Conflict-Directed Clause Learning

CIS700 (Fall '24)

Kristopher Micinski

Previous classes: the DPLL algorithm

Algorithm 2: The DPLL algorithm with the pure literal rule

DPLL($\mathcal{T}, I$):

Input : Theory $\mathcal{T}$, Interpretation I

Output: true if some interpretation that extends I satisfies $\mathcal{T}$; false otherwise

$I \leftarrow$ UNIT-PROP($\mathcal{T}, I$);

$I \leftarrow$ PURE-LITS($\mathcal{T}, I$);

if *some* $c \in \mathcal{T}$ *is conflicting in* I **then**

  return *false*;

end

if *all variables are assigned a value* **then**

  return *true*

end

$x \leftarrow$ SELECT-UNASSIGNED-VARIABLE(I);

return DPLL($\mathcal{T}, I \cup \{x\}$) **or** DPLL($\mathcal{T}, I \cup \{\overline{x}\}$)

Issues with DPLL

- DPLL incorporates *unit propagation* and the *pure literal rule*
- However, there is a better algorithm, which we'll explore today:
  - ***Conflict-Directed Clause Learning*** (CDCL)
- CDCL learns from its mistakes (learned clauses), and incorporates *non-chronological* backtracking.
  - Compared to DPLL, CDCL **analyzes** conflicts to help prune the search space, leading to significant benefits in practice
- Most modern SAT solvers based on CDCL
  - CDCL is ***so*** common that some books refer to CDCL as DPLL!

Implication Graph

A key conceptual data structure in CDCL is the **implication graph**
(Note: many solvers don't *explicitly* build the implication graph!)

The implication graph is a graph where:

- Vertices are of the form l@d where l is a literal, d a decision level
  - Examples include $\neg x_0$@5, x_7@3, and $\neg x_{15}$@0
  - These vertices represent the current partial assignment
- Labeled edges $v_0 \xrightarrow{c} v_1$ connect vertices when $\neg v_0 \in$ Antecedent(v_1)
  - Intuitively: when v_0 forces unit propagation of v_1 via the clause c
  - Notice that v_0 is not just a literal, it's a literal at a decision level

Assume that at decision level 3, the decision is $\neg x_6 @ 3$

<u>Clauses</u>

$c_1 = \neg x_1 \lor x_2$

$c_2 = \neg x_1 \lor x_3 \lor x_5$

$c_3 = \neg x_2 \lor x_4$

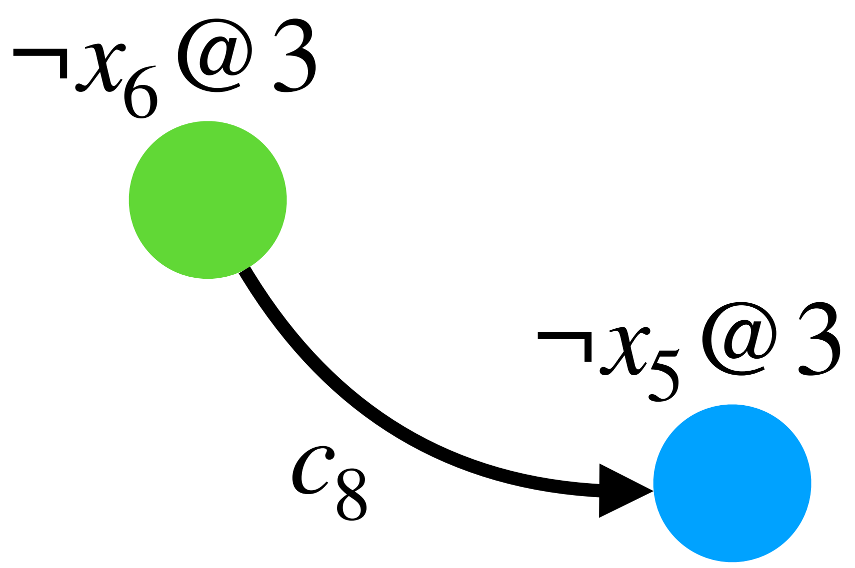
$c_4 = \neg x_3 \lor \neg x_4$

$c_5 = x_1 \lor x_5 \lor \neg x_2$

$c_6 = x_2 \lor x_3$

$c_7 = x_2 \lor \neg x_3$

$c_8 = x_6 \lor \neg x_5$



Edge labeled with the antecedent of $\neg x_5$

Due to clause c_8, unit propagation forces $\neg x_5 @ 3$
Notice: still at level 3, because unit propagation is a consequence of a decision, not a new decision level

Roots in the graph correspond to *decisions*
(Why? Because any incoming edge emanates from an antecedent. A decision *can't have* any antecedents!)

Example from the "Decision Procedures" book (Chap 2)

