2-SAT

ISIS 2801

SATisfiability

Given a propositional formula in **conjunctive normal form**
we want to know its satisfiability conditions
That is, give values to the variables such that
the proposition is **true**

multiple **disjunctions** joined by **conjunctions**

$$(p_{11} \vee \ldots \vee p_{1n}) \wedge (\neg p_{21} \vee \ldots \vee p_{2m}) \wedge \ldots \wedge (\neg p_{N1} \vee \ldots \vee p_{Nk})$$

SATisfiability

Given a propositional formula in **conjunctive normal form** we want to know its satisfiability conditions
That is, give values to the variables such that the proposition is **true**

multiple **disjunctions** joined by **conjunctions**

$$(p_{11} \vee \ldots \vee p_{1n}) \wedge (\neg p_{21} \vee \ldots \vee p_{2m}) \wedge \ldots \wedge (\neg p_{N1} \vee \ldots \vee p_{Nk})$$

trying all the possibilities takes $O(2^n)$

SAT is, in general, an NP-complete problem

However, its restricted version k-SAT are computable in polynomial time

2-SAT

Given a propositional formula in **conjunctive normal form**, where each conjunction has a **2-disjunction**, we want to know its satisfiability conditions
That is, give values to the variables such that the proposition is **true**

exactly 2 propositions per **conjunction**

$$(p_1 \vee p_3) \wedge (\neg p_1 \vee p_2) \wedge (\neg p_2 \vee p_3)$$

$p_1 := T$
$p_2 := T$
$p_3 := T$

2-SAT

Build a (implication) graph for each of the literals and their negation

- Each variable is a node
- There is an edge between two nodes if there is an implication between the variables

2-SAT

1. Implify the disjunctions

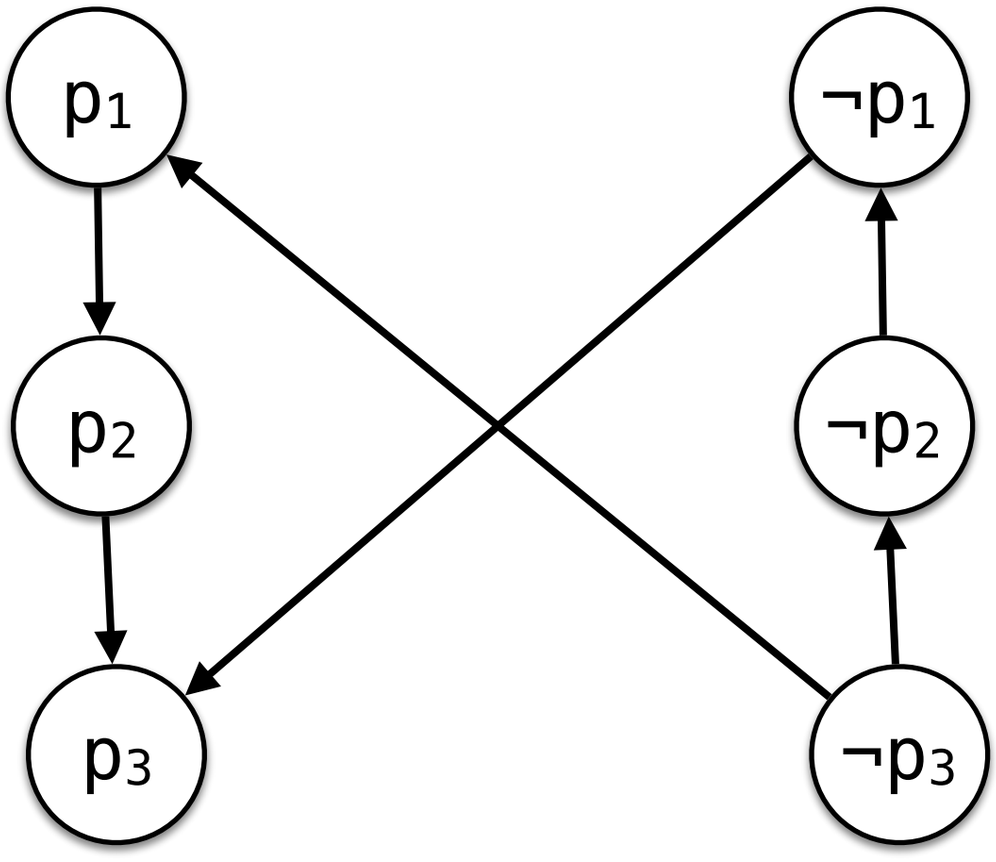
$$(p_1 \vee p_3) \equiv (\neg p_1 \Rightarrow p_3) \equiv (\neg p_3 \Rightarrow p_1)$$

$$(\neg p_1 \vee p_2) \equiv (p_1 \Rightarrow p_2) \equiv (\neg p_2 \Rightarrow \neg p_1)$$

$$(\neg p_2 \vee p_3) \equiv (p_2 \Rightarrow p_3) \equiv (\neg p_3 \Rightarrow \neg p_2)$$

