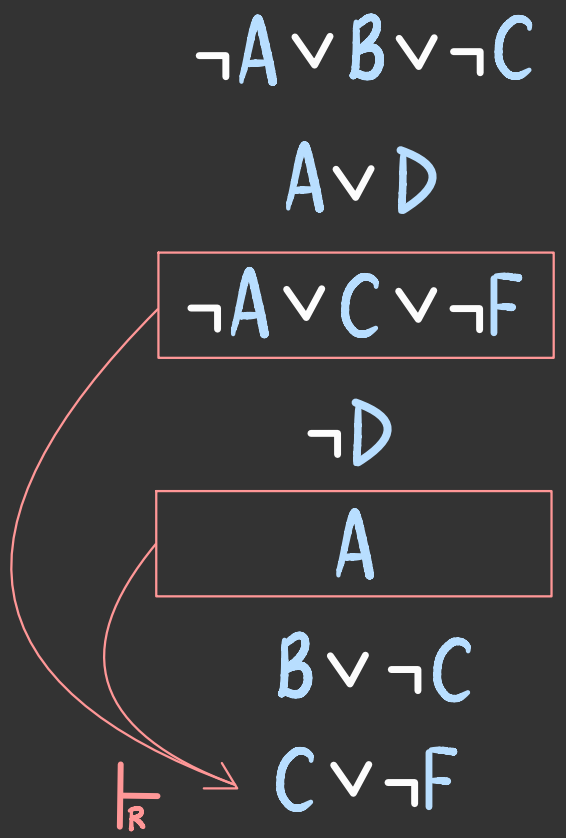
one way to save time:
only perform resolution
if one of the clauses
has length 1



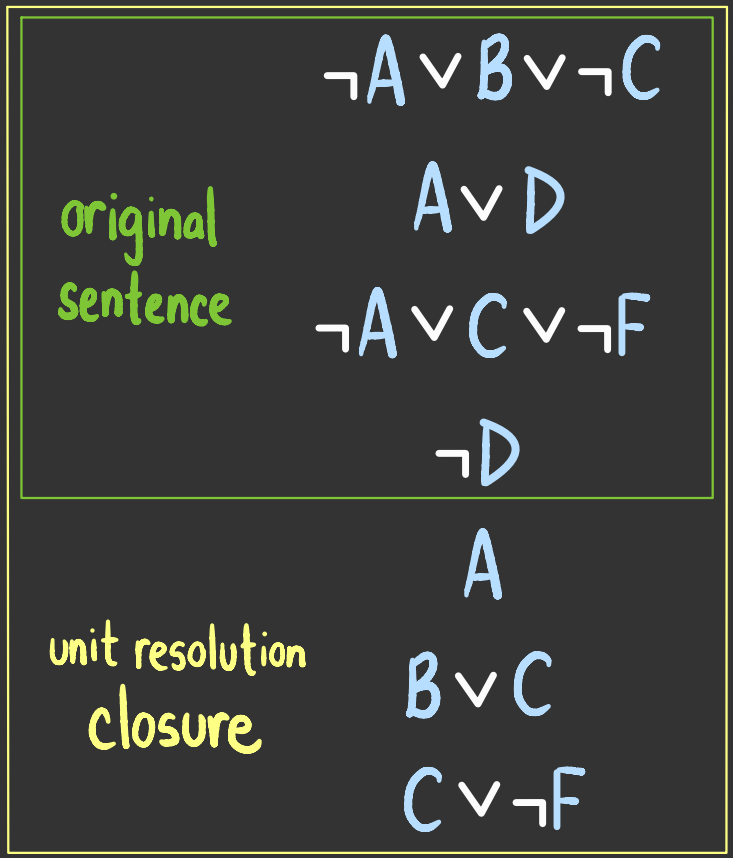
one way to save time:
only perform resolution
if one of the clauses
has length 1



one way to save time:
only perform resolution
if one of the clauses
has length 1



this is called unit resolution

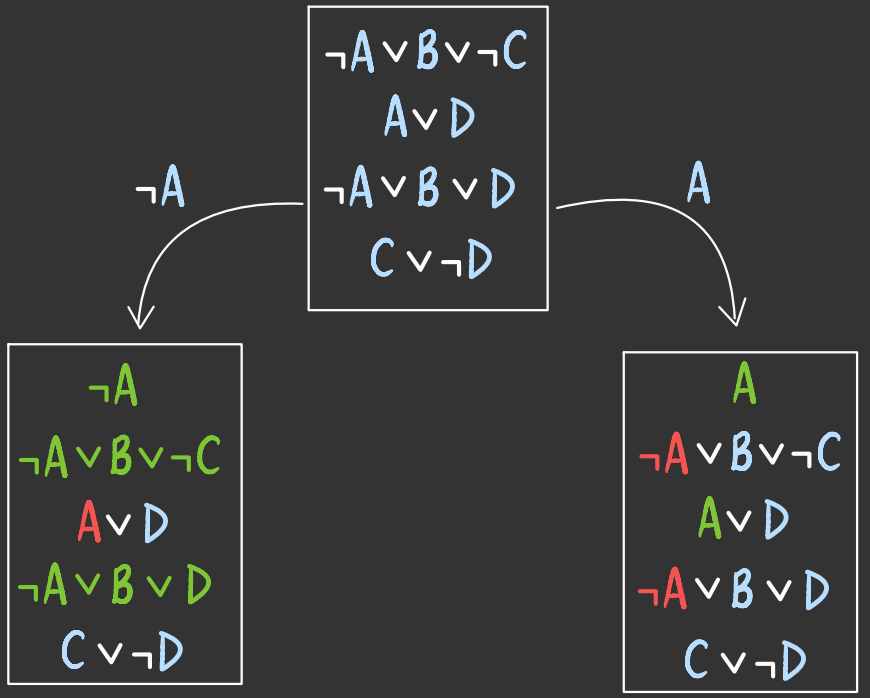


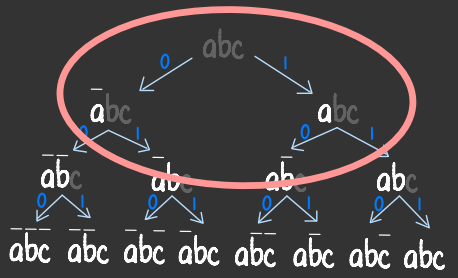
Computing the unit resolution closure turns out to be an effective strategy for pruning the search tree

```
findSatisfyingModel(α):
  container = new stack()
  container.put({})
  repeat:
    if container.empty() then return "unsatisfiable"
    m = container.get()
    if m ∈ I(α) then return m
    if ⊥ ∈ URC(α ∧ term(m)):
      do nothing
    else:
      for m′ ∈ successors(m):
        container.put(m′)
```

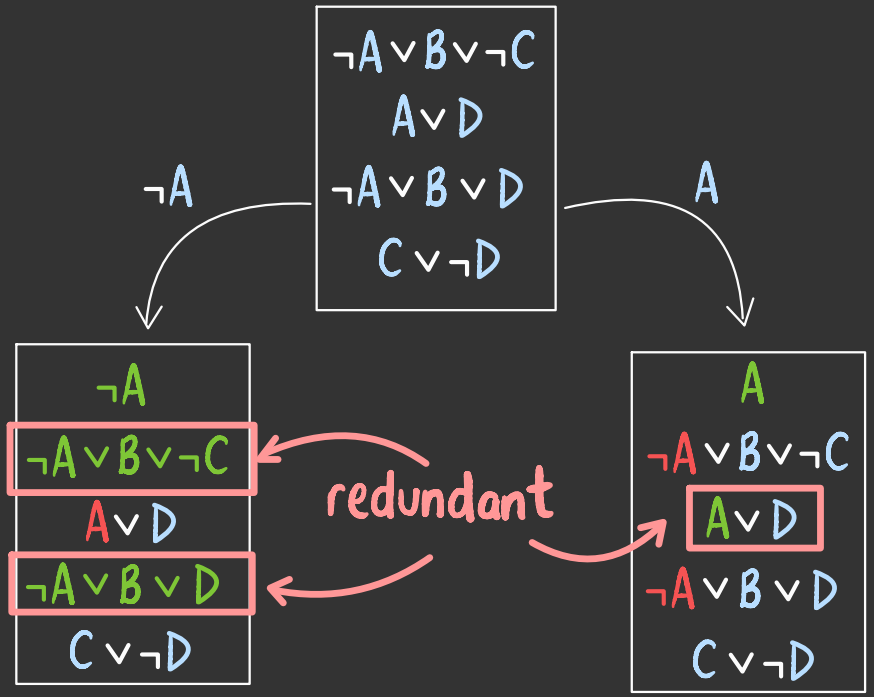
the resulting algorithm is called DPLL

```
dpll(α):
  container = new stack()
  container.put({})
  repeat:
    if container.empty() then return "unsatisfiable"
    m = container.get()
    if m ∈ I(α) then return m
    if ⊥ ∈ URC(α ∧ term(m)):
      do nothing
    else:
      for m' ∈ successors(m):
        container.put(m')
```