<u>Clauses</u>

$c_1 = \neg x_1 \vee x_2$

$c_2 = \neg x_1 \vee x_3 \vee x_5$

$c_3 = \neg x_2 \vee x_4$

$c_4 = \neg x_3 \vee \neg x_4$

$c_5 = x_1 \vee x_5 \vee \neg x_2$

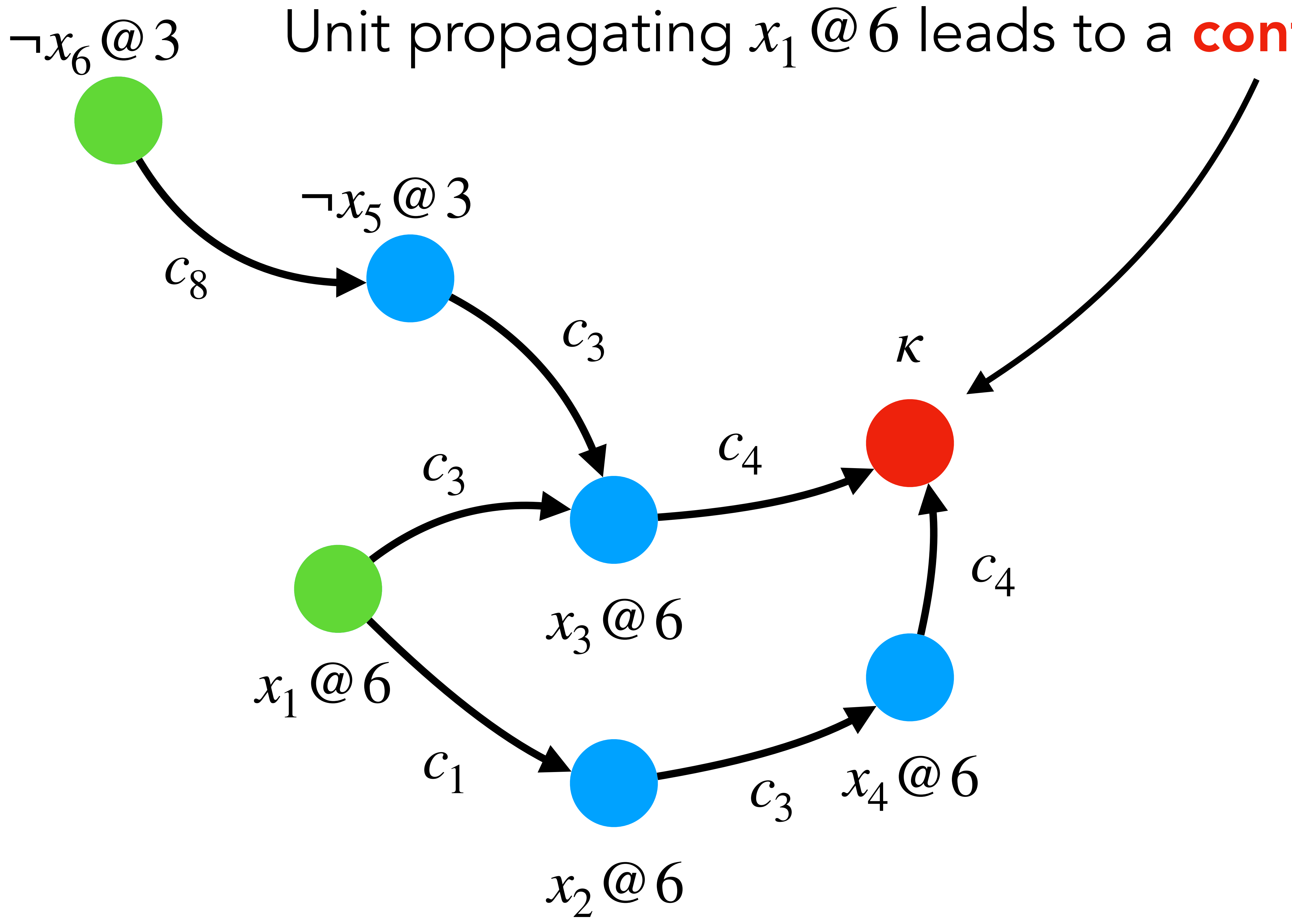
$c_6 = x_2 \vee x_3$

$c_7 = x_2 \vee \neg x_3$

$c_8 = x_6 \vee \neg x_5$

Now let's say we keep going, we made a few more decisions and the decision level is now 6
(Decisions 4/5 made choices that won't impact us here)

Unit propagating $x_1 @ 6$ leads to a **conflict**



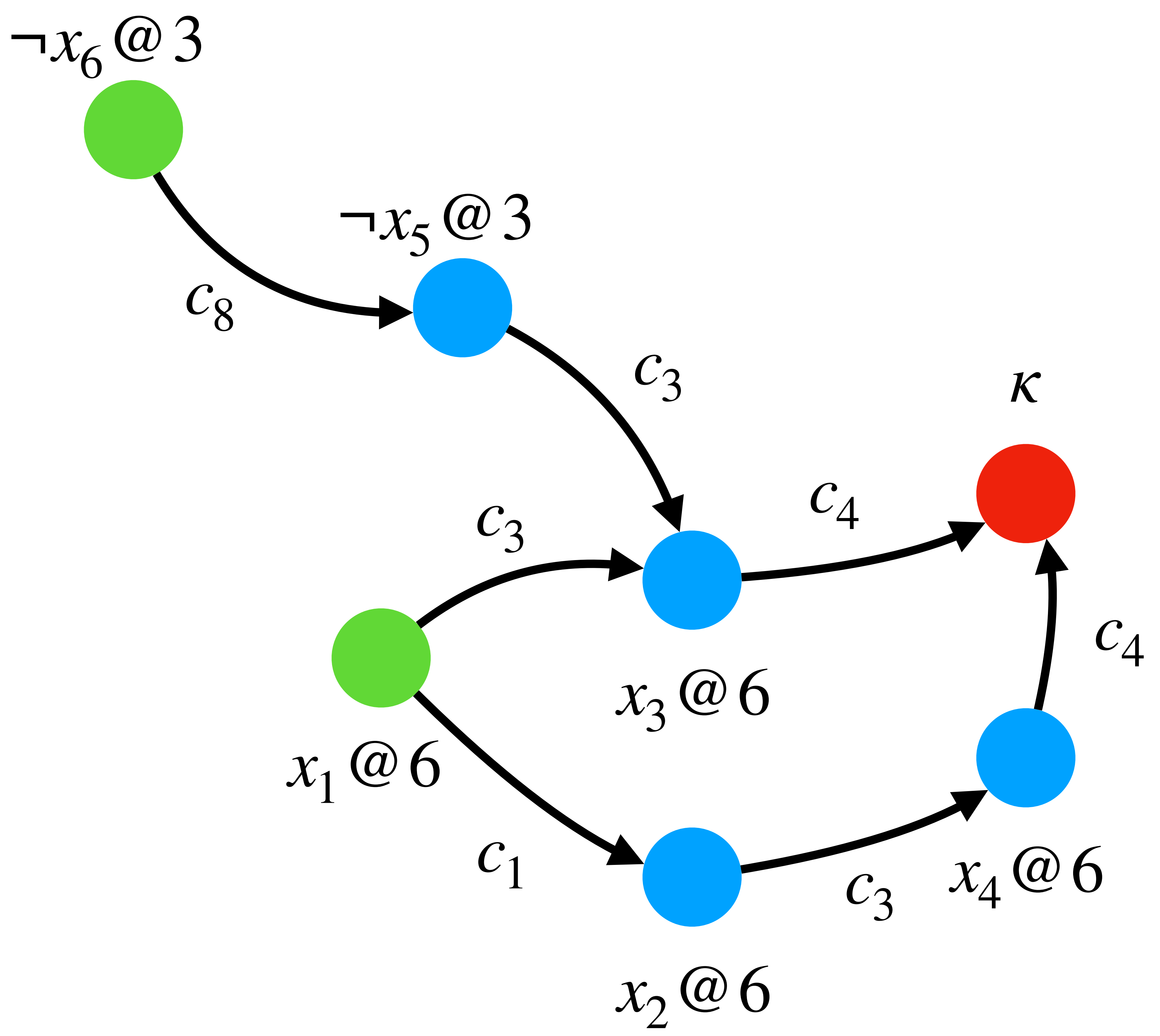
Example from the "Decision Procedures" book (Chap 2)

Now, some key observations:

- Backtracking to change x4, x2, or x3 *won't help*
- By picking $x_1 @ 6$, we are **destined** to reach a conflict!
- The **roots** of the graph represent a *sufficient condition* for the conflict
- Therefore, we can safely add a *learned* clause (such as $x_5 \vee \neg x_1$)

<u>Clauses</u>

$$c_1 = \neg x_1 \vee x_2$$
$$c_2 = \neg x_1 \vee x_3 \vee x_5$$
$$c_3 = \neg x_2 \vee x_4$$
$$c_4 = \neg x_3 \vee \neg x_4$$
$$c_5 = x_1 \vee x_5 \vee \neg x_2$$
$$c_6 = x_2 \vee x_3$$
$$c_7 = x_2 \vee \neg x_3$$
$$c_8 = x_6 \vee \neg x_5$$



Example from the "Decision Procedures" book (Chap 2)