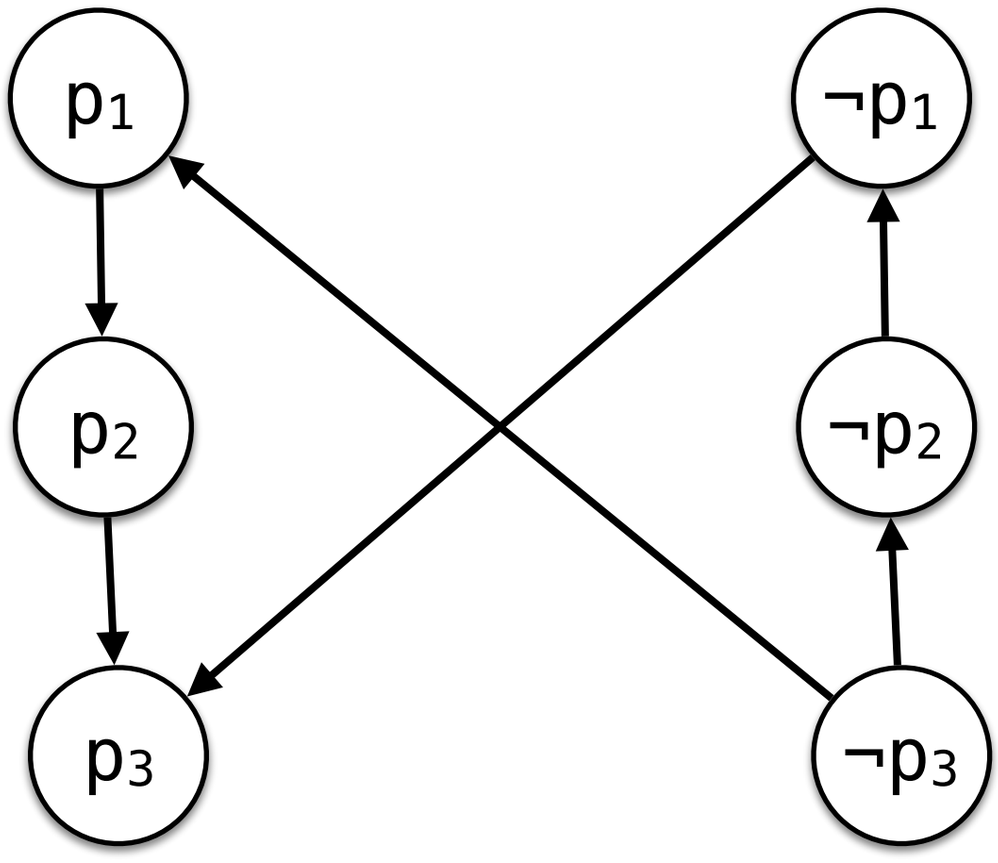
2-SAT

2. Connect the nodes (through implication)



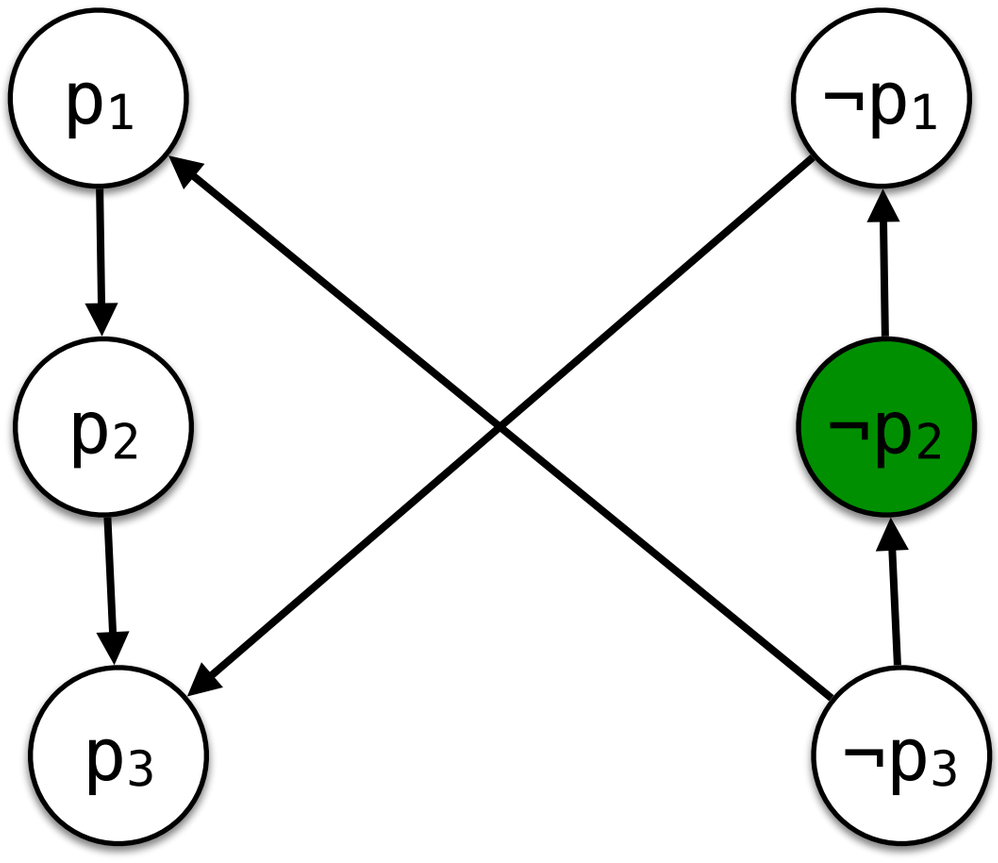
2-SAT

3. Find True values



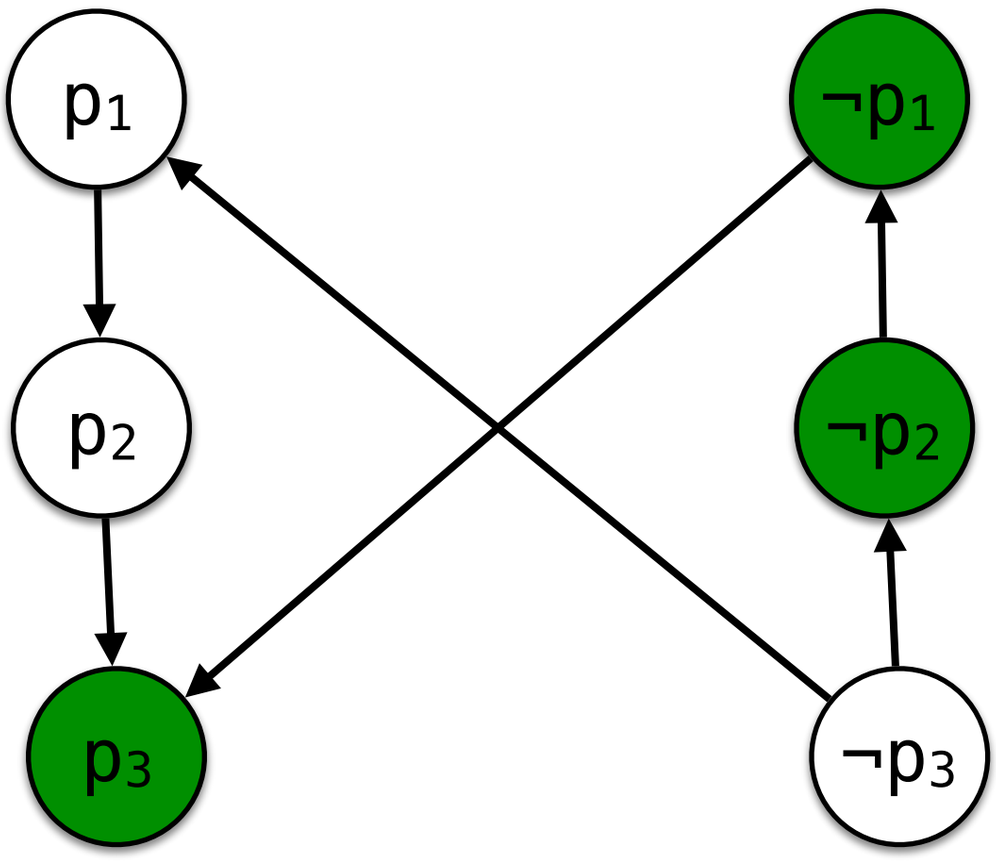
2-SAT

3. Find True values



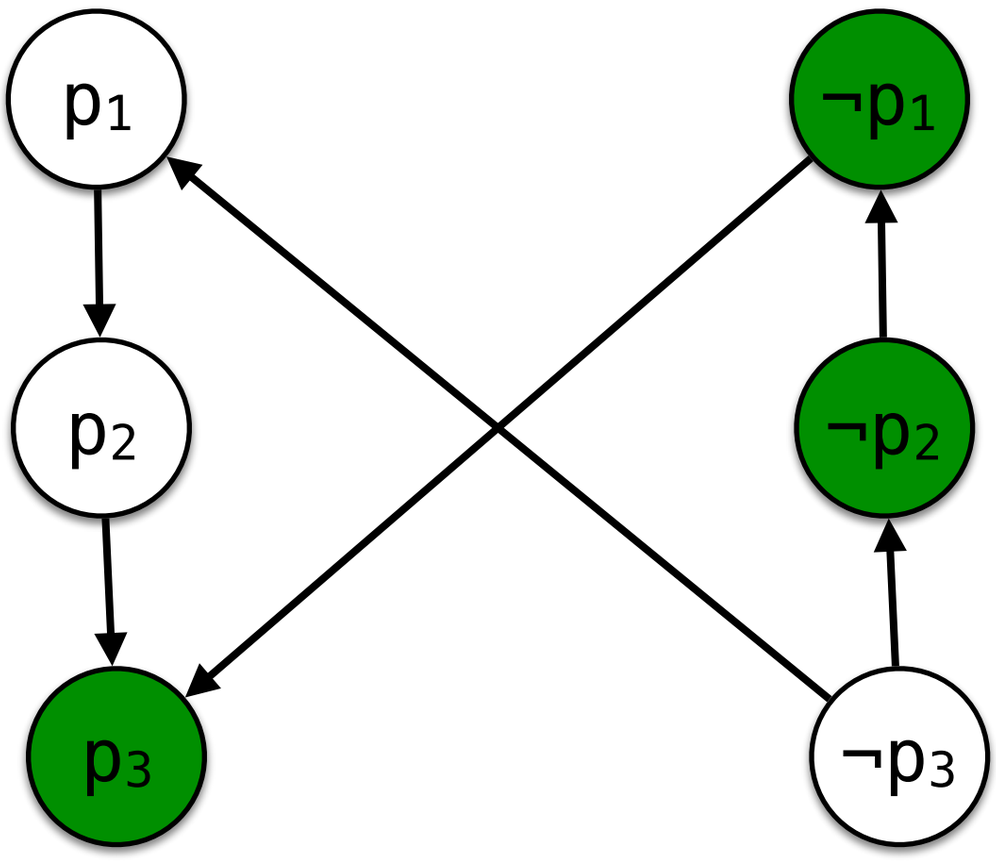
2-SAT

3. Find True values



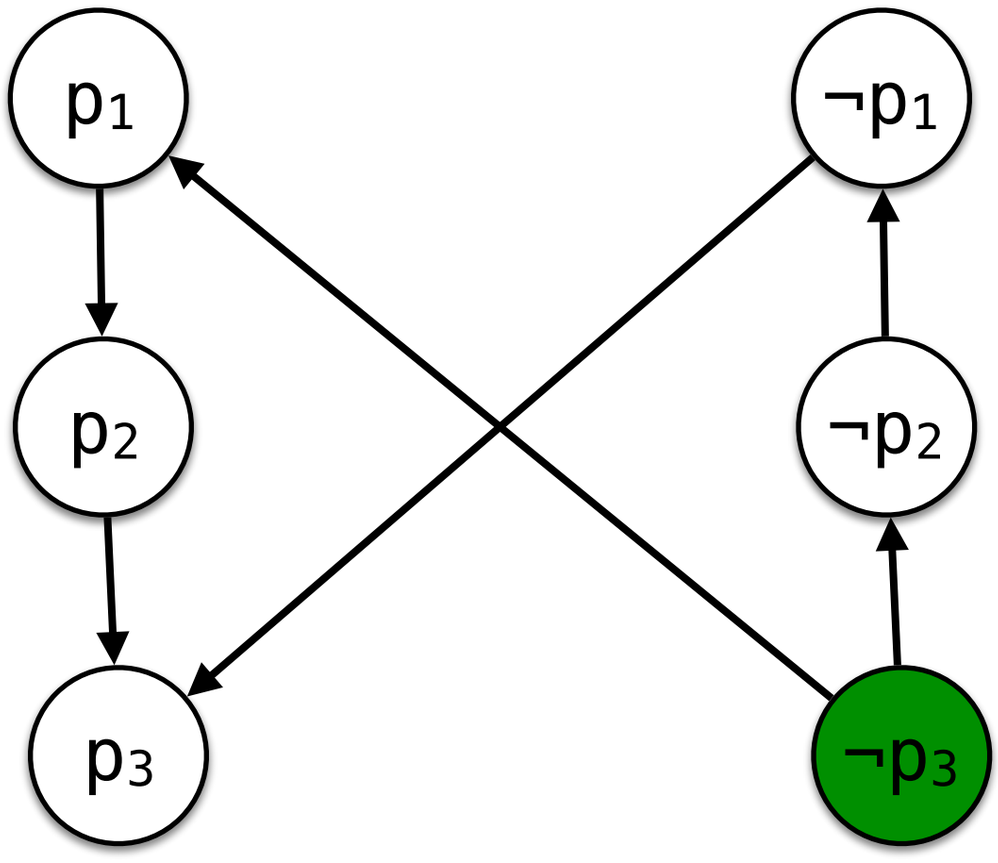
2-SAT

3. Find True values



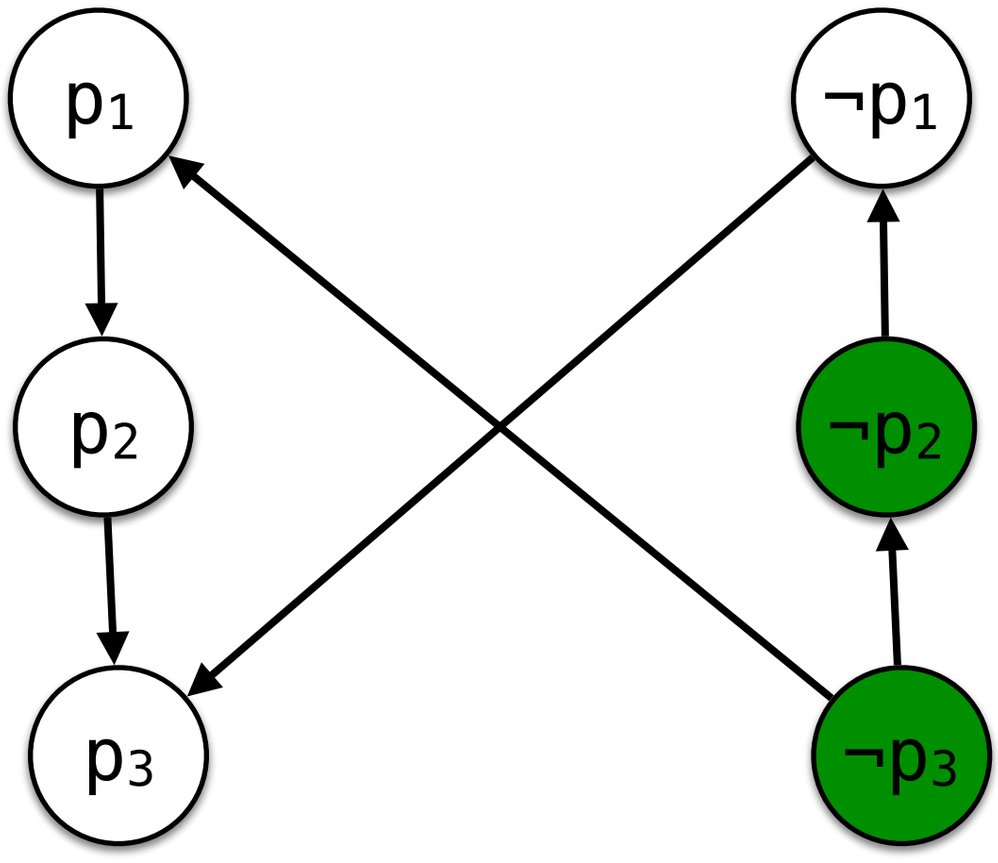
2-SAT

3. Find True values



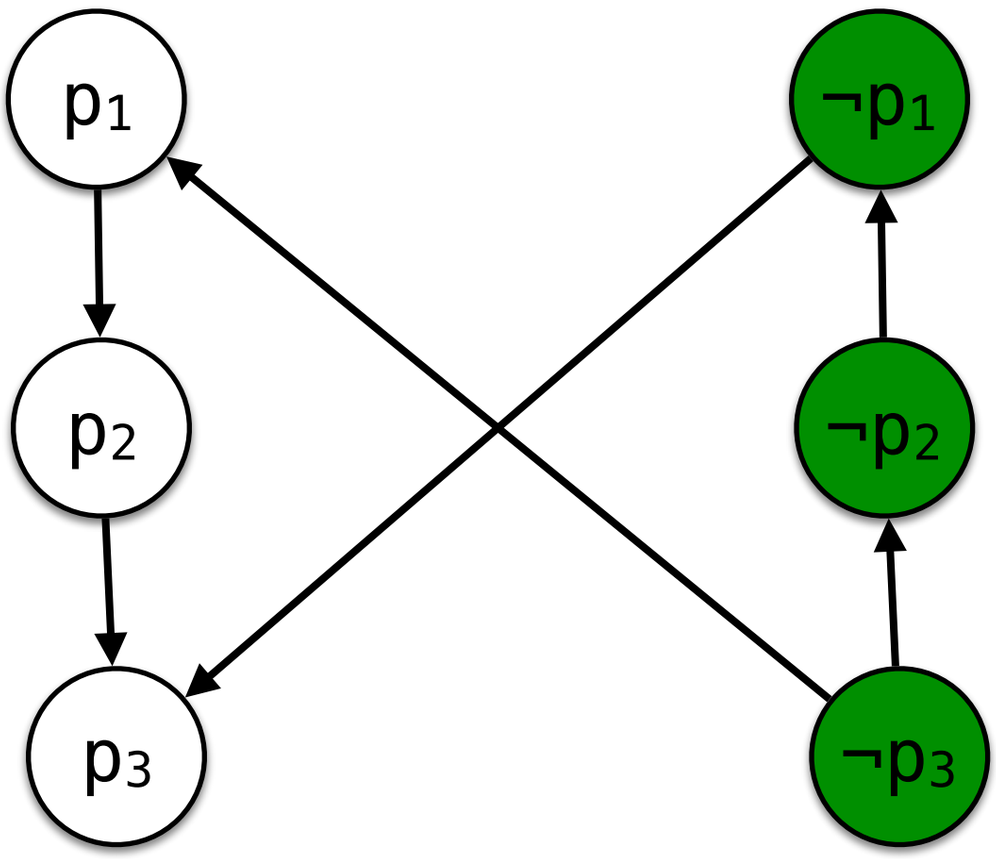
2-SAT

3. Find True values



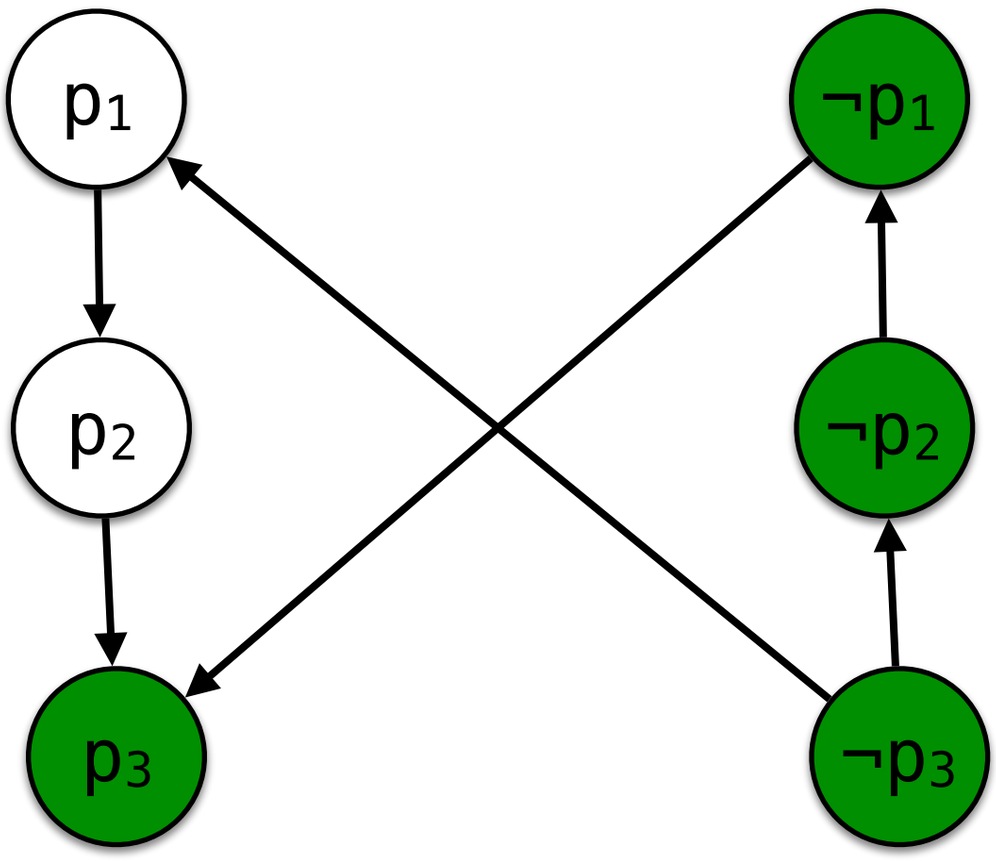
2-SAT

3. Find True values



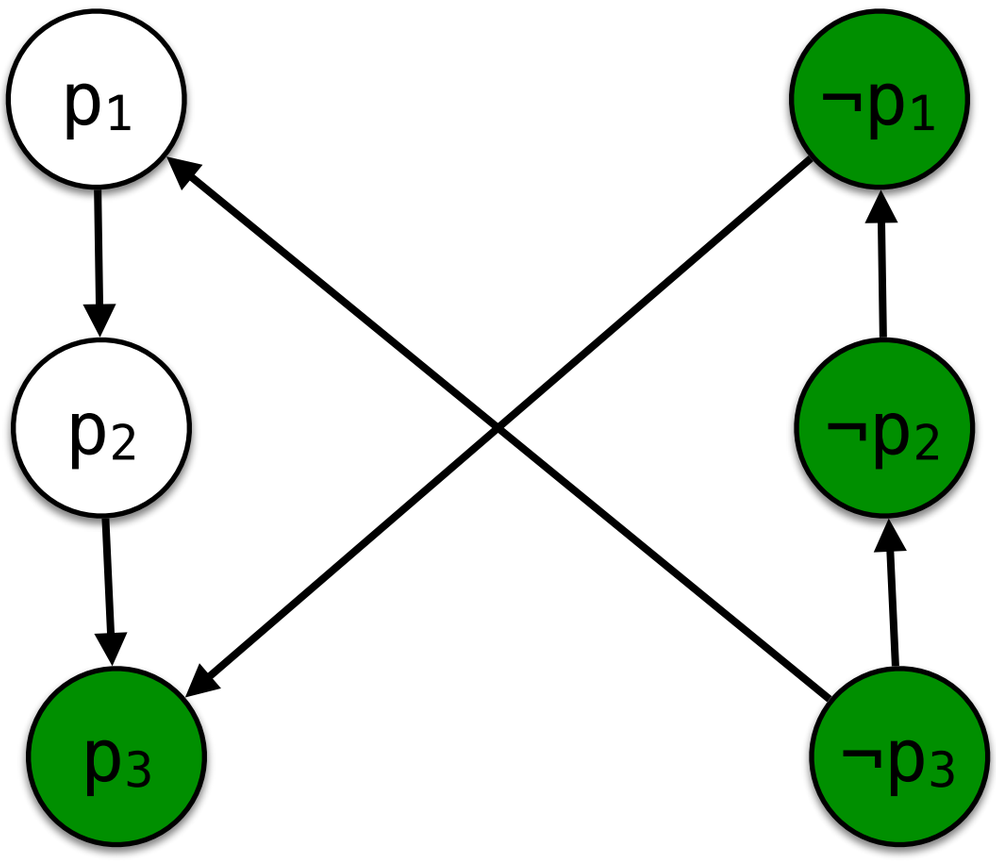
2-SAT

3. Find True values



2-SAT

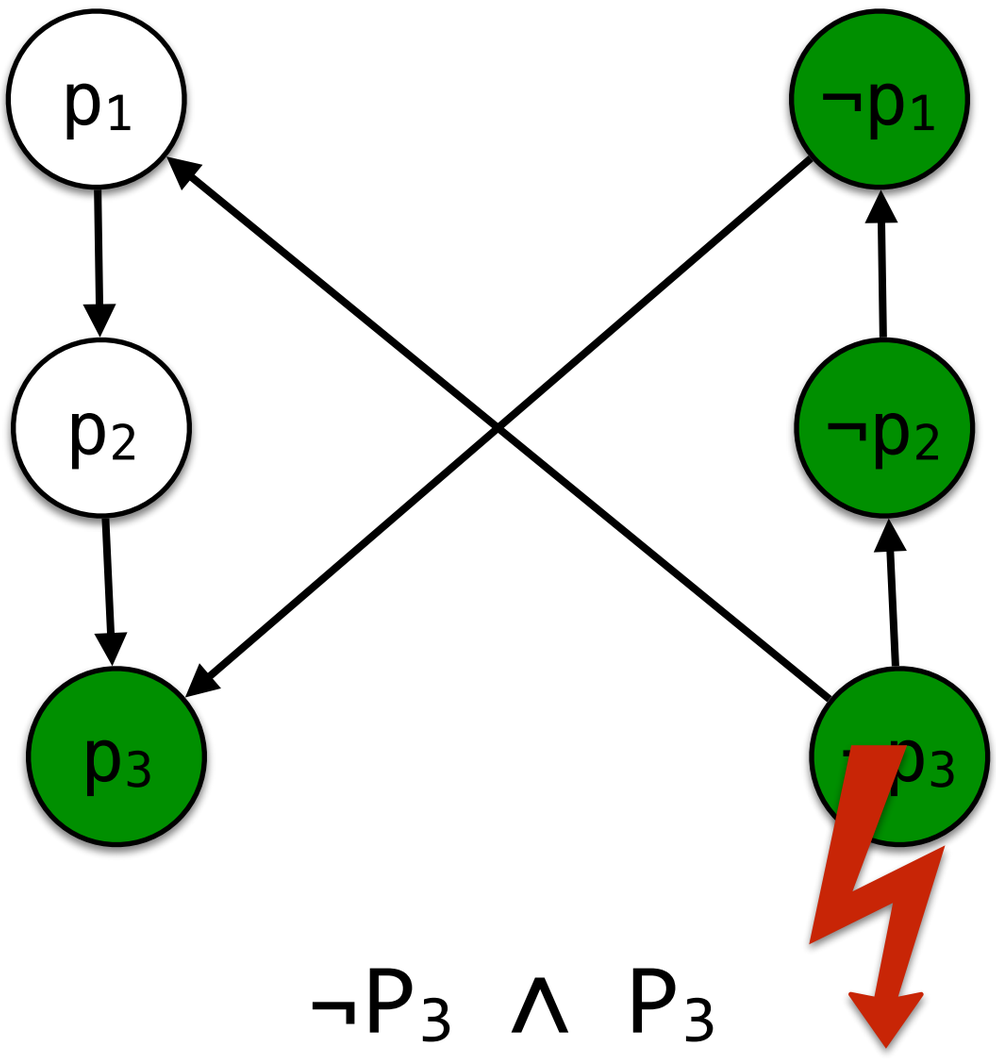
3. Find True values



$$\neg P_3 \ \wedge \ P_3$$

2-SAT

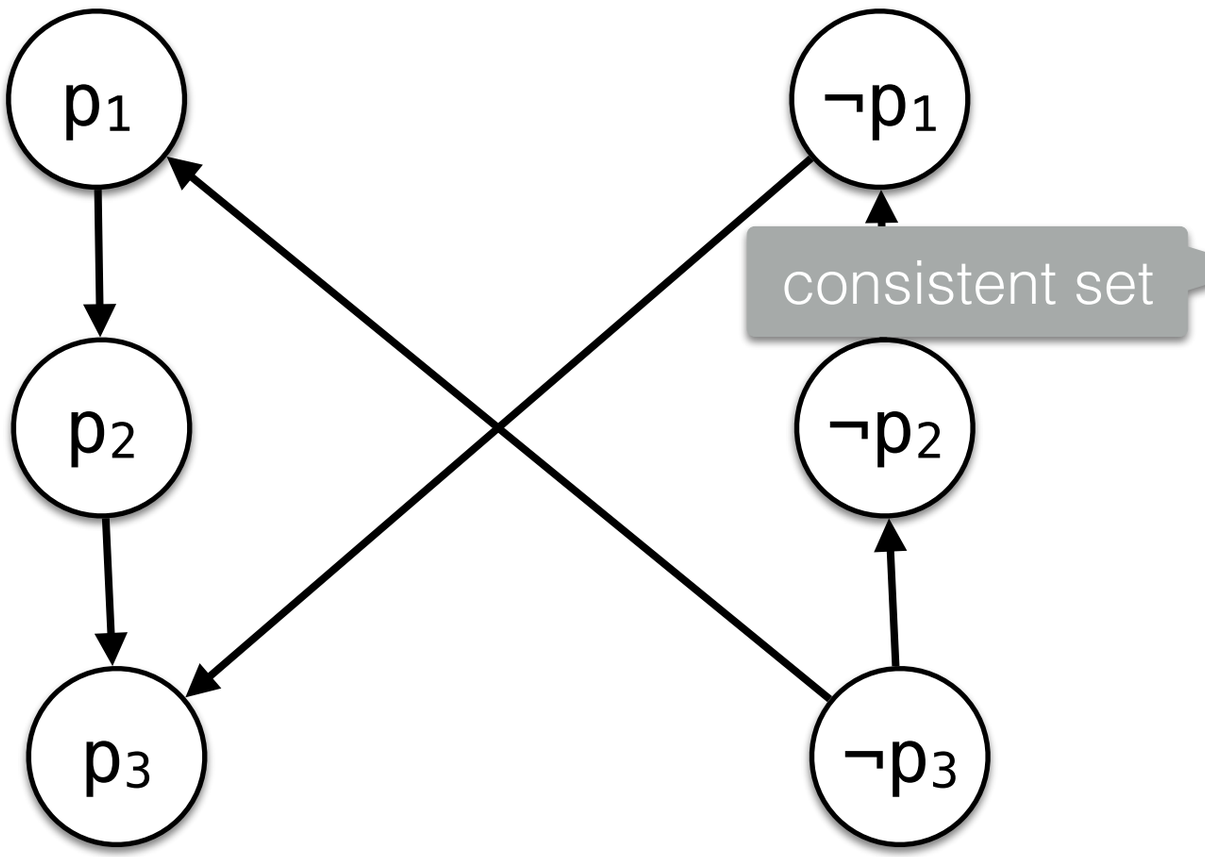
3. Find True values



2-SAT

first idea

4. building consistent sets



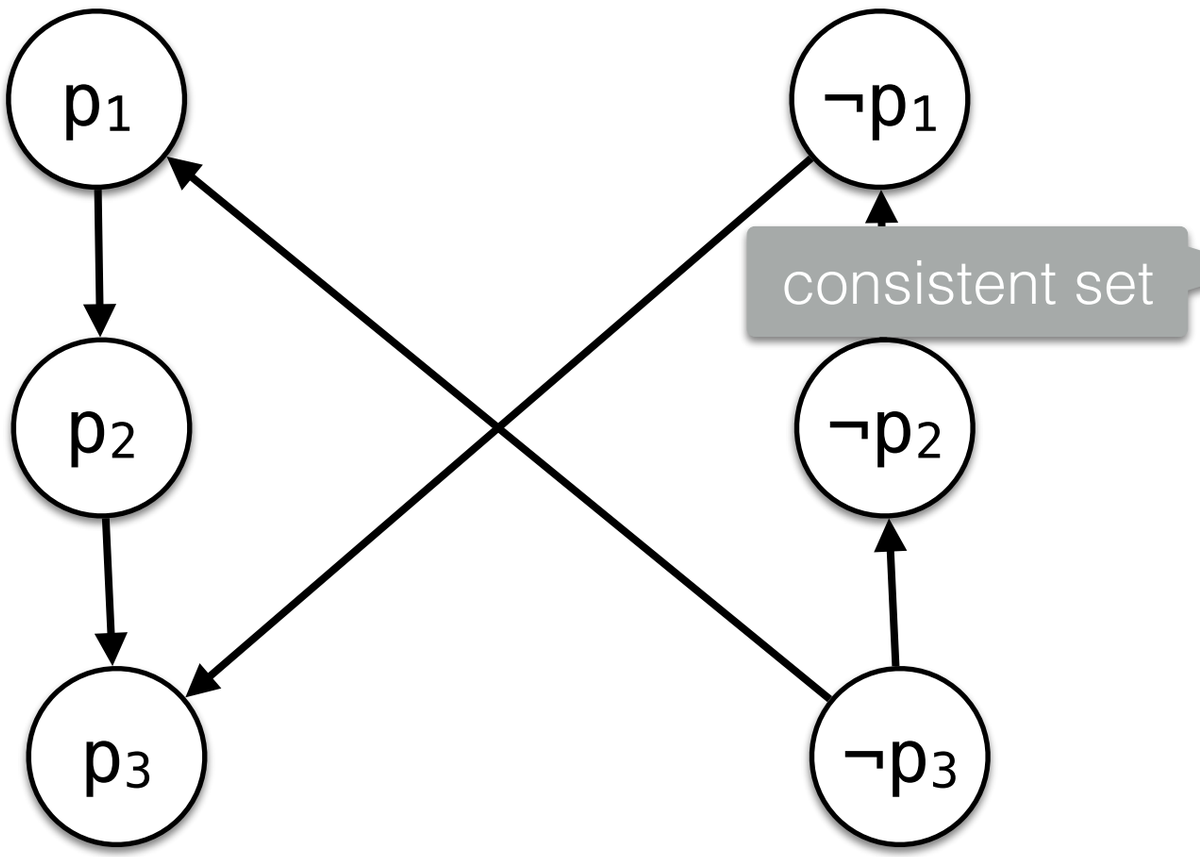
consistent set

If for **every** variable x I can **infer** both
x∨x and ¬x∨¬x, then the formula is satisfiable

2-SAT

first idea

4. building consistent sets



consistent set

If for **every** variable x I can **infer** both $x \vee x$ and $\neg x \vee \neg x$, then the formula is satisfiable

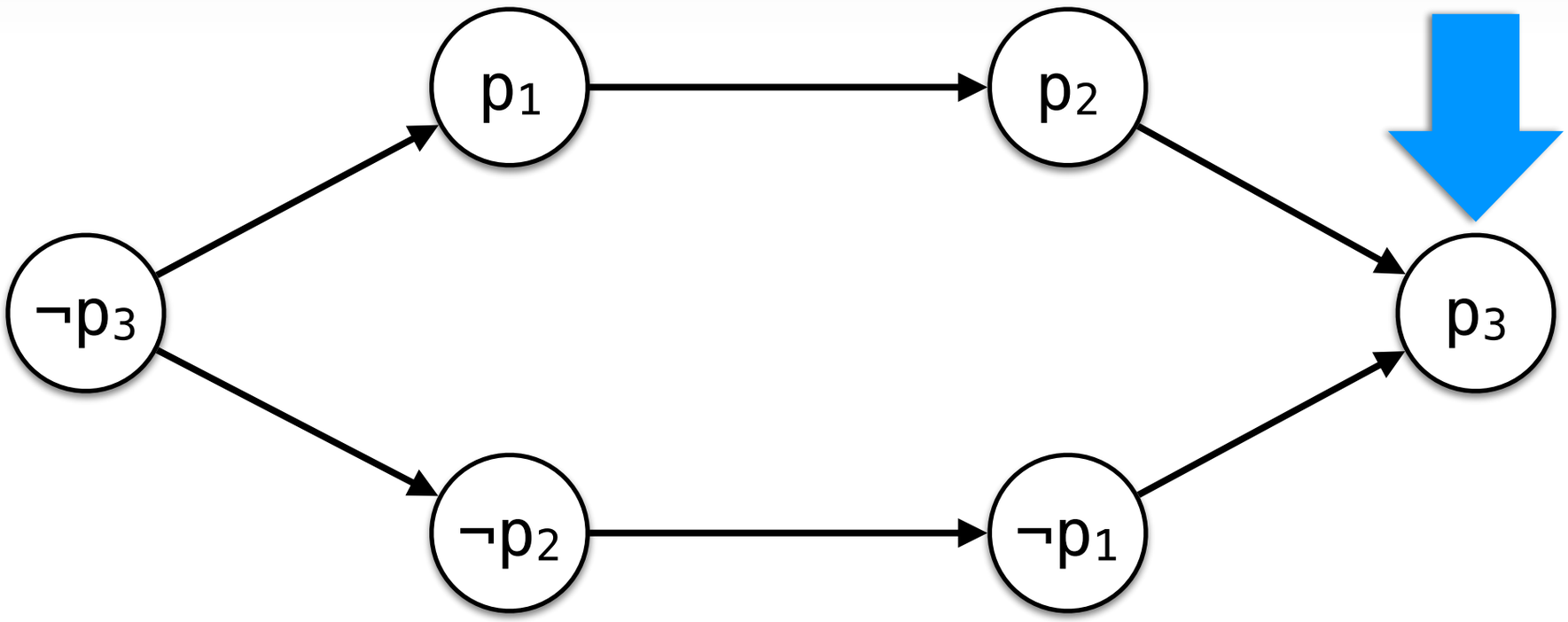
Inference happens by pairs of related variables which is $O(n^3)$. Then you need to check for consistency of the assignment, so **overall it takes $O(n^4)$**

