Resolution and DPLL

CIS700 — Fall 2023

Kris Micinski

Motivation: the resolution rule

Resolution is a simple principle that says:

$$(A \vee p) \wedge (B \vee \neg p) \wedge P \Rightarrow (A \vee B) \wedge P$$

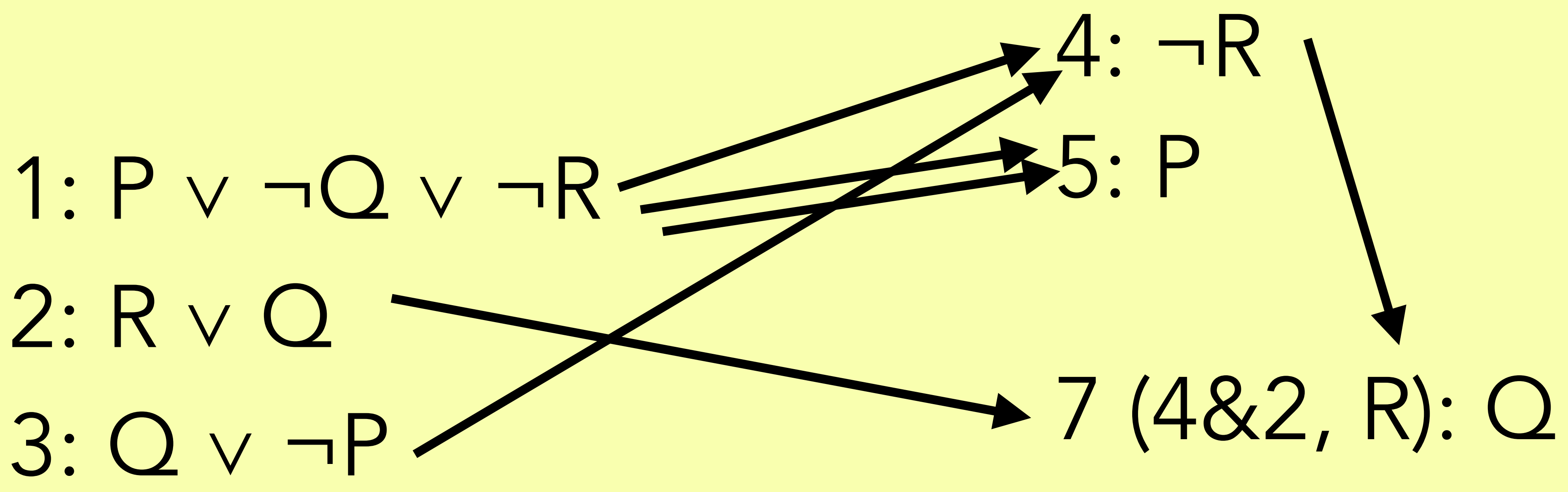
Given the following three clauses, which resolutions can you derive?

1: $P \vee \neg Q \vee \neg R$
2: $R \vee Q$
3: $Q \vee \neg P$

4 (1&3, P): $Q \vee \neg Q \vee \neg R == \neg R$
5 (1&2, R): $P \vee \neg Q \vee Q == P$
6 (1&2, Q): $P \vee \neg R \vee R == P$
7 (4&2, R): Q
8 (6&3, P): Q

Resolution is a sound reasoning principle, but not an algorithm: it doesn’t tell us SAT/UNSAT, but it tells us new information we can add

We can represent resolution as a graph with formulas as nodes and edges to indicate the resolutions—here we deduplicate



1: P ∨ ¬Q ∨ ¬R

2: R ∨ Q

3: Q ∨ ¬P

4: ¬R

5: P

7 (4&2, R): Q

Notice that the graph tracks provenance of how the decision was made

Given a large set of clauses, we could imagine iteratively applying resolution until we either (a) cannot find any more possible instances of resolution (we return SAT) or (b) produce a refutation (return UNSAT). This is the “saturation-based” approach

Given clauses with N variables, repeated application of resolution will produce at most 2^n possible clauses

Theorem: Resolution is Sound

Given a set of clauses ϕ, if P is a valid refutation then ϕ is UNSAT

Proof: induction on the structure of refutations

Theorem: Resolution is *Refutationally*-Complete

If ϕ is UNSAT, then the saturation-based method above will eventually find a refutation

Proof: induction on the number of variables k

Refutational Completeness is not *total* completeness!

I misunderstood this basic fact for a long time: resolution will only prove a formula is UNSAT if it is indeed UNSAT. This is an important difference. One important motivation for DPLL is that it is totally complete.

This motivates the discussion on the next slide…

Why don’t we use the saturation-based resolution for deciding SAT?

Unfortunately, while resolution **will infer $\emptyset$ when the formula is UNSAT** (given sufficient time/space), in practice a resolution-only approach is not scalable because of *memory blowup* (i.e., materialization overhead):

- Resolution will force enumeration of a **huge** set of derived facts
- Until a refutation is found, the resolution will keep going and going—producing combinatorial explosion
- Instead, SAT solving makes use of guided search to “guess” a model

We now discuss DPLL, one of the first search-based procedures for SAT solving. Next week, we’ll study a refinement on DPLL (CDCL) used by most modern solvers, which incorporate *aspects* of resolution (to “learn” clauses)

SAT Solving: the DPLL Algorithm

The DPLL algorithm formalizes the idea of SAT solving via *backtracking search*. The basic idea is to alternate between application of *saturation* (via unit propagation / pure variable elimination) and *guessing a variable assignment*

It is easiest to start with an example: the next few slides use an example from Wikipedia user Tamkin04iut (see “DPLL Algorithm”)

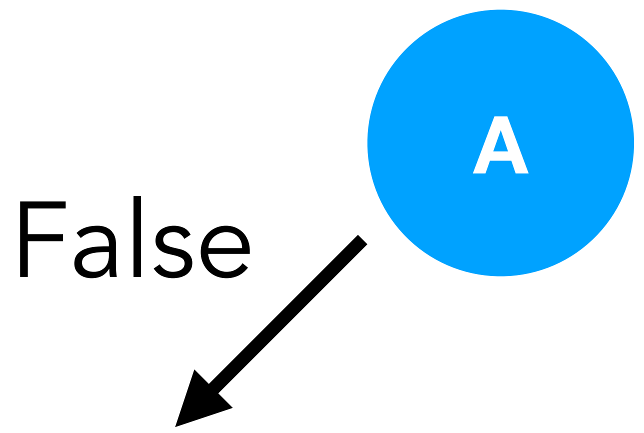
Worked Example: DPLL—Variable Choice

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

A = False
B = ???
C = ???
D = ???