Clauses

$c_1 = \neg x_1 \vee x_2$

$c_2 = \neg x_1 \vee x_3 \vee x_5$

$c_3 = \neg x_2 \vee x_4$

$c_4 = \neg x_3 \vee \neg x_4$

$c_5 = x_1 \vee x_5 \vee \neg x_2$

$c_6 = x_2 \vee x_3$

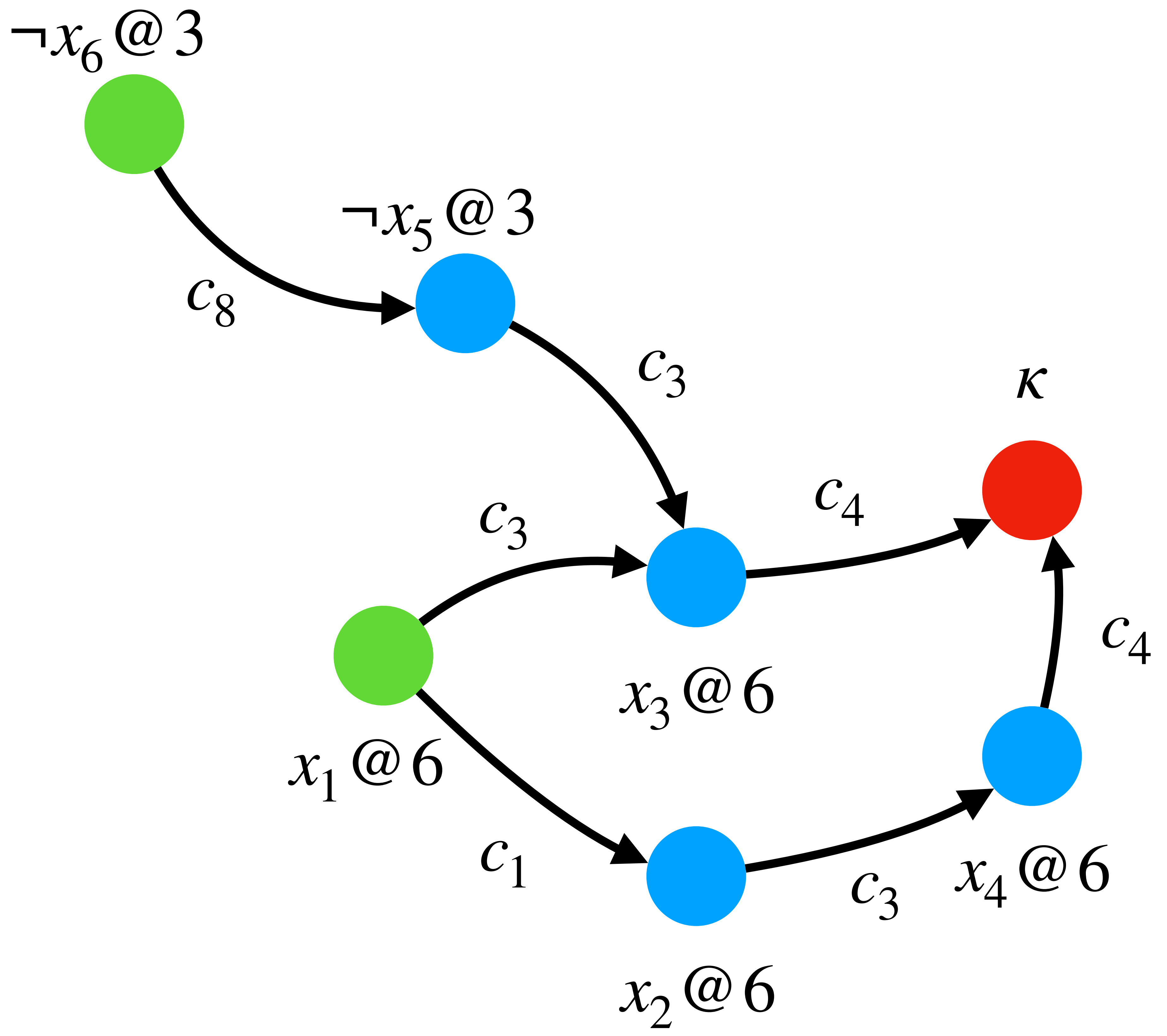
$c_7 = x_2 \vee \neg x_3$

$c_8 = x_6 \vee \neg x_5$

A learned clause is any clause implied by the original set of clauses (you can't change the meaning!)

As an example, we might negate the **entire current assignment**. The current assignment is a cube (conjunction of literals) and we know that the current assignment *must* result in failure! So we could add…

$$\neg(x_1 \wedge x_2 \wedge x_4 \wedge x_3 \wedge \neg x_5 \wedge \neg x_6) \equiv \neg x_1 \vee \neg x_2 \vee \neg x_4 \vee \neg x_3 \vee x_5 \vee x_6$$



Example from the "Decision Procedures" book (Chap 2)

In essence, this is *what DPLL already does!*

$\neg(x_1 \wedge x_2 \wedge x_4 \wedge x_3 \wedge \neg x_5 \wedge \neg x_6) \equiv \neg x_1 \vee \neg x_2 \vee \neg x_4 \vee \neg x_3 \vee x_5 \vee x_6$

However, this clause doesn't help prune the search space! We'll **never** be back in this state again (DPLL backtracks and tries another assignment), and thus *this* clause will <u>never actually be useful</u>!

