Advanced Algorithmic Techniques (COMP523)

NP-Completeness

Recap and plan

- **Previous 16 lectures:**
  - Polynomial time algorithms for solving several problems
    - Searching, sorting, graph reachability, interval scheduling, minimum spanning trees etc.
- **This lecture:**
  - Polynomial time reductions
  - Computational classes: P and NP
  - NP-hardness and NP-completeness
  - NP-Complete problems: 3SAT and Vertex Cover

Polynomial Time Reduction

- We are given a problem A that we want to solve.

Polynomial Time Reduction

- We are given a problem A that we want to solve.
- We can reduce solving problem A to solving some other problem B.

Polynomial Time Reduction

- We are given a problem A that we want to solve.
- We can reduce solving problem A to solving some other problem B.
- Assume that we had an algorithm ALG^B for solving problem B, which we can use at cost **O(*1*)**.

Polynomial Time Reduction

- We are given a problem A that we want to solve.
- We can reduce solving problem A to solving some other problem B.
- Assume that we had an algorithm ALG^B for solving problem B, which we can use at cost **O(**1**)**.
- We can construct an algorithm ALG^A for solving problem A, which uses calls to the algorithm ALG^B as a subroutine.

Polynomial Time Reduction

- We are given a problem A that we want to solve.
- We can reduce solving problem A to solving some other problem B.
- Assume that we had an algorithm ALG^B for solving problem B, which we can use at cost **O(*1*)**.
- We can construct an algorithm ALG^A for solving problem A, which uses calls to the algorithm ALG^B as a subroutine.
- If ALG^A is a polynomial time algorithm, then this is a *polynomial time reduction*.

Pictorially

Problem A

ALG^A

Do stuff …

Do stuff …

ALG^B

Do stuff…

Do stuff …

ALG^B

…

Problem B

ALG^B

Polynomial time reduction

- Can you think of any examples of such reductions?

Notation

- When problem A reduces to problem B in polynomial time, we write

  $A \leq^{p} B$

  We often say “there is a polynomial time reduction *from* A *to* B”.

How to work with reductions

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
    - I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
  - I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.
- Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
    - I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.
- Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.
    - If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
    - I can try to come up with a polynomial time reduction A $\leq^p$ B, which will give me a polynomial time algorithm for solving A.
- Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.
    - If I come up with a polynomial time reduction A $\leq^p$ B, it is also unlikely that there is a polynomial time algorithm that solves B.
    - B is “*at least as hard to solve as*” A, because if I could solve B, I could also solve A.

Types of reductions

- **Turing reduction:**
  - Notation: $A \leq_T B$
  - A reduction which solves problem A using (polynomially) many calls to an oracle (an algorithm) for solving problem B.
  - (Also known as Cook reduction).

Pictorially

Problem A

ALG^A

Do stuff …

Do stuff …

ALG^B

Do stuff…

Do stuff …

ALG^B

…

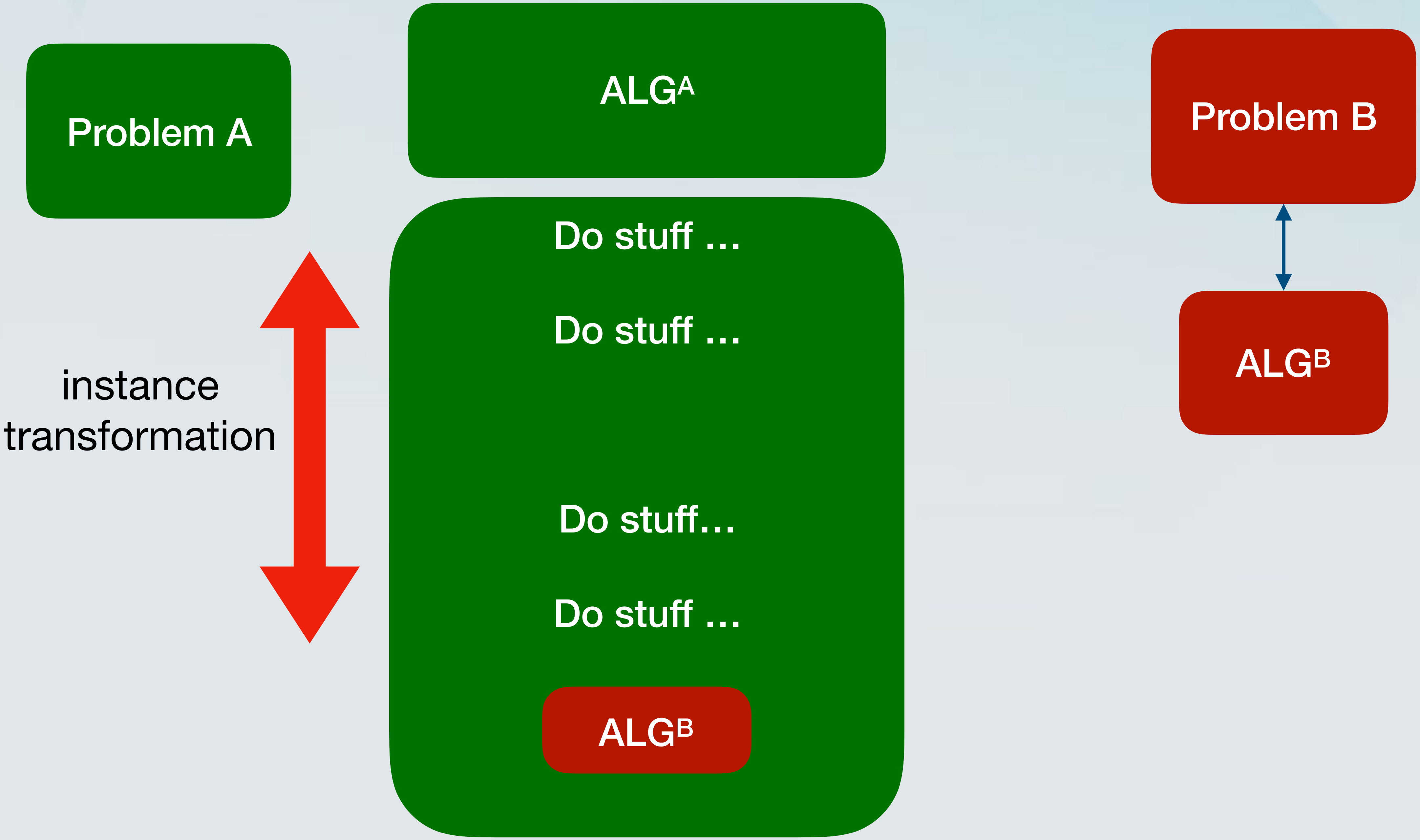
Problem B

ALG^B

Types of reductions

- **Turing reduction:**
  - Notation: $A \leq_T B$
  - A reduction which solves problem A using (polynomially) many calls to an oracle (an algorithm) for solving problem B.
  - (Also known as Cook reduction).
- **Many-one reduction:**
  - Notation: $A \leq_m B$
  - A reduction which *converts instances* of problem A to *instances* of problem B.
  - (Also known as Karp reduction).

Pictorially



Problem A

ALG^A

instance transformation

Do stuff …

Do stuff …

Do stuff…

Do stuff …

ALG^B

Problem B

ALG^B

Types of reductions

- **Turing reduction:**
  - Argument: Here is an algorithm which runs in polynomial time solving problem A, using polynomially many calls to an oracle for problem B.
- **Many-one reduction:**
  - Argument:
    - If z is a solution to instance I of problem A, then z’ is a solution of instance f(I) to problem B.
    - If z is not a solution to instance I of problem A, then z’ is not a solution of instance f(I) to problem B.
    - Equivalently: If z’ is a solution of instance f(I) to problem B, then z is a solution to instance I of problem A.

Example:

Bipartite Matching $\leq_m$ Maximum Flow

- *Maximum Bipartite Matching* or Maximum matching on a bipartite graph G.

  - Matching: A subset M of the edges E such that each node v of V appears in at most one edge e in E.

  - Maximum matching: A matching with maximum cardinality.(i.e., |M| is maximised).

From matchings to flows

- Claim: Assume that there is a matching M of size k on G. Then there is a flow f of value k in G^f.

From flows to matchings

- Claim: Assume that there is a a flow f of value k in G^f. Then there is a matching M of size k on G.

Technically speaking

- Here problem A was:

  Is there a bipartite matching of size at least k?

  and problem B was:

  Is there a flow with value at least k?

- Maximum Bipartite Matching and Maximum Flow are *optimisation problems*.

- The reduction used the corresponding *decision problems*.

- More about that later.

Running time hierarchy

$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^\alpha)$	$O(c^n)$
logarithmic	linear		quadratic	polynomial	exponential
The algorithm does not even read the whole input.	The algorithm accesses the input only a constant number of times.	The algorithm splits the inputs into two pieces of similar size, solves each part and merges the solutions.	The algorithm considers pairs of elements.	The algorithm performs many nested loops.	The algorithm considers many subsets of the input elements.

constant	$O(1)$	superlinear	$\omega(n)$
superconstant	$\omega(1)$	superpolynomial	$\omega(n^\alpha)$
sublinear	$o(n)$	subexponential	$o(c^n)$

Running time hierarchy

Polynomial time

$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^\alpha)$	$O(c^n)$
logarithmic	linear		quadratic	polynomial	exponential
The algorithm does not even read the whole input.	The algorithm accesses the input only a constant number of times.	The algorithm splits the inputs into two pieces of similar size, solves each part and merges the solutions.	The algorithm considers pairs of elements.	The algorithm performs many nested loops.	The algorithm considers many subsets of the input elements.

constant	$O(1)$	superlinear	$\omega(n)$
superconstant	$\omega(1)$	superpolynomial	$\omega(n^\alpha)$
sublinear	$o(n)$	subexponential	$o(c^n)$

Computational classes

- Every problem for which there is a known polynomial time algorithm is in the computational class P.
  - Searching, sorting, interval scheduling, minimum spanning tree, graph traversal, …
  - The class P contains computational problems *that can be solved in polynomial time*.
    - We also say that they can be solved *efficiently*.

Problems not in P

- Do you remember any problems from the lectures that we did not manage to prove that they lie in P?

Problems not in P

- Do you remember any problems from the lectures that we did not manage to prove that they lie in P?
  - Weighted interval scheduling?

Problems not in P

- Do you remember any problems from the lectures that we did not manage to prove that they lie in P?
    - Weighted interval scheduling?
    - Subset sum?

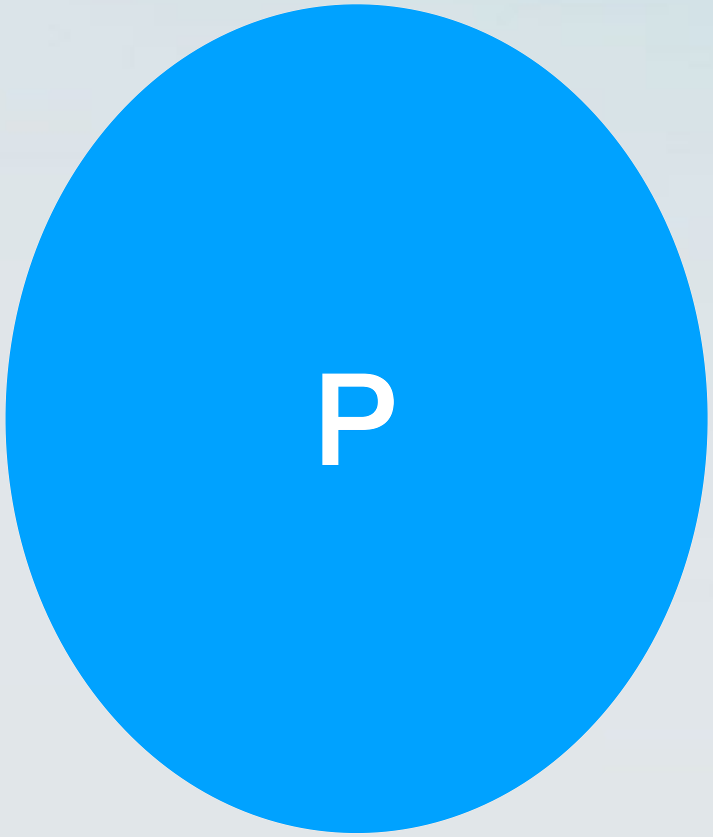
Problems not in P

- Do you remember any problems from the lectures that we did not manage to prove that they lie in P?
  - Weighted interval scheduling?
  - Subset sum?
  - Knapsack?