First, we pick a variable: Let’s pick A
Variable choice is important, we will discuss heuristics later

Arbitrarily, let’s guess A is false



The underlined clauses are now **satisfied**

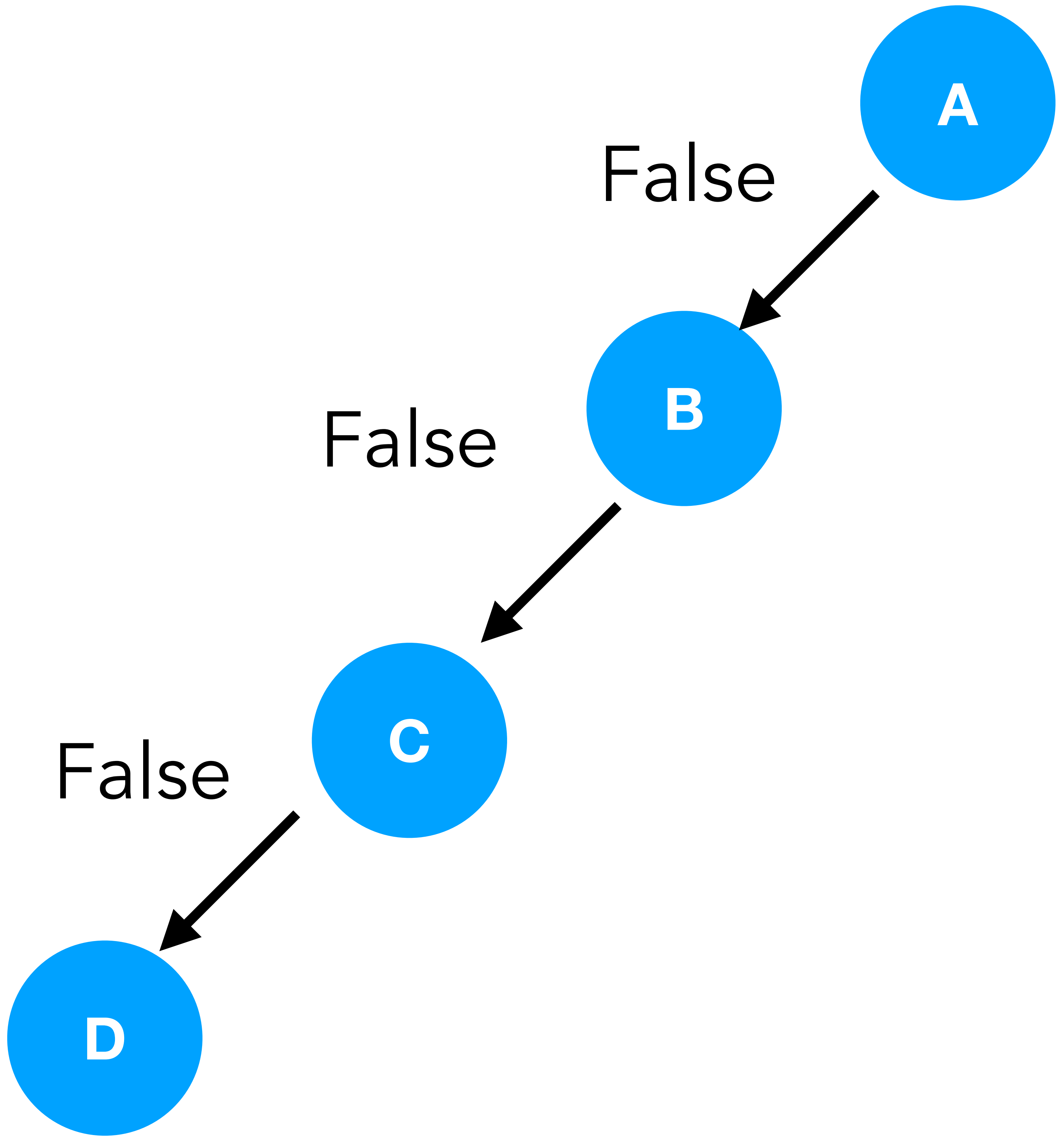
We’re not done yet—we still need to underline the rest!

Worked Example: DPLL—Variable Choice

Looking only at A is “fine” but still leaves things unfinished:
Let’s keep going—we guess that B and C are also False

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

A = False
B = False
C = False
D = ???



Worked Example: DPLL—Unit Propagation

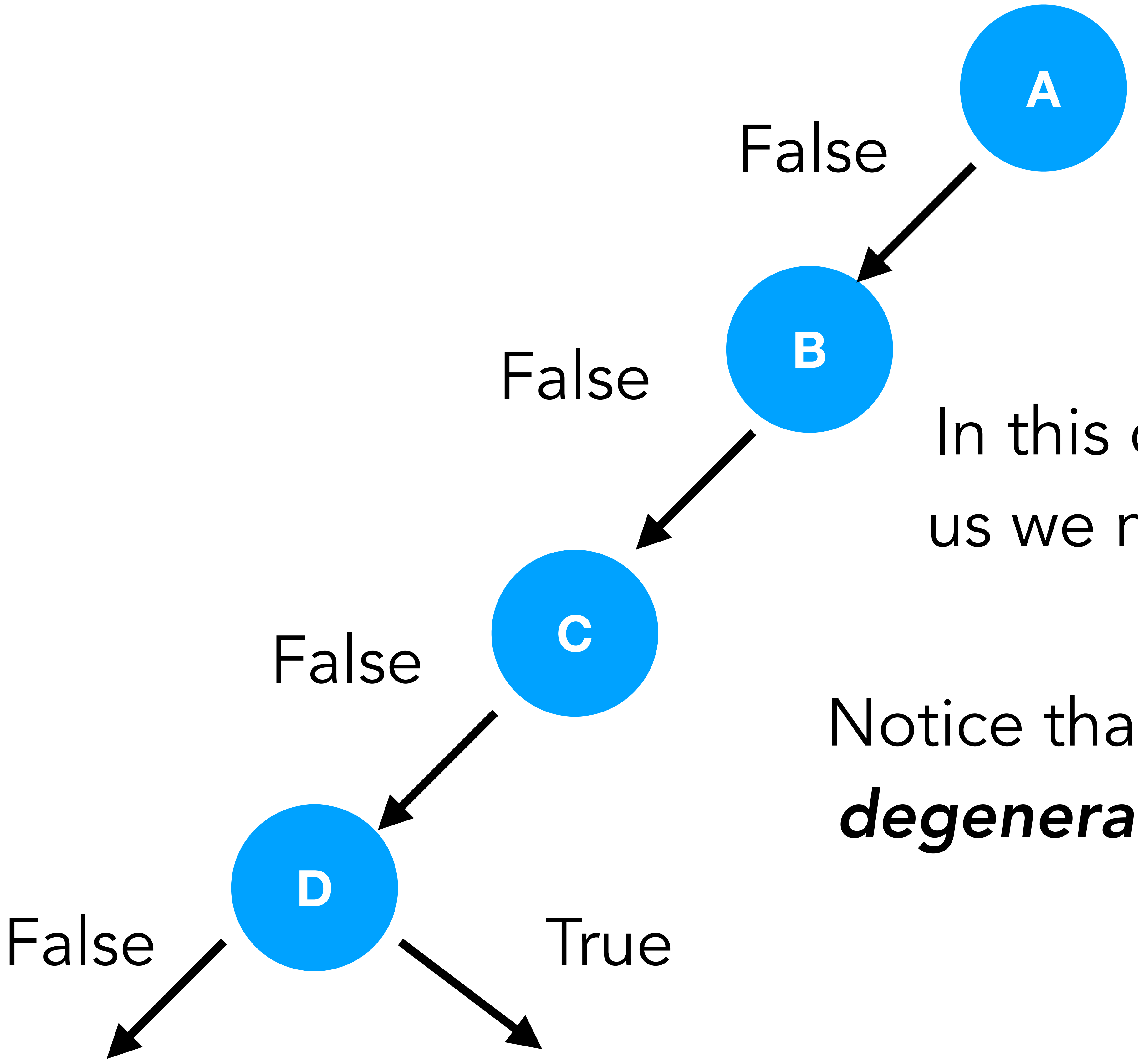
Any clause which has *a single unassigned literal* is called a **unit clause**