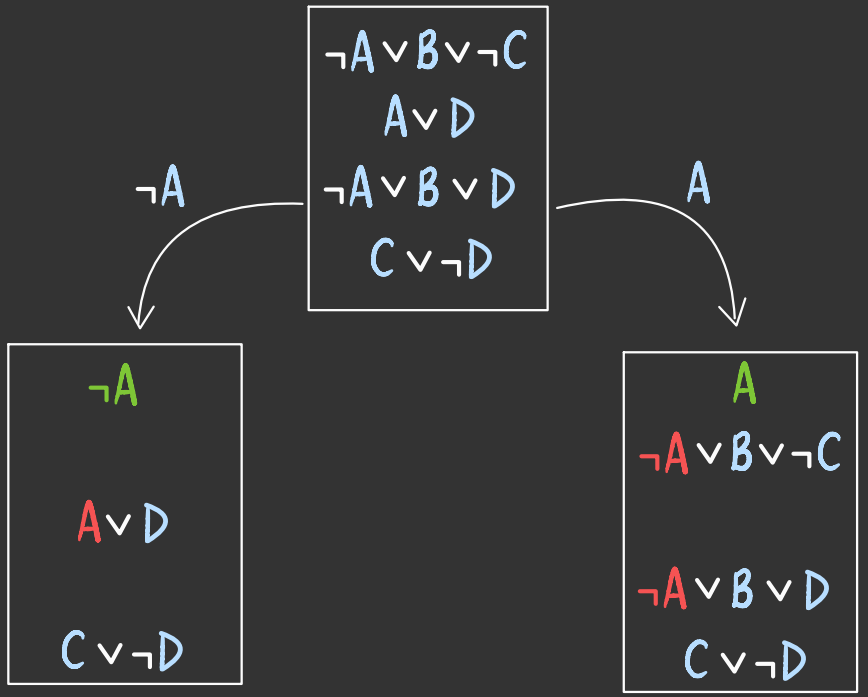




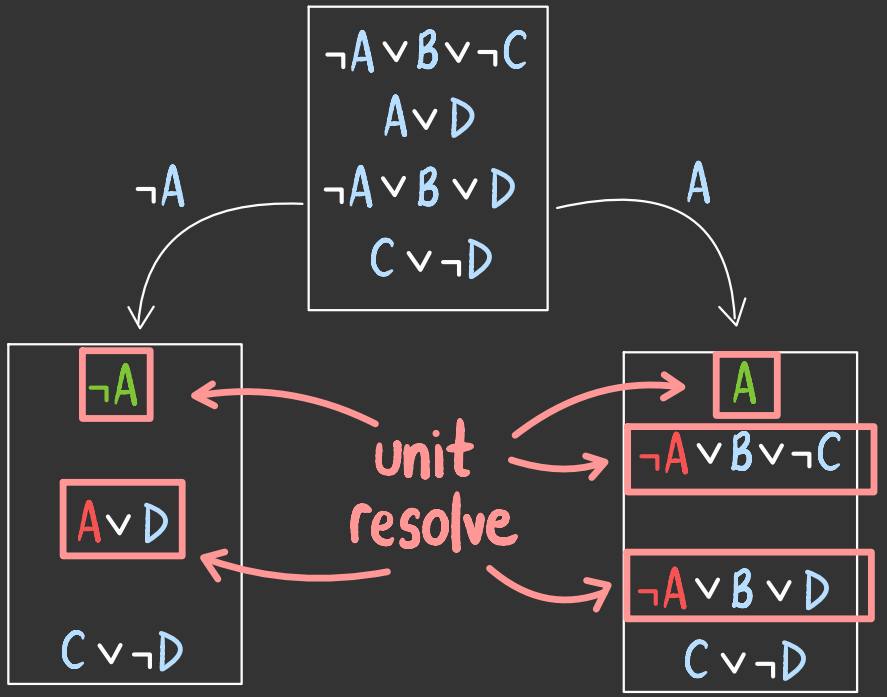
unit resolution in practice



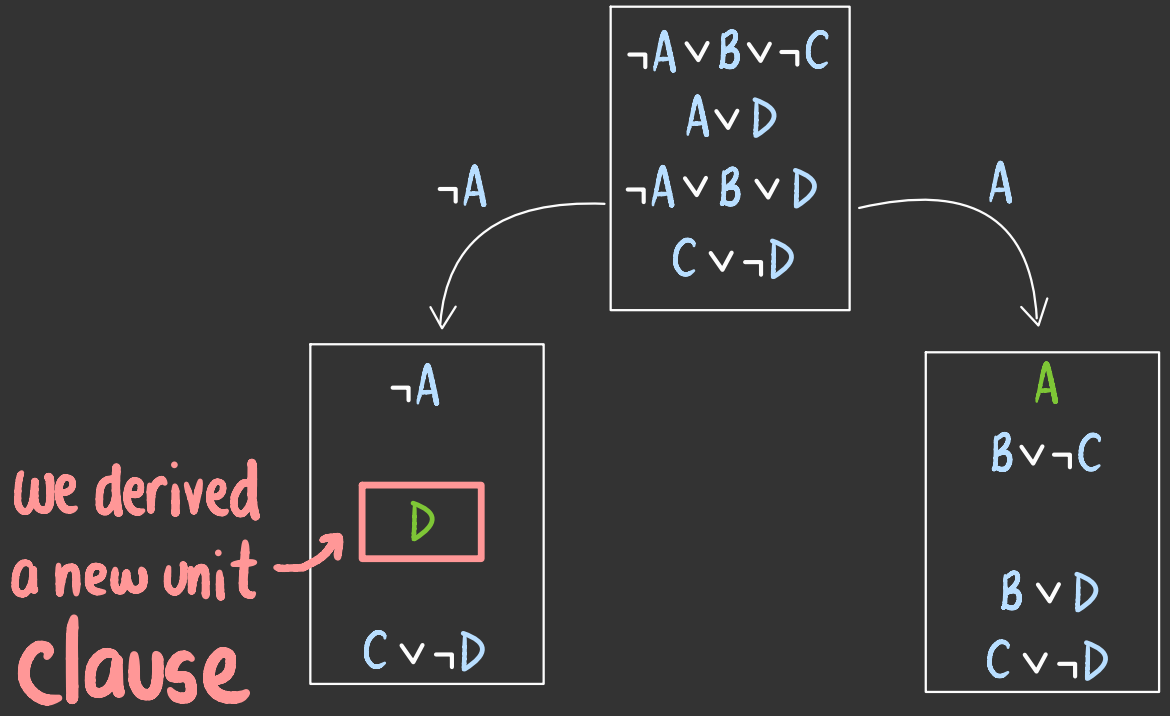
unit resolution in practice



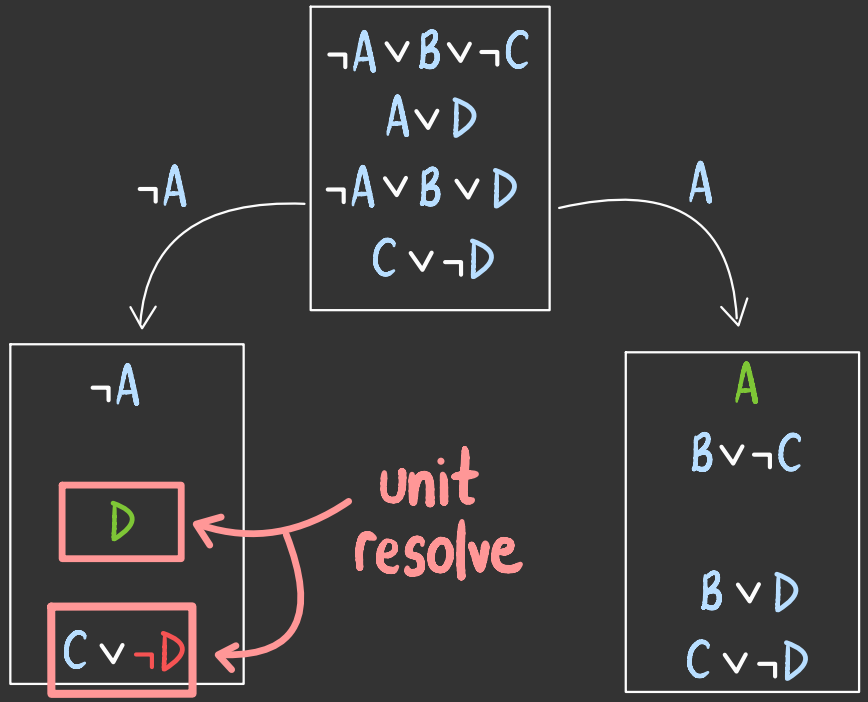