<u>Clauses</u>

$c_1 = \neg x_1 \vee x_2$

$c_2 = \neg x_1 \vee x_3 \vee x_5$

$c_3 = \neg x_2 \vee x_4$

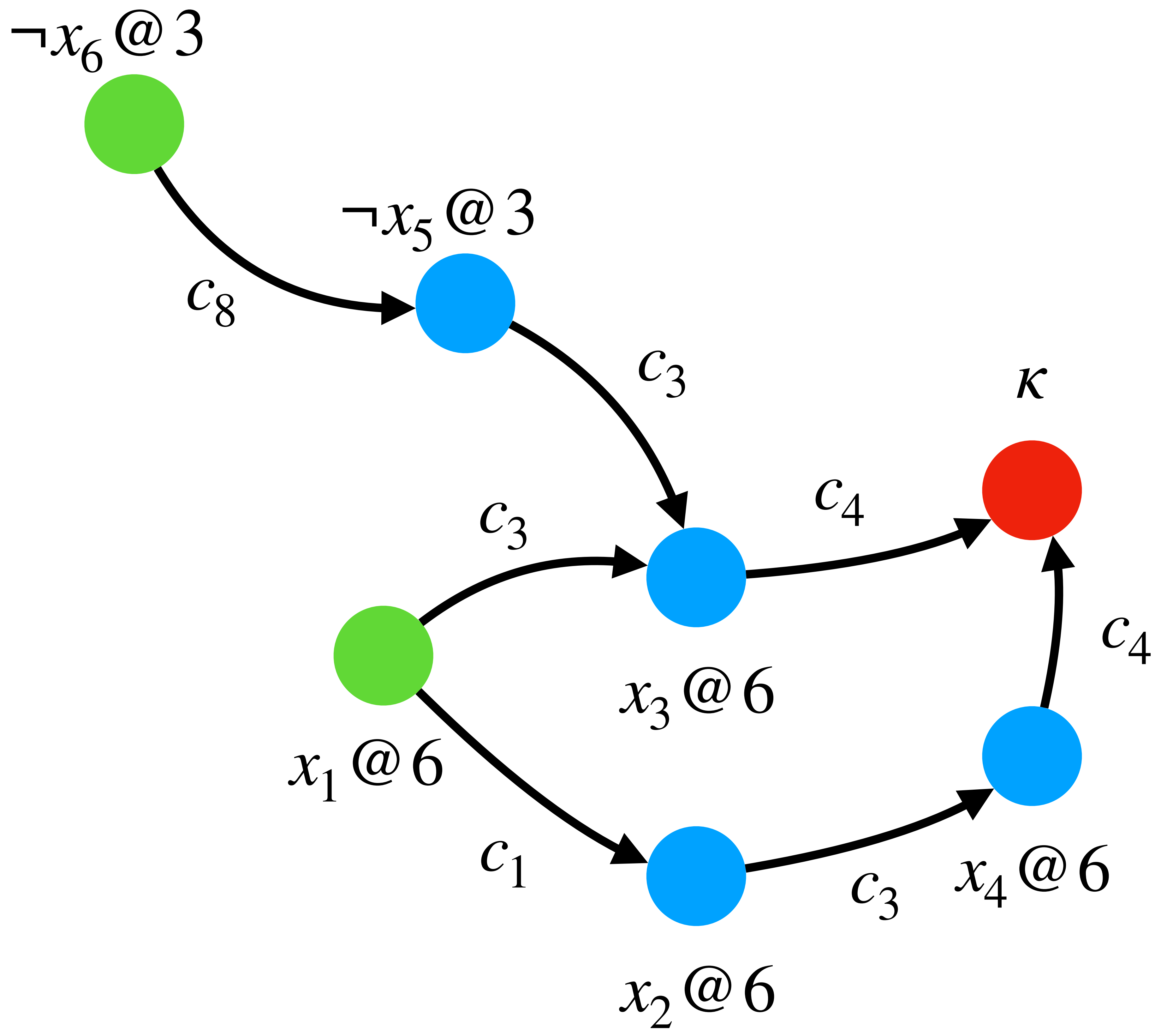
$c_4 = \neg x_3 \vee \neg x_4$

$c_5 = x_1 \vee x_5 \vee \neg x_2$

$c_6 = x_2 \vee x_3$

$c_7 = x_2 \vee \neg x_3$

$c_8 = x_6 \vee \neg x_5$



Example from the "Decision Procedures" book (Chap 2)

Instead, CDCL attempts to learn ***smaller*** clauses, which represent the "root cause" of the conflict while eliding irrelevant literals

<u>Clauses</u>

$c_1 = \neg x_1 \vee x_2$

$c_2 = \neg x_1 \vee x_3 \vee x_5$

$c_3 = \neg x_2 \vee x_4$

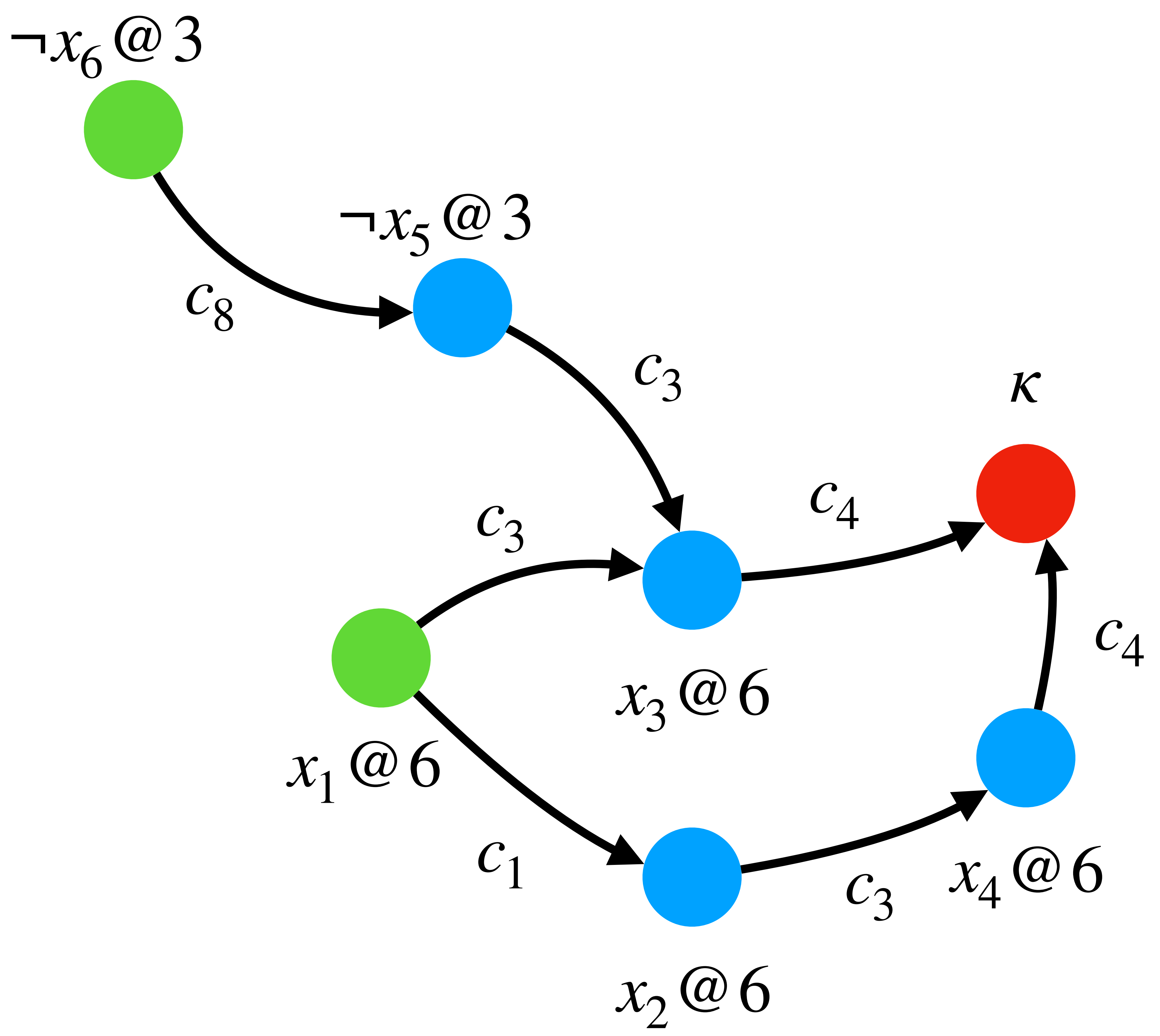
$c_4 = \neg x_3 \vee \neg x_4$

$c_5 = x_1 \vee x_5 \vee \neg x_2$

$c_6 = x_2 \vee x_3$

$c_7 = x_2 \vee \neg x_3$

$c_8 = x_6 \vee \neg x_5$



Example from the "Decision Procedures" book (Chap 2)

The procedure AnalyzeConflict is triggered upon a conflict, it ***analyzes*** the implication graph to (a) produce a learned clause and (b) compute the backtracking level

In **this** case, the learned clause will be $x_5 \vee \neg x_1$ (we will see why)