Intuitively, a unit clause is a clause which is **forcing the assignment of the single remaining literal**

In this case, *these* clauses are unit

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

A = False
B = False
C = False
D = ???



In this case, unit propagation tells us we must assign **both** D and ¬D

Notice that unit propagation is a ***degenerate form of resolution***

Worked Example: DPLL—Conflicts

In this case, *these* clauses are unit

¬A ∨ B ∨ C

A ∨ C ∨ D

A ∨ C ∨ ¬D

A ∨ ¬C ∨ D

A ∨ ¬C ∨ ¬D

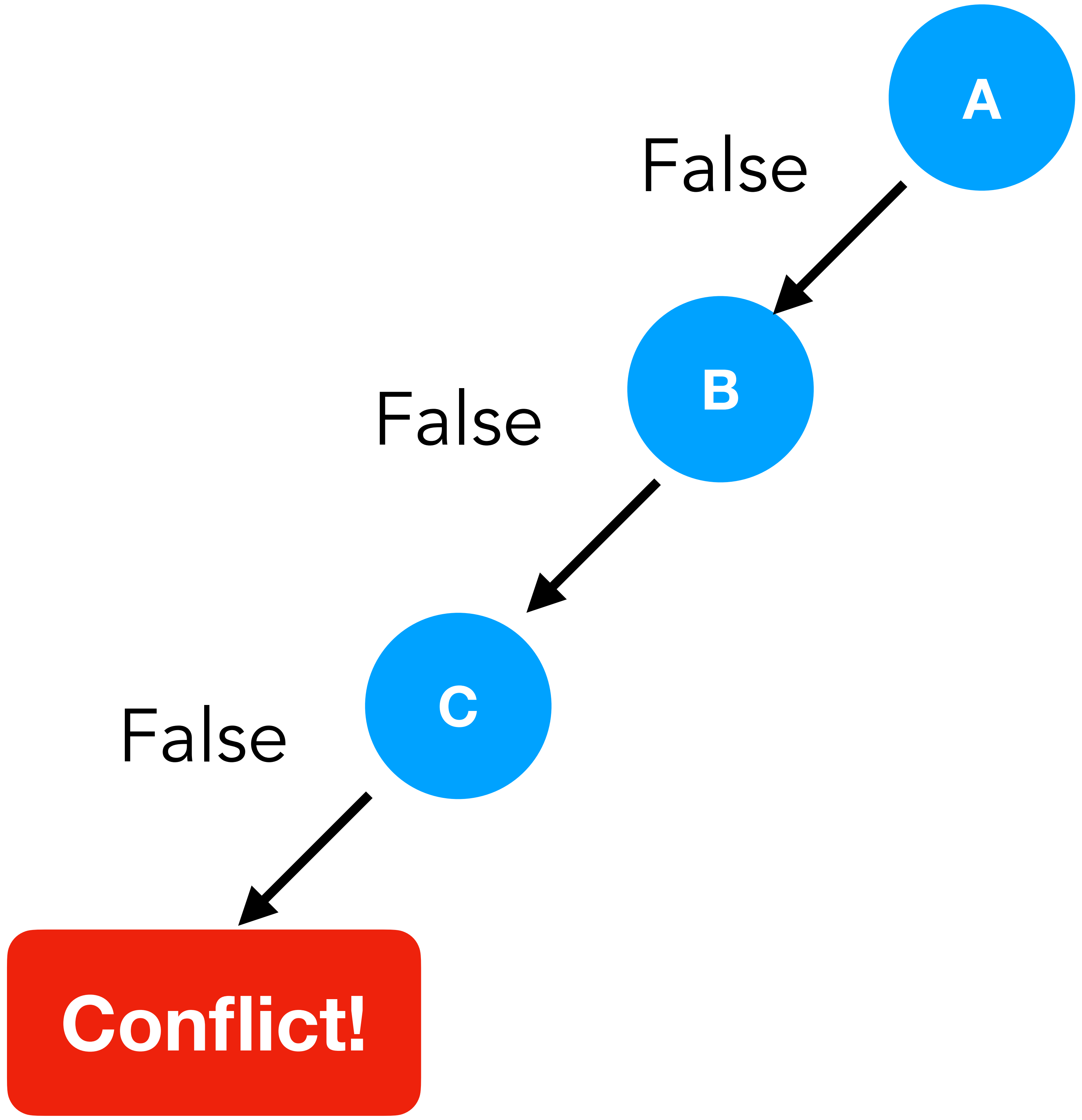
¬B ∨ ¬C ∨ D

¬A ∨ B ∨ ¬C

¬A ∨ ¬B ∨ C

A = False
B = False
C = False
D = ???