2-SAT

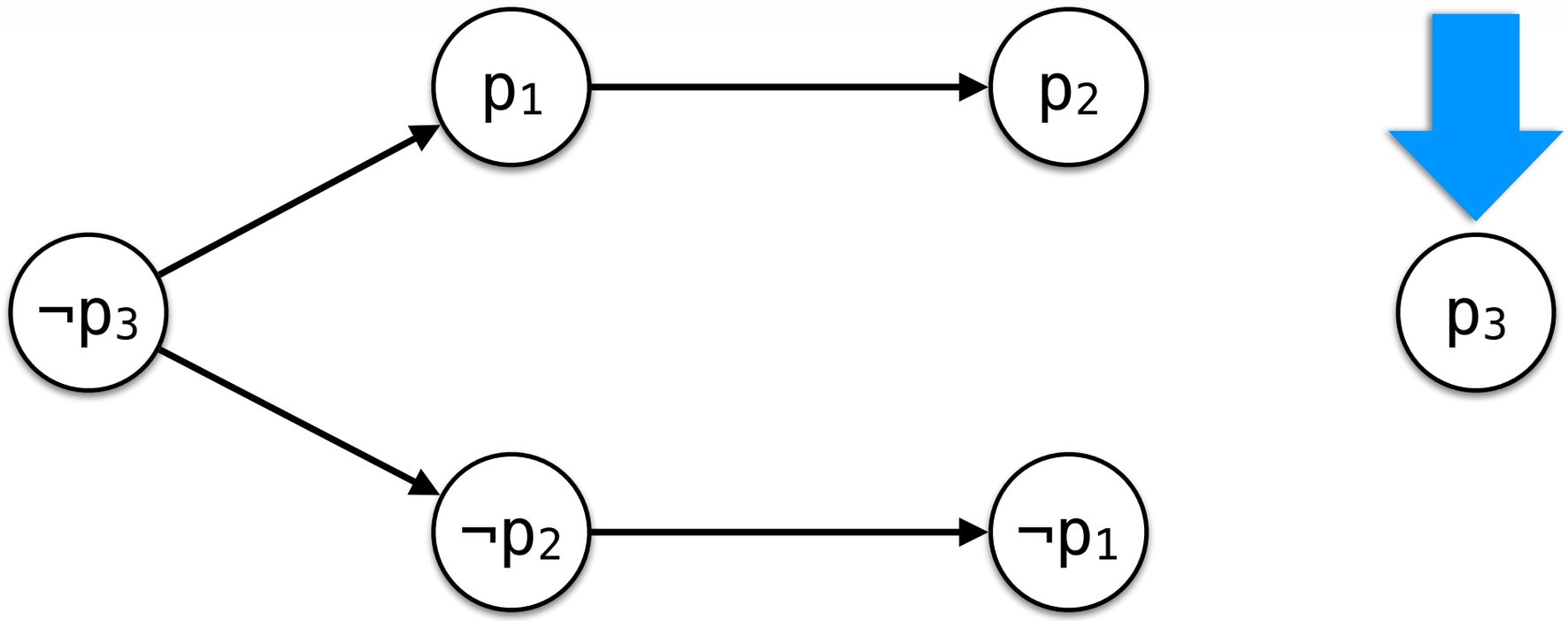
Enforcing variables

$$\neg p_1 \Rightarrow p_1 \equiv p_1$$

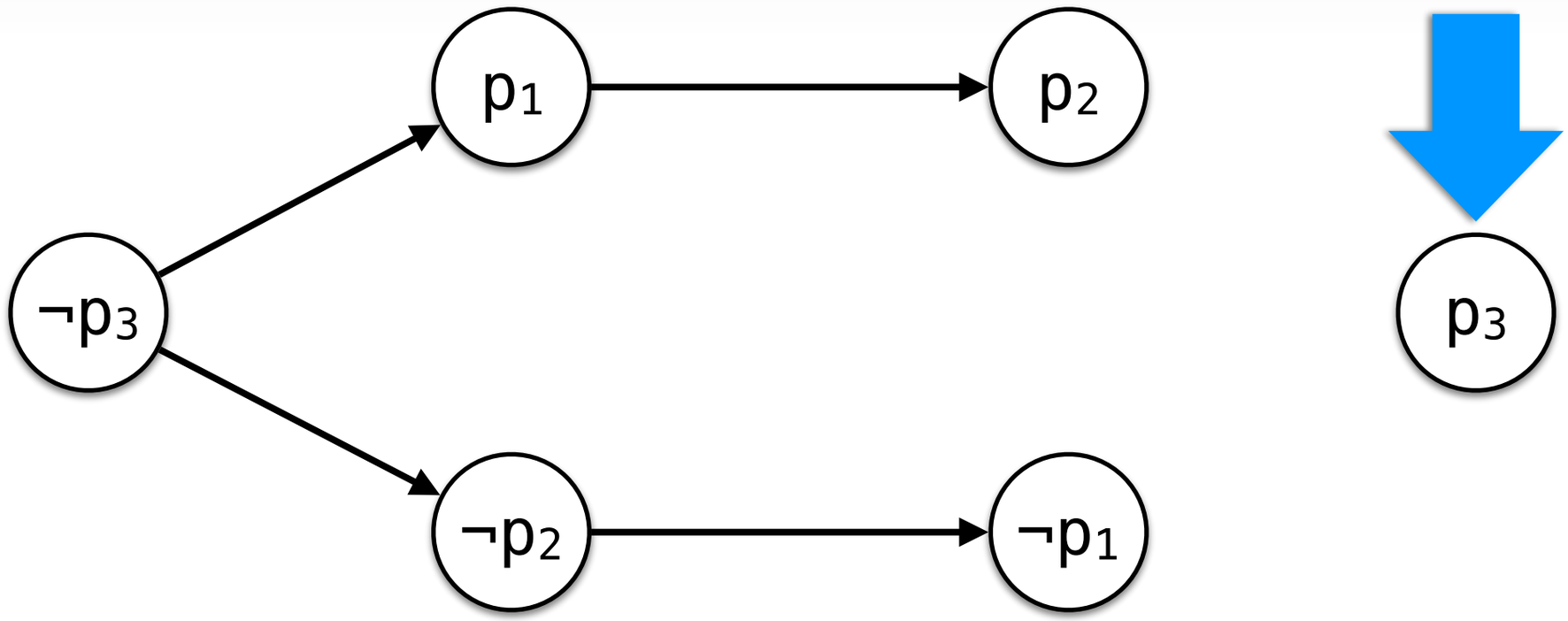
2-SAT



2-SAT

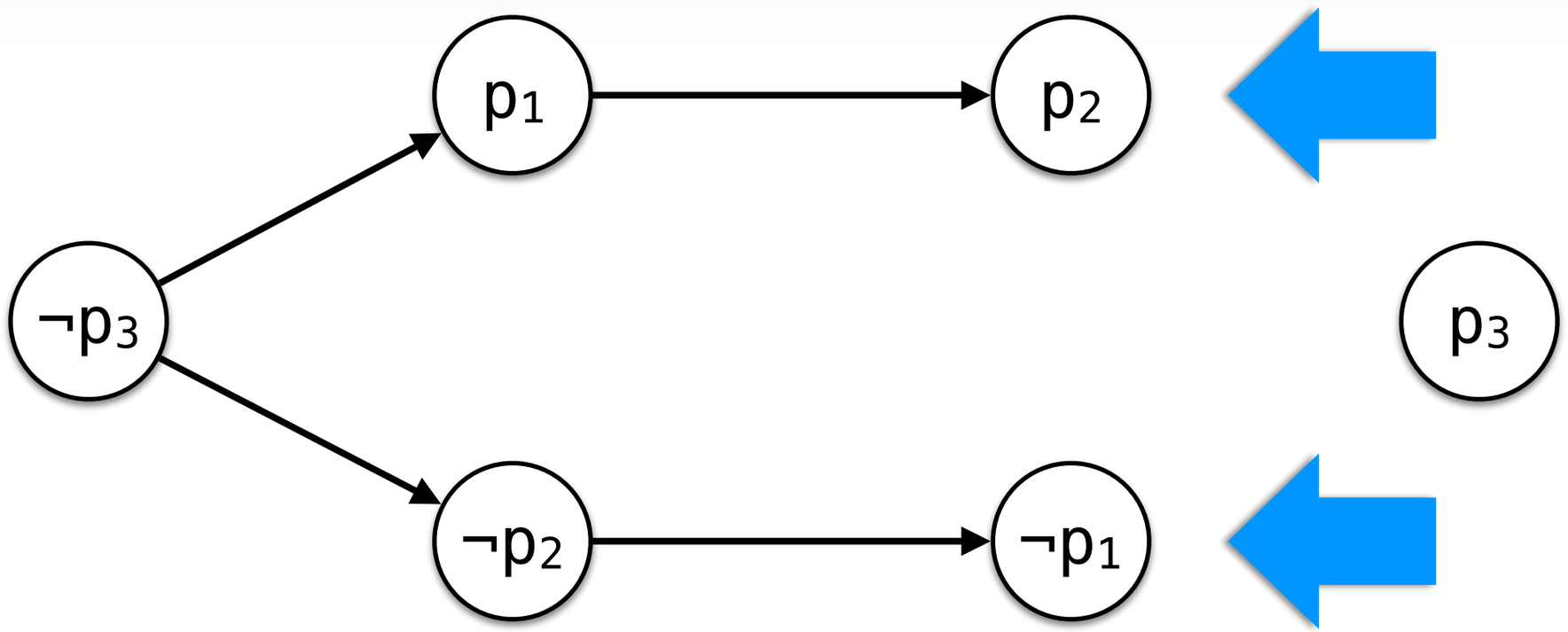


2-SAT



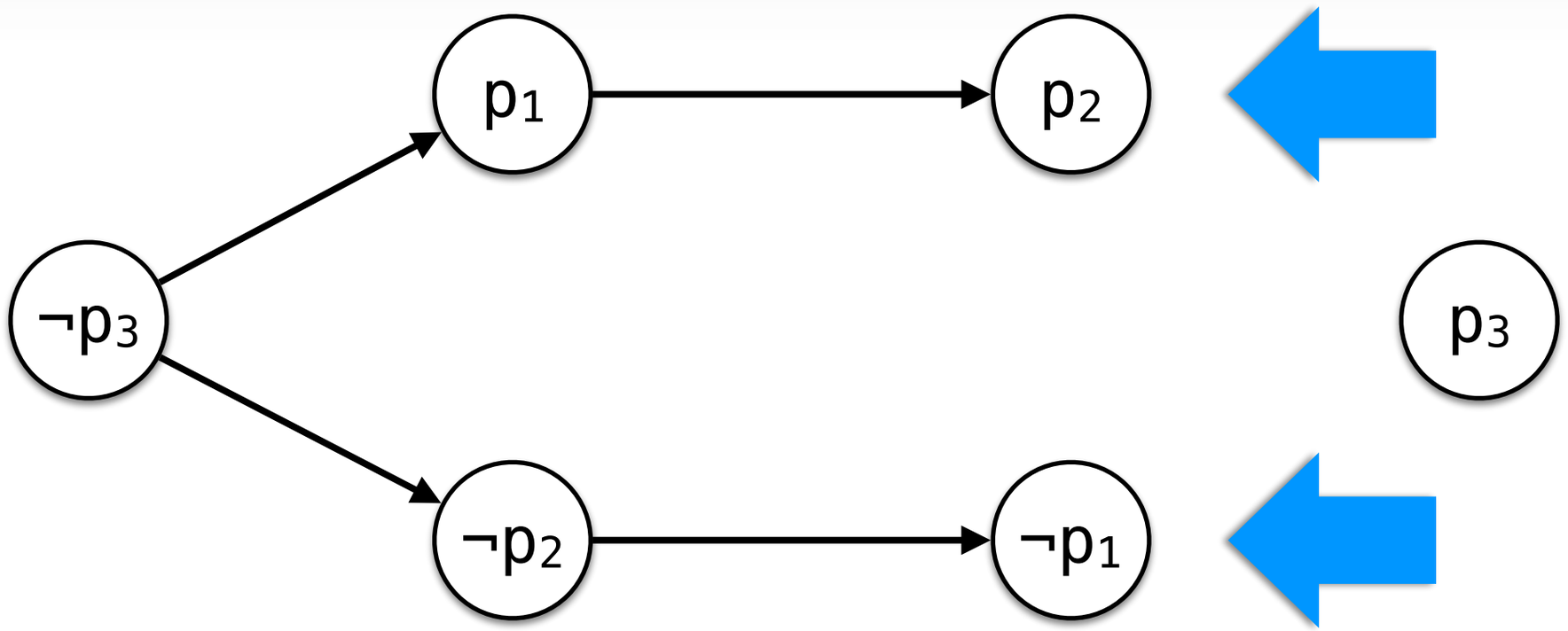
$p_3 \equiv T$

2-SAT



$p_3 \equiv T$

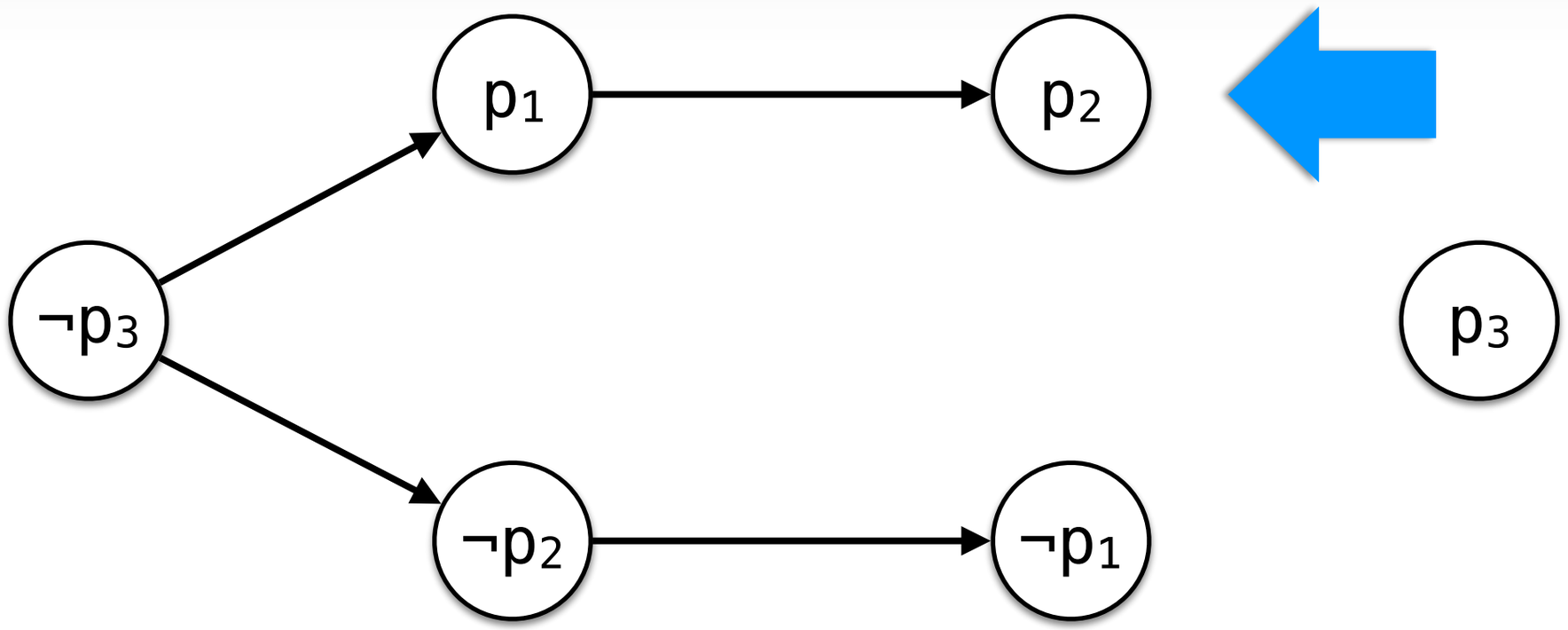
2-SAT



$p_3 \equiv T$

$p_2 \equiv T \quad \neg p_1 \equiv T$

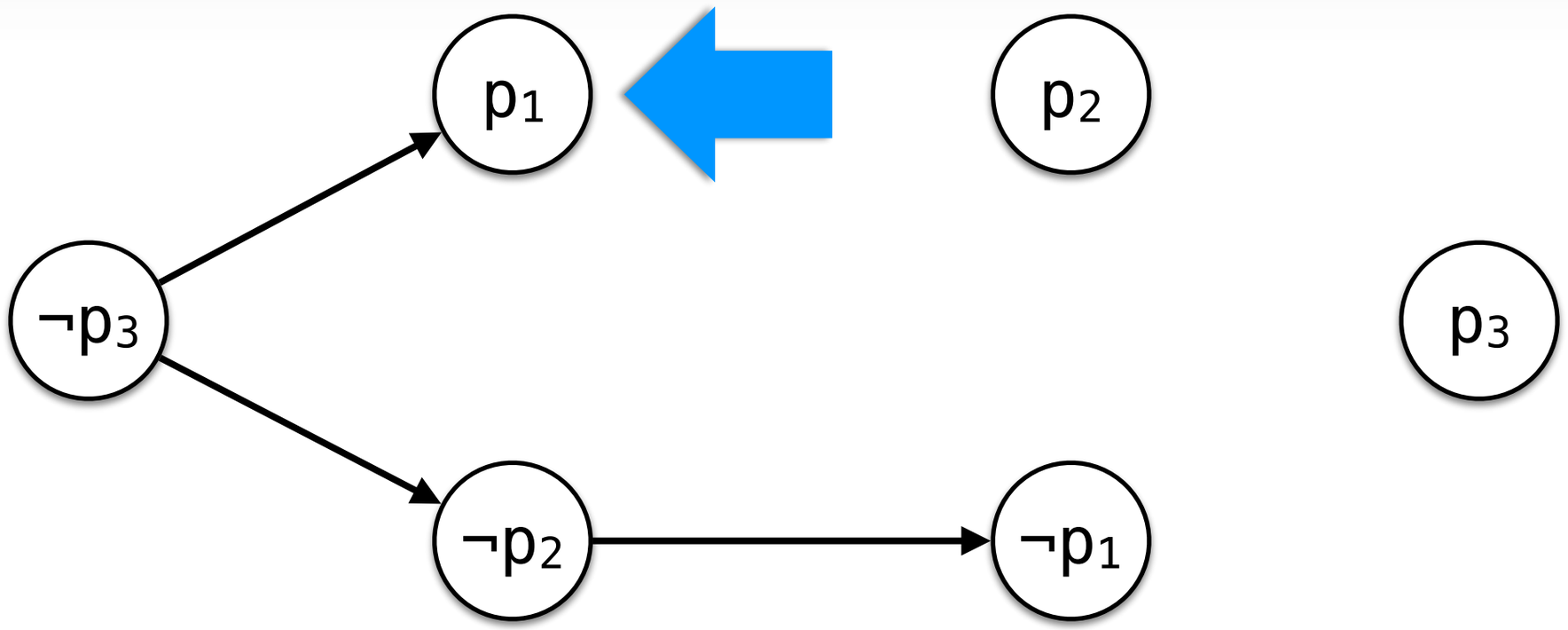
2-SAT



$p_3 \equiv T$

$p_2 \equiv T$

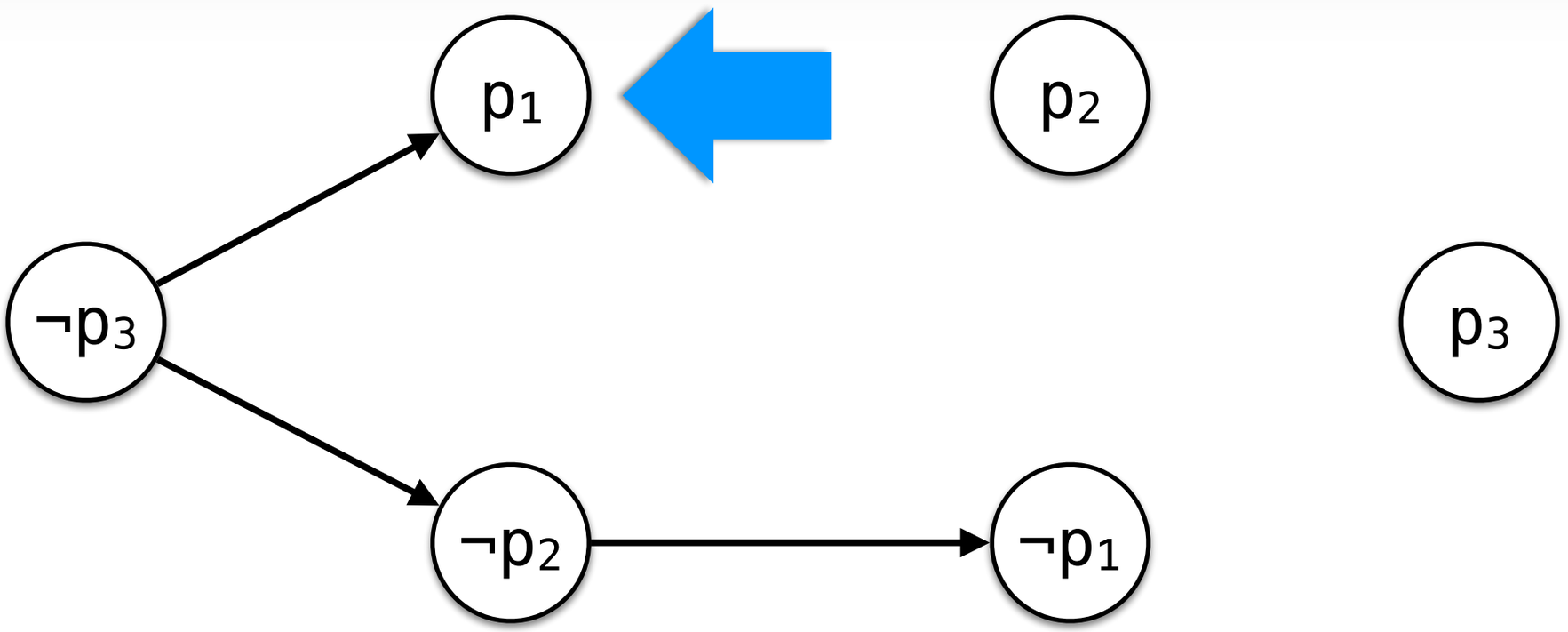
2-SAT



$p_3 \equiv T$

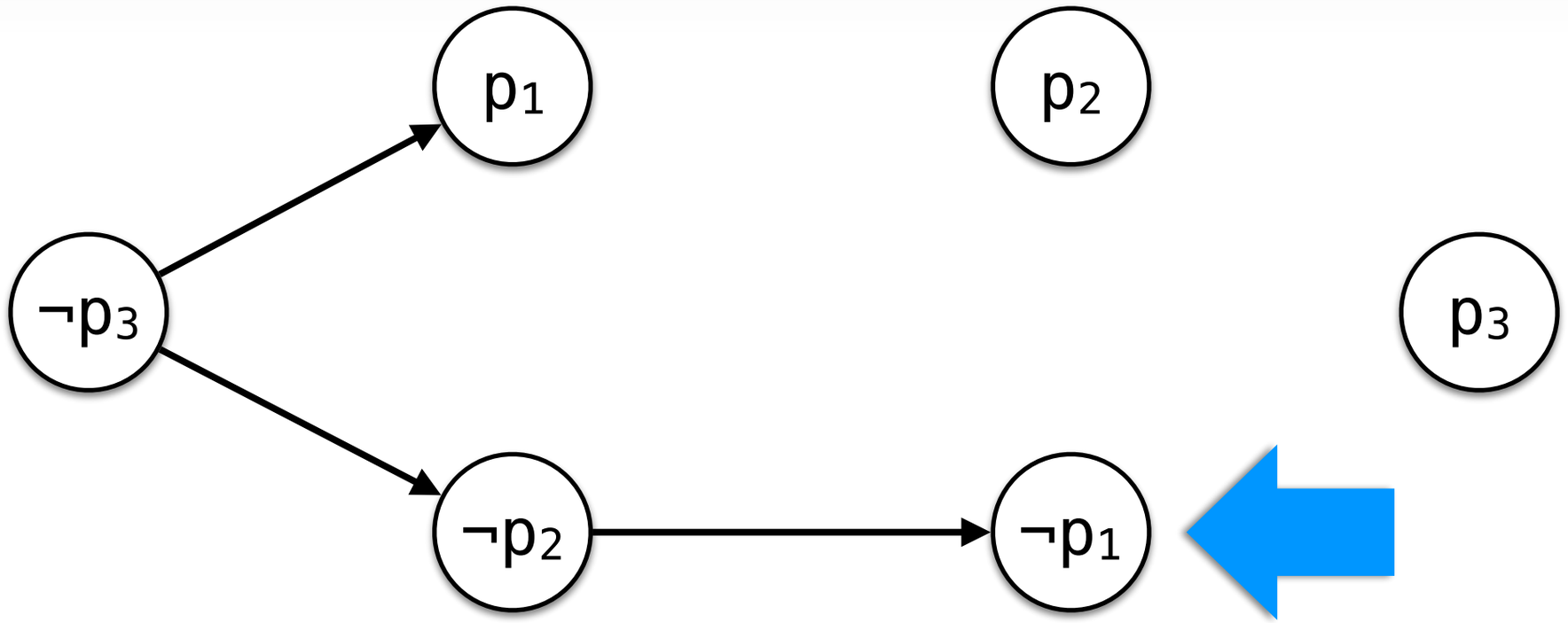
$p_2 \equiv T$

2-SAT



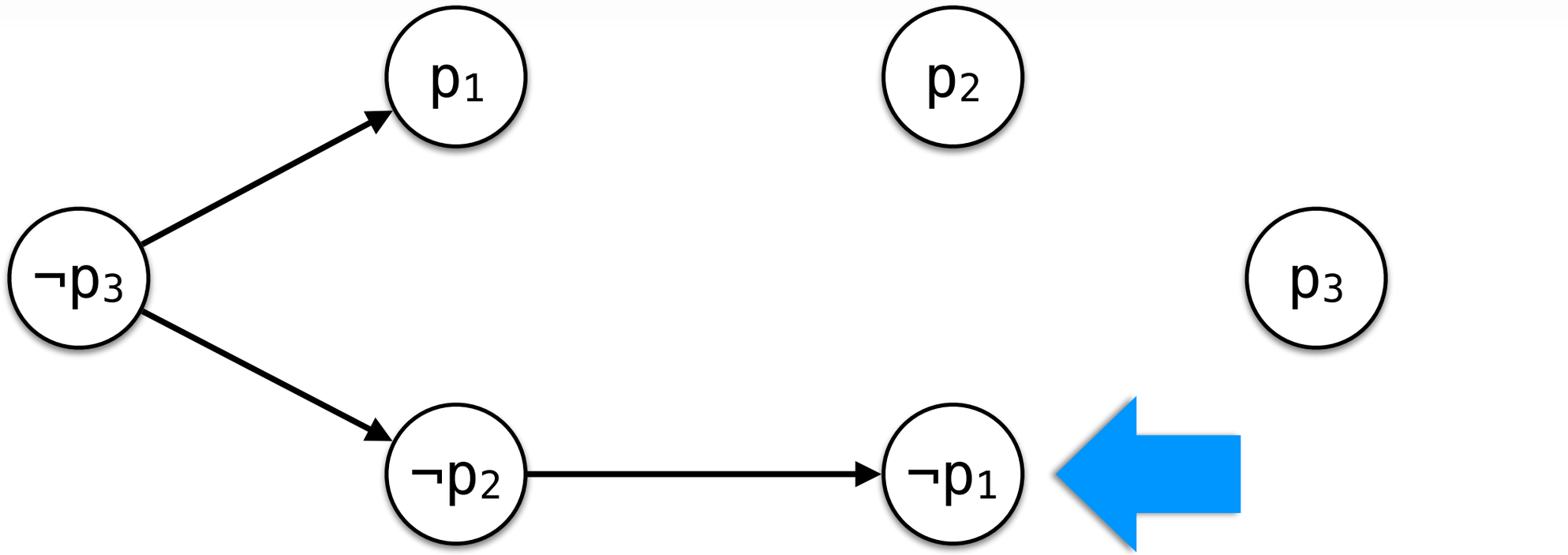
$p_3 \equiv T$
$p_2 \equiv T$
$p_1 \equiv T$

2-SAT



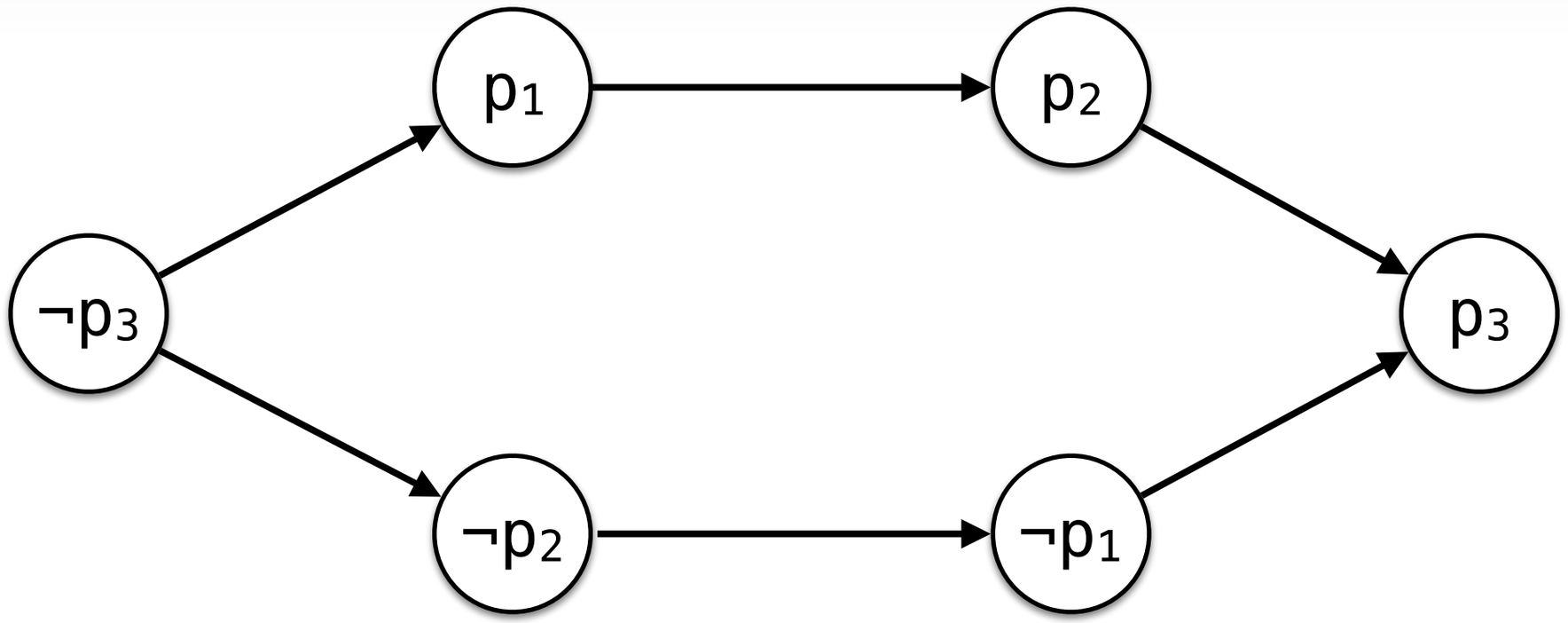
$p_3 \equiv T$
$p_2 \equiv T$
$p_1 \equiv T$

2-SAT



$p_3 \equiv T$
$p_2 \equiv T$
$p_1 \equiv T$

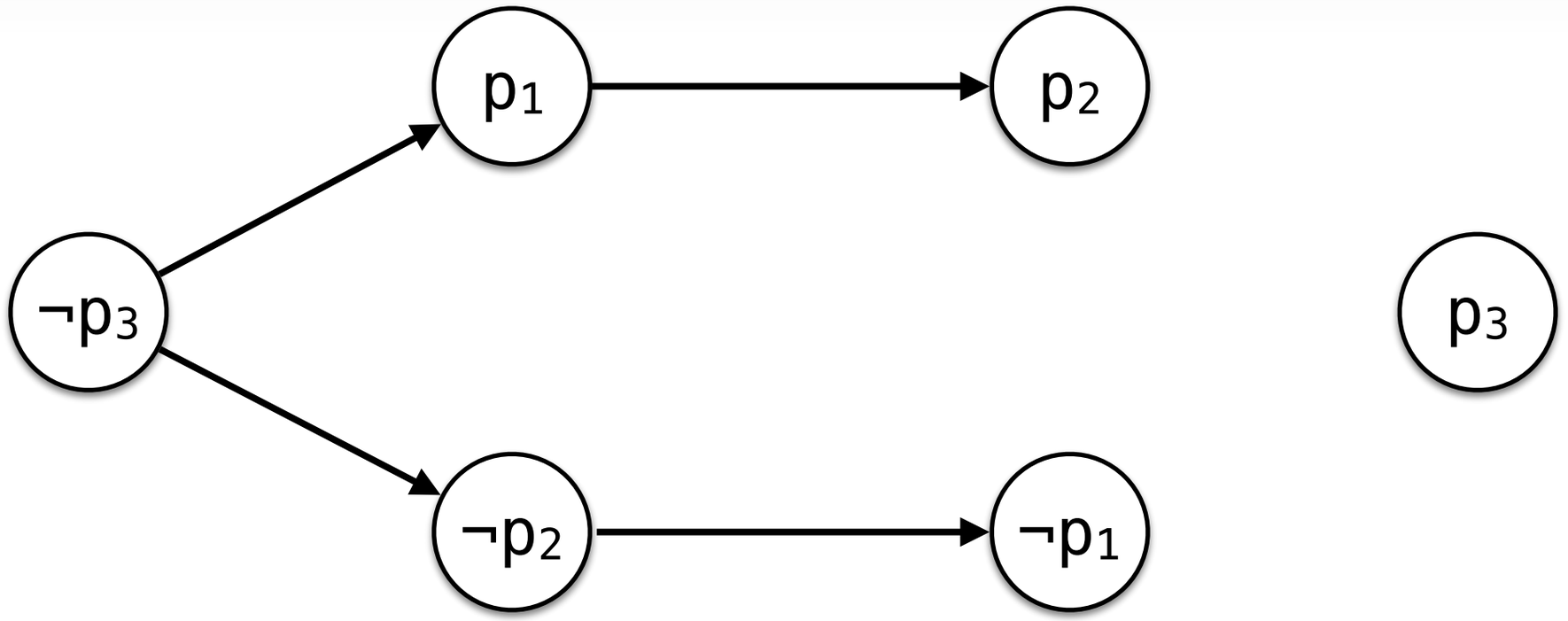
2-SAT



$p_3 \equiv T$
$p_2 \equiv T$
$p_1 \equiv T$