Problems not in P

- Do you remember any problems from the lectures that we did not manage to prove that they lie in P?
  - Weighted interval scheduling?
  - Subset sum?
  - Knapsack?
  - Maximum flow?

The landscape of complexity



contains all problems that
can be solved in polynomial time.

The class **NP**

The class NP

- Stands for “*non deterministic polynomial time*”.

The class NP

- Stands for “*non deterministic polynomial time*”.
- Problems that can be solved in polynomial time by a non-deterministic Turing machine.

The class NP

- Stands for “*non deterministic polynomial time*”.
- Problems that can be solved in polynomial time by a non-deterministic Turing machine.
- More intuitive definition:

The class **NP**

- Stands for “*non deterministic polynomial time*”.
- Problems that can be solved in polynomial time by a non-deterministic Turing machine.
- More intuitive definition:
  - Problems such that, *if a solution is given*, it can be *checked* that it is indeed a solution in polynomial time.

The class **NP**

- Stands for “*non deterministic polynomial time*”.
- Problems that can be solved in polynomial time by a non-deterministic Turing machine.
- More intuitive definition:
  - Problems such that, *if a solution is given*, it can be *checked* that it is indeed a solution in polynomial time.
  - *Efficiently verifiable*.

The subset sum problem

- We are given a set of n items {*1*, 2, … , *n*}.
- Each item *i* has a non-negative integer weight w_i.
- We are given an integer bound W.
- Goal: Select a subset S of the items such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} w_i$ is maximised.

Equivalent formulation decision version

- We are given a set of n items {*1*, 2, … , *n*}.
- Each item *i* has a non-negative integer weight w_i.
- We are given an integer bound W.
- Goal: Decide if there exists a subset S of the items such that

$$\sum_{i \in S} w_i = W$$

Subset Sum is in NP

- If we are given a candidate solution S, we can easily check whether the following holds or not:

$$\sum_{i \in S} w_i = W$$

Problem classification

Problem classification

- Problems in P:
  - Searching, sorting, minimum spanning tree, graph traversal, maximum flow, minimum cut, Weighted Interval Scheduling, …

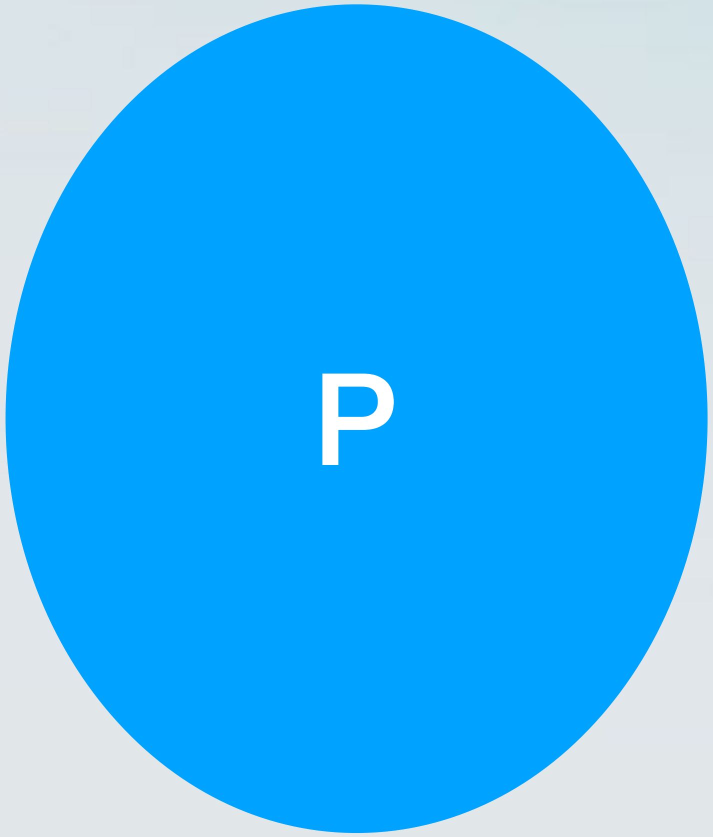
Problem classification

- Problems in P:
  - Searching, sorting, minimum spanning tree, graph traversal, maximum flow, minimum cut, Weighted Interval Scheduling, …
- Problems in NP:
  - Subset Sum, Knapsack

Problem classification

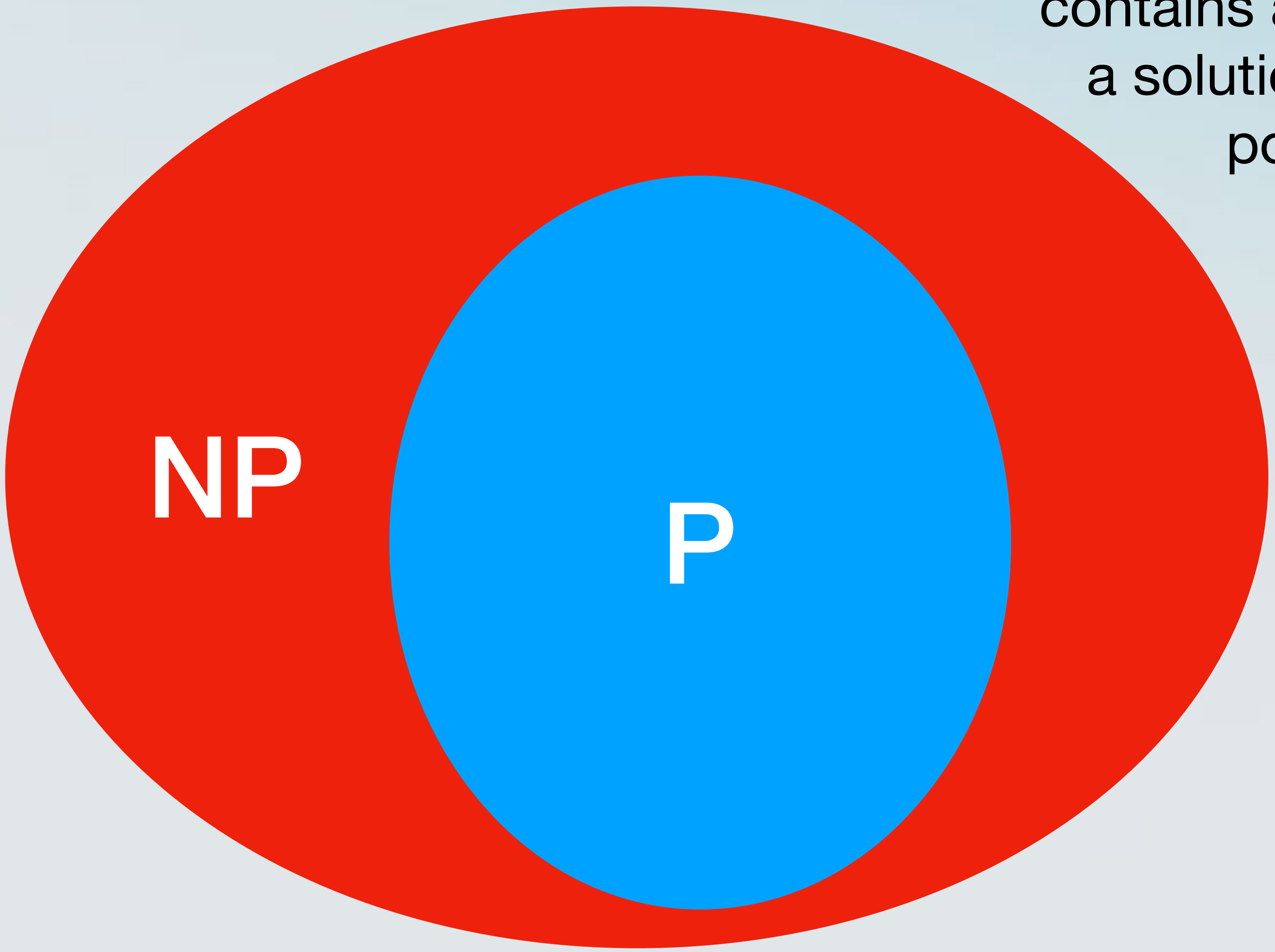
- Problems in P:
  - Searching, sorting, minimum spanning tree, graph traversal, maximum flow, minimum cut, Weighted Interval Scheduling, …
- Problems in NP:
  - Subset Sum, Knapsack, Weighted Interval Scheduling, Searching, sorting, minimum spanning tree, graph traversal, maximum flow, minimum cut, …

The landscape of complexity



contains all problems that
can be solved in polynomial time.

The landscape of complexity



contains all problems for which a solution can be verified in polynomial time.

contains all problems that can be solved in polynomial time.

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
  - I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.
- Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.
  - If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.
  - B is "*at least as hard to solve as*" A, because if I could solve B, I could also solve A.

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
  - I can try to come up with a polynomial time reduction A $\leq^p$ B, which will give me a polynomial time algorithm for solving A.
- Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.
  - If I come up with a polynomial time reduction A $\leq^p$ B, it is also unlikely that there is a polynomial time algorithm that solves B.
  - B is "*at least as hard to solve as*" A, because if I could solve B, I could also solve A.

NP-hardness

- A problem B is NP-hard if for every problem A in NP, it holds that A $\leq^p$ B.
  - If every problem in NP is “polynomial time reducible to B”.
  - This captures the fact that B is *at least as hard as the hardest problems* in NP.

NP-hardness

- A problem B is NP-hard if for every problem A in NP, it holds that $A \leq^{p} B$.

- To prove NP-hardness, it seems that we have to construct a reduction from every problem A in NP.
    - This is not very useful!

NP-completeness

- A problem B is NP-complete if

NP-completeness

- A problem B is NP-complete if
    - *It is in NP*.

NP-completeness

- A problem B is NP-complete if
  - *It is in NP*.
    - i.e., it has a polynomial-time verifiable solution.

NP-completeness

- A problem B is NP-complete if
  - *It is in NP*.
    - i.e., it has a polynomial-time verifiable solution.
  - *It is NP-hard*.

NP-completeness

- A problem B is NP-complete if
  - *It is in NP*.
    - i.e., it has a polynomial-time verifiable solution.
  - *It is NP-hard*.
    - i.e., every problem in NP can be efficiently reduced to it.

NP-completeness

NP-completeness

- Assume problem P is NP-complete.

NP-completeness

- Assume problem P is NP-complete.
    - Then every problem in NP is efficiently reducible to P. (why?)

NP-completeness

- Assume problem P is NP-complete.
  - Then every problem in NP is efficiently reducible to P. (why?)
- To prove NP-hardness of problem B, it seems that we have to construct a reduction from every problem A in NP.