We can now observe a **conflict**: unit propagation reveals that both D and ¬D must hold along this branch



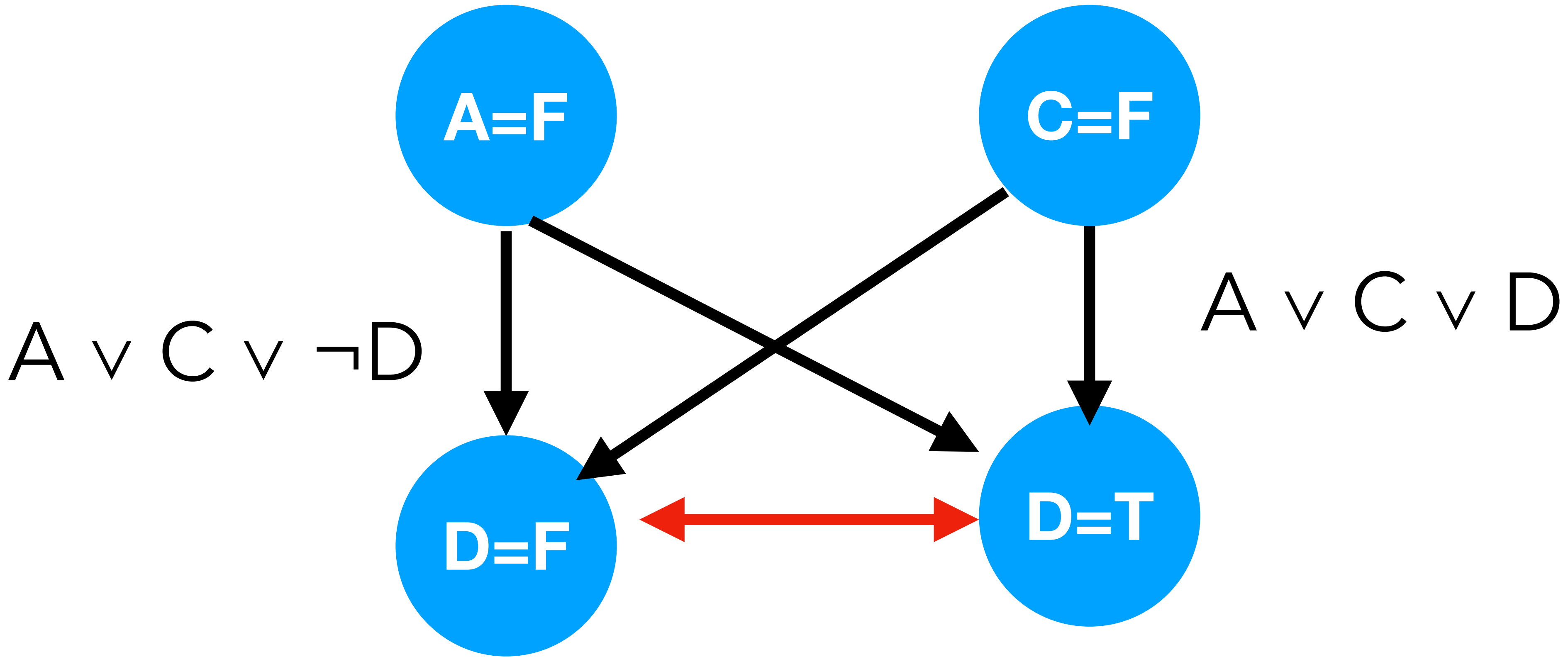
Worked Example: DPLL—Implication Graph

In this case, *these* clauses are unit

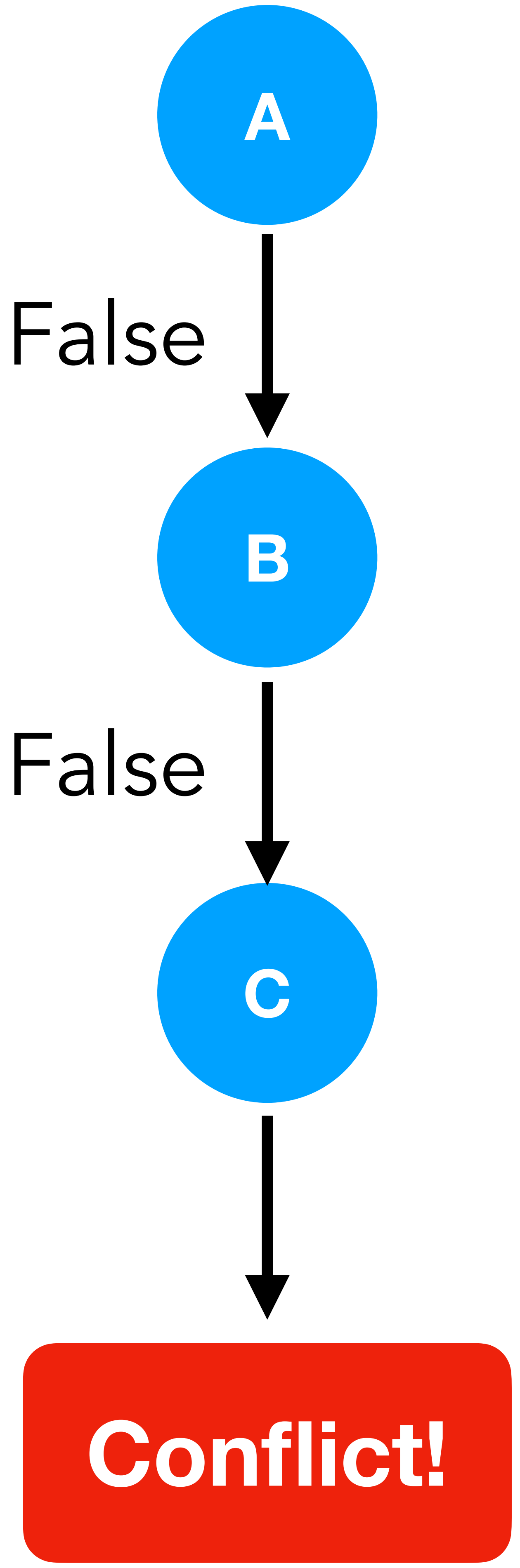
¬A ∨ B ∨ C
A ∨ C ∨ D
A ∨ C ∨ ¬D
A ∨ ¬C ∨ D
A ∨ ¬C ∨ ¬D
¬B ∨ ¬C ∨ D
¬A ∨ B ∨ ¬C
¬A ∨ ¬B ∨ C

A = False
B = False
C = False
D = ???

We can assemble a **(forced) implication graph**, which allows us to record which clauses forced unit propagation to build assignments:



Conflict!