Any (reverse) topological order of the graph will give you a **satisfiable set**

2-SAT



$p_3 \equiv T$
$p_2 \equiv T$
$p_1 \equiv T$

Any (reverse) topological order of the graph will give you a **satisfiable set**

WHAT IF YOU DONT HAVE A DAG OR THE FORMULA IS NOT SATISFIABLE

2-SAT

$(p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_3) \wedge (\neg p_1 \vee p_3) \wedge (p_1 \vee \neg p_2)$

2-SAT

$(p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_3) \wedge (\neg p_1 \vee p_3) \wedge (p_1 \vee \neg p_2)$

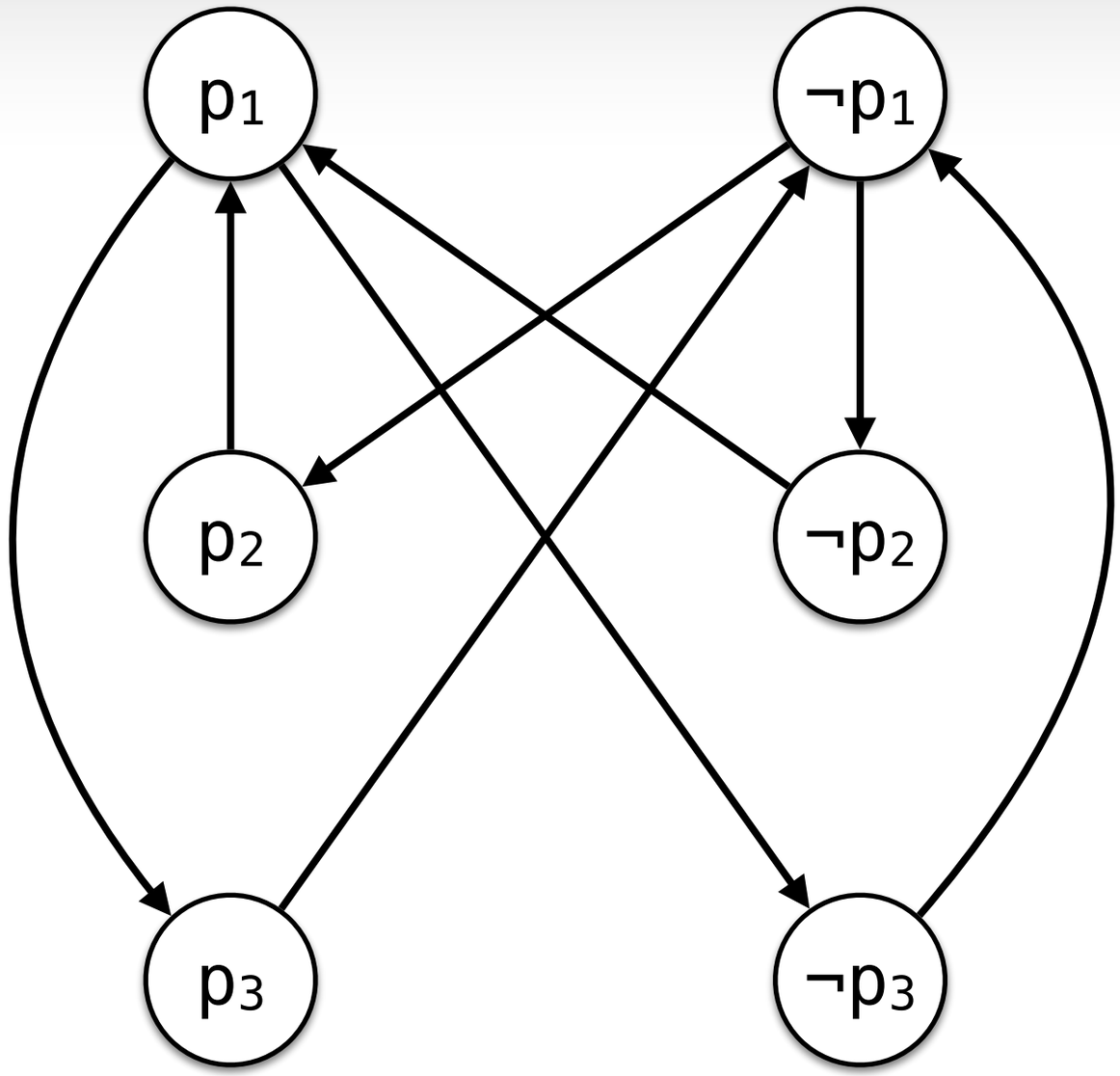
$(p_1 \vee p_2) \equiv (p_1 \Rightarrow p_2) \equiv (\neg p_2 \Rightarrow \neg p_1)$

$(\neg p_1 \vee \neg p_3) \equiv (p_1 \Rightarrow \neg p_3) \equiv (p_3 \Rightarrow \neg p_1)$

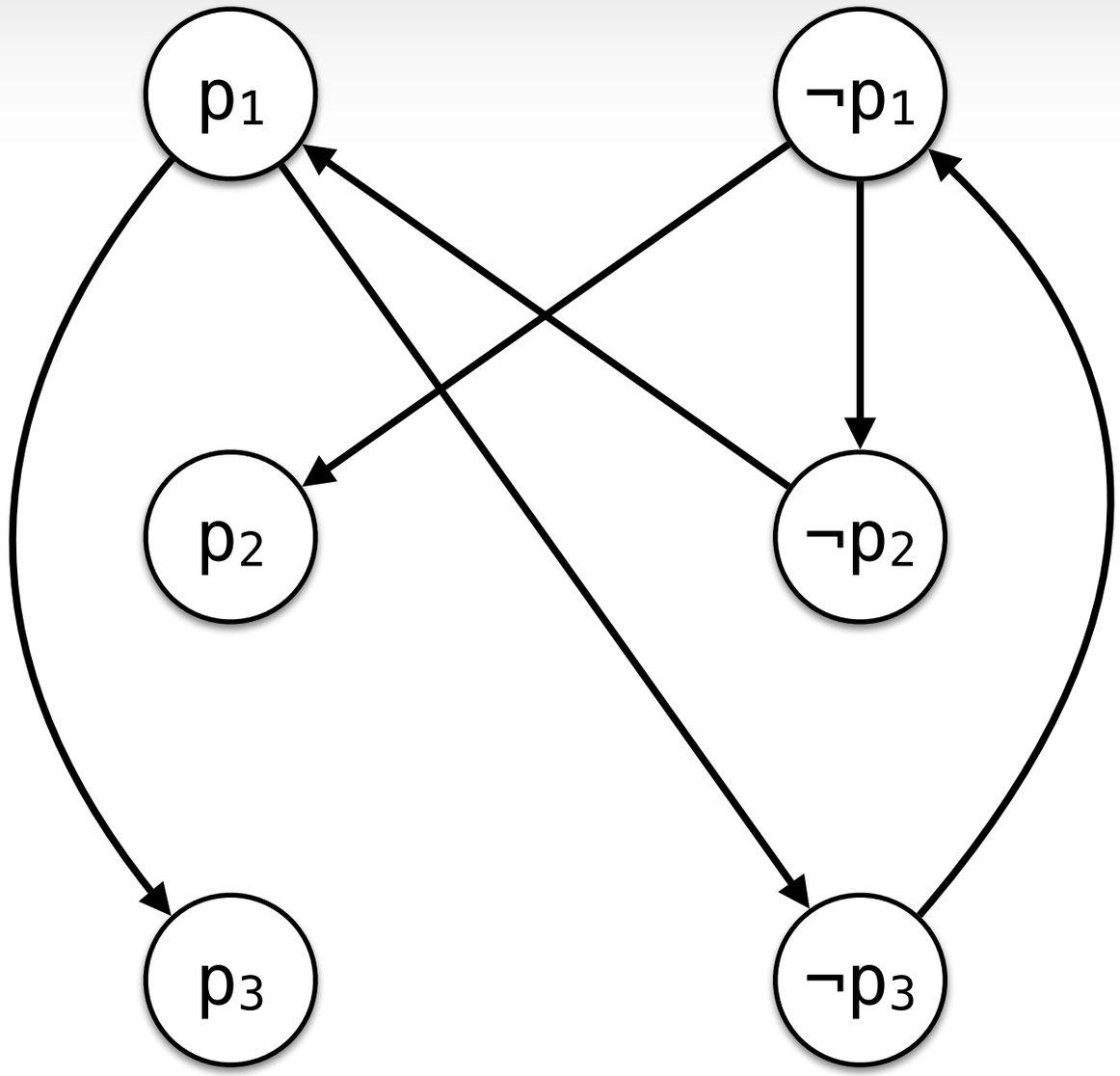
$(\neg p_1 \vee p_3) \equiv (p_1 \Rightarrow p_3) \equiv (\neg p_3 \Rightarrow \neg p_1)$

$(p_1 \vee \neg p_2) \equiv (p_2 \Rightarrow p_1) \equiv (\neg p_1 \Rightarrow \neg p_2)$