unit resolution in practice



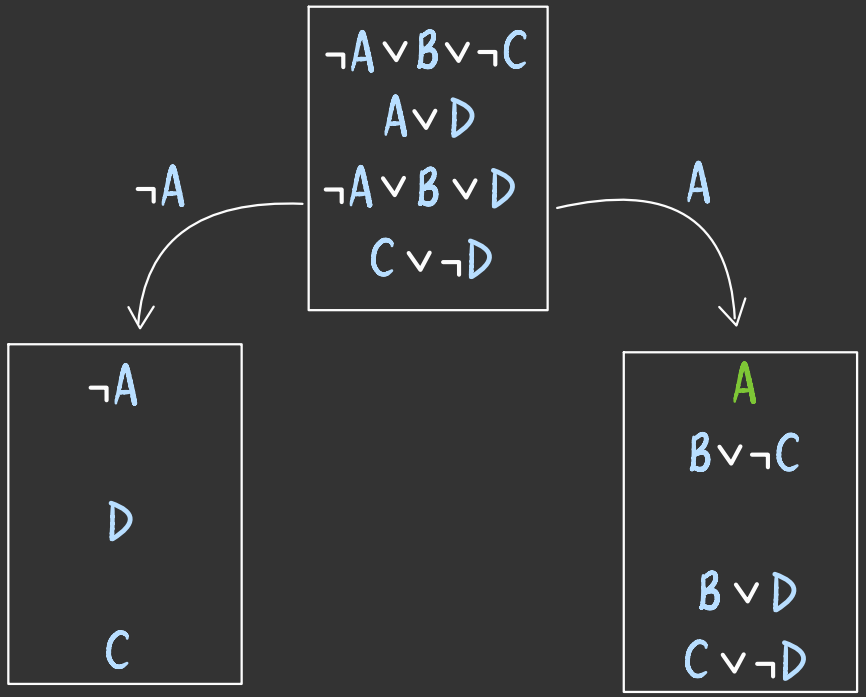
unit resolution in practice



unit resolution in practice

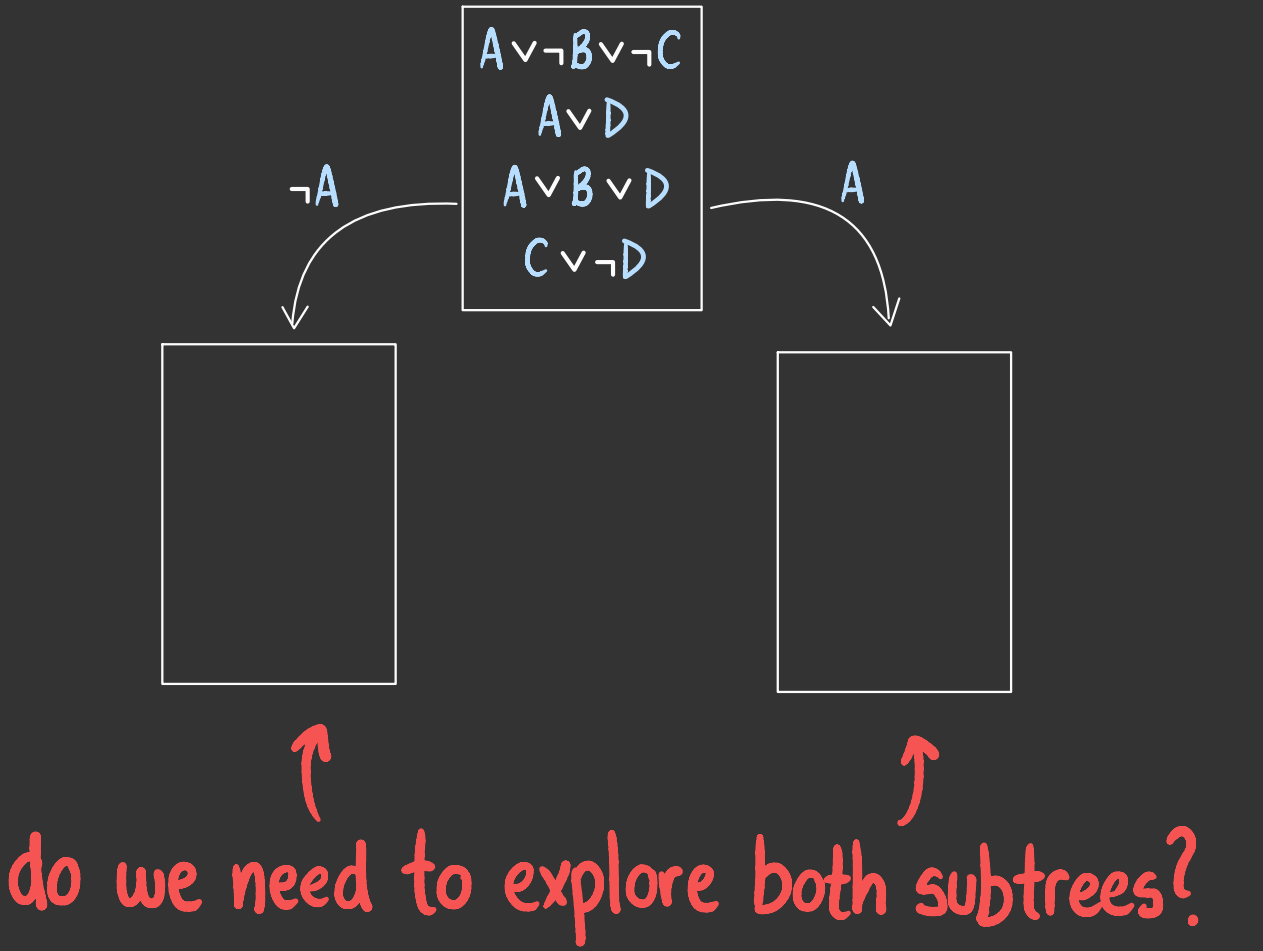


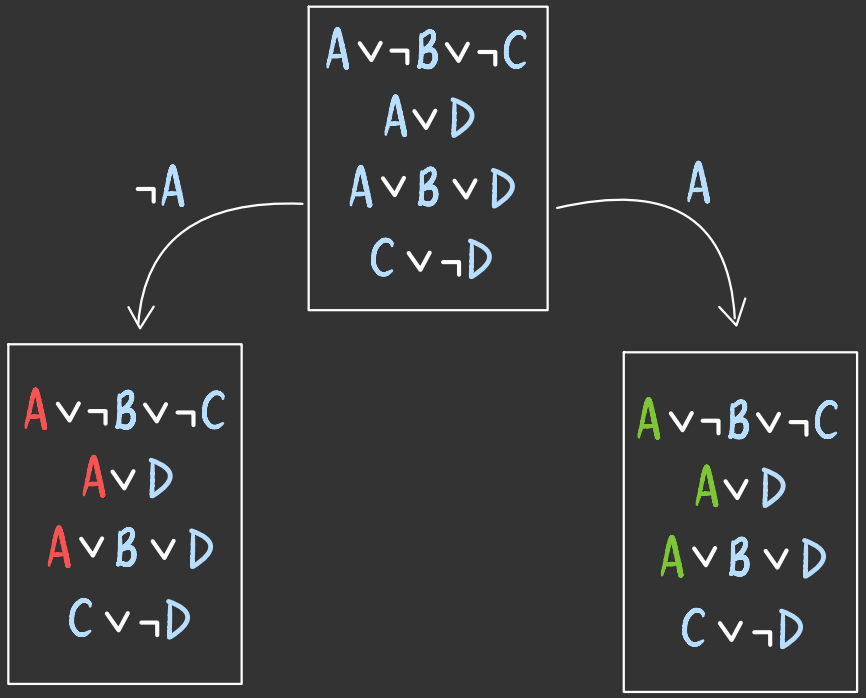
unit resolution in practice