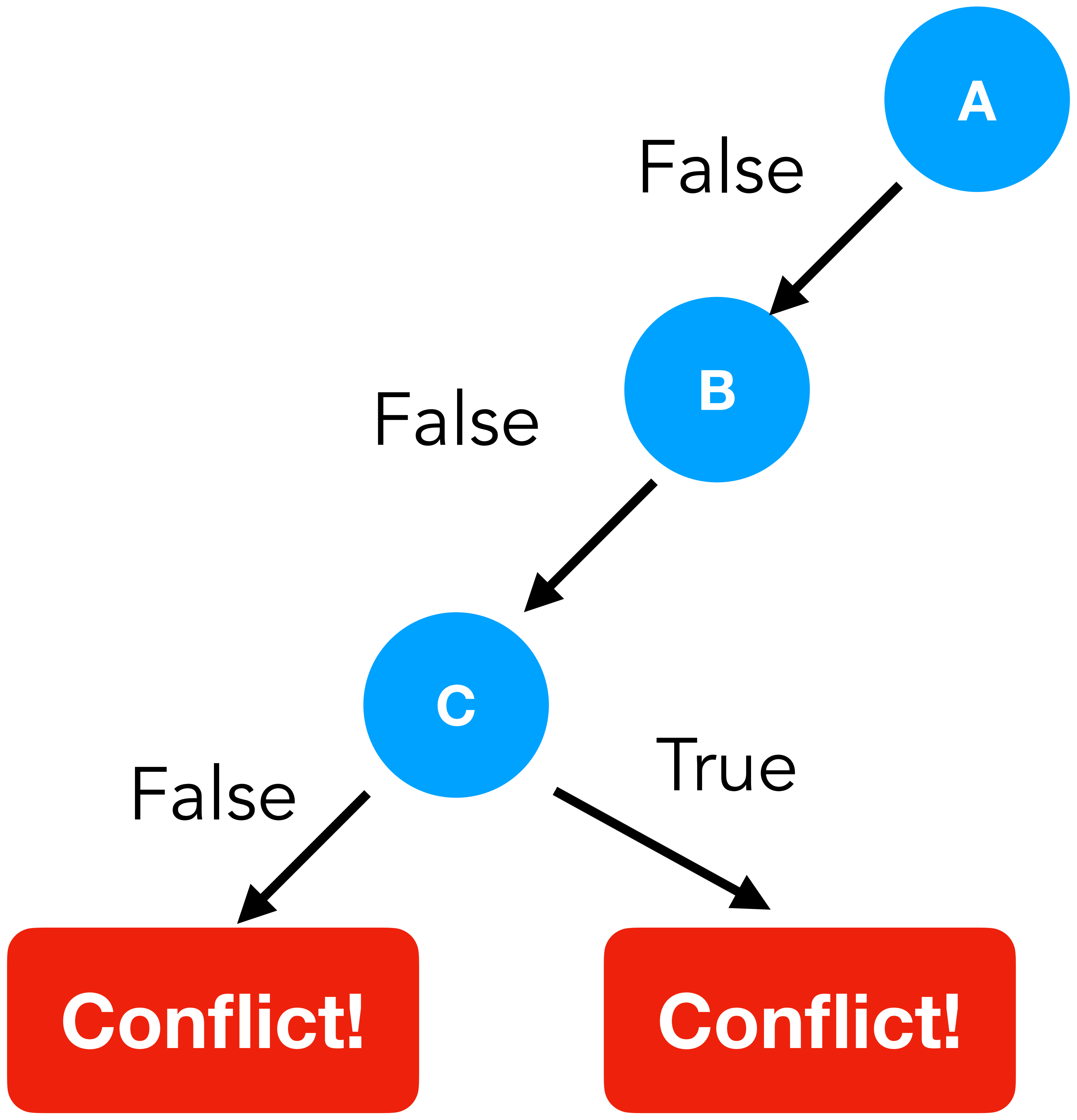
Once we get to a conflict, we backtrack. In this case we show chronological backtracking back to try C=True

Worked Example: DPLL—Backtracking

Now, *these* clauses are unit

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

A = False
B = False
C = True
D = ???



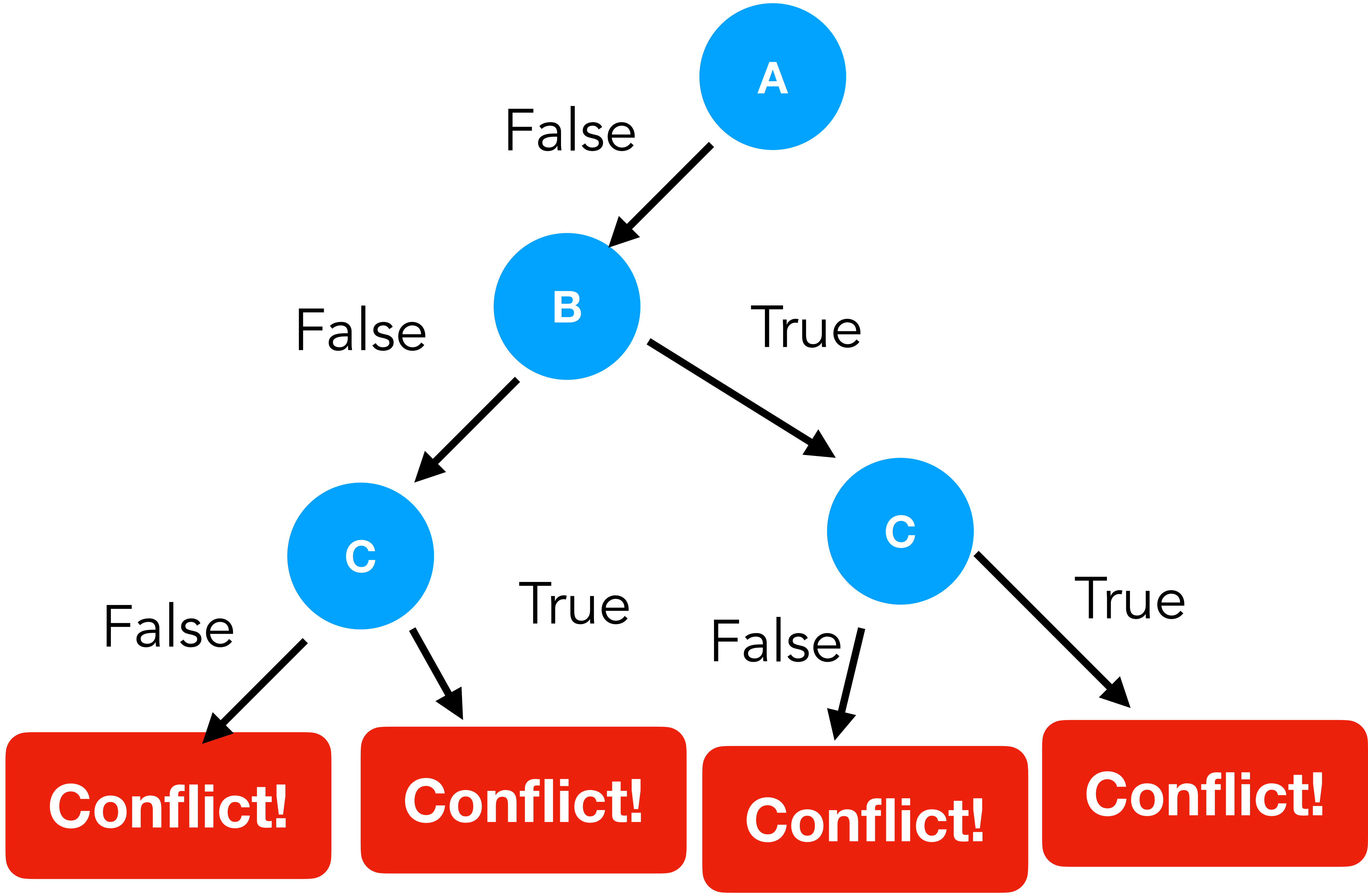
Before each decision, we must repeatedly apply unit propagation—now again, we get a conflict!