2-SAT

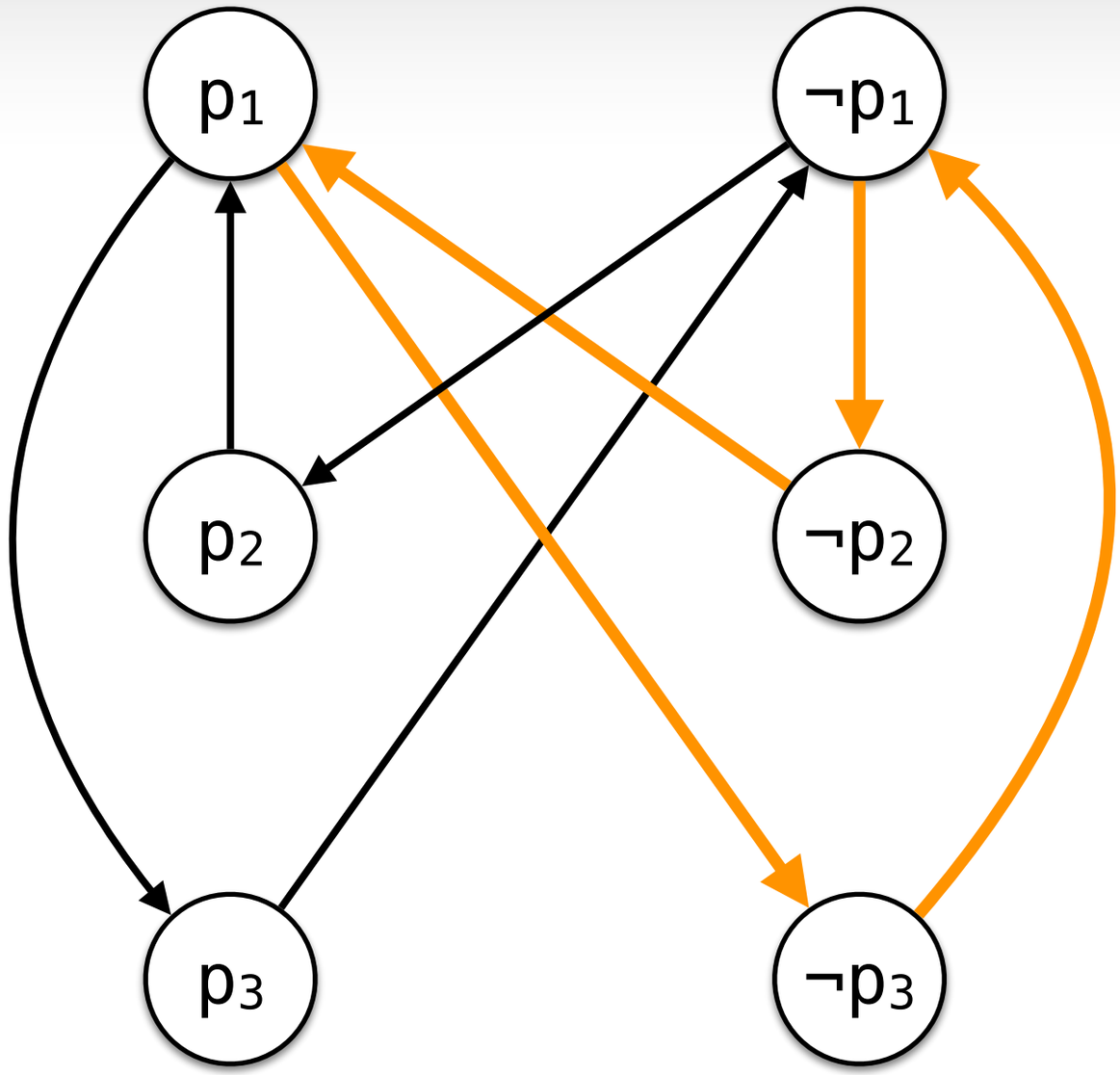


2-SAT



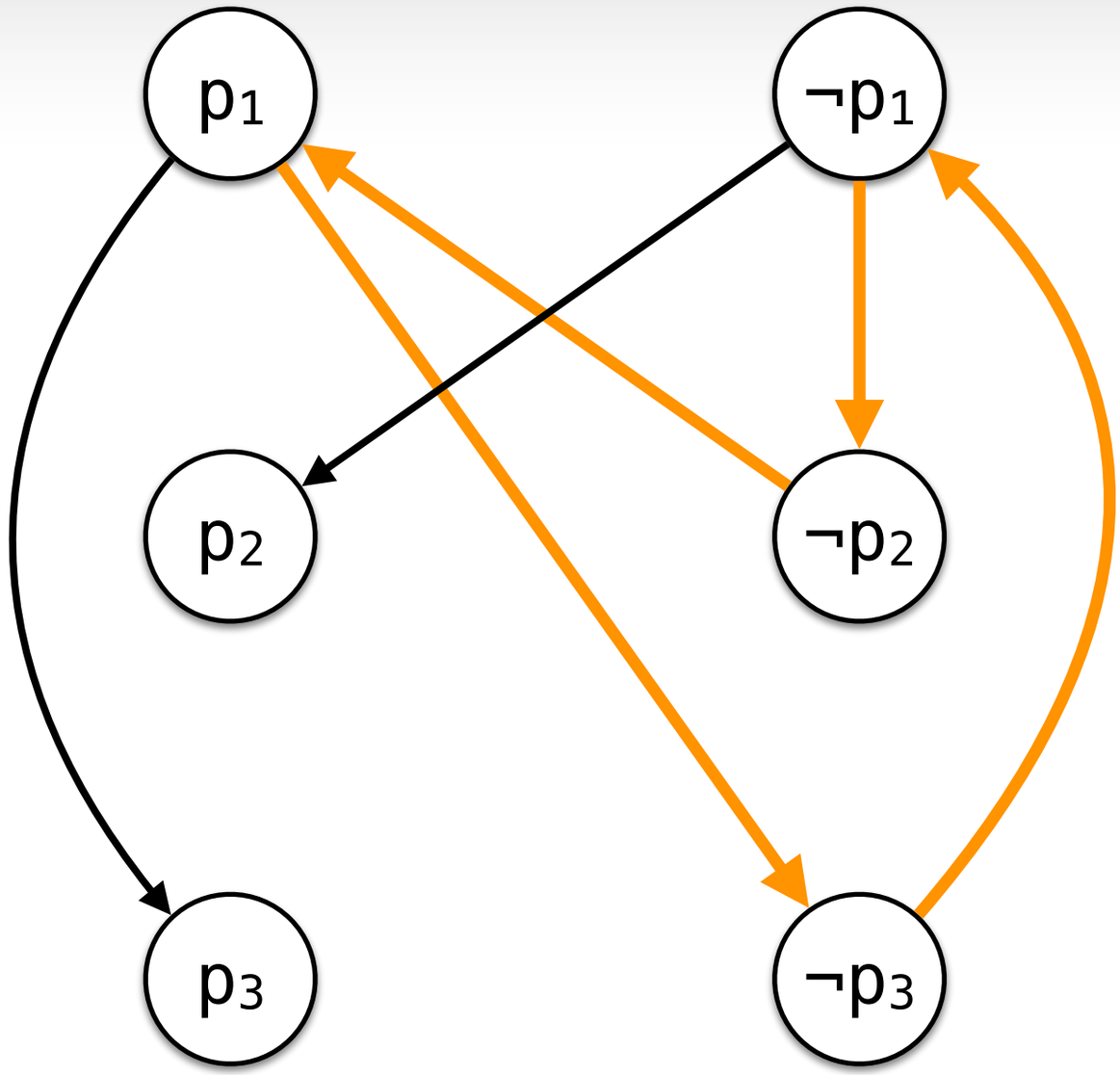
2-SAT

cycle

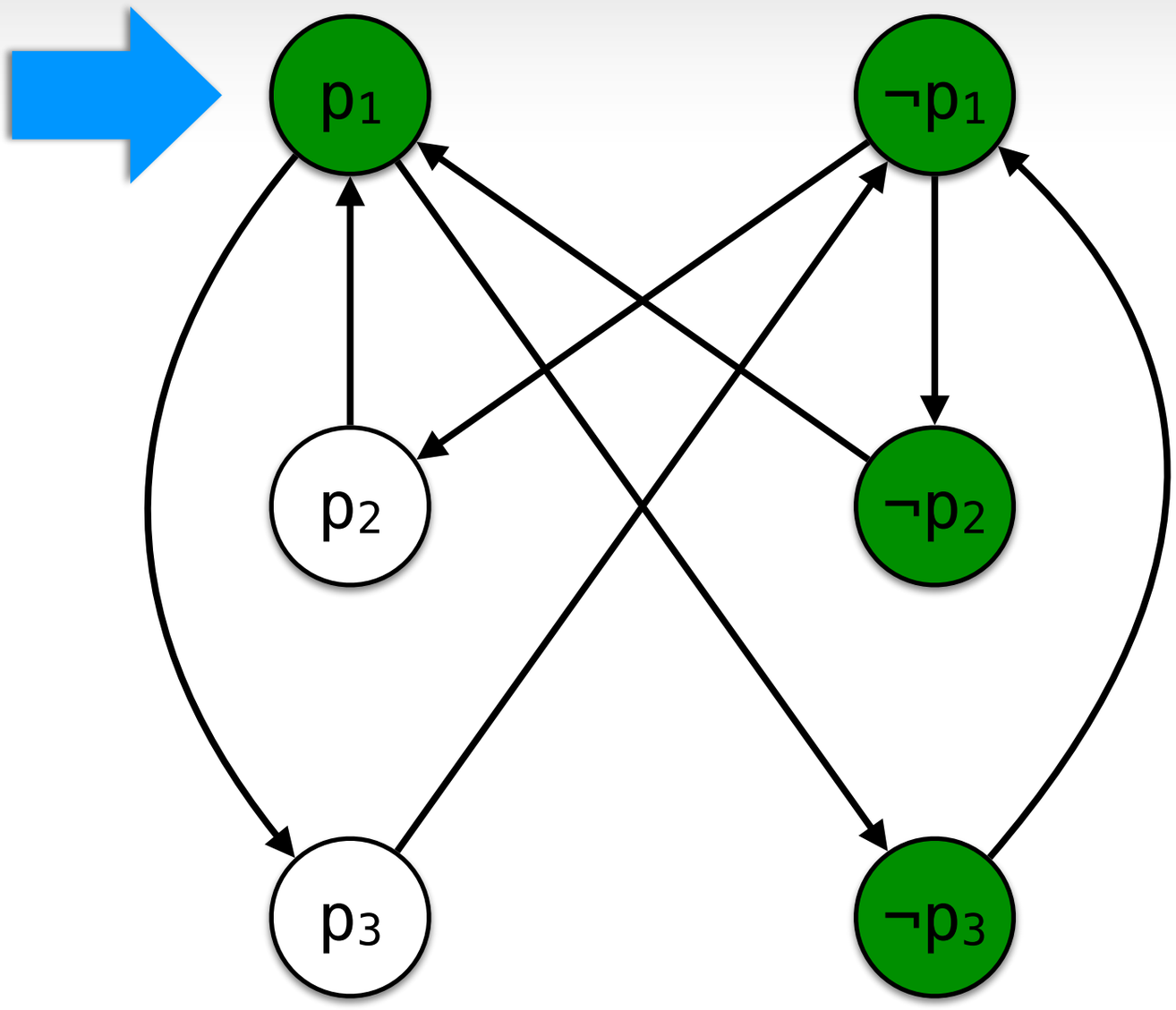


2-SAT

cycle

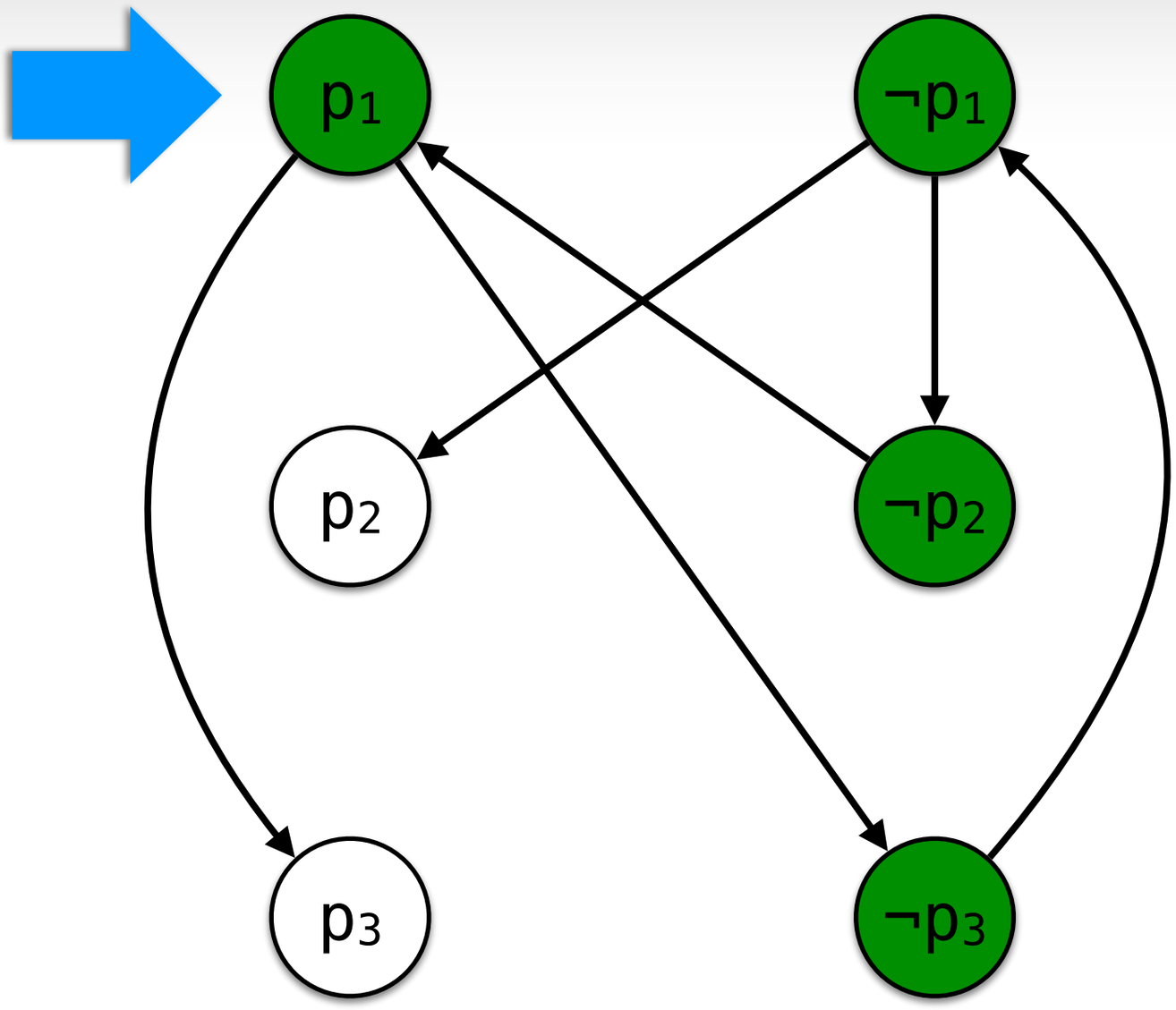


2-SAT



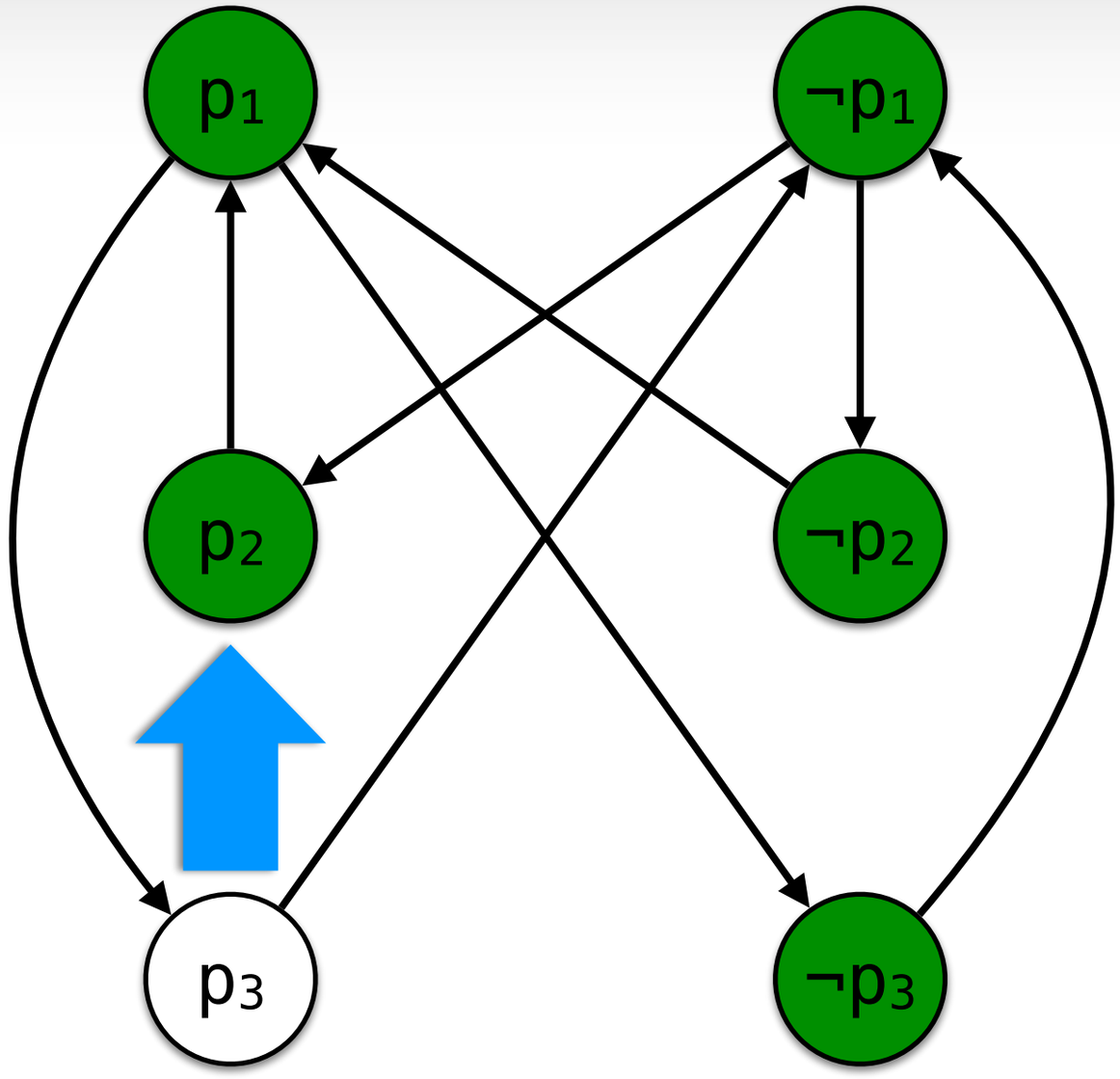
can't take p_1

2-SAT



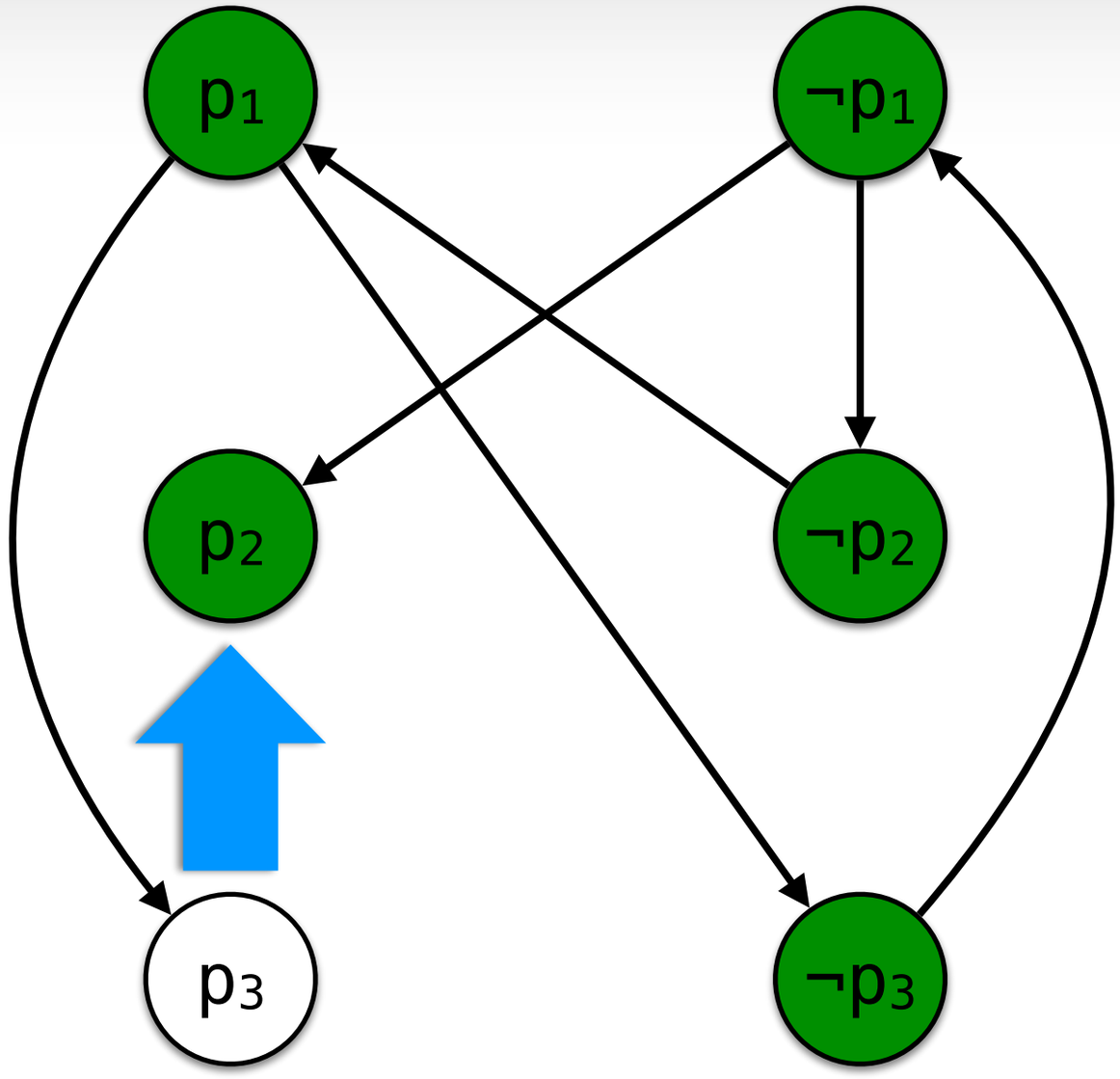
can't take p_1

2-SAT



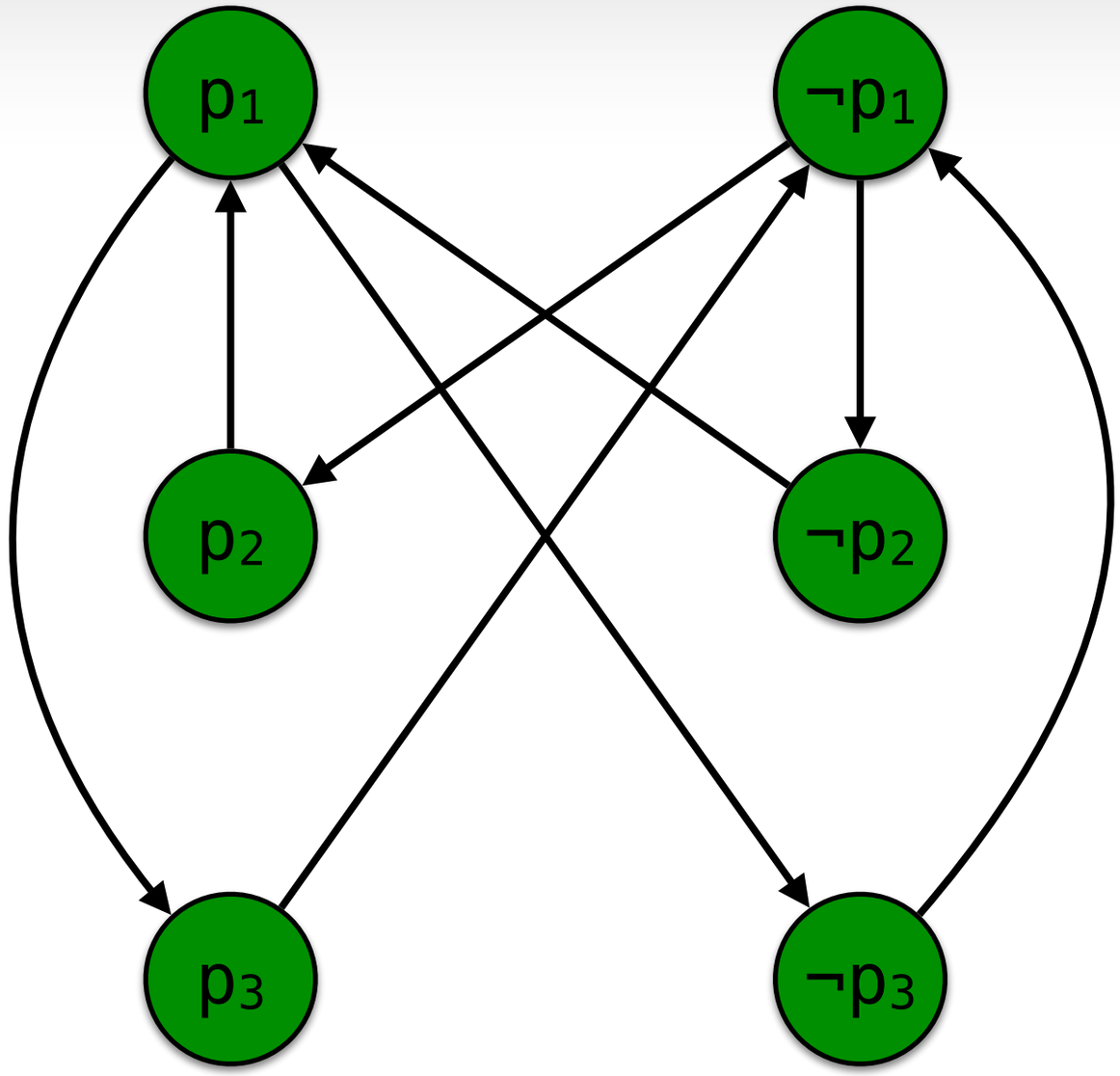
can't take p_2

2-SAT



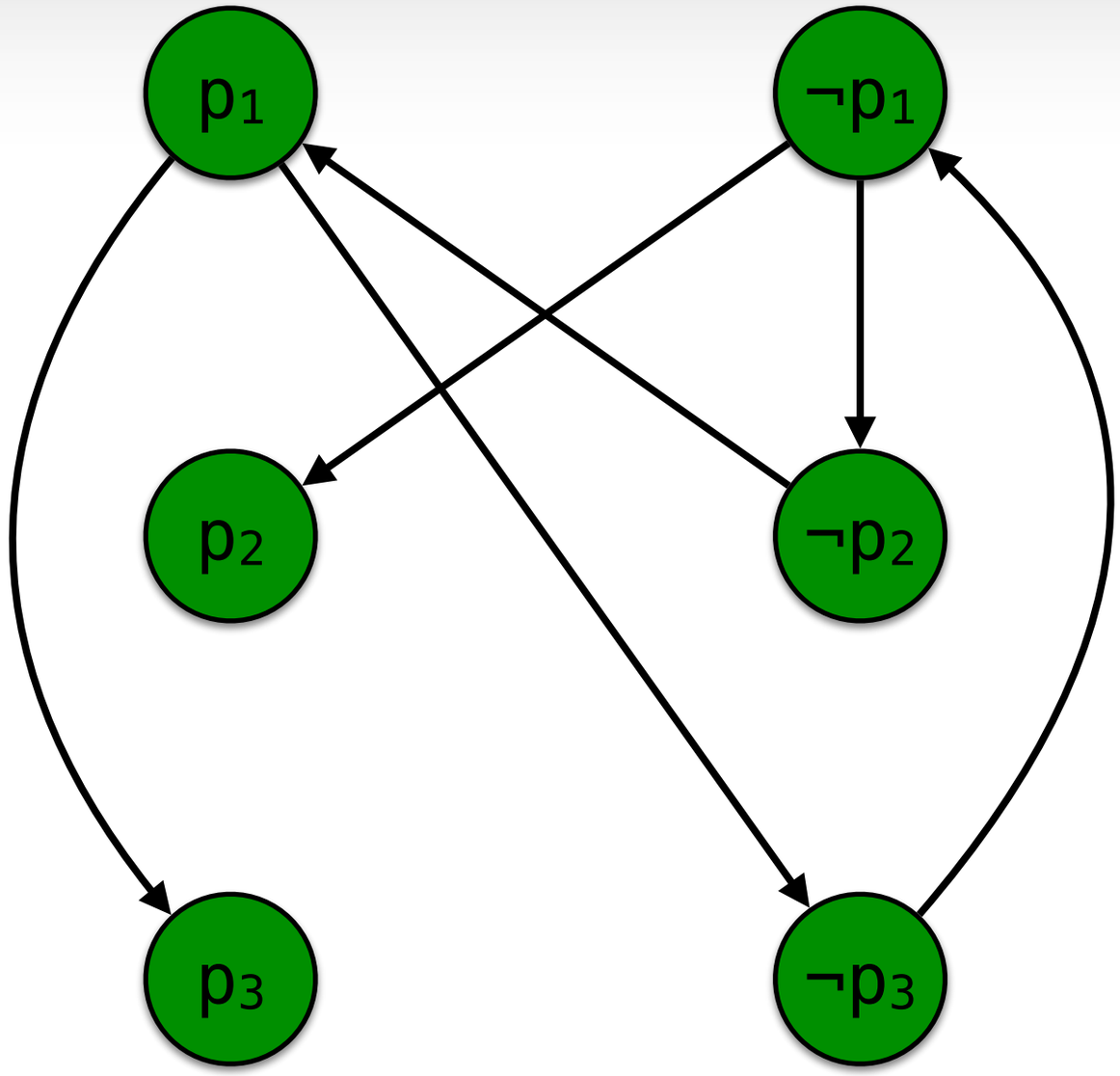