Clauses

$c_1 = \neg x_1 \vee x_2$

$c_2 = \neg x_1 \vee x_3 \vee x_5$

$c_3 = \neg x_2 \vee x_4$

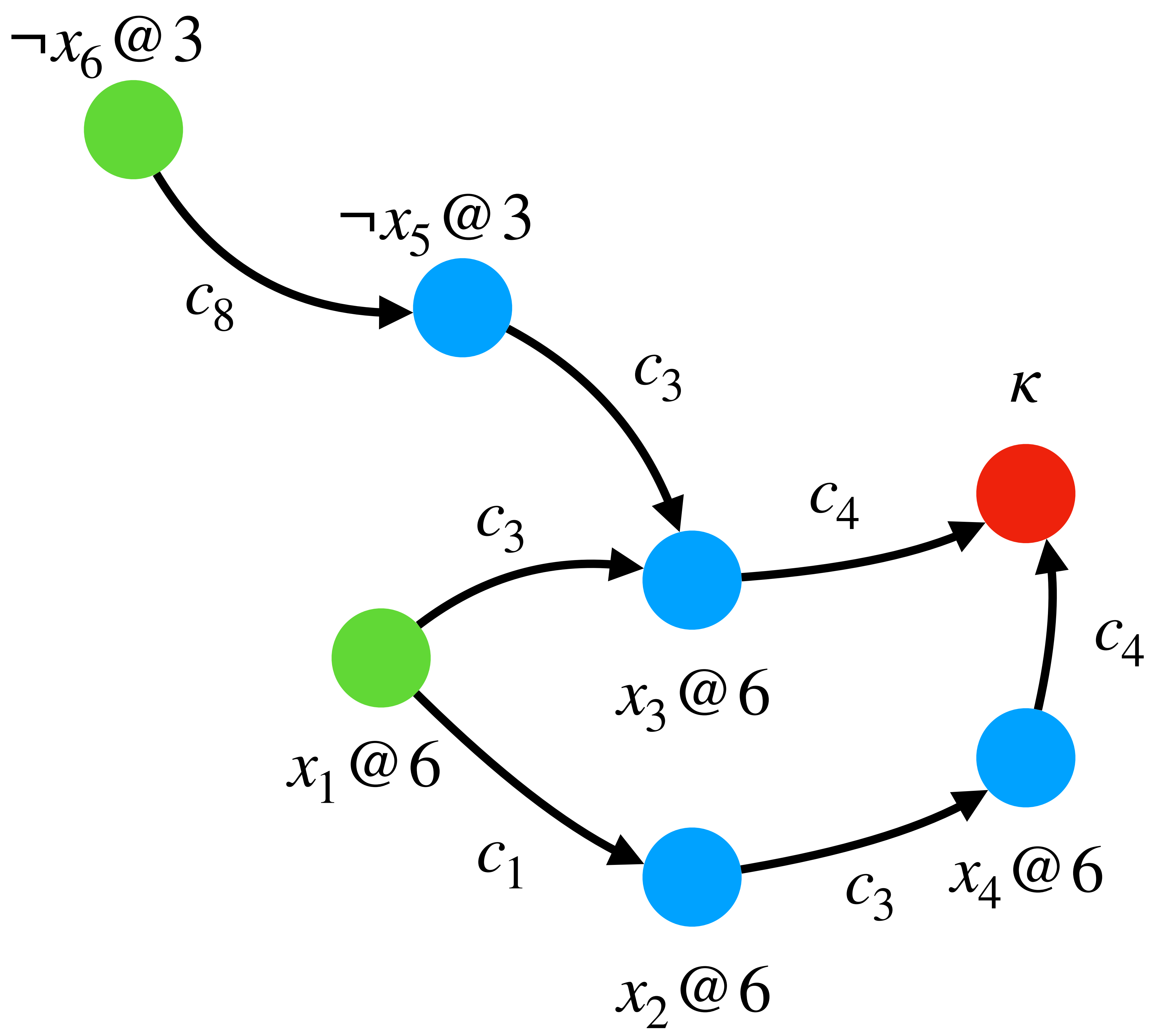
$c_4 = \neg x_3 \vee \neg x_4$

$c_5 = x_1 \vee x_5 \vee \neg x_2$

$c_6 = x_2 \vee x_3$

$c_7 = x_2 \vee \neg x_3$

$c_8 = x_6 \vee \neg x_5$



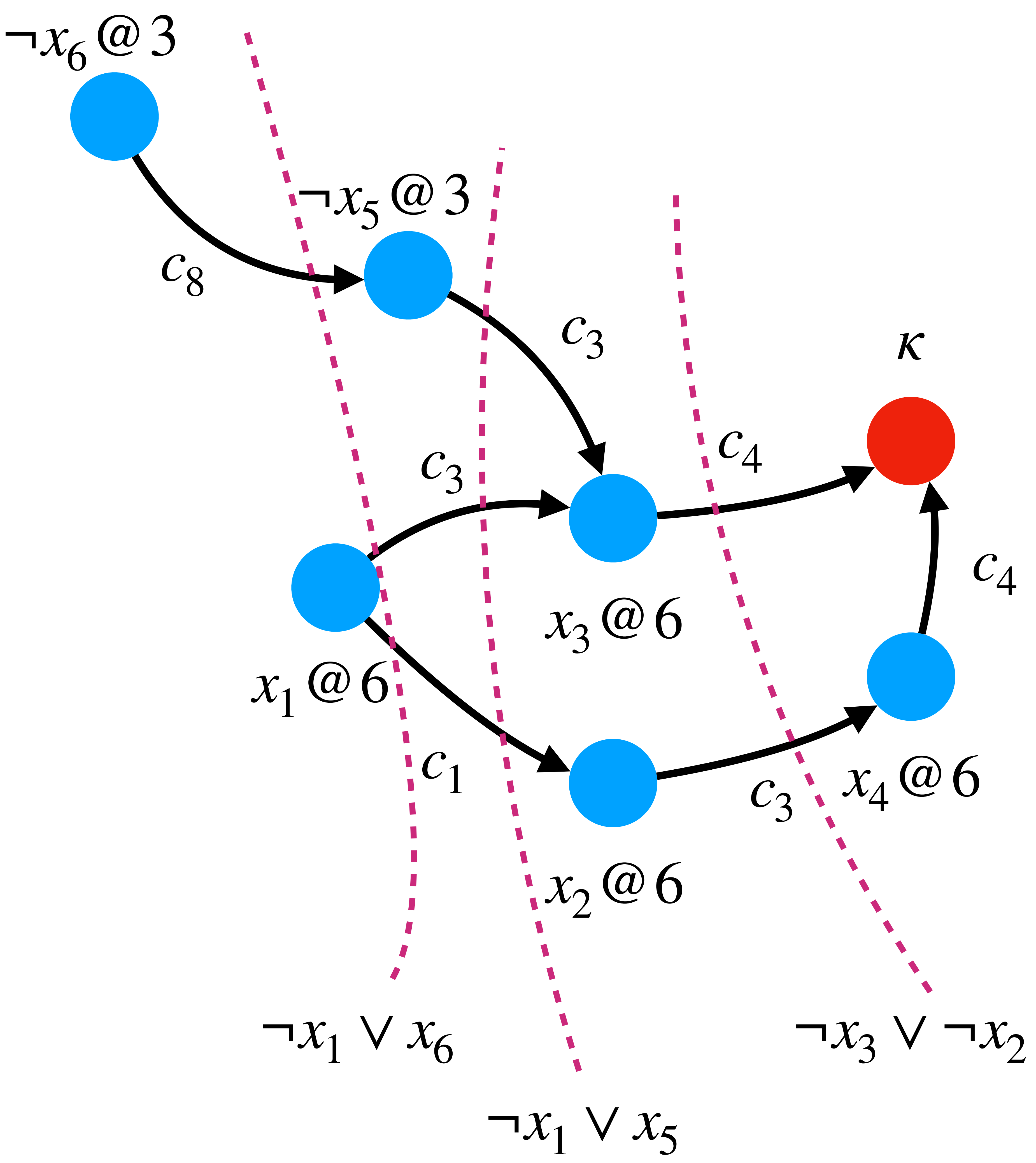
Example from the "Decision Procedures" book (Chap 2)

Separating Cuts

A *separating cut* in a conflict graph is a minimal set of edges C where—if you remove every edge in C from the graph—you break all paths from root notes to the conflict nodes

A cut partitions nodes into the *reason* side and the *conflict* side. The set of nodes on the reason side immediately "to the left" of the cut constitute a sufficient condition for the conflict. Thus, their negation constitutes a conflict clause