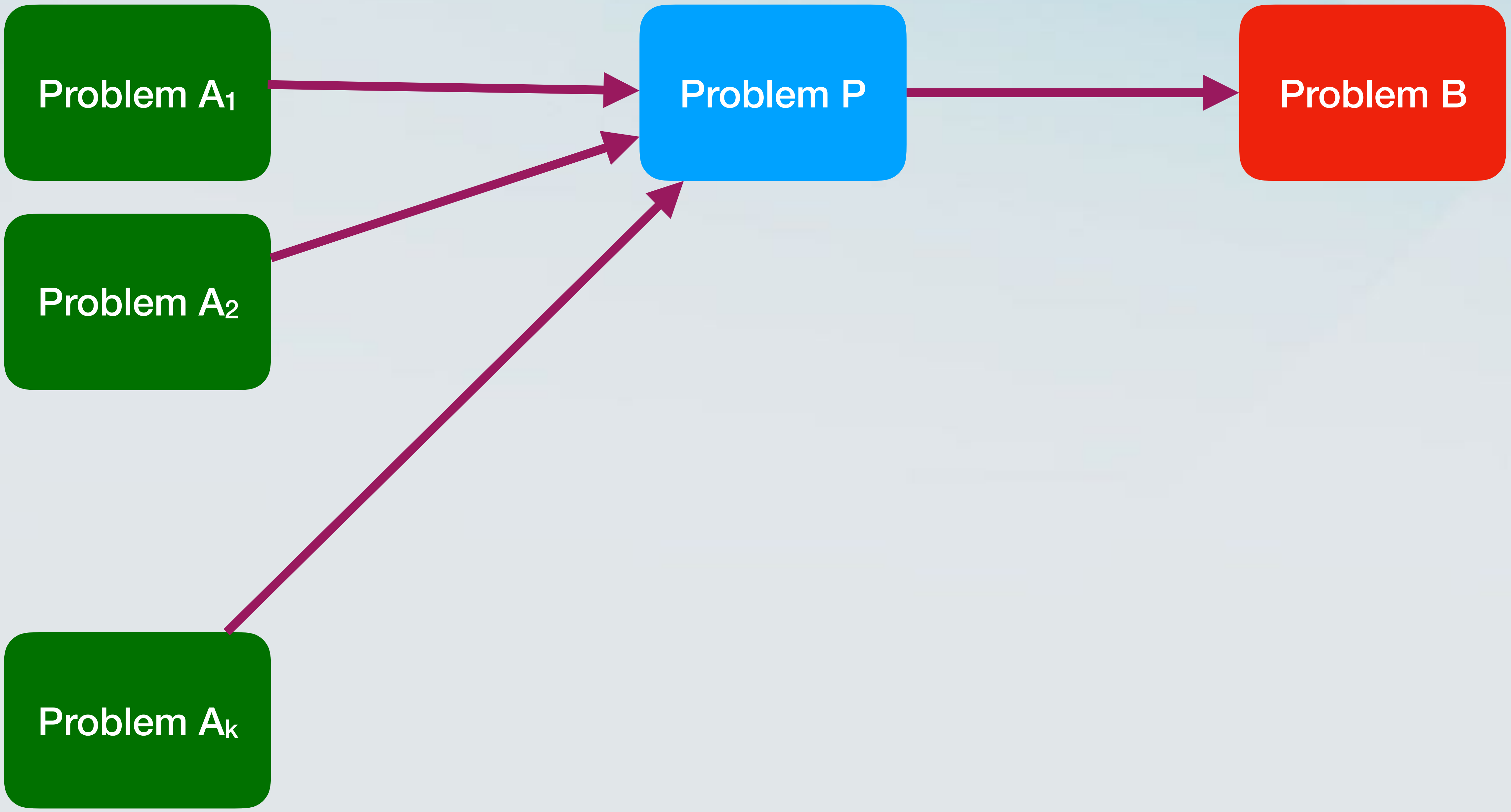
NP-completeness

- Assume problem P is NP-complete.
  - Then every problem in NP is efficiently reducible to P. (why?)
- To prove NP-hardness of problem B, it seems that we have to construct a reduction from every problem A in NP.
  - Actually, it suffices to construct a reduction from P to B.

NP-completeness

- Assume problem P is NP-complete.
    - Then every problem in NP is efficiently reducible to P. (why?)
- To prove NP-hardness of problem B, it seems that we have to construct a reduction from every problem A in NP.
    - Actually, it suffices to construct a reduction from P to B.
    - A reduction from any other problem A to B goes “via” P.

NP-hardness via P.



Problem A_1

Problem A_2

Problem A_k

Problem P

Problem B

NP-completeness

NP-completeness

- Assume problem P is NP-complete.

NP-completeness

- Assume problem P is NP-complete.
- This all works if we have an NP-complete problem to start with.

3 SAT

- A CNF formula with m clauses and k literals.

  $\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

- ("An AND of ORs").
- Each clause has three literals.

3 SAT

- A CNF formula with m clauses and k literals.

  $\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

- (“An AND of ORs”).

- Each clause has three literals.

- Truth assignment: A value in {0,1} for each variable x_i.

3 SAT

- A CNF formula with m clauses and k literals.

  $\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

- ("An AND of ORs").

- Each clause has three literals.

- Truth assignment: A value in {0,1} for each variable x_i.

- Satisfying assignment: A truth assignment which makes the formula evaluate to 1 (= true).

3 SAT

- A CNF formula with m clauses and k literals.

  $\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

- ("An AND of ORs").
- Each clause has three literals.
- Truth assignment: A value in {0,1} for each variable x_i.
- Satisfying assignment: A truth assignment which makes the formula evaluate to 1 (= true).
- Computational problem 3SAT : Decide if the input formula ϕ has a satisfying assignment.

3 SAT is NP-complete

3 SAT is NP-complete

- 3 SAT is in NP (why?)

3 SAT is NP-complete

- 3 SAT is in NP (why?)
- 3 SAT is NP-hard.

3 SAT is NP-complete

- 3 SAT is in NP (why?)
- 3 SAT is NP-hard.
- Remarks:
  - The first problem shown to be NP-complete was the SAT problem (more general than 3 SAT), and this reduces to 3SAT.
  - Several textbooks start from Circuit SAT, a version of the SAT problem defined on circuits with boolean gates AND, OR or NOT.

Proving NP-completeness

Proving NP-completeness

- Suppose that you are given a problem A and you want to prove that it is NP-complete.

Proving NP-completeness

- Suppose that you are given a problem A and you want to prove that it is NP-complete.
- First, prove that A is in NP.
    - Usually by observing that a solution is efficiently checkable.

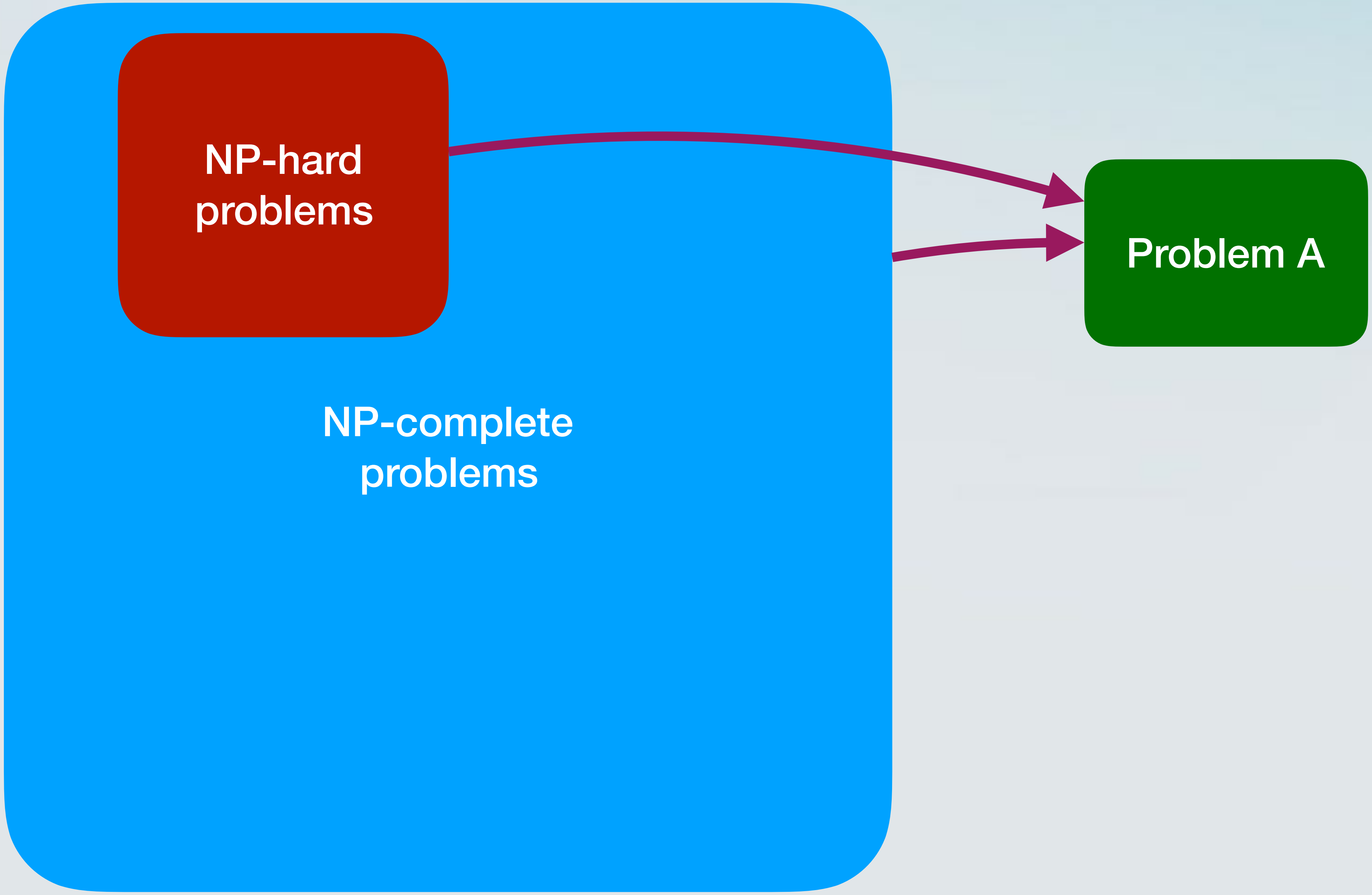
Proving NP-completeness

- Suppose that you are given a problem A and you want to prove that it is NP-complete.
- First, prove that A is in NP.
    - Usually by observing that a solution is efficiently checkable.
- Then prove that A is NP-hard.
    - Construct a polynomial time reduction from some NP-complete problem P.

In fact ...

- Suppose that you are given a problem A and you want to prove that it is NP-complete.
- First, prove that A is in NP.
  - Usually by observing that a solution is efficiently checkable.
- Then prove that A is NP-hard.
  - Construct a polynomial time reduction from some NP-hard problem P.

Pictorially



Enough with the definitions. Let’s see how it works.

- We will prove that a well-known problem on graphs, called Vertex Cover is NP-complete.

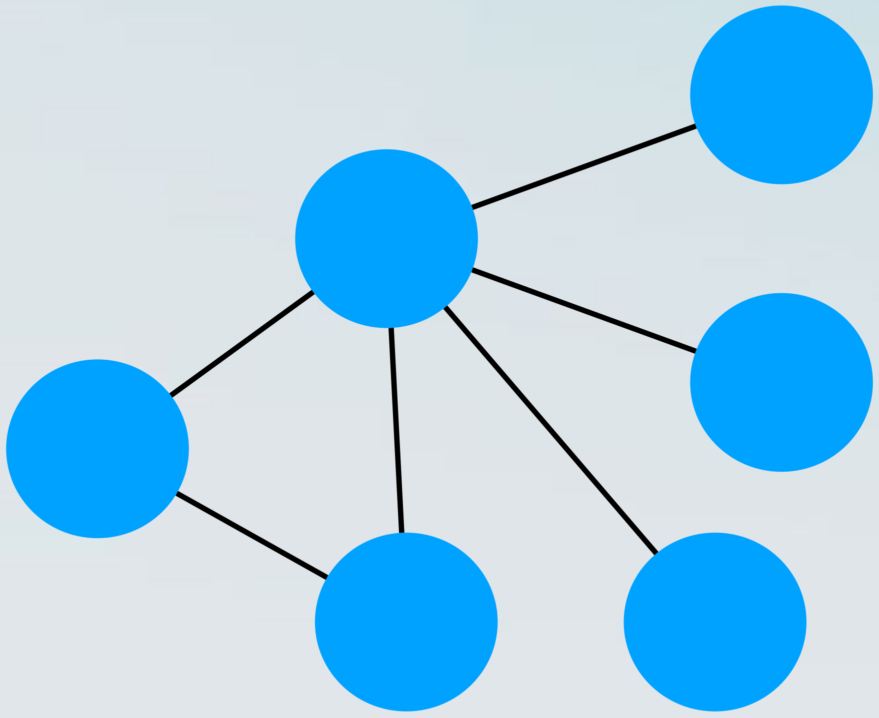
Vertex Cover

- Definition: A vertex cover C of a graph G=(V, E) is a subset of the nodes such that every edge e in the graph has at least one endpoint in C.