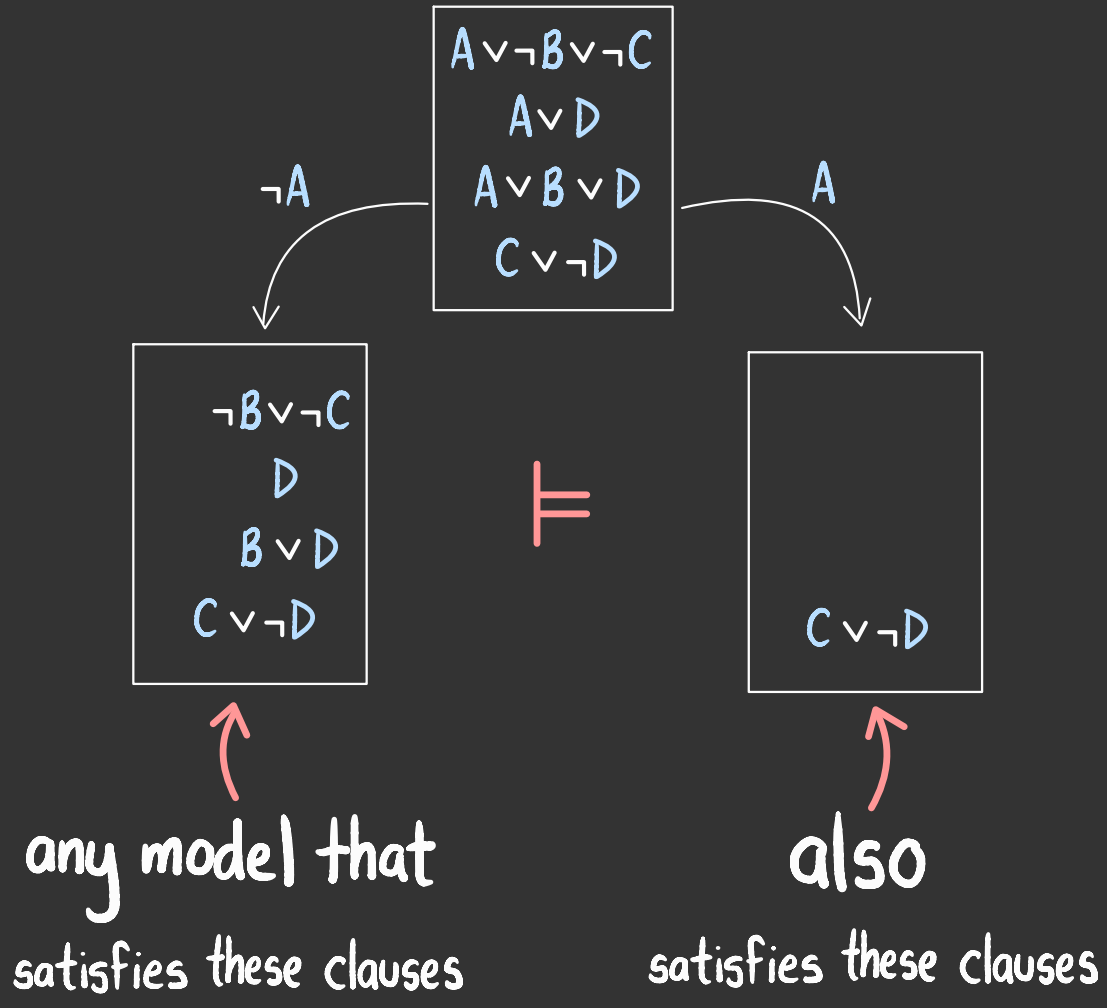


unit resolution in practice

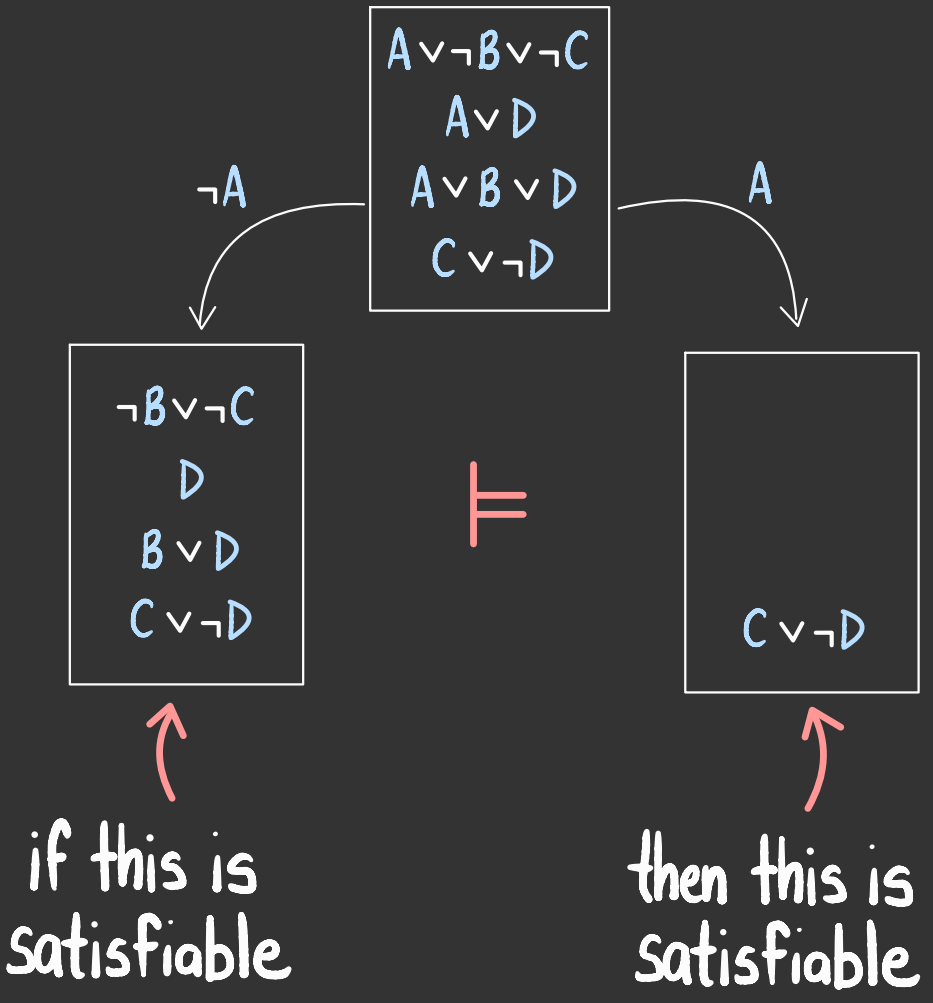
DPLL often incorporates additional optimizations


A∨¬B∨¬C
A∨D
A∨B∨D
C∨¬D
¬A
A
do we need to explore both subtrees?