can’t take p_2

2-SAT



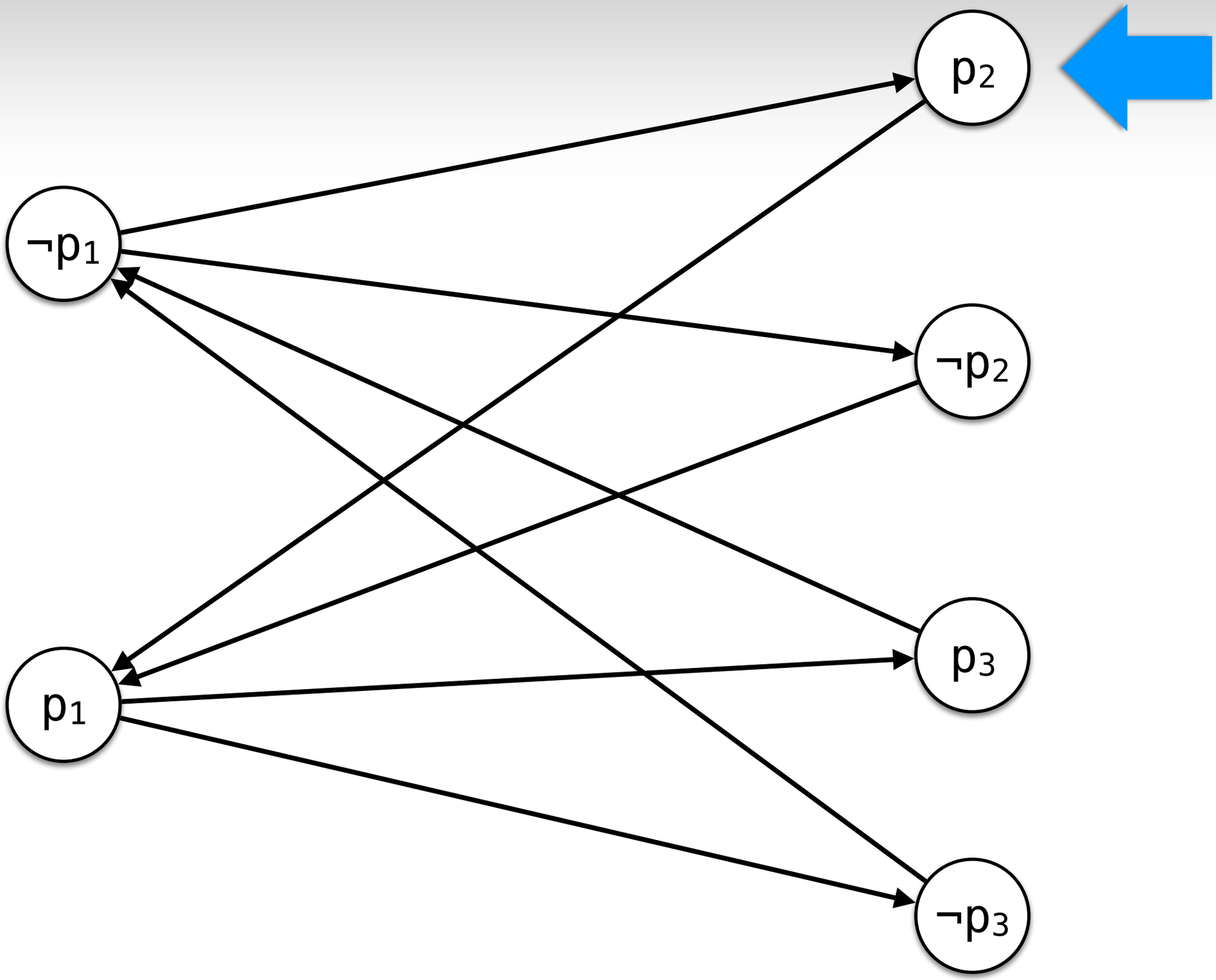
can't take p_3

2-SAT

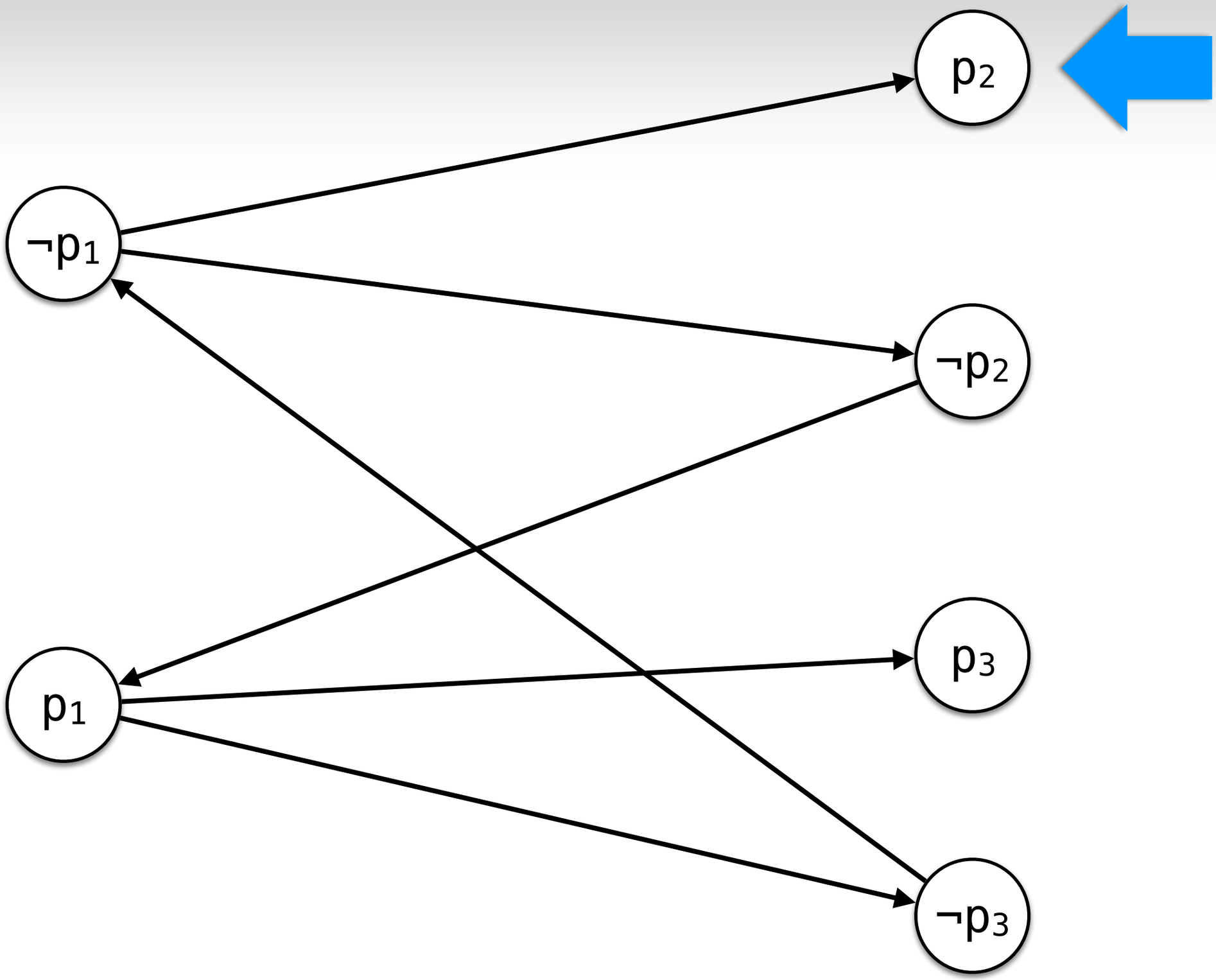


can't take p_3

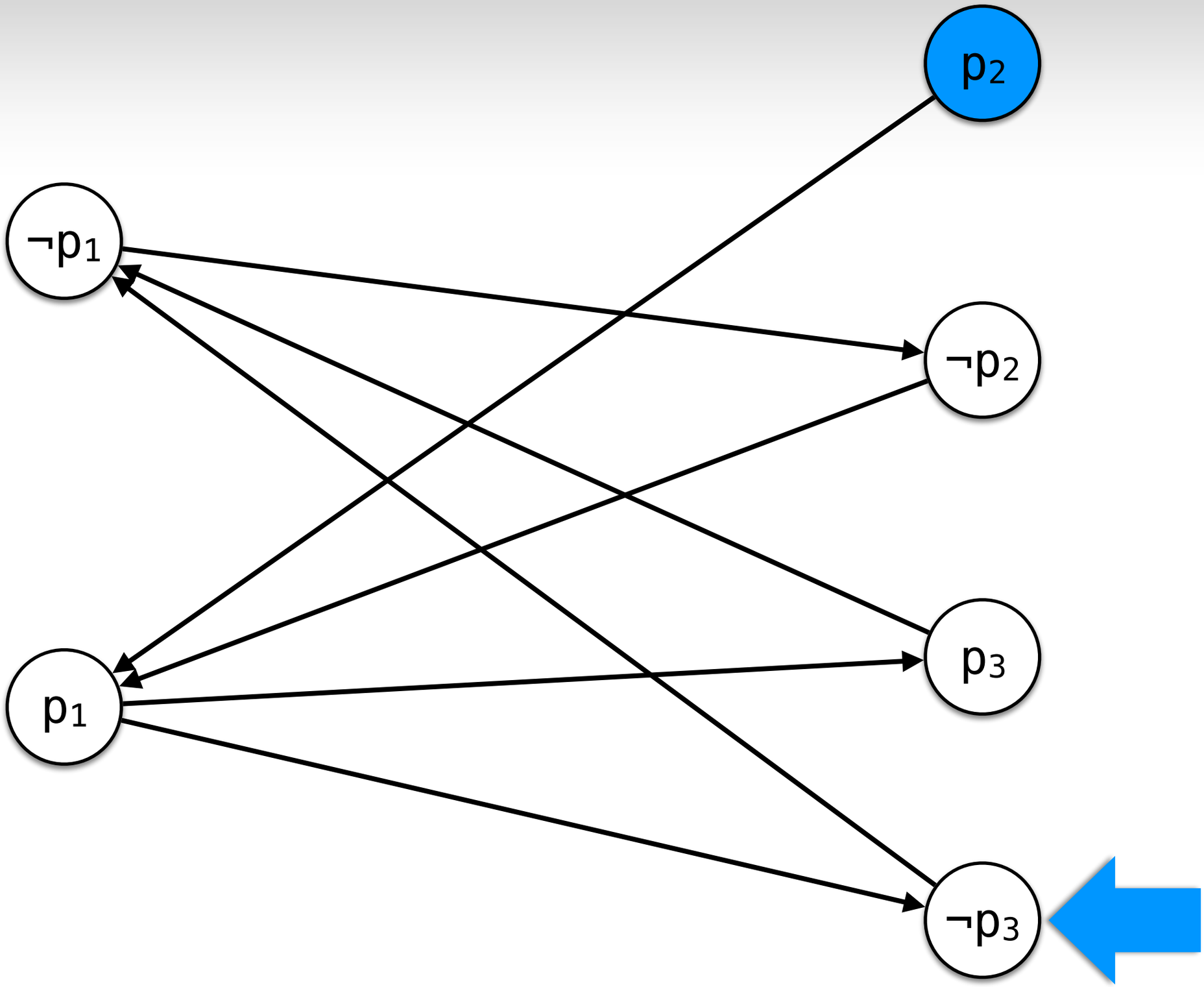
2-SAT



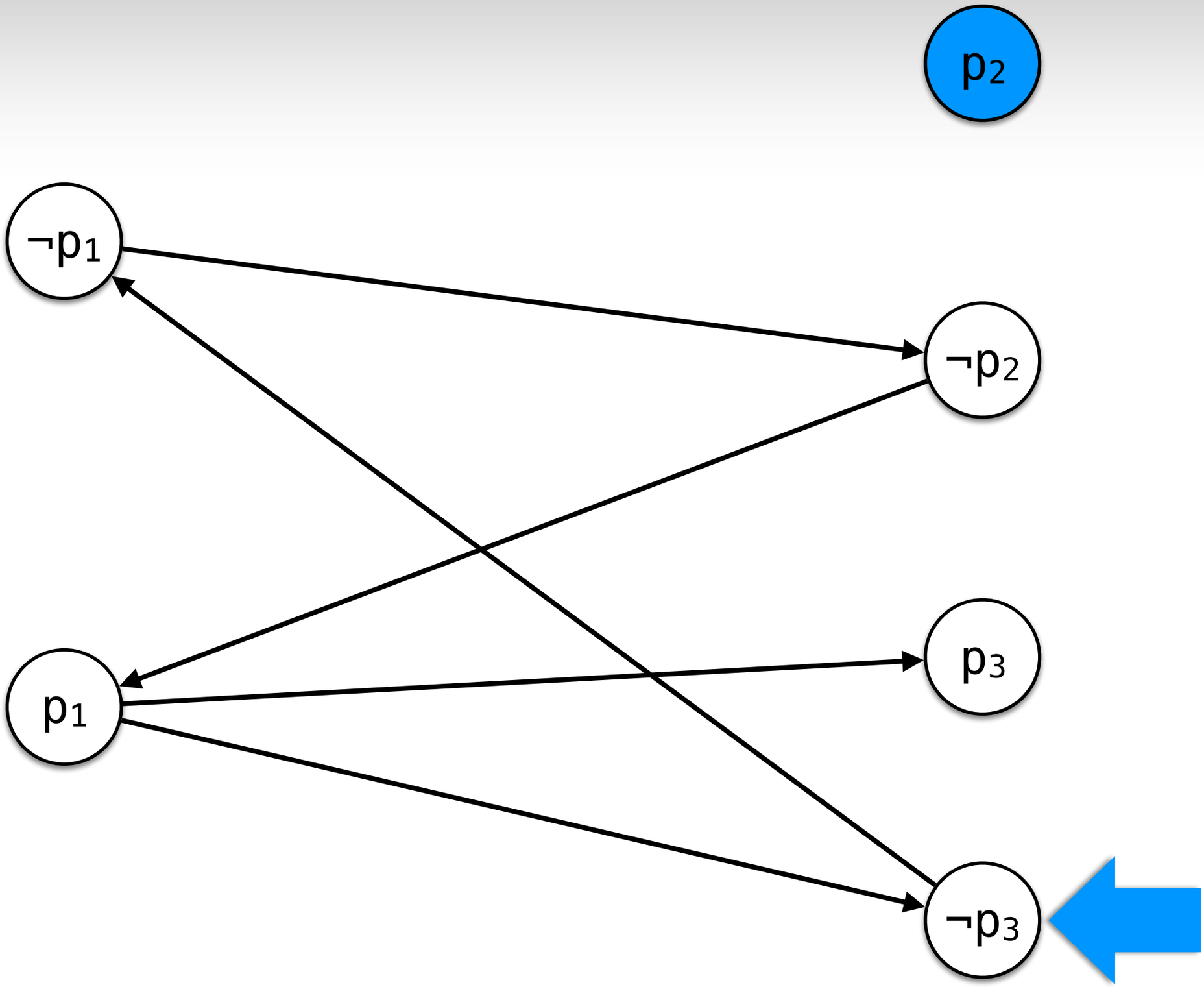
2-SAT



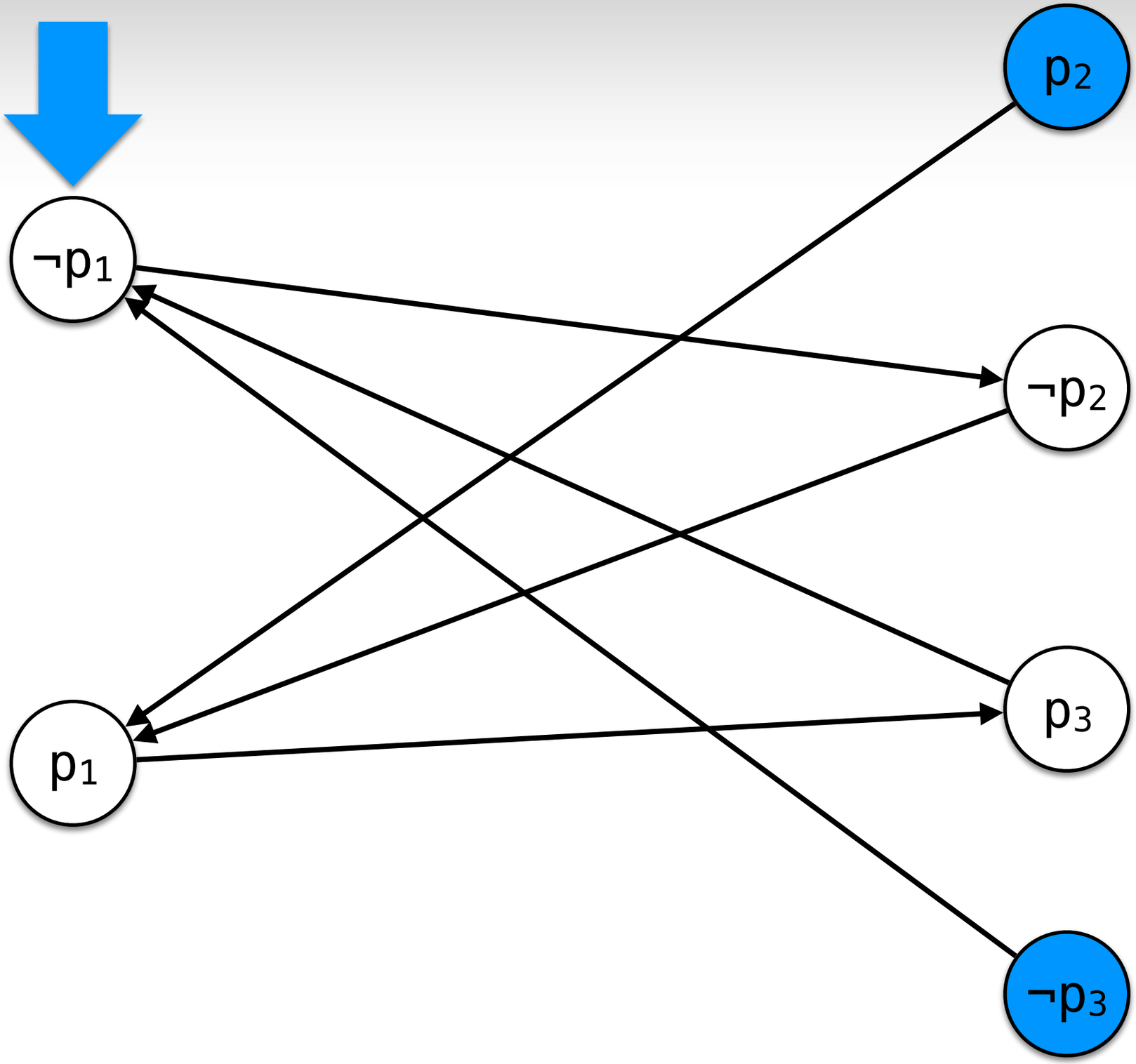
2-SAT



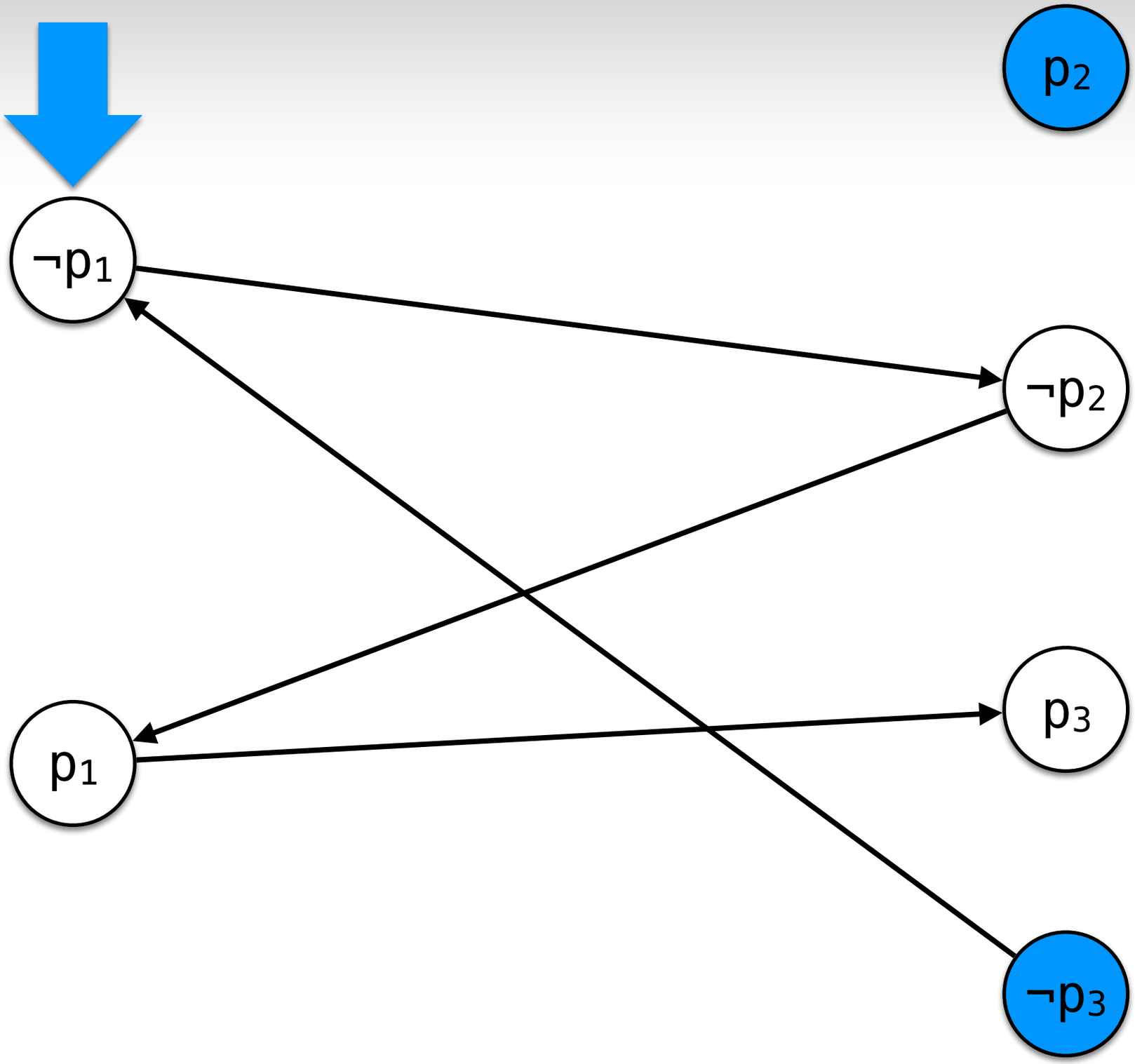
2-SAT



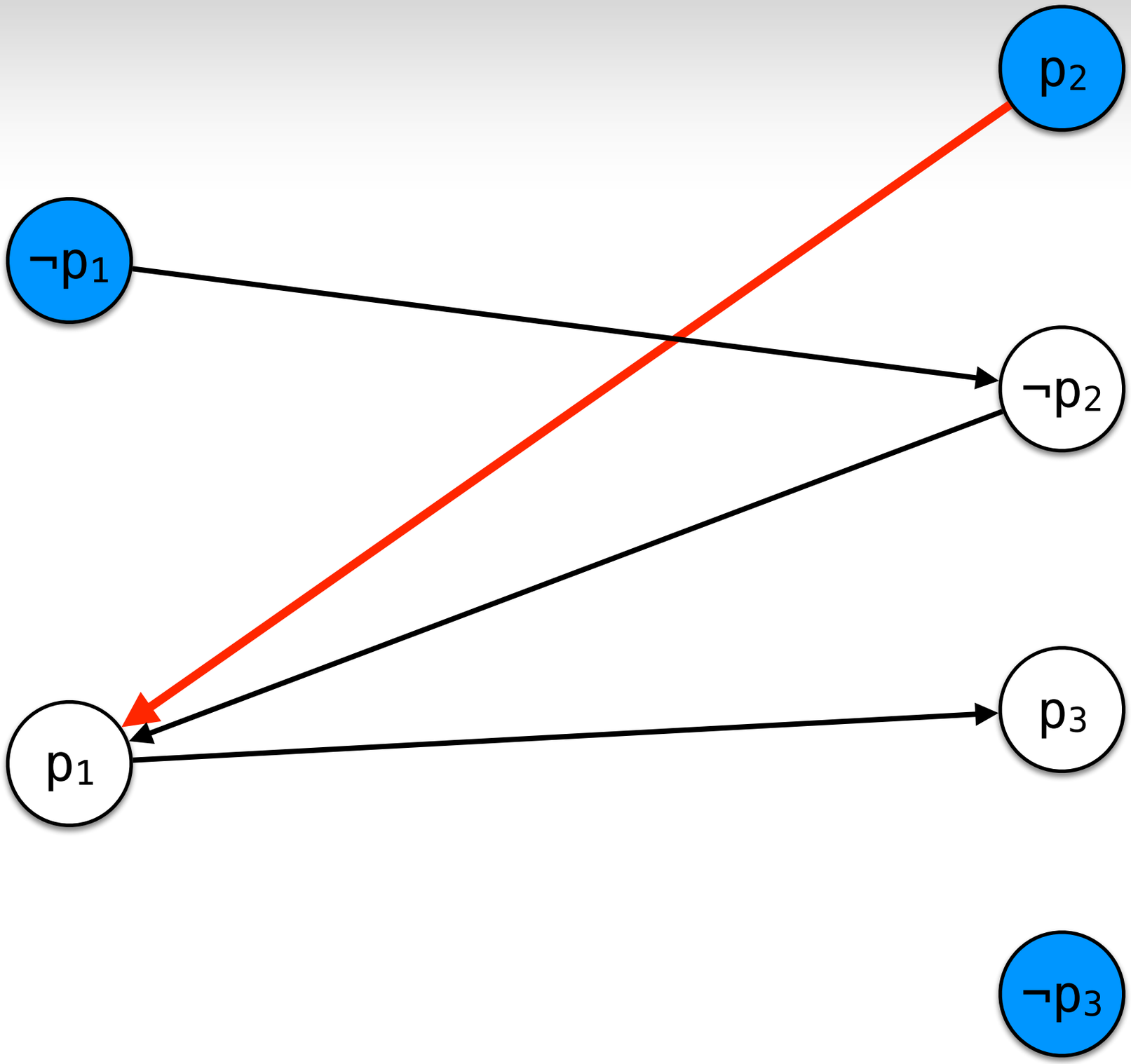
2-SAT



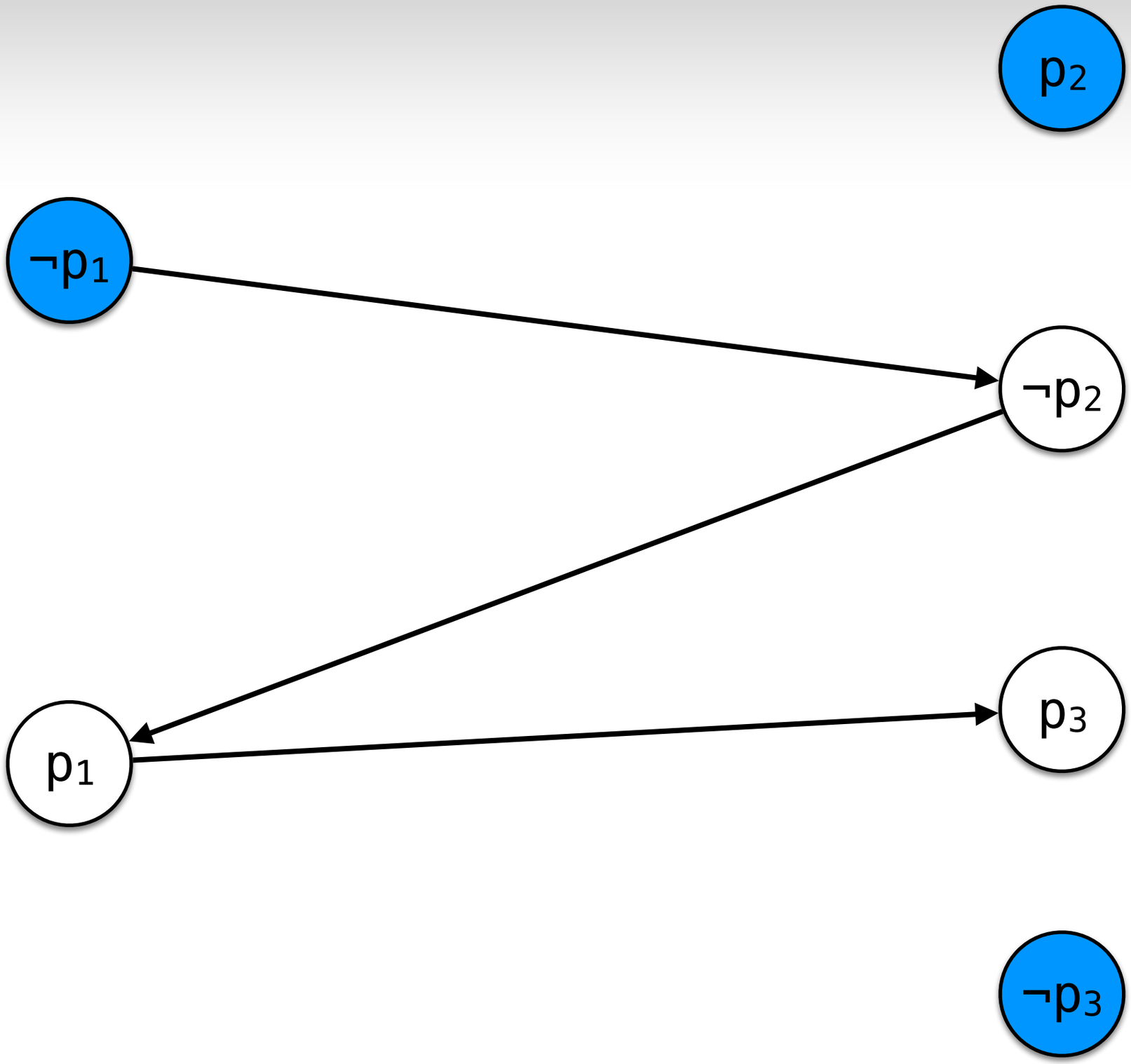