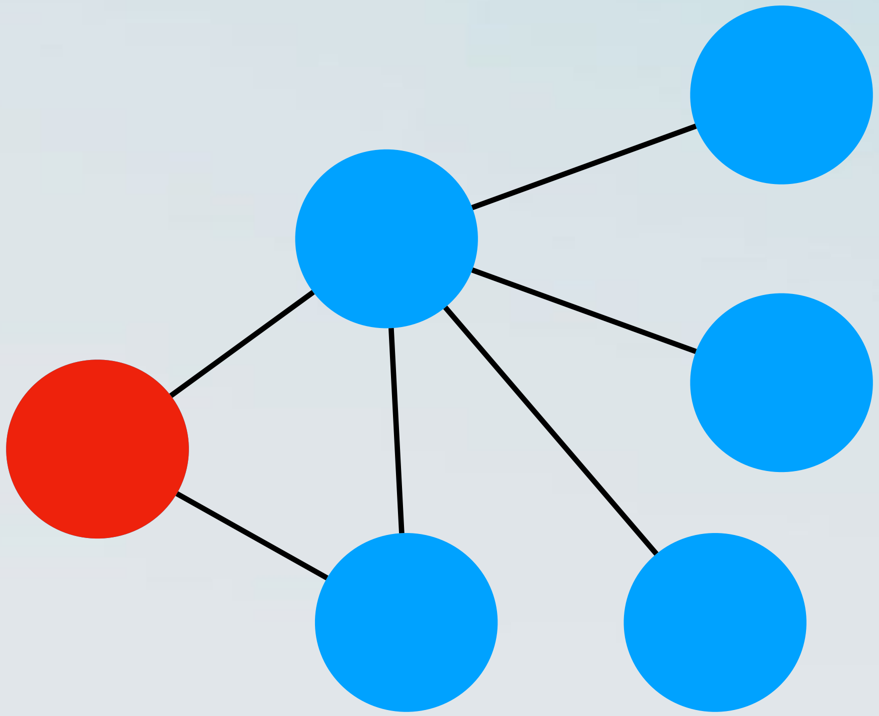
- Definition: A minimum vertex cover is a vertex cover of the smallest possible size.

- Vertex Cover
  Input: A graph G=(V, E)
  Output: A minimum vertex cover.