Worked Example: DPLL—Keep backtracking

Now, *these* clauses are unit

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

A = False
B = True
C = True
D = ???



So does guessing C = True

Yet again, we get a conflict!

Worked Example: DPLL—Even *more* backtracking…

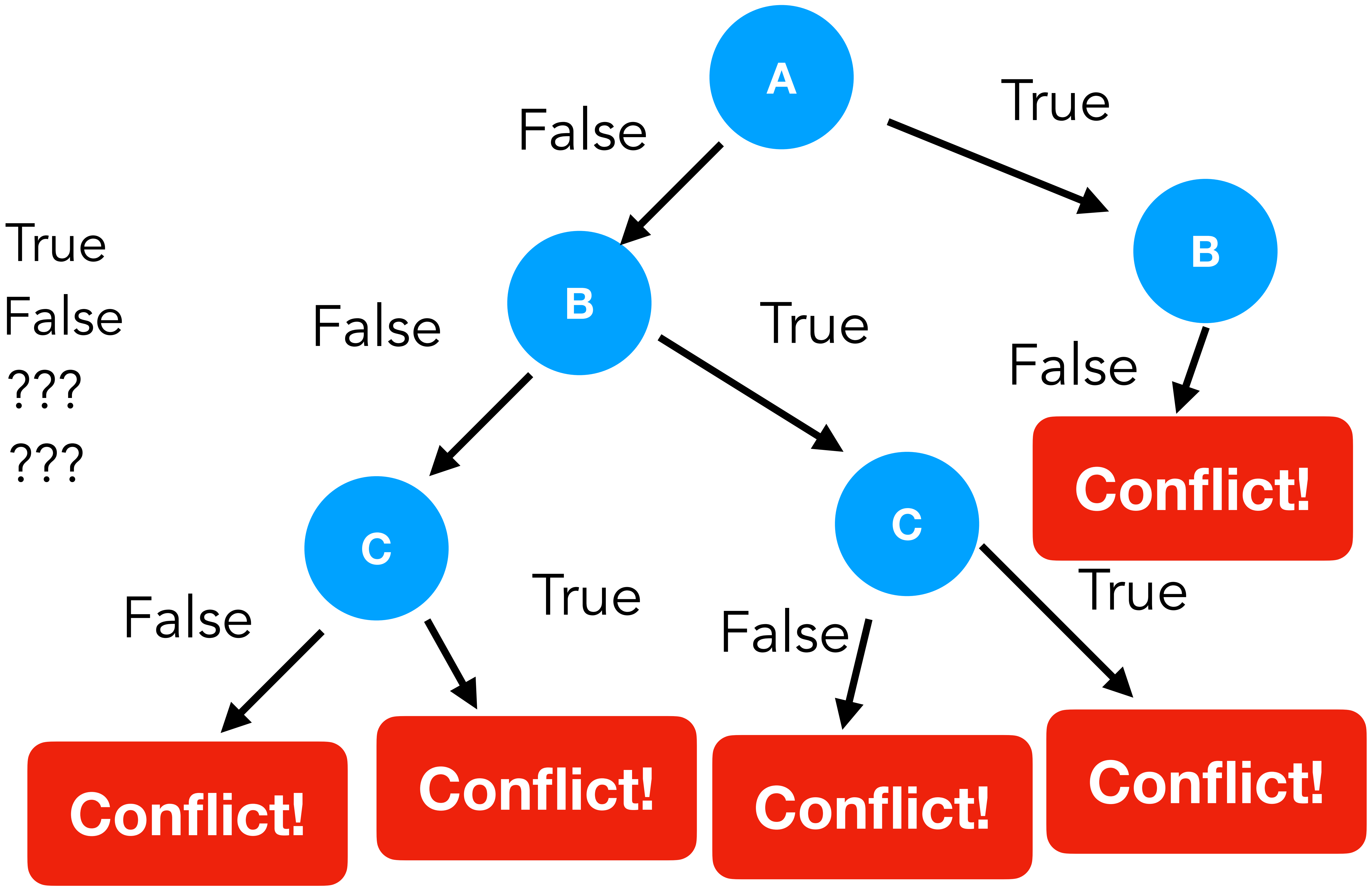
We backtrack all the way up to guessing A=True, and then guess B=False

These unit clauses conflict!

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

A = True
B = False
C = ???
D = ???

Still a conflict!



Worked Example: DPLL—Success!

The forced assignment of C=True via unit propagation leads to the discovery of a **new** unit clause