2-SAT



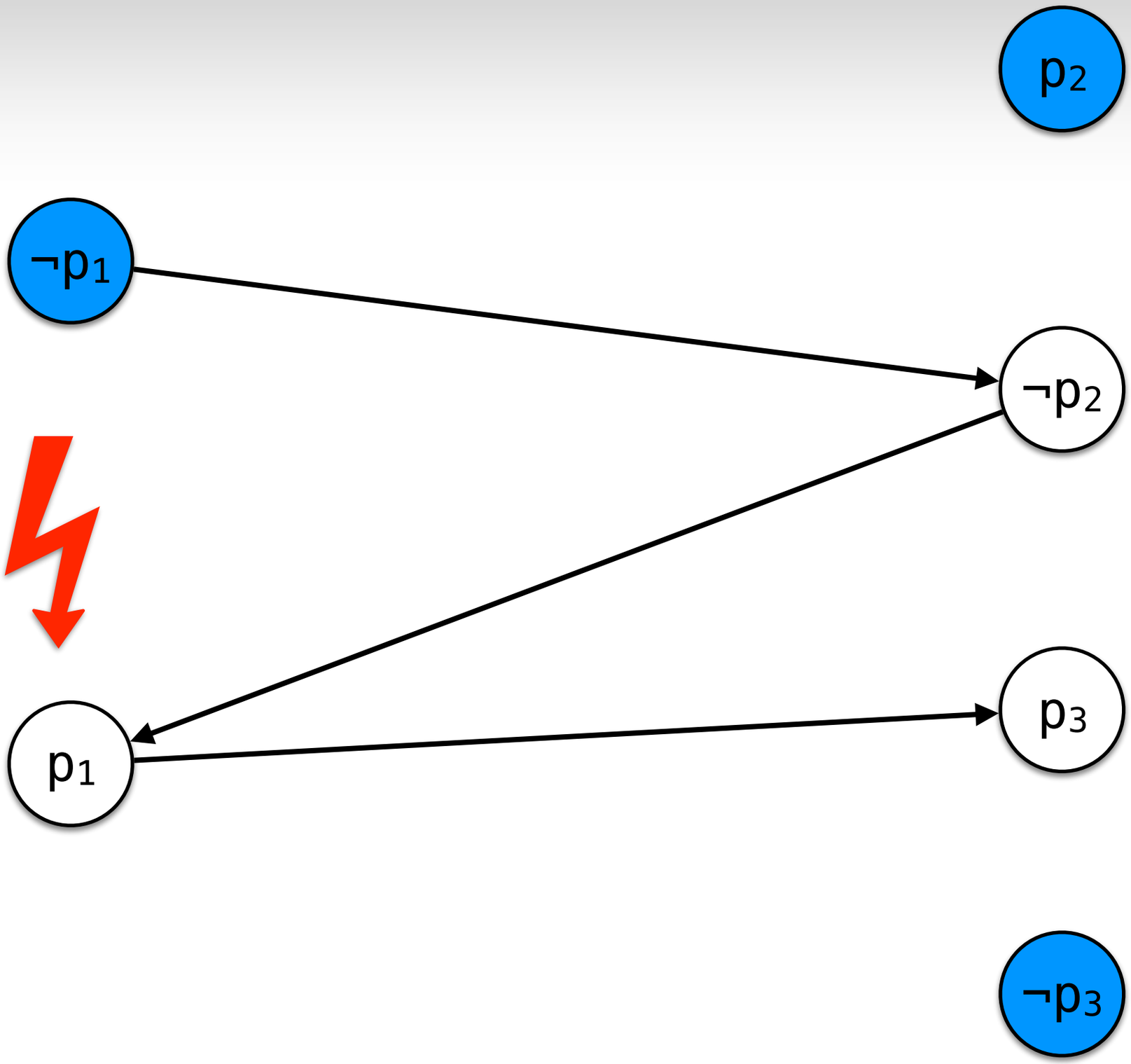
2-SAT



2-SAT



2-SAT

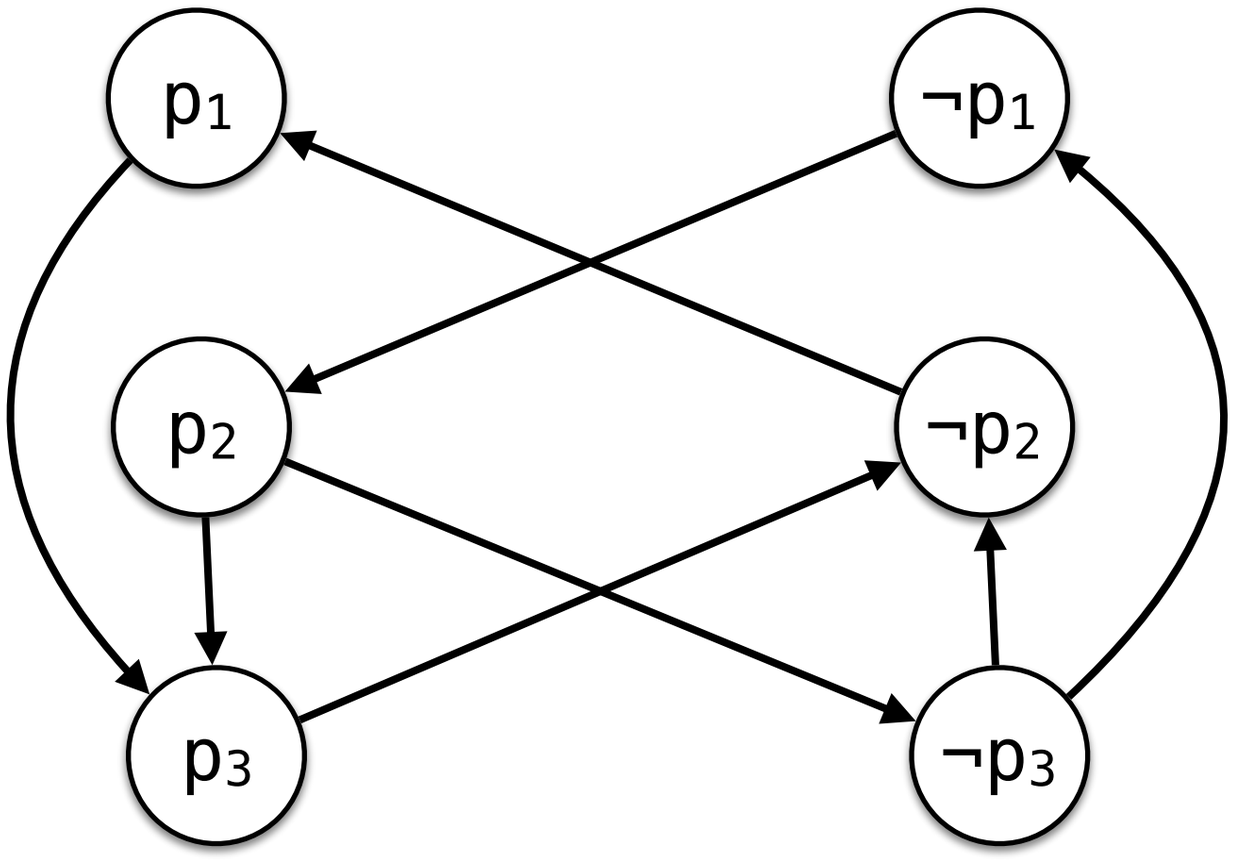


CALCULATE THE STRONG CONNECTED COMPONENTS

2-SAT

$(p_1 \vee p_2) \wedge (\neg p_3 \vee \neg p_2) \wedge (\neg p_2 \vee p_3) \wedge (\neg p_1 \vee p_3)$

1. **Build the implication graph**



2-SAT

1. Build the implication graph

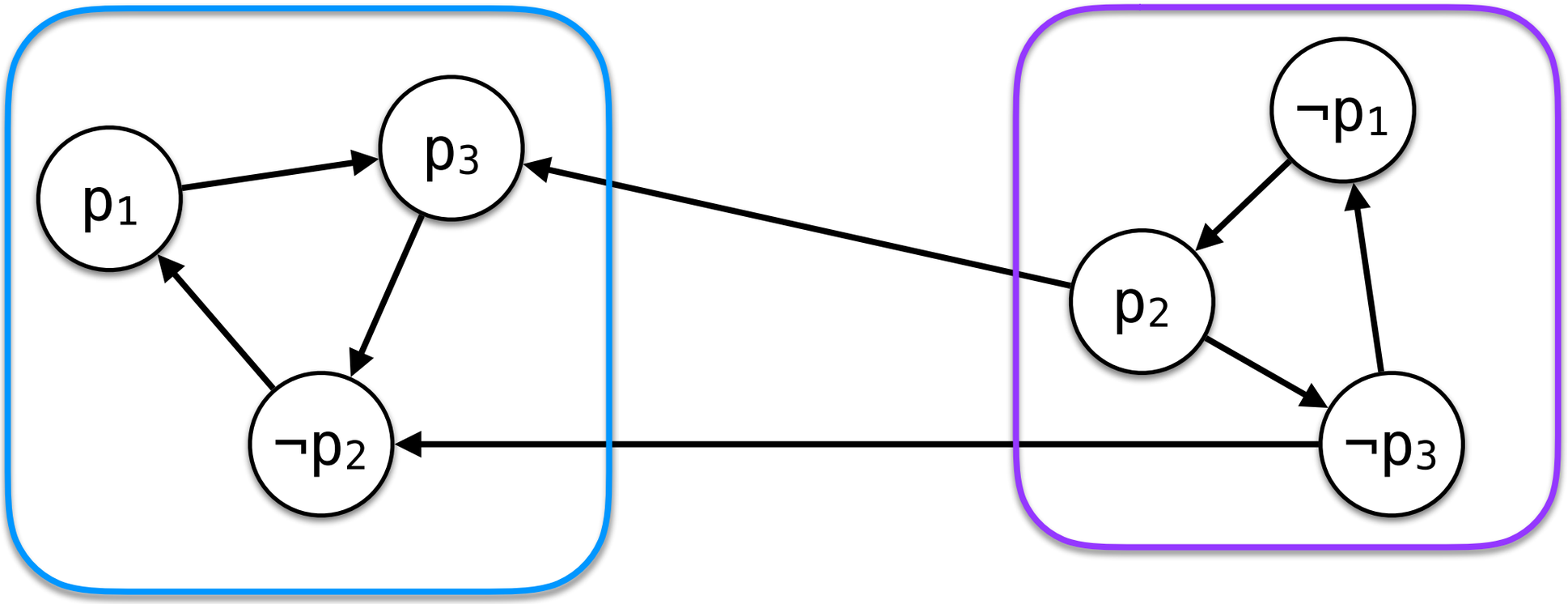
```
vector<vector<int>> g, gt;

g[u].push(v);
gt[v].push(u);
```