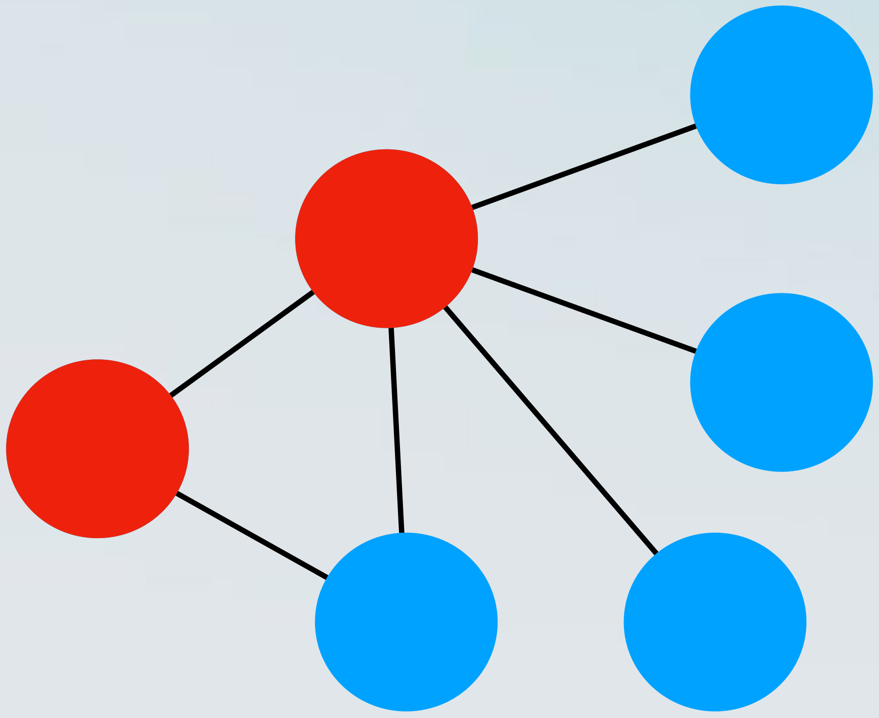
Example



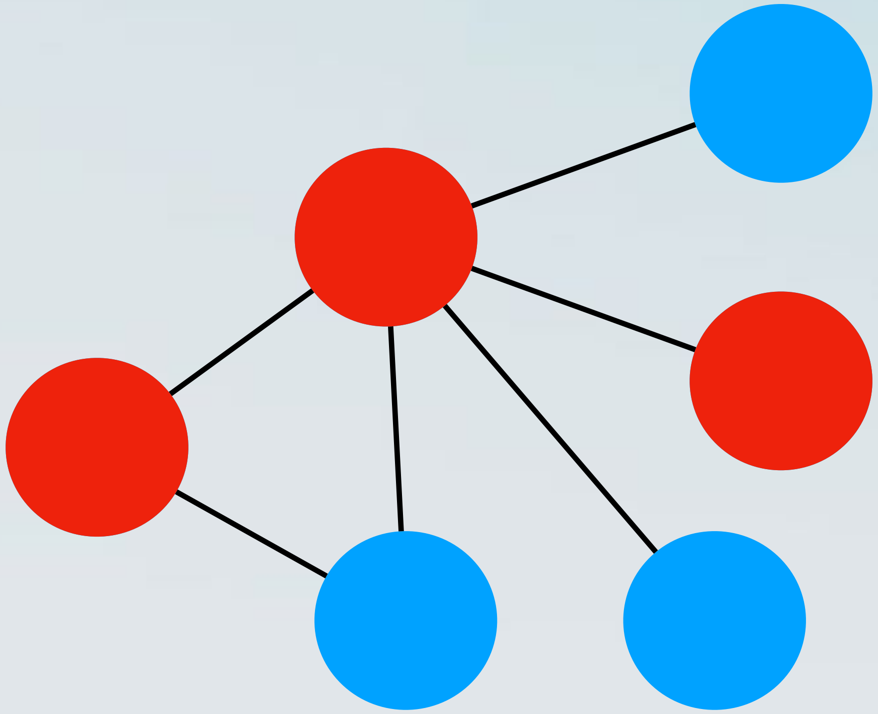
Example



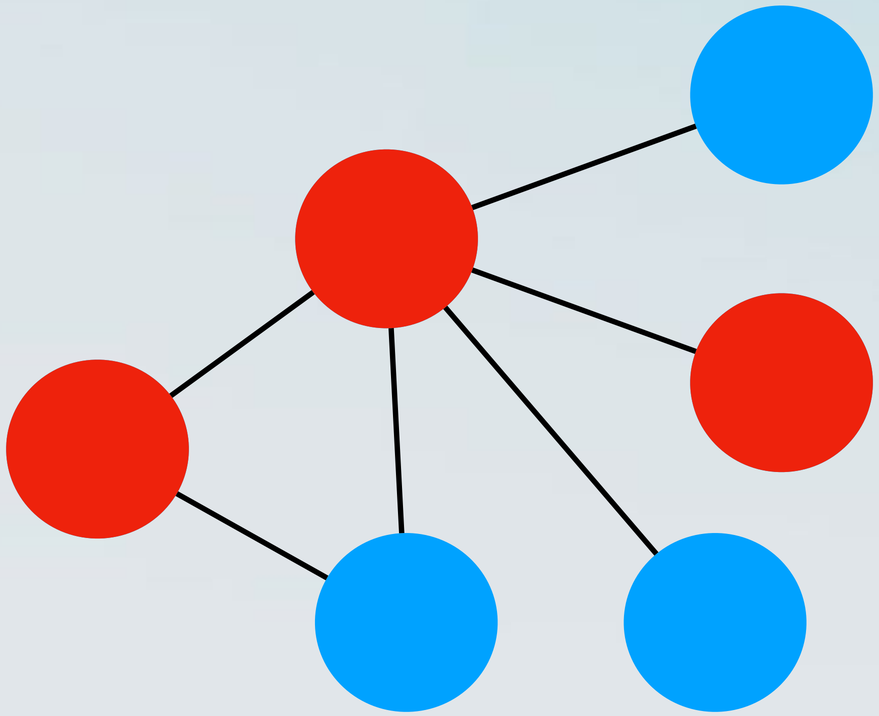
Example



Example

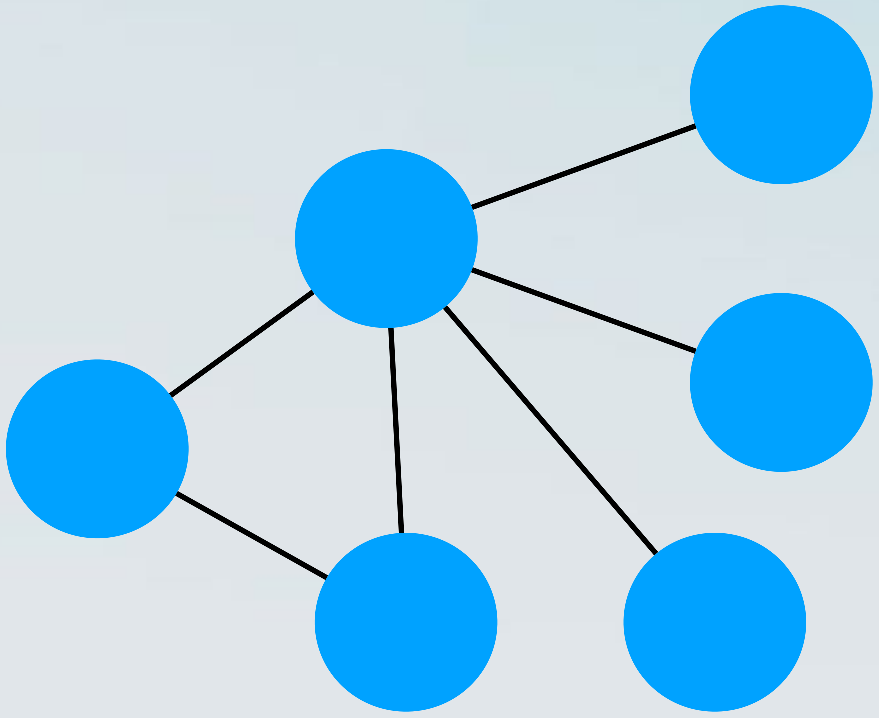


Example

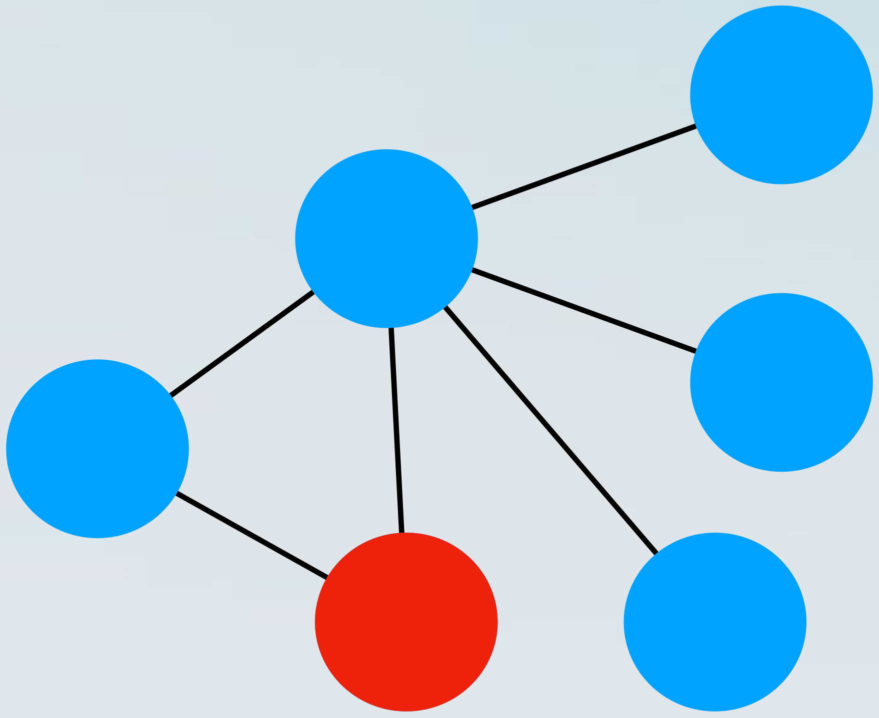


A vertex cover

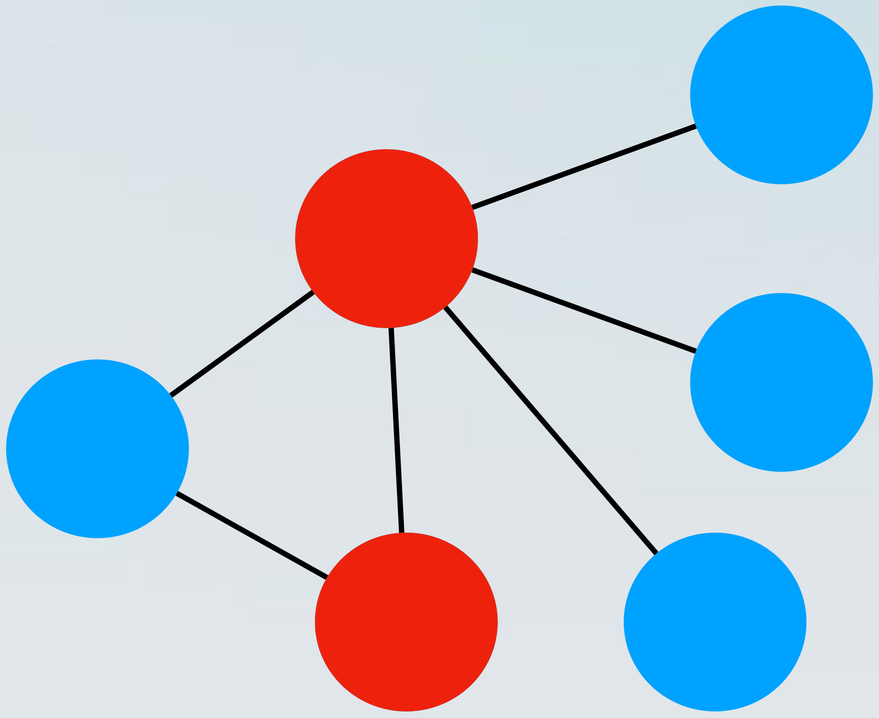
Example



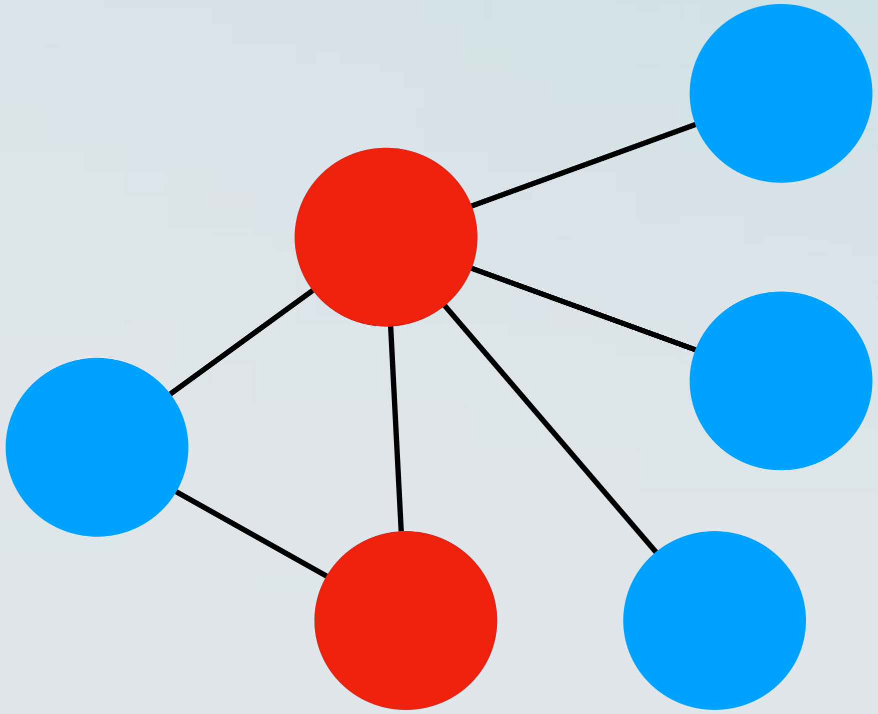
Example



Example



Example



A minimum vertex cover

Vertex Cover

- Definition: A vertex cover C of a graph G=(V, E) is a subset of the nodes such that every edge e in the graph has at least one endpoint in C.

- Definition: A minimum vertex cover is a vertex cover of the smallest possible size.

- Vertex Cover
  Input: A graph G=(V, E)
  Output: A minimum vertex cover.

Vertex Cover decision version

- Definition: A vertex cover C of a graph G=(V, E) is a subset of the nodes such that every edge e in the graph has at least one endpoint in C.

- Definition: A minimum vertex cover is a vertex cover of the smallest possible size.

- Vertex Cover
  Input: A graph G=(V, E) and a number k
  Output: Is there a vertex cover of size ≤ k?.

Vertex cover

Vertex cover

- Vertex Cover is in NP.

Vertex cover

- Vertex Cover is in NP.
- Assume that we are given a vertex cover.
  - We can check that is has size k and that it is a vertex cover in polynomial time.

Vertex cover

Vertex cover

- Vertex Cover is in NP-hard.

Vertex cover

- Vertex Cover is in NP-hard.
- We will construct a polynomial time reduction from 3SAT.
  - i.e., we will prove that 3SAT $\leq^p$ Vertex Cover.

The reduction

- Let ϕ be a 3-CNF formula with m clauses and d variables.
- We construct, in polynomial time, an instance <G, k> of Vertex Cover such that
    - If ϕ is satisfiable => G has a vertex cover of size at most k.
    - If ϕ is not satisfiable => G does not have any vertex cover of size at most k.

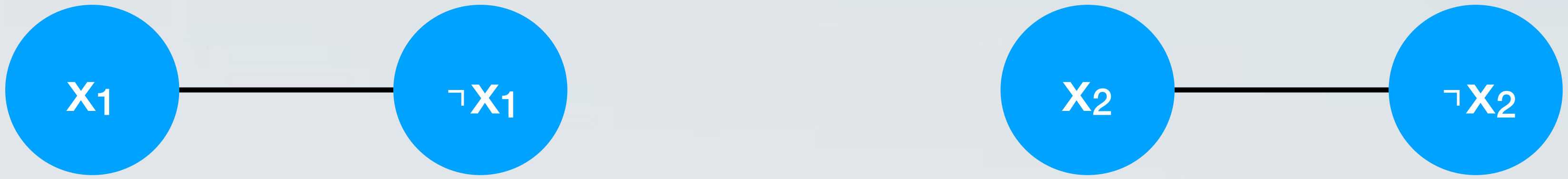
The reduction

- For every variable x in ϕ, we create two nodes x and $\neg x$ in G and we connect them with an edge $e = (x , \neg x)$.

Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

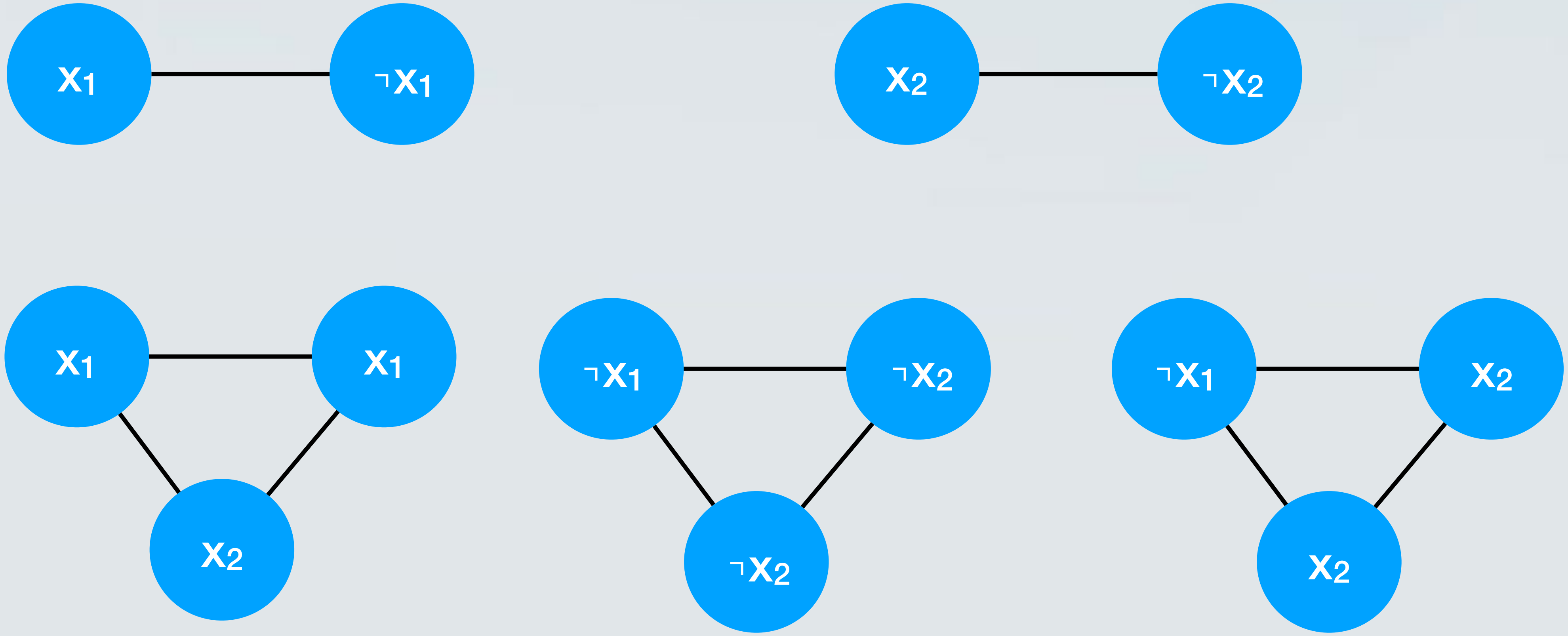
- For every variable x in ϕ, we create two nodes x and ¬x in G and we connect them with an edge e = (x , ¬x).