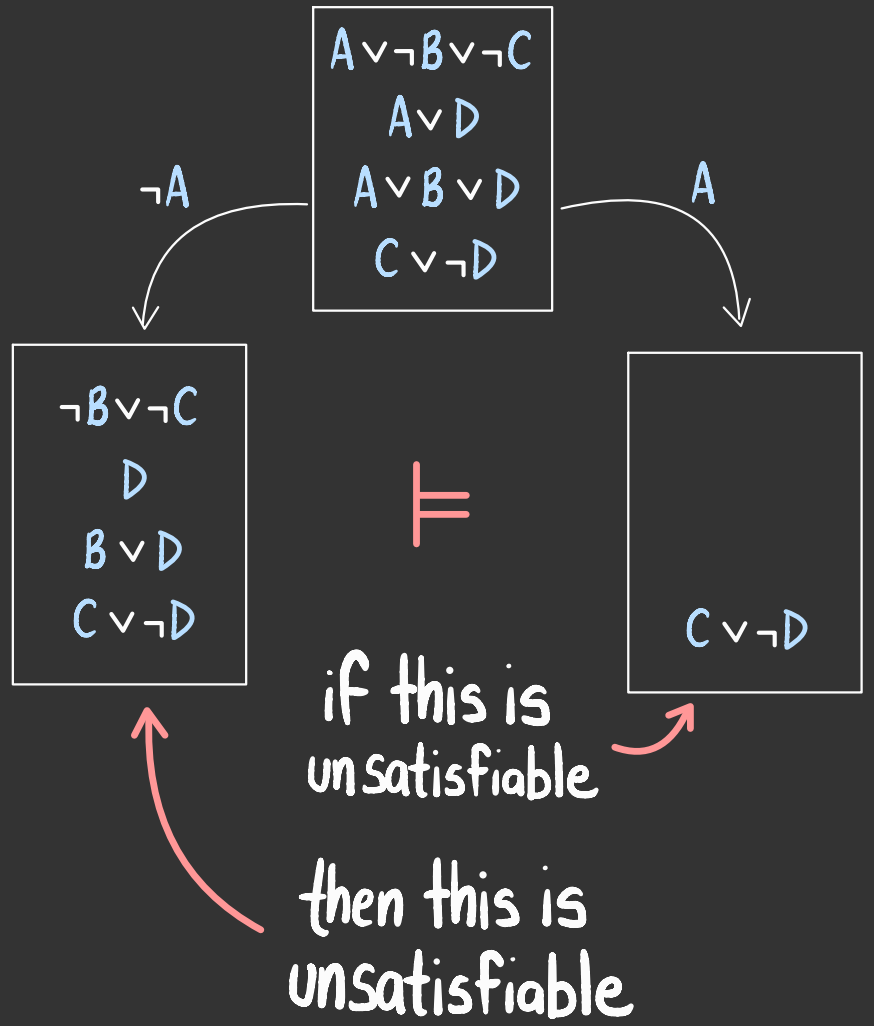

A∨¬B∨¬C
A∨D
A∨B∨D
C∨¬D
¬A
A
A∨¬B∨¬C
A∨D
A∨B∨D
C∨¬D
A∨¬B∨¬C
A∨D
A∨B∨D
C∨¬D


A∨¬B∨¬C
A∨D
A∨B∨D
C∨¬D
¬A
A
¬B∨¬C
D
B∨D
C∨¬D
⊨
C∨¬D
any model that
satisfies these clauses
also
satisfies these clauses

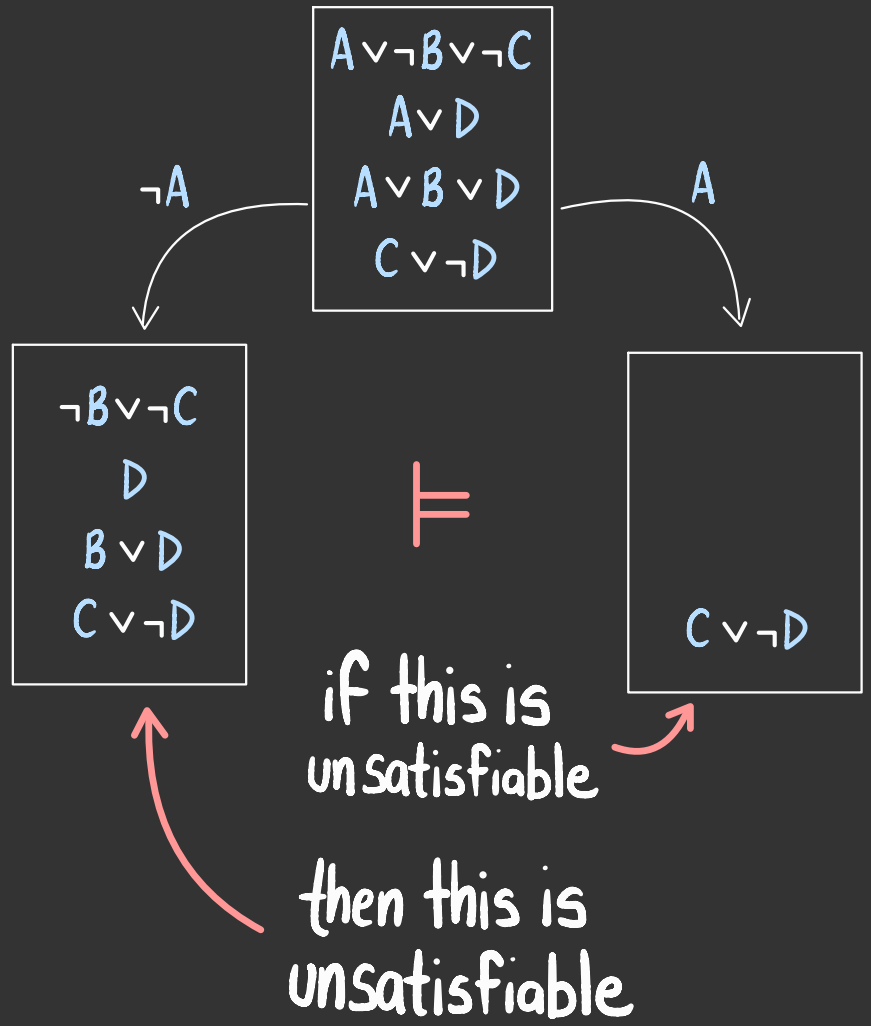
So which subtree do we explore?