Three example separating cuts

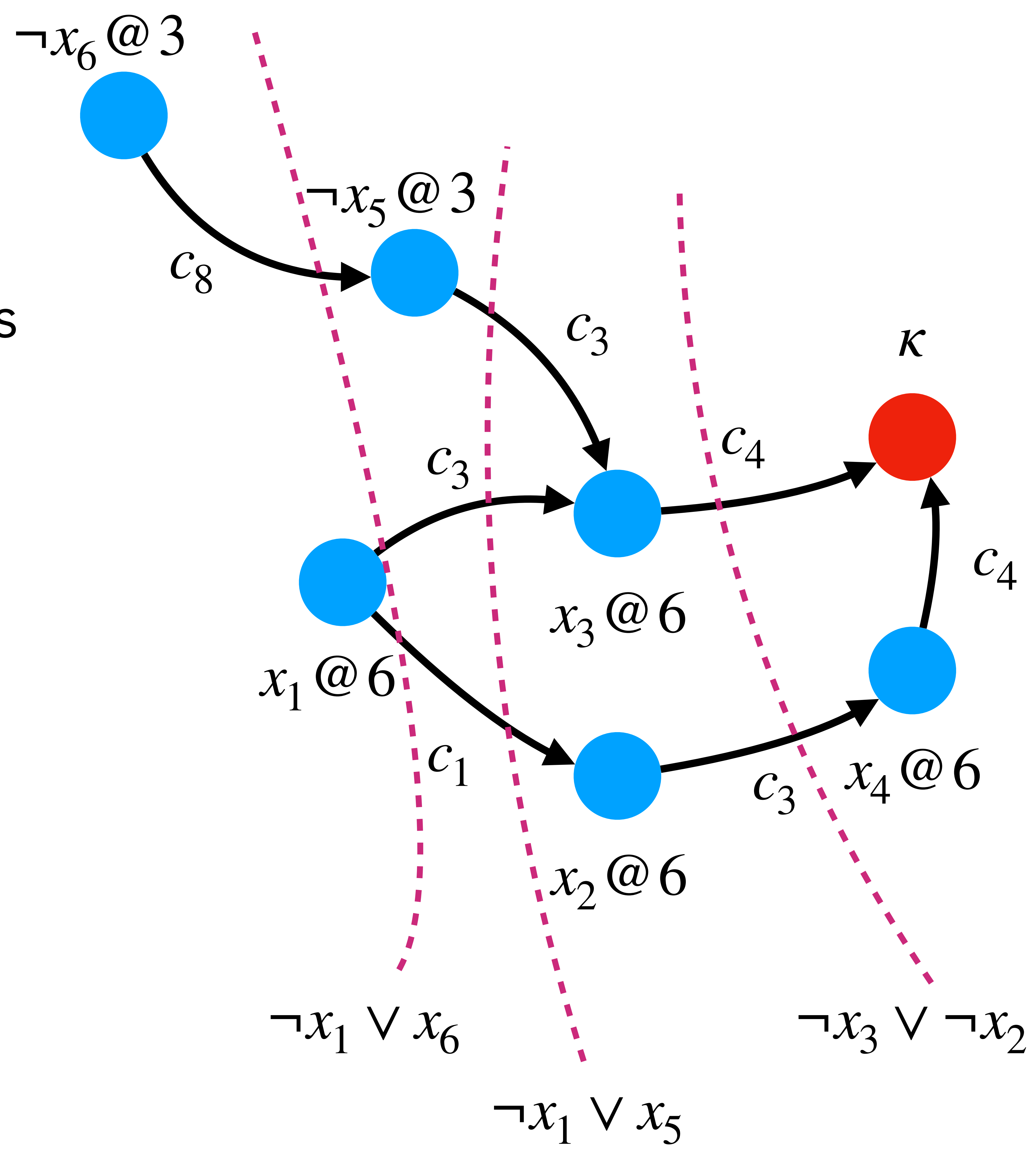


And associated conflict clauses

Any (or all) of these separating cuts could be used to generate a conflict clause, which may or may not be useful to prune the search space in subsequent decisions

In practice, many modern solvers add a *single* clause which is an *asserting* clause, whose purpose is to force unit propagation

In this case, the conflict clause will be $x_5 \vee \neg x_1$, choosing the center cut (we'll see why). Backtracking will occur based on the asserting clause chosen



CDCL will typically backtrack to the ***second most recent*** decision level in the conflict clause. In the case of $x_5 \lor \neg x_1$, this is level 3.