transpose of the graph,
we will use it later

0	1	2	3	4	5
p_1	$\neg p_1$	p_2	$\neg p_2$	p_3	$\neg p_3$

2-SAT

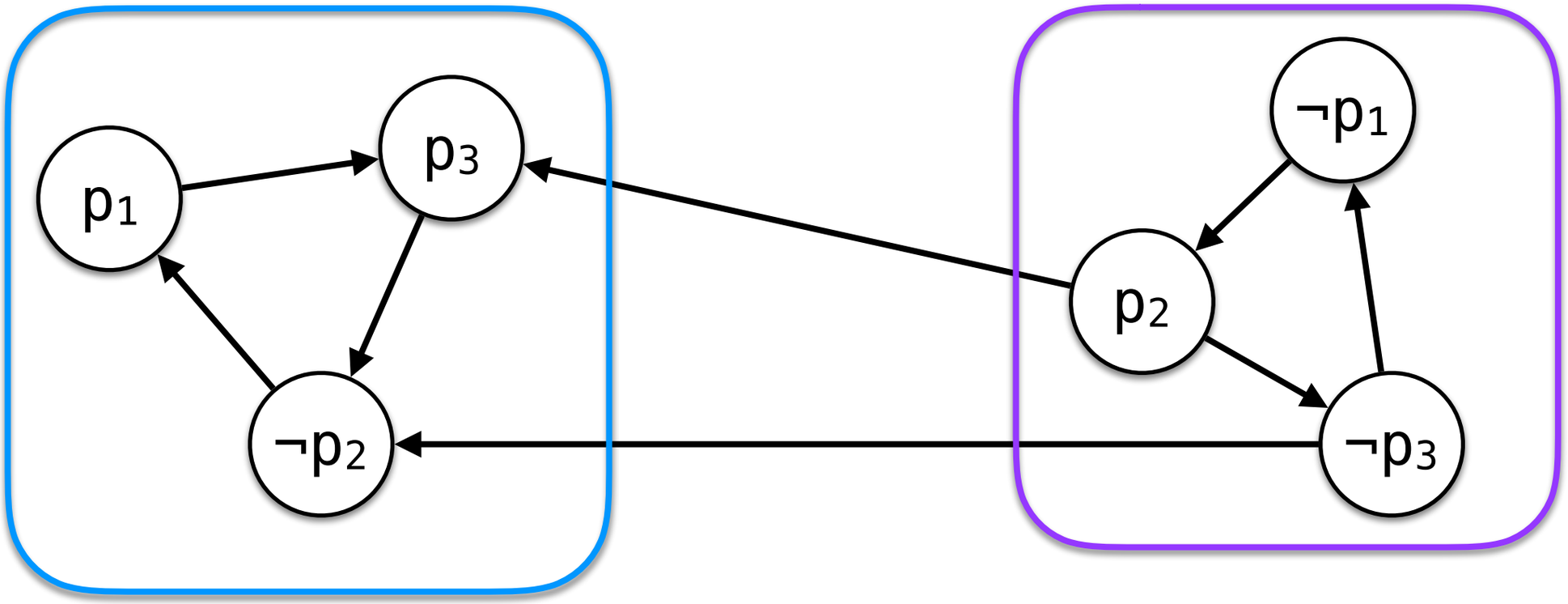
2. Run SCC



2-SAT

linear time algorithm
for scc

2. Run SCC



2-SAT

2. Run SCC

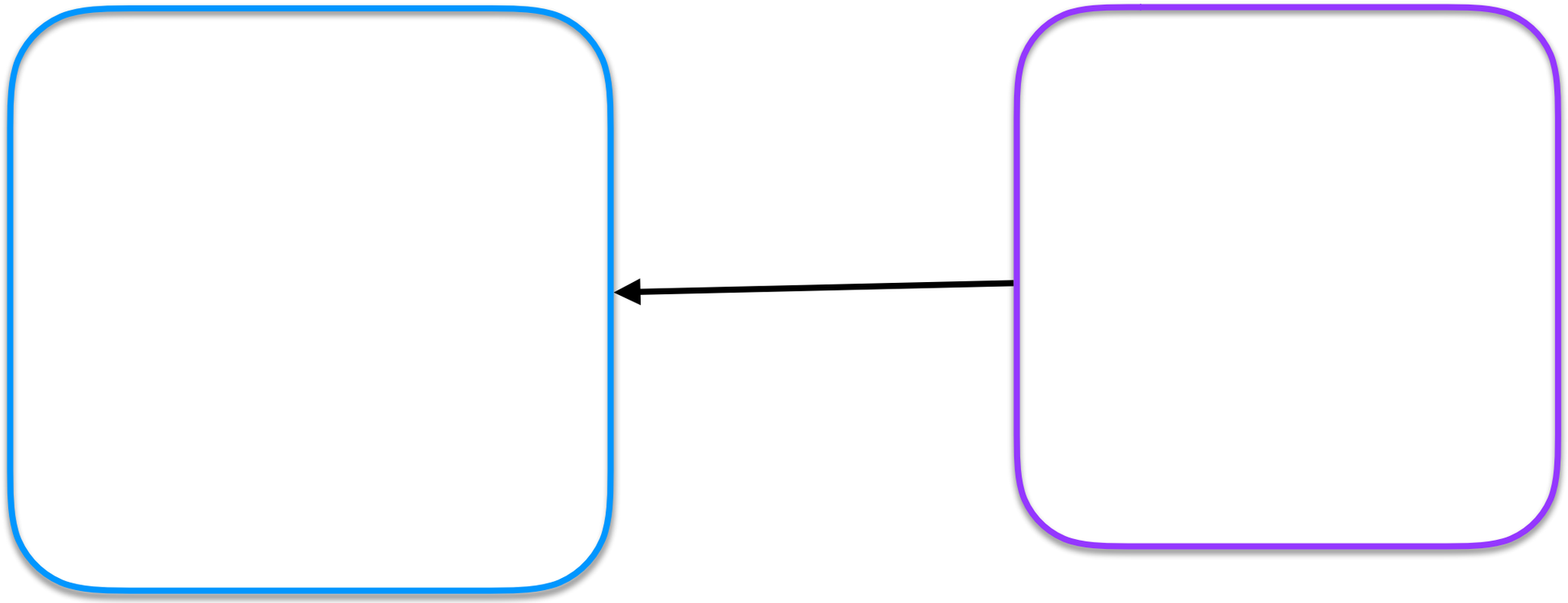
```
vector<bool> used;
vector<int> order, comp;

void topo_sort(int v) {
    used[v] = true;
    for(int u : g[v]) {
        if (!used[u])
            topo_sort(u);
    }
    order.push_back(v);
}
```

DFS

```
void trans_sort(int v, int cl){
    comp[v] = cl;
    for(int u : gt[v]) {
        if(comp[u] == -1)
            trans_sort(u, cl);
    }
}
```

2-SAT

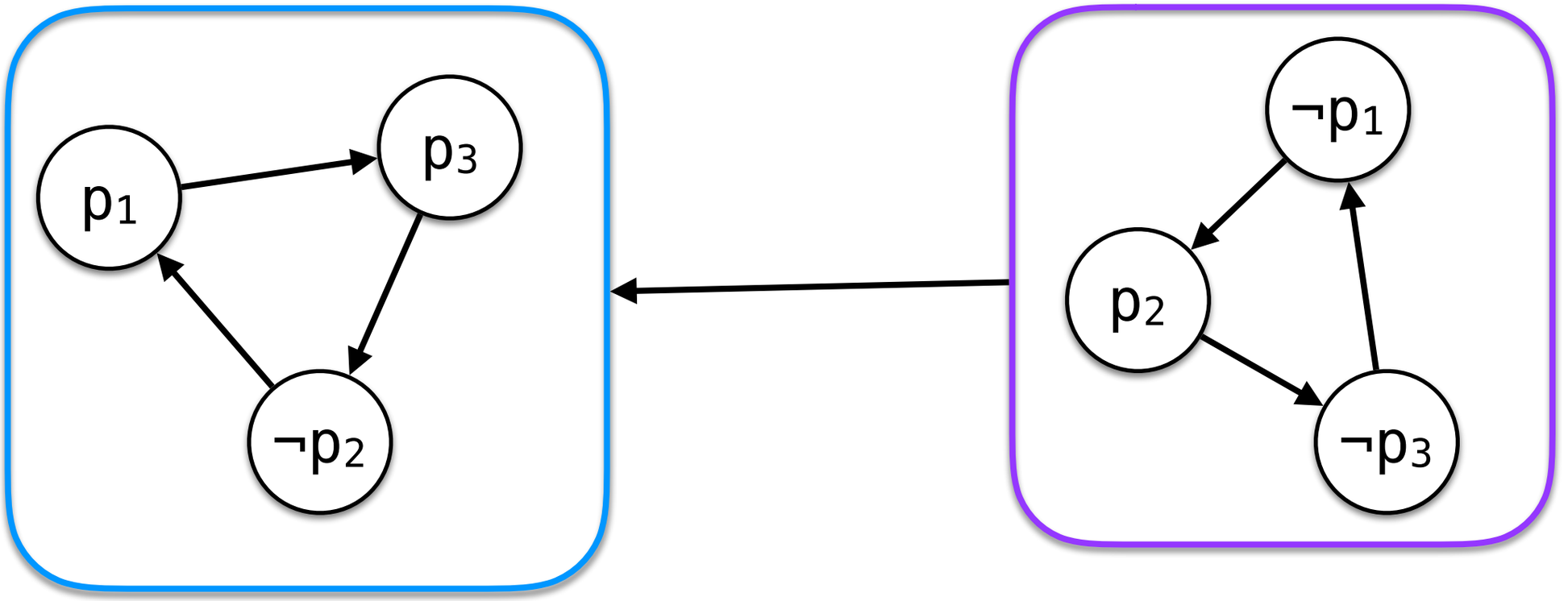
3. Construct the condensation of the implication graph



2-SAT

side effect of the scc

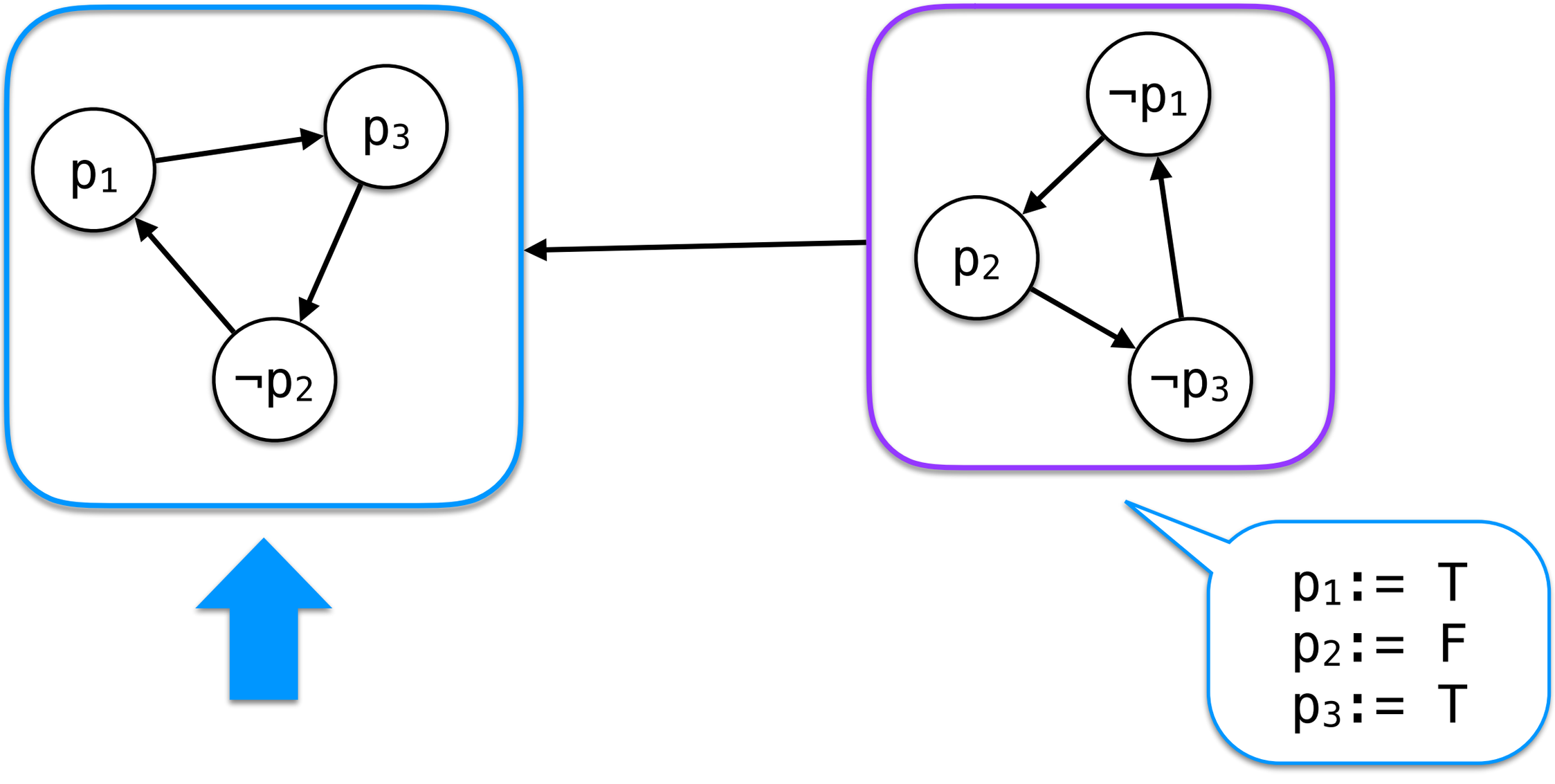
4. Topological order vertices



```
p1:= ?
p2:= ?
p3:= ?
```

2-SAT

5. Assign truth values to the variables in each component



2-SAT

5. Assign truth values to the variables in each component

```
for (int i = 0; i < n; i += 2) {
    if (comp[i] == comp[i + 1])
        return false;
    assignment[i / 2] = comp[i] > comp[i + 1];
}
return true;
```

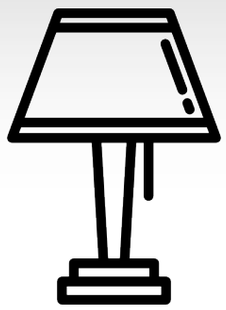
contradiction in the same component

the set **is satisfiable**, the truth values are in assignment

WHICH PROBLEMS ARE
2-SAT

Illumination

Illumination



You inherited a haunted house. Its floor plan is an n-by-n square grid with l lamps in fixed locations and no interior walls. Each lamp can either illuminate its row or its column, but not both simultaneously. The illumination of each lamp extends by r squares in both directions, so a lamp unobstructed by an exterior wall of the house can illuminate as many as $2r + 1$ squares.

If a square is illuminated by more than one lamp in its row, or by more than one lamp in its column, the resulting bright spot will scare away ghosts forever, diminishing the value of your property. Is it possible for all lamps to illuminate a row or column, without scaring any ghosts? Note that a square illuminated by two lamps, one in its row and the other in its column, will not scare away the ghosts.

Input

The first line of input contains three positive integers, n, r and l ($1 \leq n, r, l \leq 1{,}000$).

Each of the next l lines contains two positive integers r_i and c_i ($1 \leq r_i, c_i \leq n$), indicating that there is a lamp in row r_i and column c_i.

It is guaranteed that all lamps are in distinct locations.

Output

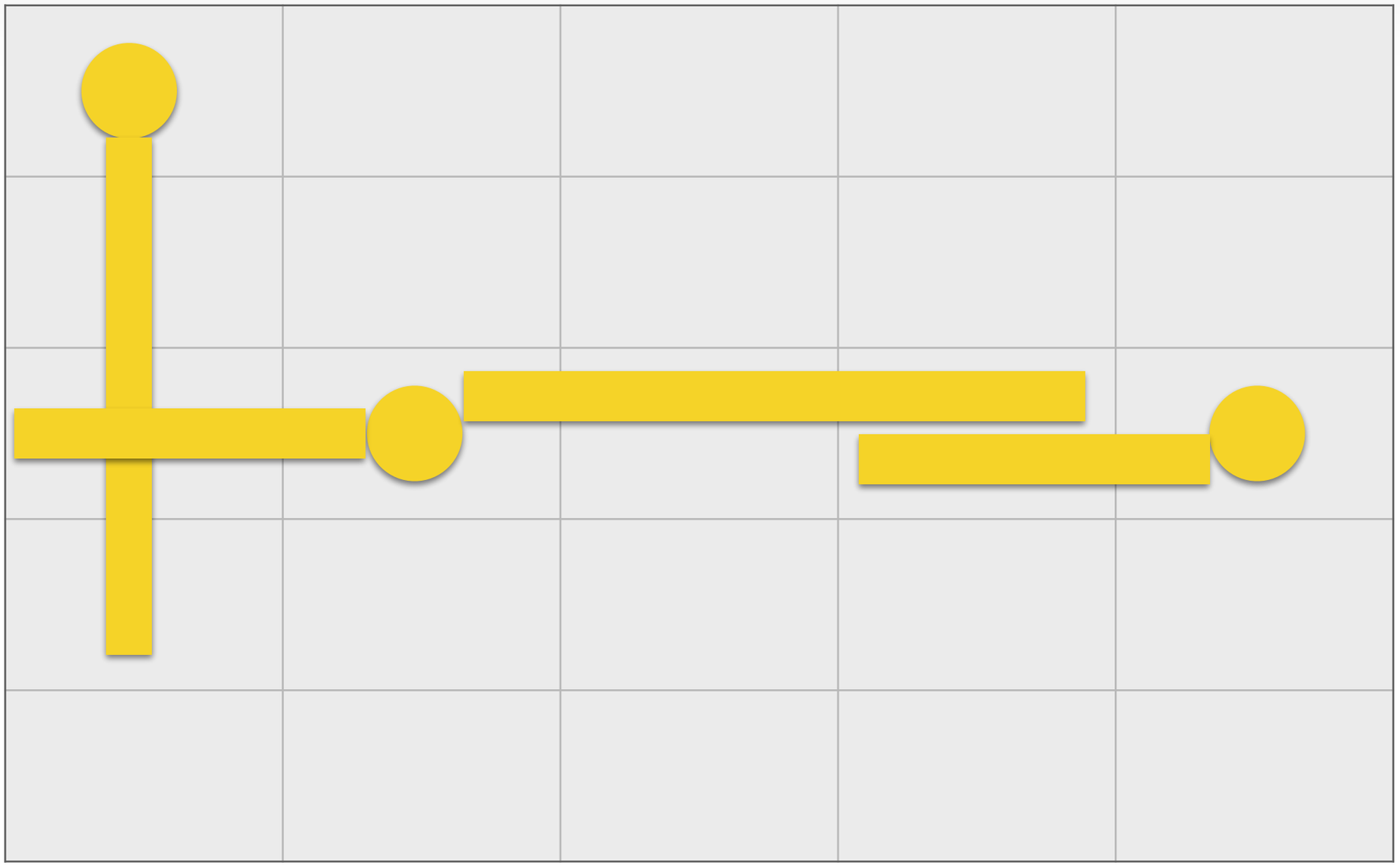
Print, on a single line, `YES` if it is possible to illuminate all lamps as stated above; otherwise, print `NO`.

Illumination

Sample Input	Sample Output
3 2 5 1 1 1 3 3 1 3 3 2 2	YES

Sample Input	Sample Output
3 2 6 1 1 1 2 1 3 3 1 3 2 3 3	NO

Illumination



WHAT IF NO SQUARE CAN BE ILLUMINATED TWICE?

Manhattan

You are the mayor of a city with severe traffic problems. To deal with the situation, you have decided to make a new plan for the street grid. As it is impossible to make the streets wider, your approach is to make them one-way (only traffic in one direction is allowed on a street), thus creating a more efficient flow of traffic.

The streets in the city form an orthogonal grid — like on Manhattan avenues run in north-south-direction, while streets run in east-west-direction. Your mission is to make all the streets and avenues one-way, i.e. fix the direction in which traffic is allowed, while maintaining a short driving distance between some ordered pairs of locations. More specifically, a route in the city is defined by two street-avenue crossings, the start and goal location. On a one-way street grid, a route has a legal path if it is possible to drive from the start location to the goal location along the path passing streets and avenues in their prescribed direction only. A route does not define a specific path between the two locations - there may be many possible paths for each route. A legal path in a one-way street grid is considered simple if it requires at most one turn, i.e. a maximum of one street and one avenue need to be used for the path.

When traveling by car from one location to another, a simple path will be preferred over a non-simple one, since it is faster. However, as each street in the grid is one-way, there may always be routes for which no simple path exists. On your desk lies a list of important routes which you want to have simple paths after the re-design of the street grid.

Your task is to write a program that determines if it is possible to fix the directions of the one-way streets and avenues in such a way that each route in the list has at least one simple path.

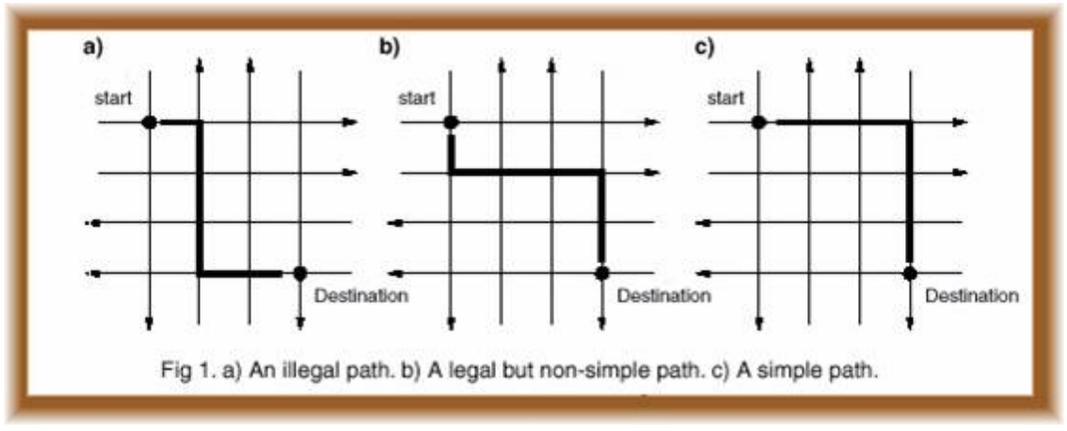


Fig 1. a) An illegal path. b) A legal but non-simple path. c) A simple path.

Input

On the first line of the input, there is a single integer n, telling how many city descriptions that follows. Each city description begins with a line containing three integers: the number of streets $0 < S \leq 30$ and avenues $0 < A \leq 30$ in the street grid, and the number of routes $0 < m \leq 200$ that should have at least one simple path. The next m lines define these routes, one on each line. Each route definition consists of four integers, $s1$, $a1$, $s2$, $a2$, where the start location of the route is at the crossing of street $s1$ and avenue $a1$, and the goal location is at the crossing of street $s2$ and avenue $a2$. Obviously, $0 < s1, s2 \leq S$ and $0 < a1, a2 \leq A$.

Output

For each city, your program should output '`Yes`' on a single line if it is possible to make the streets and avenues one-way, so that each route has at least one simple path. Otherwise the text '`No`' should be printed on a line of its own.

IF MAX-FLOW (DINIC-SOSA) DOESN'T WORK
2-SAT MAY