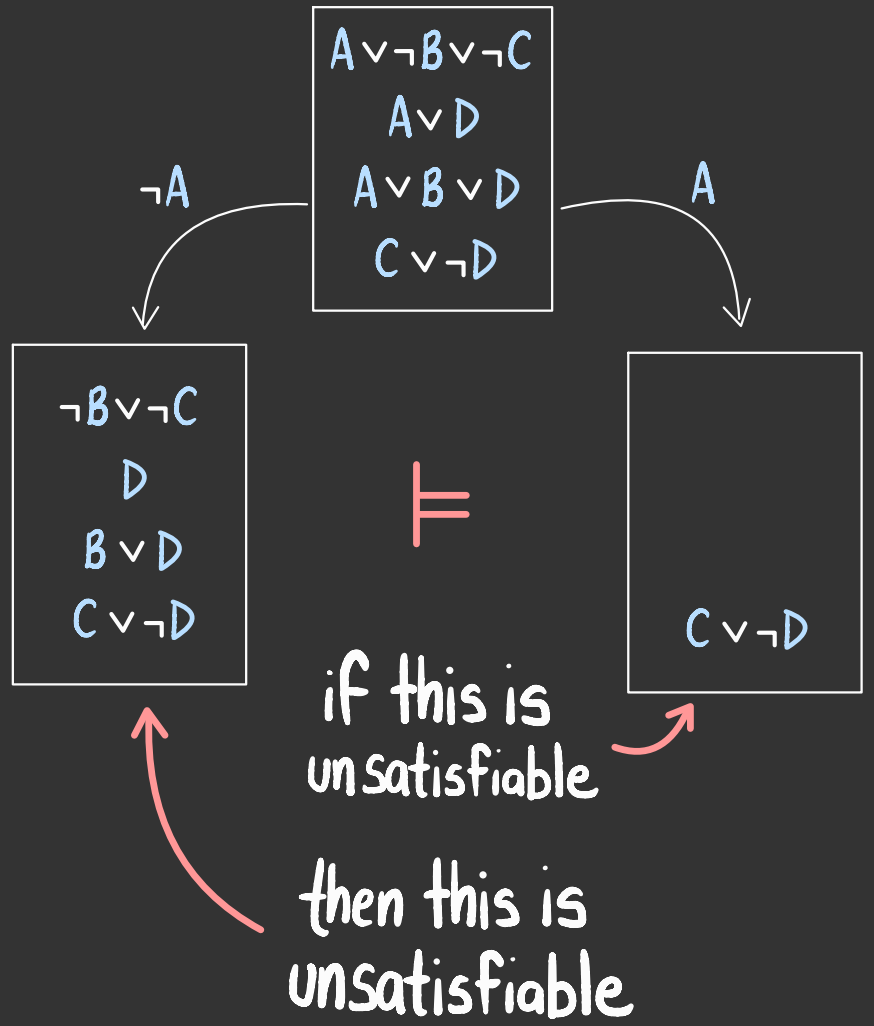
So which subtree do we explore?



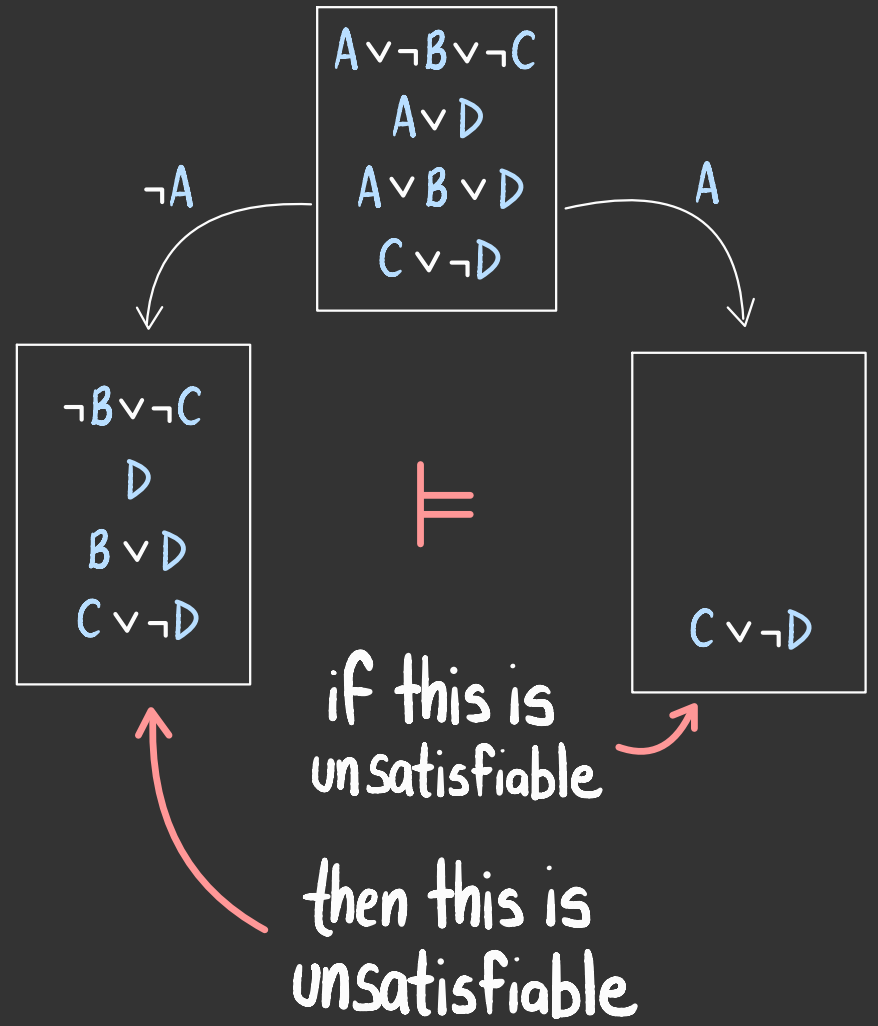
if we explore either subtree and find a satisfying model, then we can conclude the sentence is satisfiable



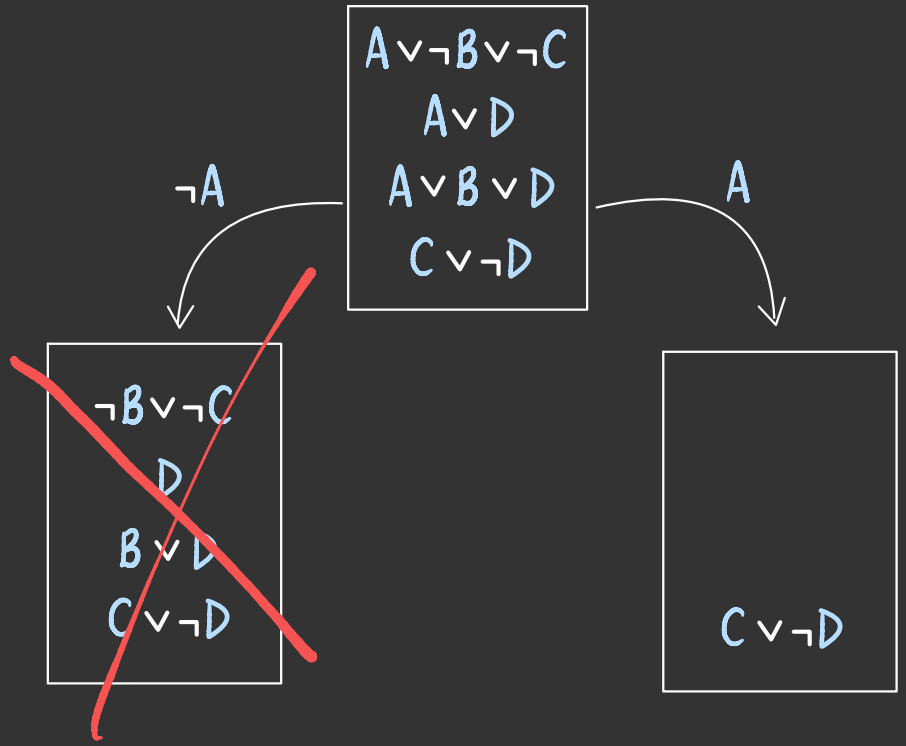
if we explore the left subtree and find no satisfying models, then the right subtree might still be satisfiable