Clauses

$c_1 = \neg x_1 \lor x_2$

$c_2 = \neg x_1 \lor x_3 \lor x_5$

$c_3 = \neg x_2 \lor x_4$

$c_4 = \neg x_3 \lor \neg x_4$

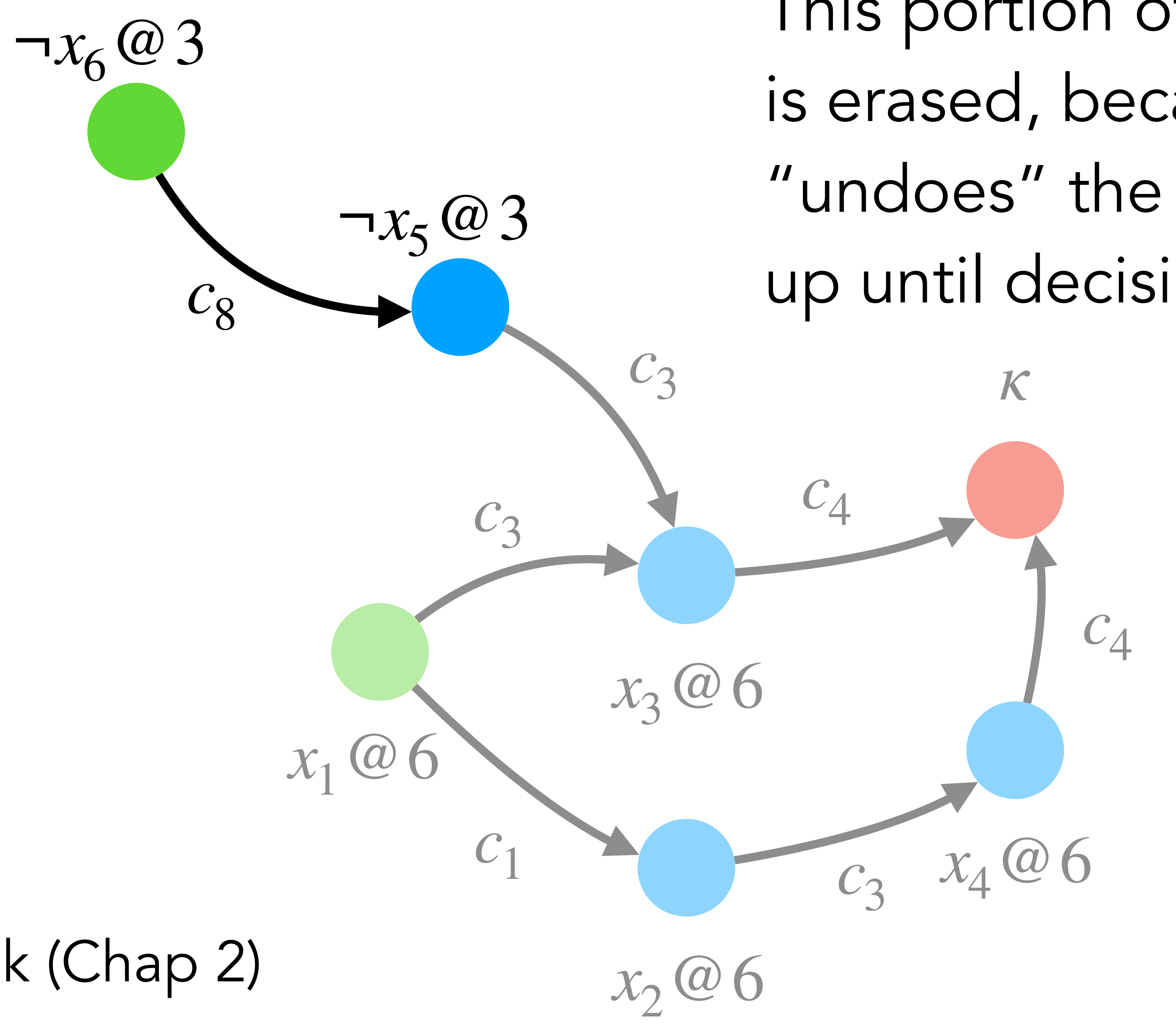
$c_5 = x_1 \lor x_5 \lor \neg x_2$

$c_6 = x_2 \lor x_3$

$c_7 = x_2 \lor \neg x_3$

$c_8 = x_6 \lor \neg x_5$

$c_9 = x_5 \lor \neg x_1$



This portion of the implication graph is erased, because backtracking "undoes" the implication graph back up until decision level 3

Example from the "Decision Procedures" book (Chap 2)

Now, the newly-learned clause c_9 **forces** unit propagation of $\neg x_1$
This is <u>not</u> a coincidence, it is by construction!

Clauses

$c_1 = \neg x_1 \vee x_2$

$c_2 = \neg x_1 \vee x_3 \vee x_5$

$c_3 = \neg x_2 \vee x_4$

$c_4 = \neg x_3 \vee \neg x_4$

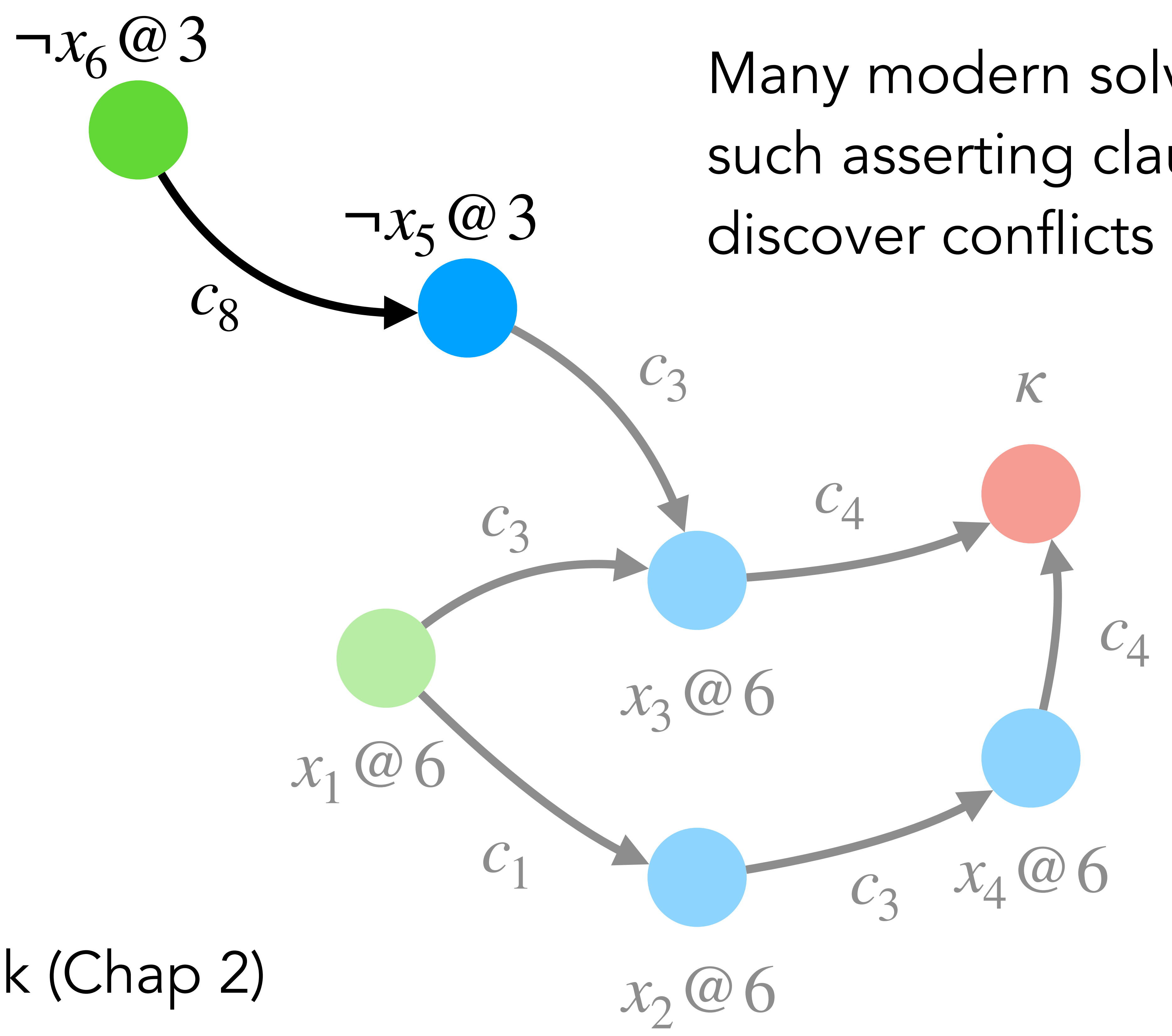
$c_5 = x_1 \vee x_5 \vee \neg x_2$

$c_6 = x_2 \vee x_3$

$c_7 = x_2 \vee \neg x_3$

$c_8 = x_6 \vee \neg x_5$

$c_9 = x_5 \vee \neg x_1$



c_9 is called an *asserting clause* because, post-backtracking, it **becomes unit** and *forces* propagation

Many modern solvers exclusively learn such asserting clauses, which tend to discover conflicts quickly

Example from the "Decision Procedures" book (Chap 2)

CDCL Algorithm

Iteratively perform the following

- Apply **unit propagation**
  - Unit propagation must be fast! Modern solvers use the *two-watched literal* trick / data structure, which enables efficient indexing based on the current (partial) assignment
- **Decide** variable air cheap (to maintain) decision heuristic
  - E.g., Variable State Independent Decay Sum (VSIDS)
- Both of these are fast (good data structures), but dominate the work
- Reasoning kicks in at *conflict analysis*, which analyzes a conflict state to **learn** a conflict clause and **restart** the search (at a lower decision level)