¬A ∨ B ∨ C

A ∨ C ∨ D

A ∨ C ∨ ¬D

A ∨ ¬C ∨ D

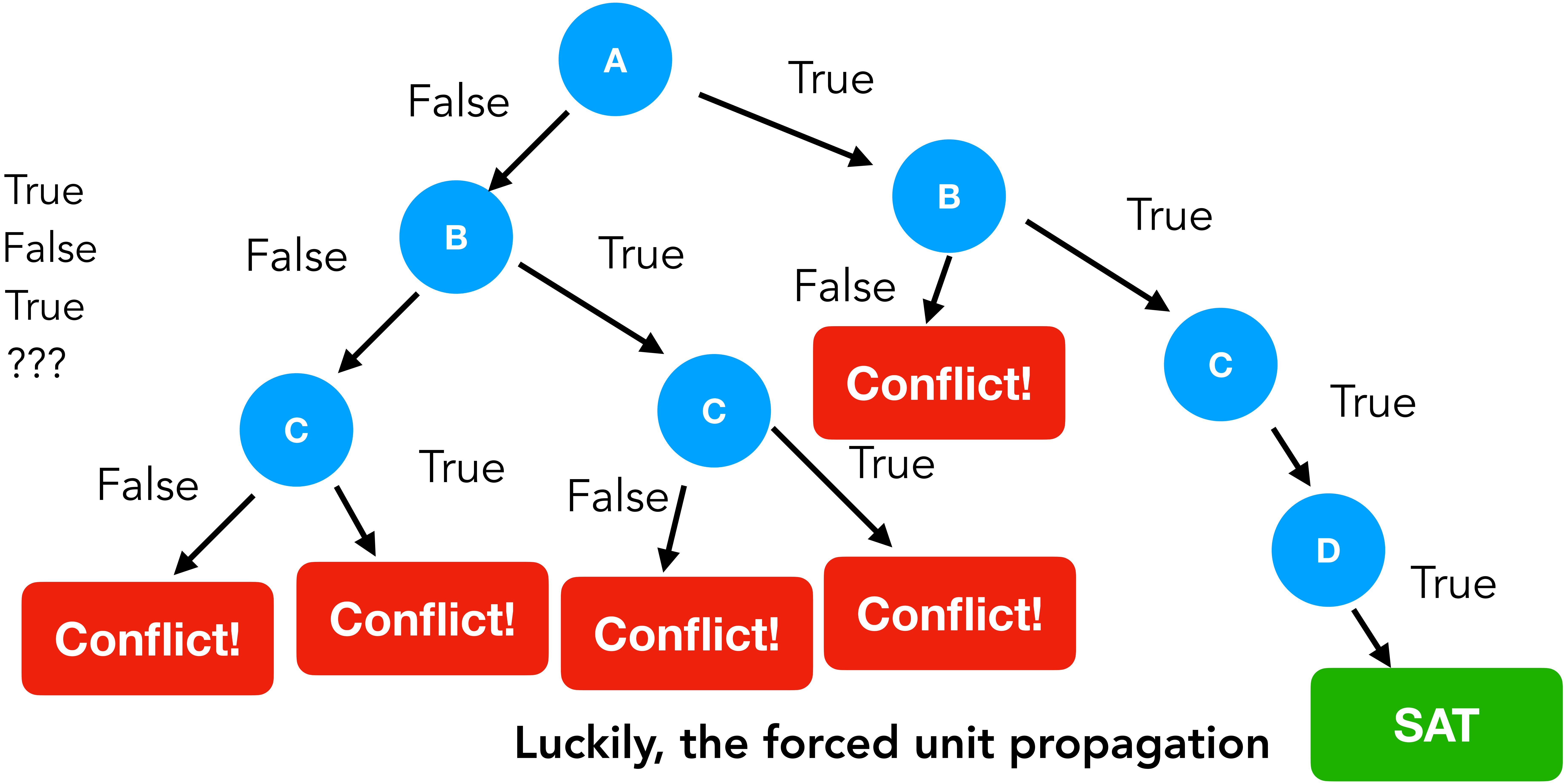
A ∨ ¬C ∨ ¬D

¬B ∨ ¬C ∨ D

¬A ∨ B ∨ ¬C

¬A ∨ ¬B ∨ C

A = True
B = False
C = True
D = ???



Luckily, the forced unit propagation D=True leads us to success!

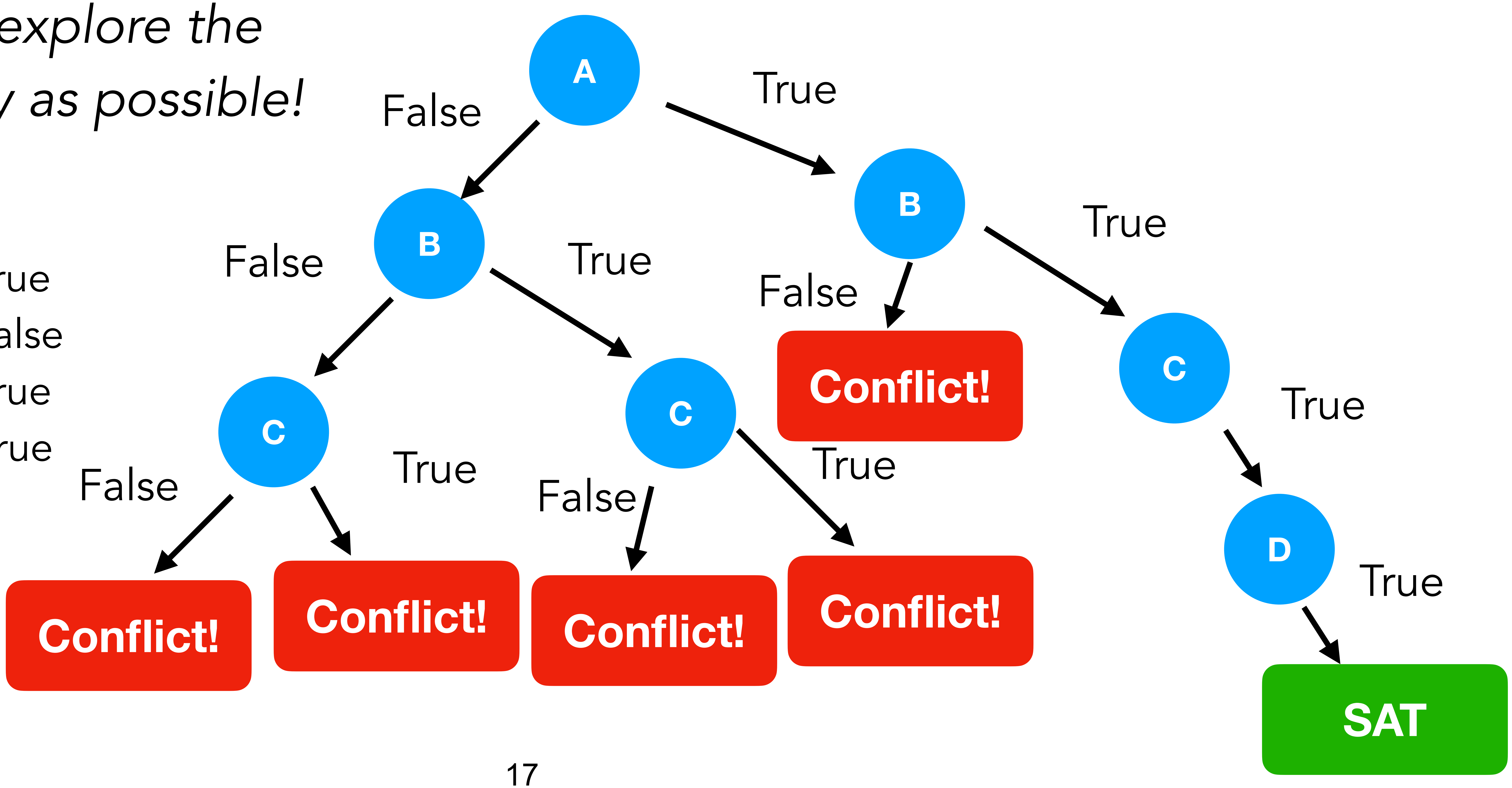
In sum, the DPLL algorithm says to iteratively:

- Apply unit propagation to assignments forced by unit clauses
- If unit propagation leads to conflict, backtrack chronologically (back to most recent guess)
- Decide a variable after all unit clauses taken care of—this is the search phase

The goal is to explore the tree as efficiently as possible!