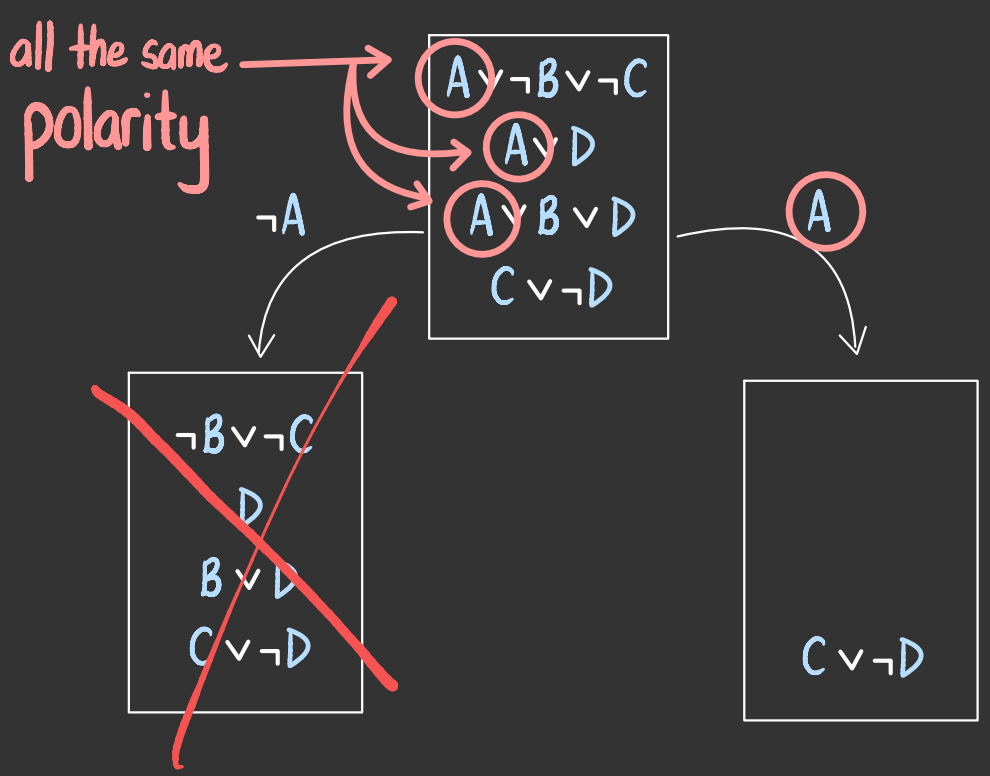
but if we explore the right subtree and find no satisfying models, then the left subtree is also unsatisfiable



so it is sufficient
to just explore
the right subtree

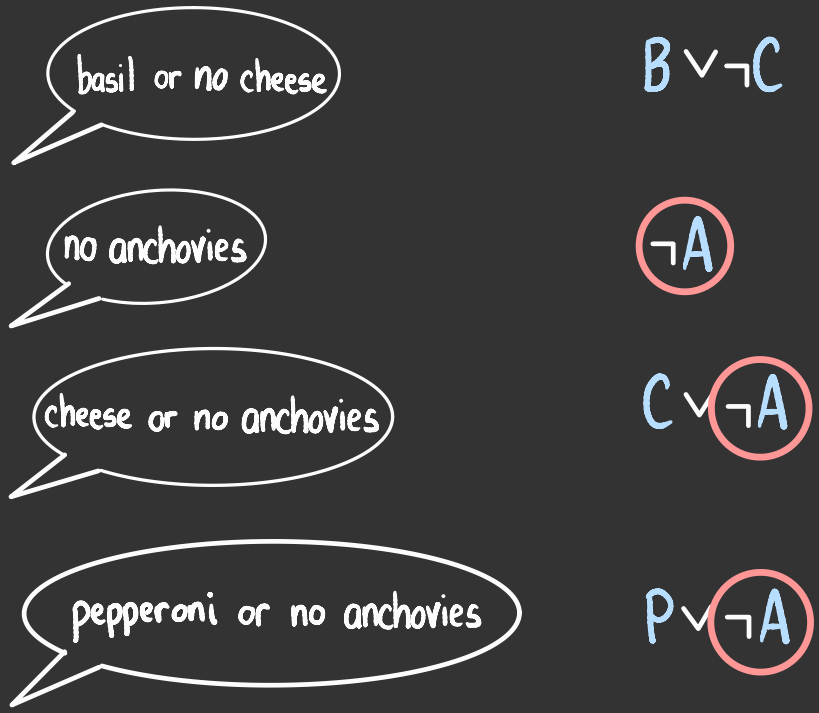


this is an example of the pure symbol heuristic



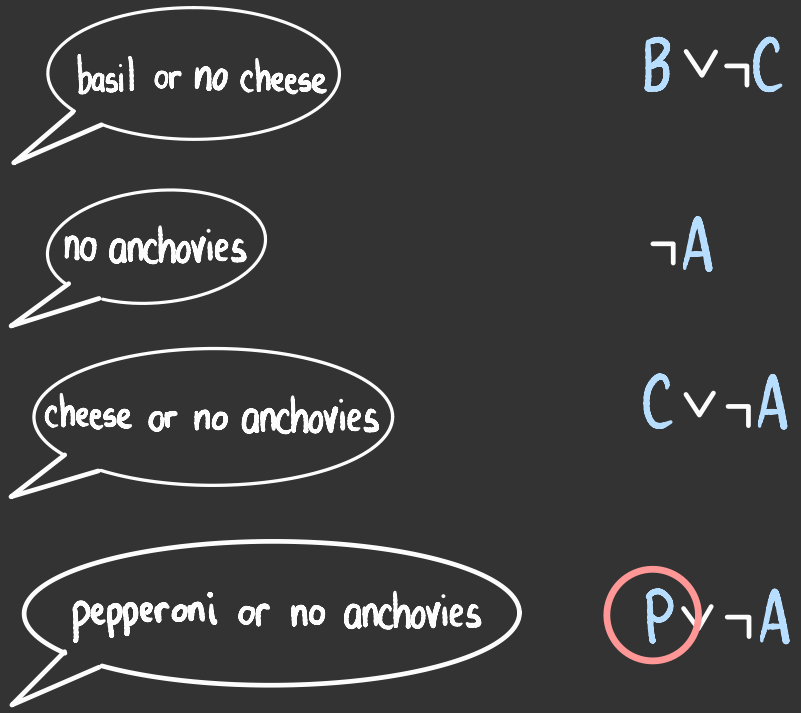
if a literal appears with only one "polarity", then we don't need to consider the opposite polarity

the pure symbol heuristic



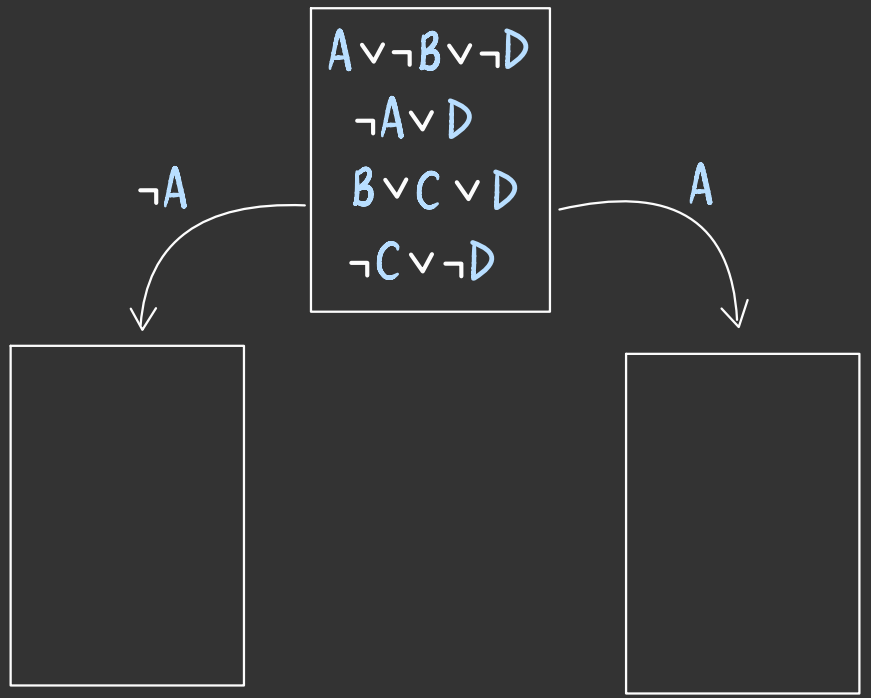
if no one wants anchovies, then don't put anchovies on the pizza