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

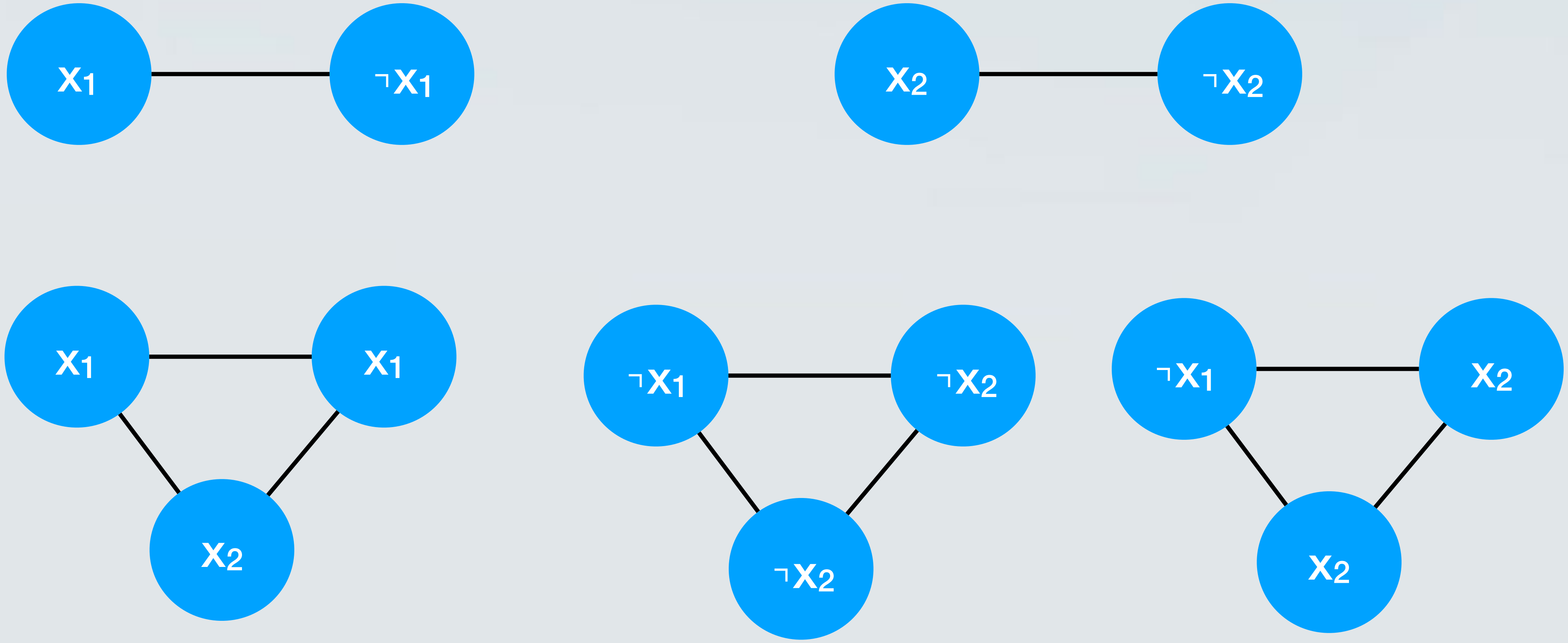
- For every clause $\ell = (\ell_1, \ell_2, \ell_3)$ in ϕ, we create three nodes ℓ_1, ℓ_2, ℓ_3 in G and we connect them all with each other.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

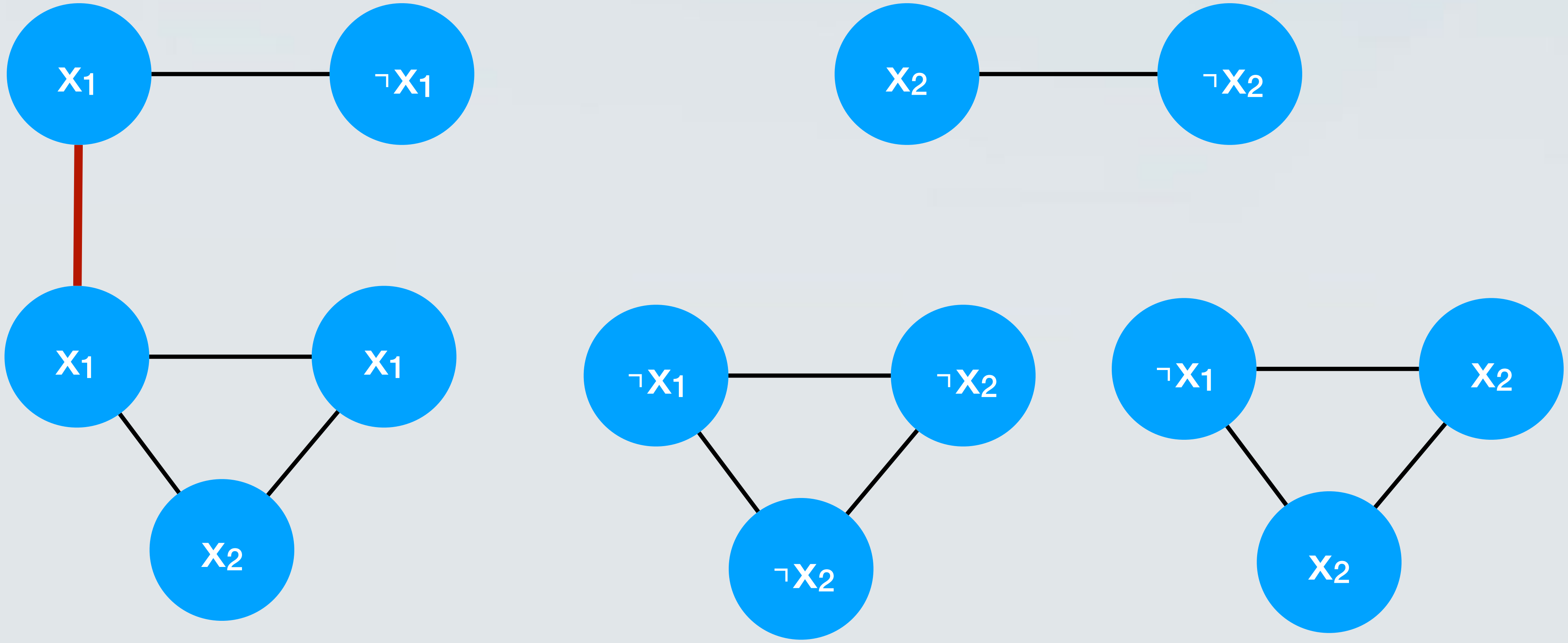
- We add an edge between all nodes with the same label on the top and on the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

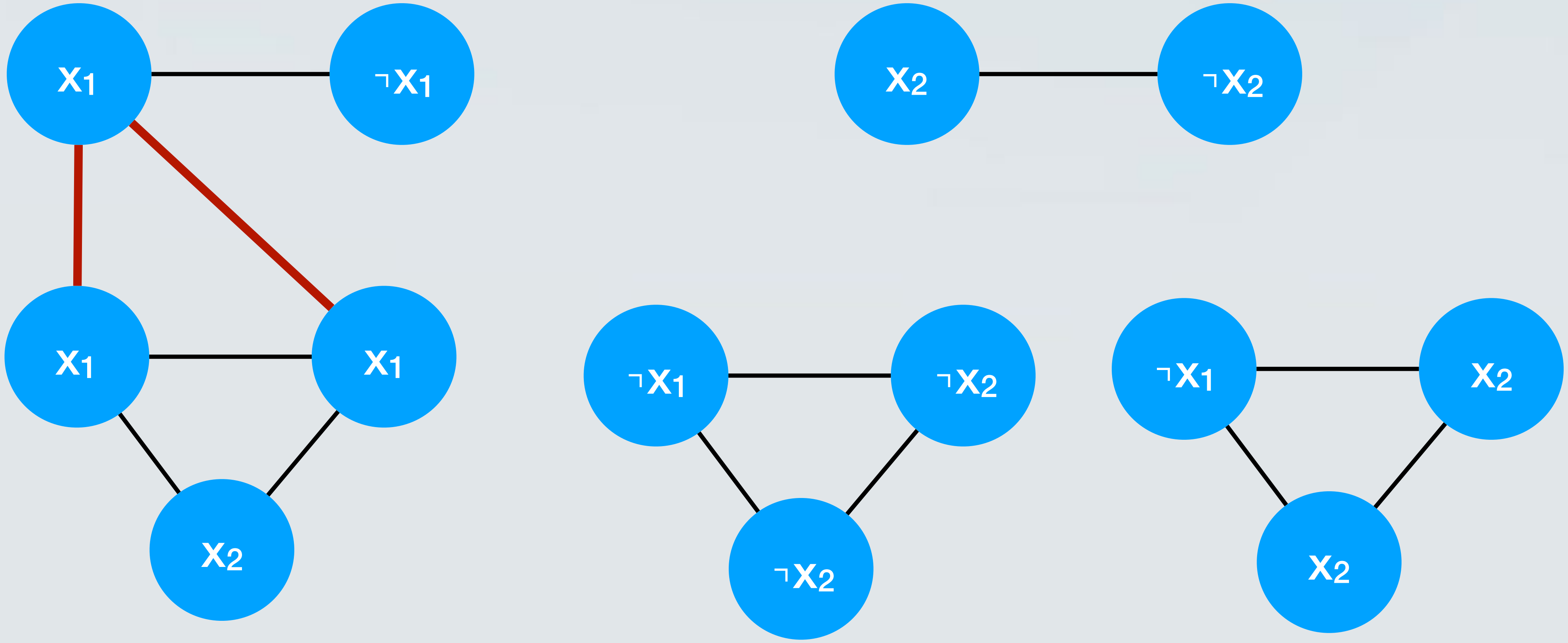
- We add an edge between all nodes with the same label on the top and on the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

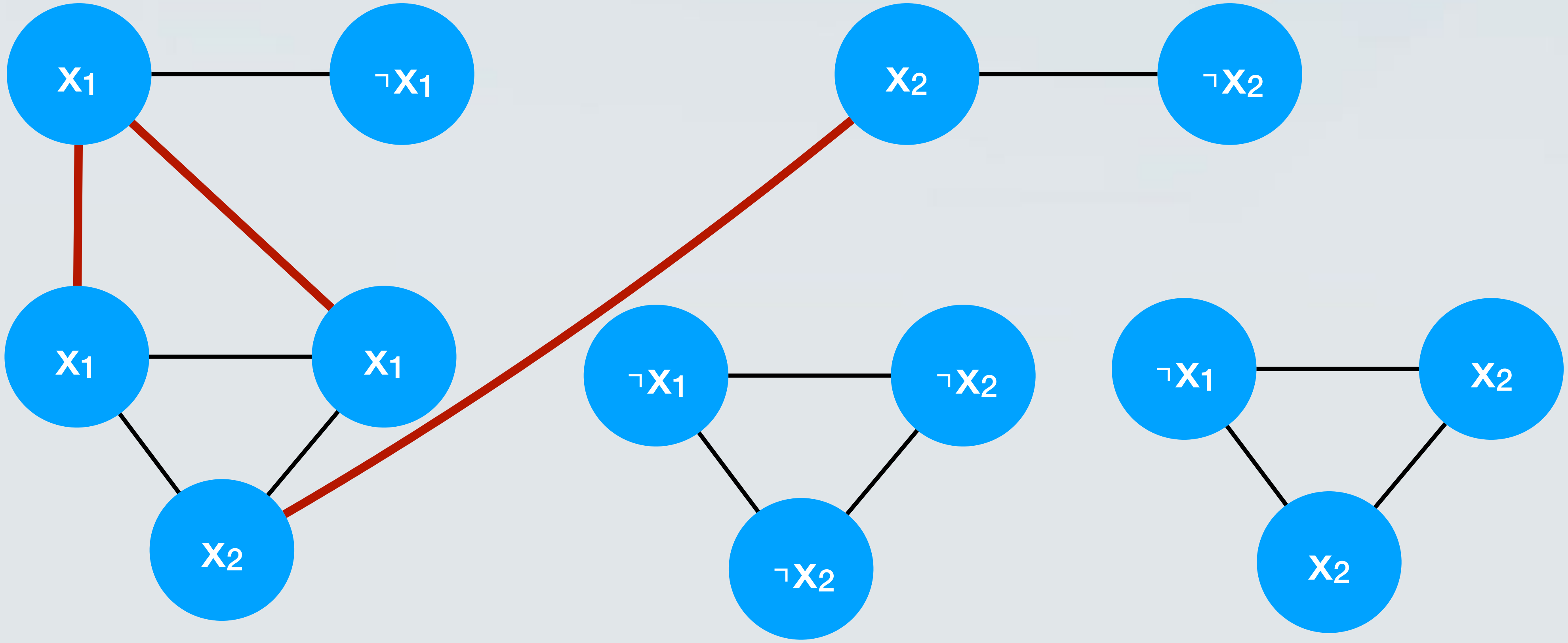
- We add an edge between all nodes with the same label on the top and on the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