CDCL Algorithm

Iteratively

- **unit propagation**
  -
- **Decide**
  -
-
-

learn and **restart**

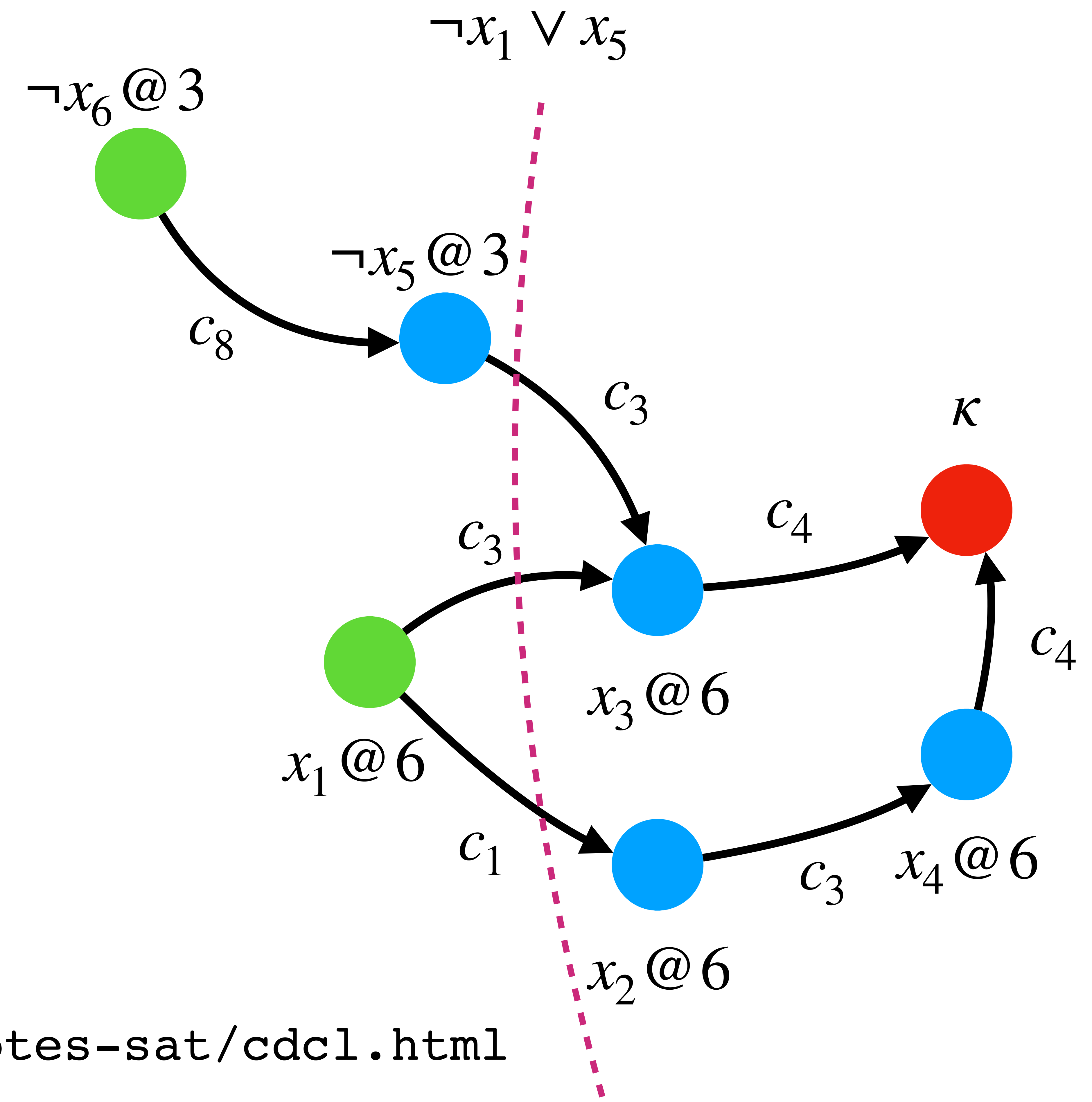
Algorithm: Analyze-Conflict
Input: A conflict graph (implication graph ending in conflict)
Output: Backtracking decision level and a new conflict clause

```
1. If current-decision-level = 0 then return UNSAT
2. cl := current-conflicting-clause
3. while (¬stop-criterion-met(cl)) do
4.   lit := Last-Assigned-Literal(cl);
5.   var := Variable-Of-Literal(lit);
6.   ante := Antecedent(lit);
7.   cl := Resolve(cl, ante, var);
8. Add-Clause-To-Database(cl);
9. return Clause-Asserting-Level (cl); // 2nd highest decision level in cl
```

Unique Implication Points (UIPs)

- A vertex *l* in the implication graph is a **unique implication point** (UIP) if all paths from the latest decision literal vertex that reach the conflict node go through *l*
  - In graph theory terms, a UIP is a *dominator*
- If *l* is a UIP, then a **UIP cut** is a cut (A,B) such that:
  - B (the “right” side of the cut) contains all the successors of *l* such that there is a from that successor to the conflict
  - A is everything else
- The **first UIP** is the UIP that is closest to the conflict node

Here, the UIP is just the decision literal. The UIP cut is a separating cut, and leads to a learned clause which is implied by the original formula, in this case $\neg x_1 \vee x_5$



Some more great example of UIP cuts are here

`https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/cdcl.html`

- Most CDCL implementations generate the learned clause *C* by using the first UIP cut
- Then they perform **non-chronological backtracking**
  - Let *m* be the second-largest decision level of of *C*, the learned clause
  - Remove all literals with decision level greater than *m* from the trail (assignment queue)
- Because *C* is derived from a UIP cut, by construction, it contains *exactly one* literal in the latest decision level before buckjumping
  - Thus, after buckjumping, *C* ***necessarily*** forces unit propagation
  - (Again, we say that C is an "asserting clause.")

Exit Conditions for CDCL

- If the formula is **satisfiable**, CDCL will eventually find the satisfying truth assignment
  - Learned clauses preserve satisfiability
- For **unsatisfiable** formulas, the algorithm eventually derives UNSAT because each conflict results in a new learned clause
  - Eventually, we derive a conflict **at level 0**
  - Only finitely many clauses, thus at some point we'll produce enough unit clauses at level 0 to hit UNSAT

Two-Watched Literals

- We need assignment to be **blazing** fast
  - When we assign some literal (say $\neg x_1$), we need to be able to *quickly* identify which clauses should be considered
- Naively, we would have to look through **all** the clauses
- But, there's an insight: we only need to detect when a clause becomes *unit*!
- Thus, only need to watch **two** (unassigned) lits in each clause
- This yields the **two watched literals** approach:
  - For each **variable**, we keep a linked list of *watched clauses*

$$c_1 = \neg x_1 \vee x_2$$
$$c_2 = \neg x_1 \vee x_3 \vee x_5$$
$$c_3 = \neg x_2 \vee x_4$$
$$c_4 = \neg x_3 \vee \neg x_4$$
$$c_5 = x_1 \vee x_5 \vee \neg x_2$$
$$c_6 = x_2 \vee x_3$$
$$c_7 = x_2 \vee \neg x_3$$
$$c_8 = x_6 \vee \neg x_5$$

- For example, we might say…
  - x1 -> {c1, c2, c5}
  - x2 -> {c1, c3, c6, c7}
  - x3 -> {c2, c4, c6, c7}
  - …
- When x3 is assigned, we look only at {c2, c4, c6, c7}
  - We discuss some more details in our discussion on MiniSAT

$$
\begin{aligned}
c_1 &= \neg x_1 \vee x_2 \\
c_2 &= \neg x_1 \vee x_3 \vee x_5 \\
c_3 &= \neg x_2 \vee x_4 \\
c_4 &= \neg x_3 \vee \neg x_4 \\
c_5 &= x_1 \vee x_5 \vee \neg x_2 \\
c_6 &= x_2 \vee x_3 \\
c_7 &= x_2 \vee \neg x_3 \\
c_8 &= x_6 \vee \neg x_5
\end{aligned}
$$