the pure symbol heuristic

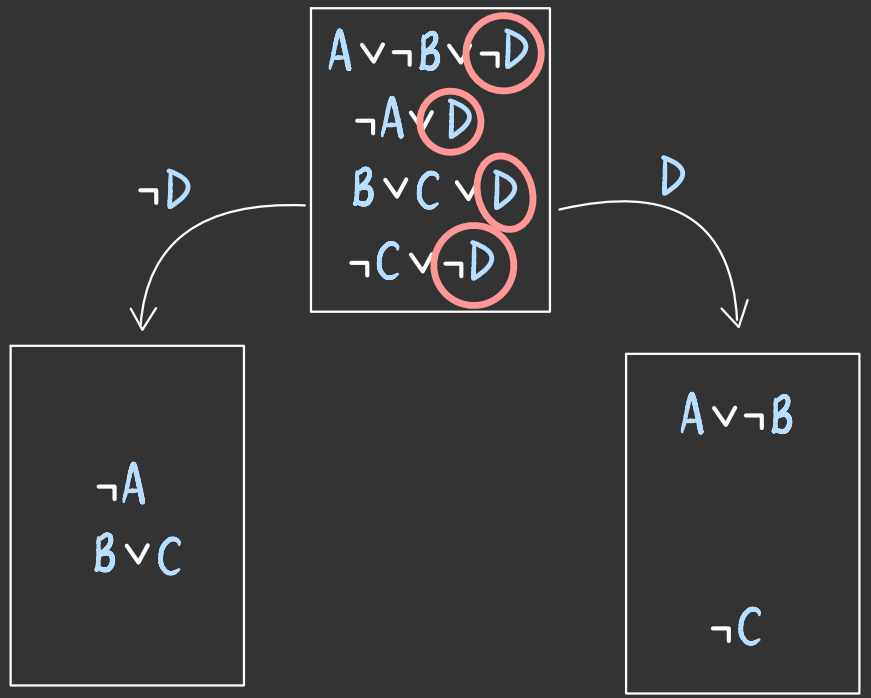


if no one objects to pepperoni, then just put pepperoni on the pizza

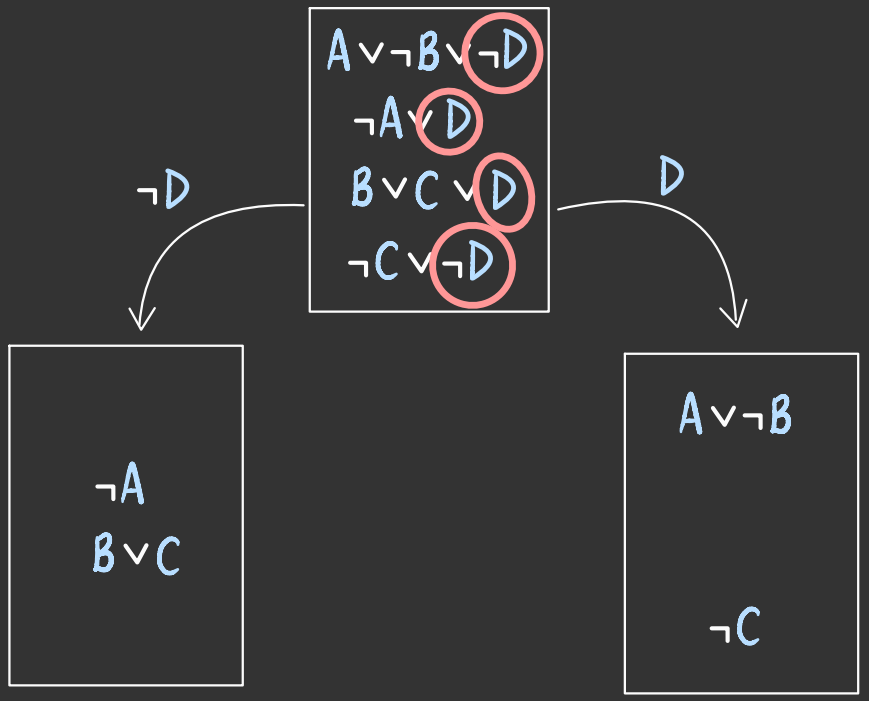
is A the best first variable to assign?



D is involved in more clauses, and so will more likely simplify our clauses



this is an example of the **degree heuristic**



assign the symbol that appears in the most clauses

but!

no matter what we do,
satisfiability is NP-complete,
so dpll will always* take
(in the worst-case)
exponential time
in the number of clauses

*unless P=NP

however, can we characterize these worst-case scenarios?

no matter what we do, satisfiability is NP-complete, so dpll will always* take (in the worst-case) exponential time in the number of clauses

*unless P=NP

suppose we create a cnf sentence by randomly creating m three-literal clauses from a signature with n symbols

$$\Sigma = \{A_1, \ldots, A_n\}$$

$m = 2$

$$A_3 \vee A_7 \vee \neg A_8$$
$$\neg A_3 \vee \neg A_{10} \vee A_{15}$$

$m = 4$

$$\neg A_1 \vee A_3 \vee \neg A_7$$
$$A_6 \vee \neg A_7 \vee A_{11}$$
$$A_3 \vee A_5 \vee \neg A_{15}$$
$$\neg A_7 \vee \neg A_{11} \vee A_{14}$$

suppose we create a cnf sentence by randomly creating m three-literal clauses from a signature with n symbols

$$\Sigma = \{A_1, \ldots, A_n\}$$

$m = 2$ ← which is more likely to be unsatisfiable? → $m = 4$

$$A_3 \vee A_7 \vee \neg A_8$$
$$\neg A_3 \vee \neg A_{10} \vee A_{15}$$

$$\neg A_1 \vee A_3 \vee \neg A_7$$
$$A_6 \vee \neg A_7 \vee A_{11}$$
$$A_3 \vee A_5 \vee \neg A_{15}$$
$$\neg A_7 \vee \neg A_{11} \vee A_{14}$$

suppose we create a cnf sentence by randomly creating m three-literal clauses from a signature with n symbols

$$\Sigma = \{A_1, \ldots, A_n\}$$

as m grows, the sentence is more likely to be unsatisfiable

$m = 4$

$$\neg A_1 \vee A_3 \vee \neg A_7$$
$$A_6 \vee \neg A_7 \vee A_{11}$$
$$A_3 \vee A_5 \vee \neg A_{15}$$
$$\neg A_7 \vee \neg A_{11} \vee A_{14}$$

suppose we create a cnf sentence by randomly creating m three-literal clauses from a signature with n symbols

$$\Sigma = \{A_1, \ldots, A_n\}$$

$m = 4, n = 5$ ← which is more → $m = 4, n = 15$

which is more likely to be unsatisfiable?

$$\neg A_1 \vee A_3 \vee \neg A_4$$
$$A_1 \vee \neg A_2 \vee A_3$$
$$A_2 \vee A_4 \vee \neg A_5$$
$$\neg A_1 \vee \neg A_2 \vee A_5$$

$$\neg A_1 \vee A_3 \vee \neg A_7$$
$$A_6 \vee \neg A_7 \vee A_{11}$$
$$A_3 \vee A_5 \vee \neg A_{15}$$
$$\neg A_7 \vee \neg A_{11} \vee A_{14}$$

suppose we create a cnf sentence by randomly creating m three-literal clauses from a signature with n symbols

$$\Sigma = \{A_1, \ldots, A_n\}$$

$m = 4$, $n = 5$

$$\neg A_1 \vee A_3 \vee \neg A_4$$
$$A_1 \vee \neg A_2 \vee A_3$$
$$A_2 \vee A_4 \vee \neg A_5$$
$$\neg A_1 \vee \neg A_2 \vee A_5$$

as n shrinks, the sentence is more likely to be unsatisfiable

to the laptop!