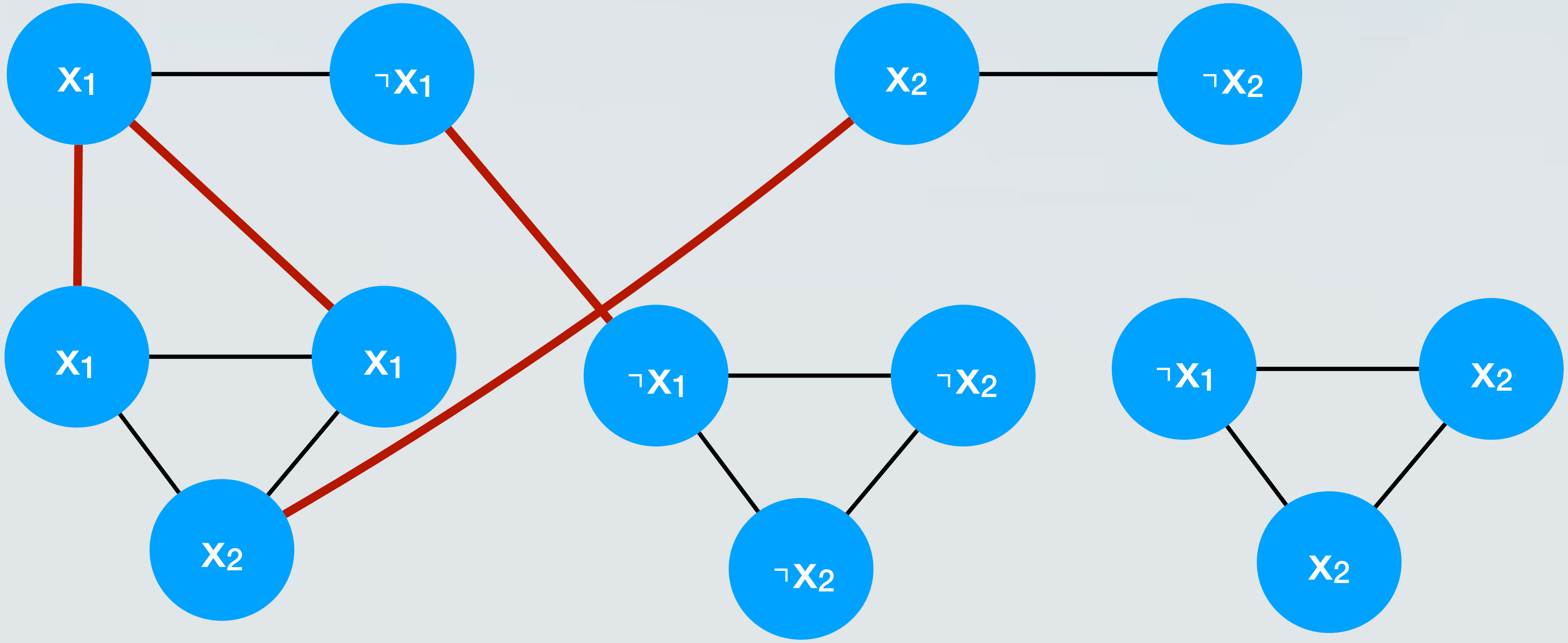
- We add an edge between all nodes with the same label on the top and on the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

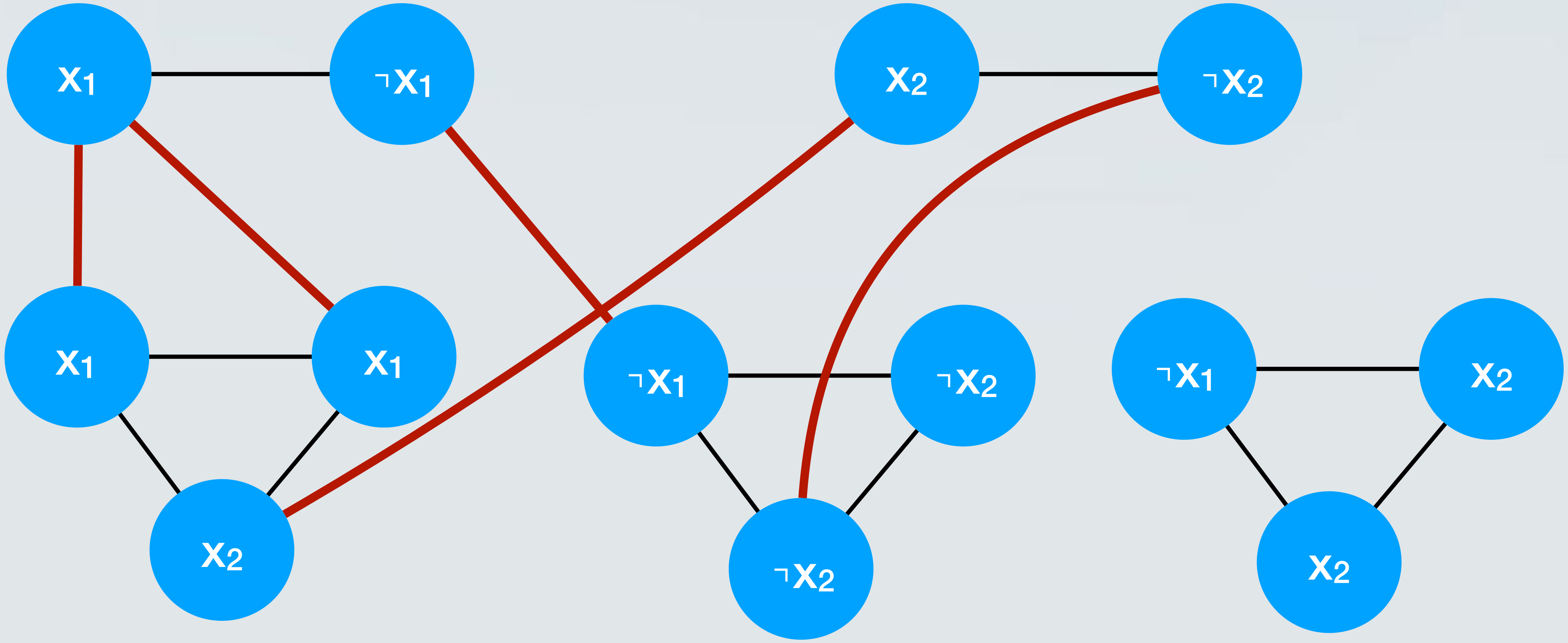
- We add an edge between all nodes with the same label on the top and on the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

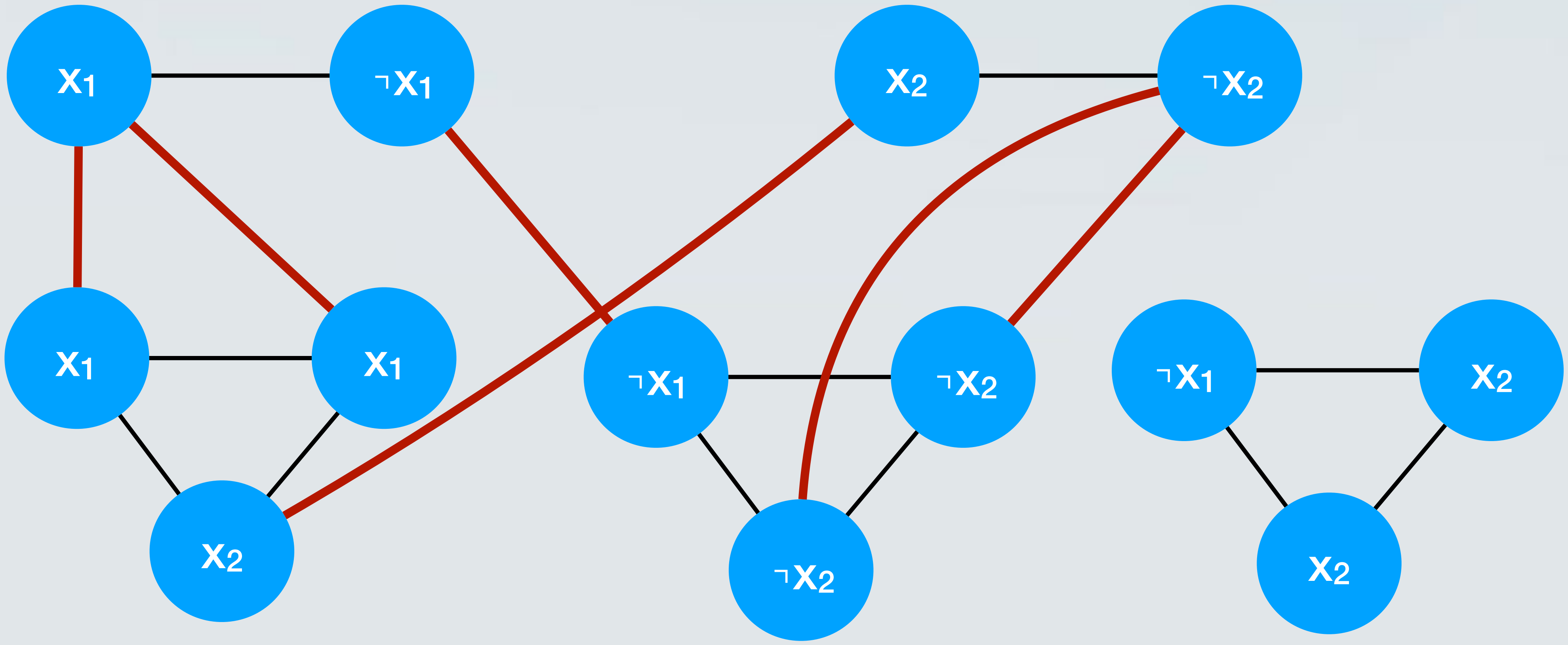
- We add an edge between all nodes with the same label on the top and on the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

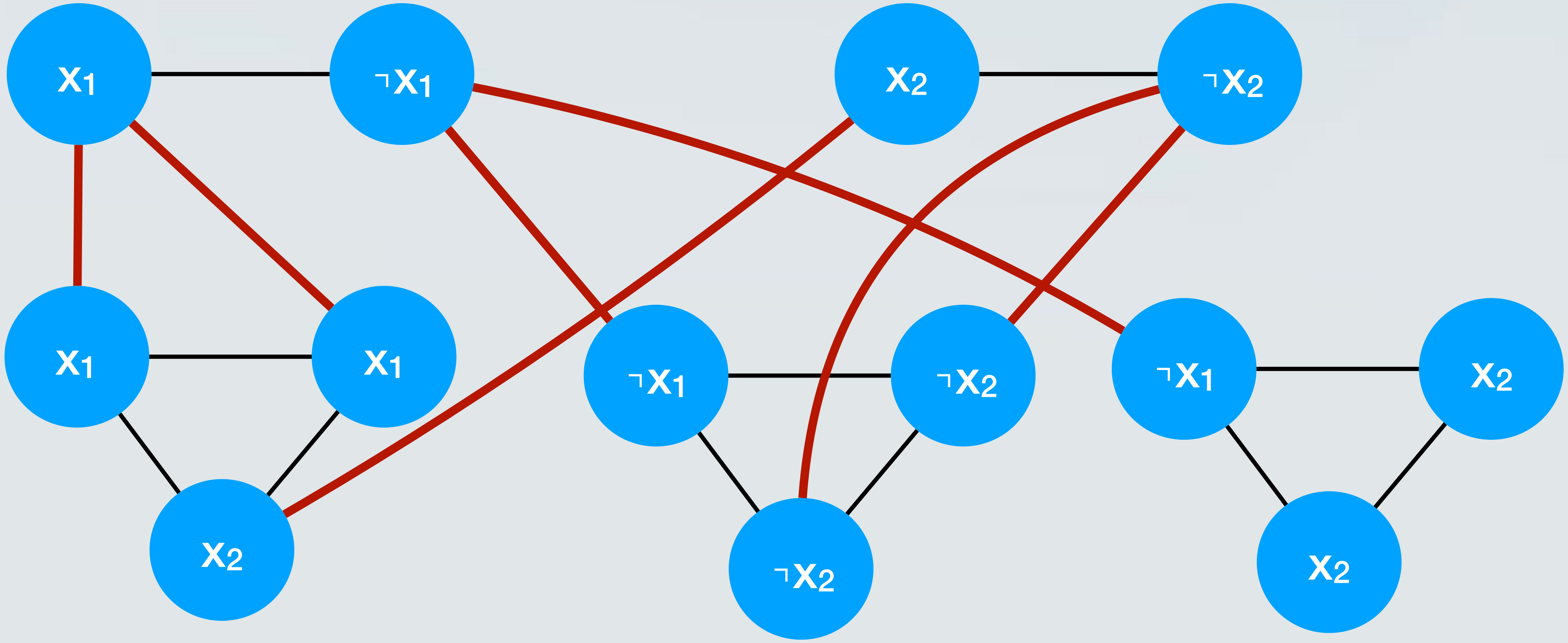
- We add an edge between all nodes with the same label on the top and on the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