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

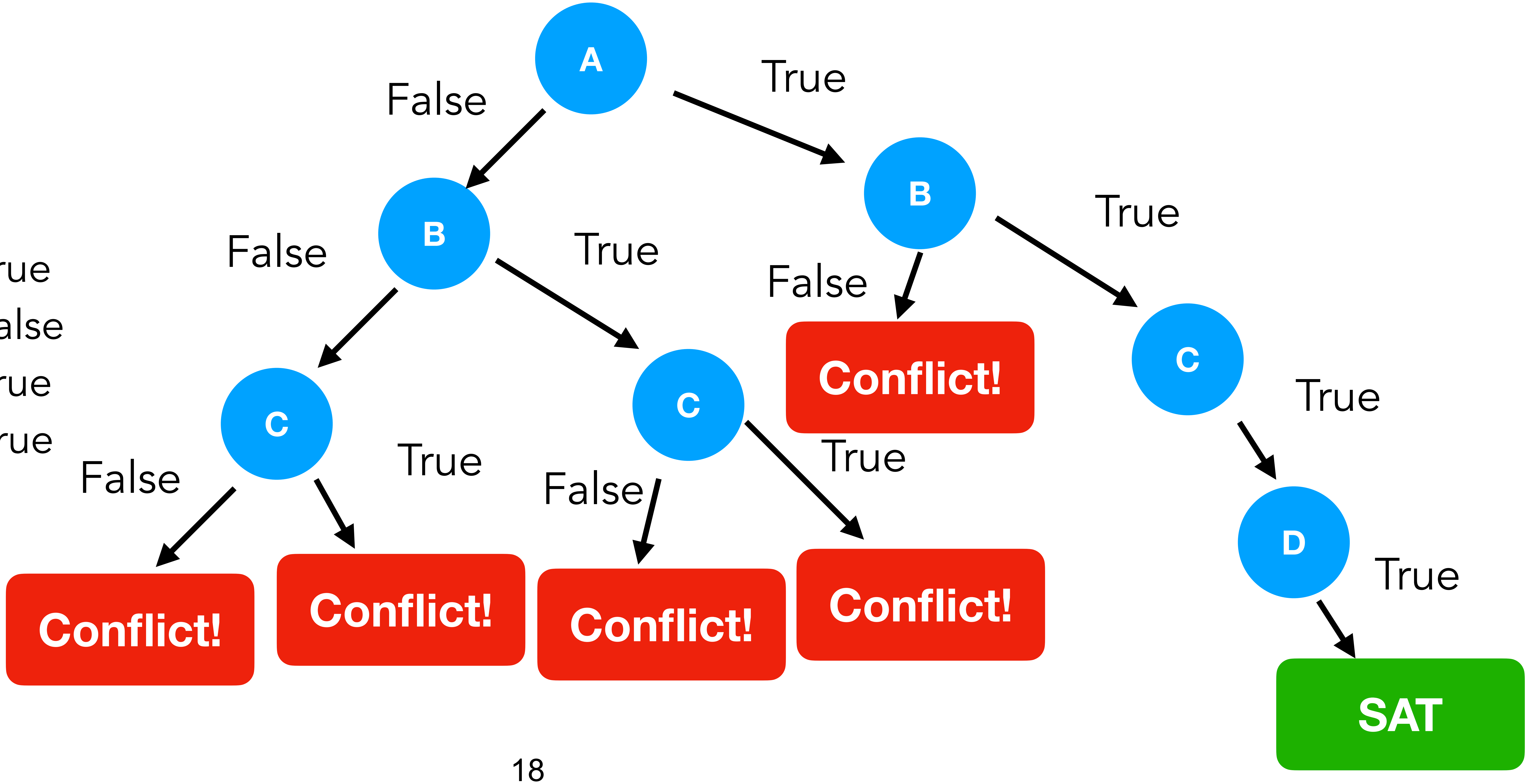
A = True
B = False
C = True
D = True



- Notice that the first choice led us to do a lot of redundant work
- Why didn't we just pick A = True?
- In general: picking variables optimally is tantamount to the halting problem—no general-purpose algorithm exists
- Next week, we'll discuss a **better algorithm** (CDCL) which analyzes conflicts to learn derived clauses that help cut off the search space based on clauses learned on-the-fly

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

A = True
B = False
C = True
D = True



DPLL Algorithm (Wikipedia’s definition)

```
function DPLL(Φ)
    // unit propagation:
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    // stopping conditions:
    if Φ is empty then
        return true; // SAT
    if Φ contains an empty clause then
        return false; // UNSAT
    // DPLL procedure:
    l ← choose-literal(Φ);
    return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l});
```

Pure Literals

Literals are **pure** when they are both (a) unassigned at the current point in the search and (b) they only occur in a single polarity (i.e., only A or only ¬A) in the formula.

Pure variables may simply be discarded—assigning them as either True or False is fine, and so they do not force decisions.

DPLL *alternates* between **inferring immediate consequences** (i.e., “saturation”) and **guessing** (i.e., “decision”)

```
function DPLL(Φ)
    // unit propagation:
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    // stopping conditions:
    if Φ is empty then
        return true; // SAT
    if Φ contains an empty clause then
        return false; // UNSAT
    // DPLL procedure:
    l ← choose-literal(Φ);
    return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l});
```

Saturation Phase

Decision Phase

Aside: Horn clauses and Datalog

Horn clauses are clauses with **at most one** positive (i.e., non-negated) literal

P ← Q, R (Logic programming style)
or Q ∧ R → P (implication style)
or (definition of →, DeMorgan…) $\neg(Q \land R) \lor P \equiv \neg Q \lor \neg R \lor P$

Datalog also allows facts: atomically known propositions (which can be interpreted as → P, i.e., nothing needed to infer P)

P ← Q, R
Q ← K
K
G
R ← J
J ←G

Query: P?

Datalog is **easier** to decide than SAT—the degenerate nature of Horn clauses means that we never have to *guess*. Datalog can be solved via saturation, **without** the need for guessing or backtracking. It’s complexity lies in PSPACE (<< than k-SAT!)

DPLL Algorithm

```
Input — set of clauses $\phi$ in CNF
Output — True (SAT) or False (UNSAT)

DPLL($\phi$):
  Forever:
    While there exist any unit clauses $\{l\} \in \phi$:
      $\phi$ := unit_propagate($\phi$,l)
    If $\phi$ contains no more clauses: return True
    Elif $\phi$ contains any empty clauses: return False
    Else:
      Choose a literal l which is unassigned in $\phi$
      Return DPLL($\Phi \wedge \{l\}$) $\vee$ DPLL($\Phi \wedge \{\neg l\}$)
```

A few remarks:

```
DPLL(ϕ):
  Forever:
    // Want to avoid scanning over all of ϕ
    While there exist any unit clauses {l} ∈ ϕ:
      // Unit Propagation needs to be fast
      ϕ := unit_propagate(ϕ,l)
    If ϕ contains no more clauses: return True
    Elif ϕ contains any empty clauses: return False
    Else:
      // How do we pick variables?
      Choose a literal l which is unassigned in ϕ
      Return DPLL(Φ ∧ {l}) ∨ DPLL(Φ ∧ {¬l})
```

Grading / Rubric

Please email me submissions, kkmicins@syr.edu, CCing all of your group mates.

I will test your submissions with a variety of DIMACS inputs, e.g., the ones from this page. https://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html

Please tell me how to invoke and run your program when you email me. Ensure that it can run on either a Mac or Linux machine (I have both of these)—give me sources and instructions to build your project.