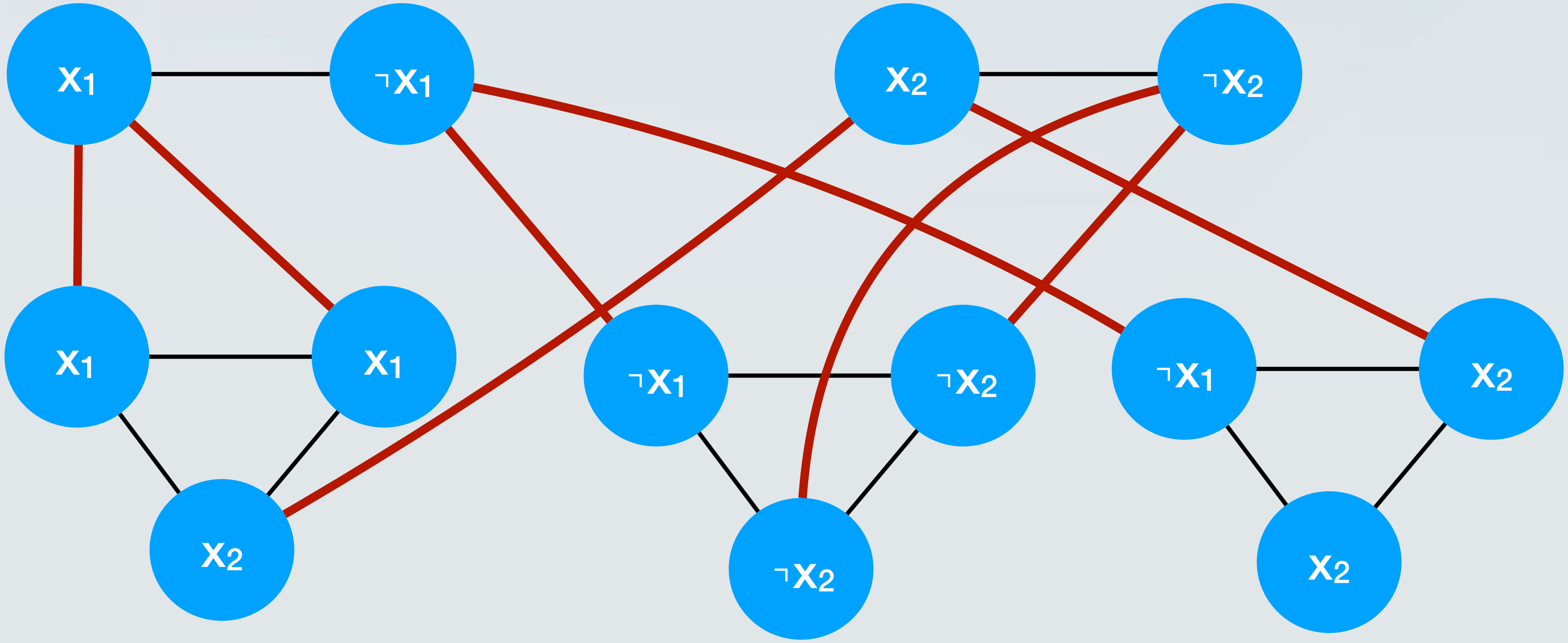
- We add an edge between all nodes with the same label on the top and on the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

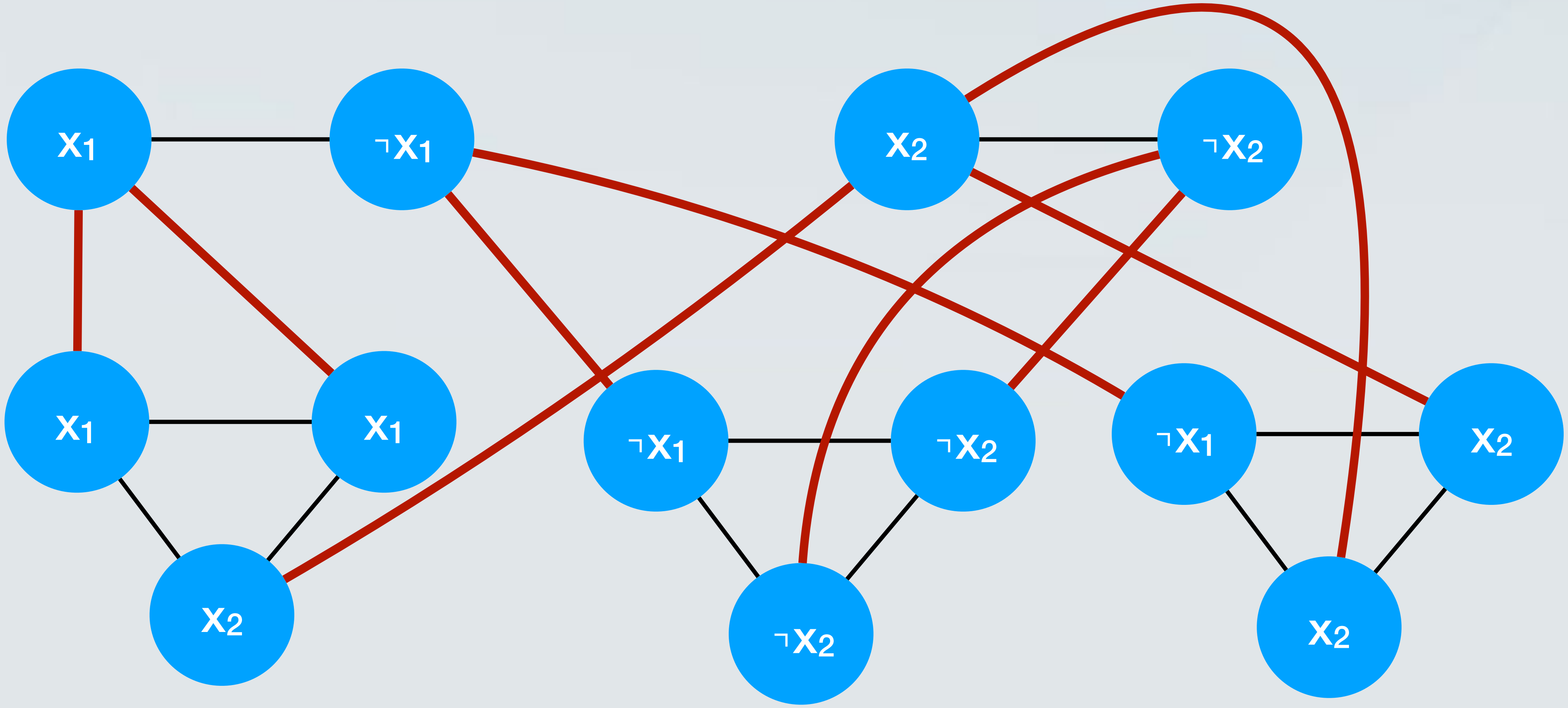
- We add an edge between all nodes with the same label on the top and on the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

- We add an edge between all nodes with the same label on the top and on the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

The reduction

- Let ϕ be a 3-CNF formula with m clauses and d variables.
- We construct, in polynomial time, an instance $\langle G, k \rangle$ of Vertex Cover, with $k = d + 2m$ such that
  - If ϕ is satisfiable => G has a vertex cover of size at most k.
  - If ϕ is not satisfiable => G does not have any vertex cover of size at most k.

One direction

One direction

- If ϕ is satisfiable => G has a vertex cover of size at most k.

One direction

- If ϕ is satisfiable => G has a vertex cover of size at most k.
- Let $(y_1, y_2, \ldots, y_k)$ in $\{0,1\}^n$ be a satisfying assignment for ϕ.

One direction

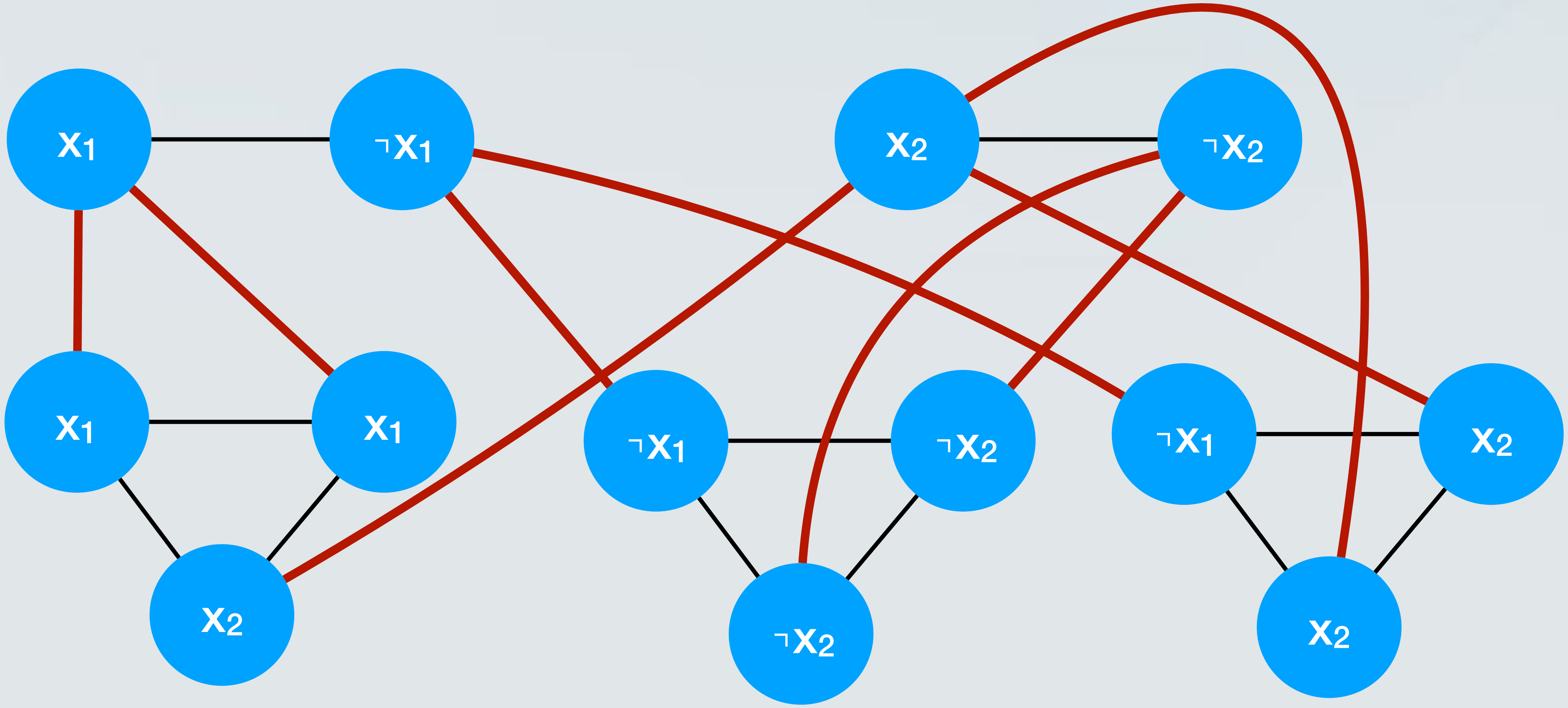
- If ϕ is satisfiable => G has a vertex cover of size at most k.
- Let $(y_1, y_2, \ldots, y_k)$ in $\{0,1\}^n$ be a satisfying assignment for ϕ.
- For the nodes on the top: If $y_i = 1$, include node x_i in the vertex cover C, otherwise, include node $\neg x_i$.

One direction

- If ϕ is satisfiable => G has a vertex cover of size at most k.
- Let $(y_1, y_2 , \dots , y_k)$ in $\{0,1\}^n$ be a satisfying assignment for ϕ.
- For the nodes on the top: If $y_i = 1$, include node x_i in the vertex cover C, otherwise, include node $\neg x_i$.
- For the nodes on the bottom: In each triangle, choose a note x_i that has been picked on the top and do not include it in the vertex cover. Include the other two nodes.

Example

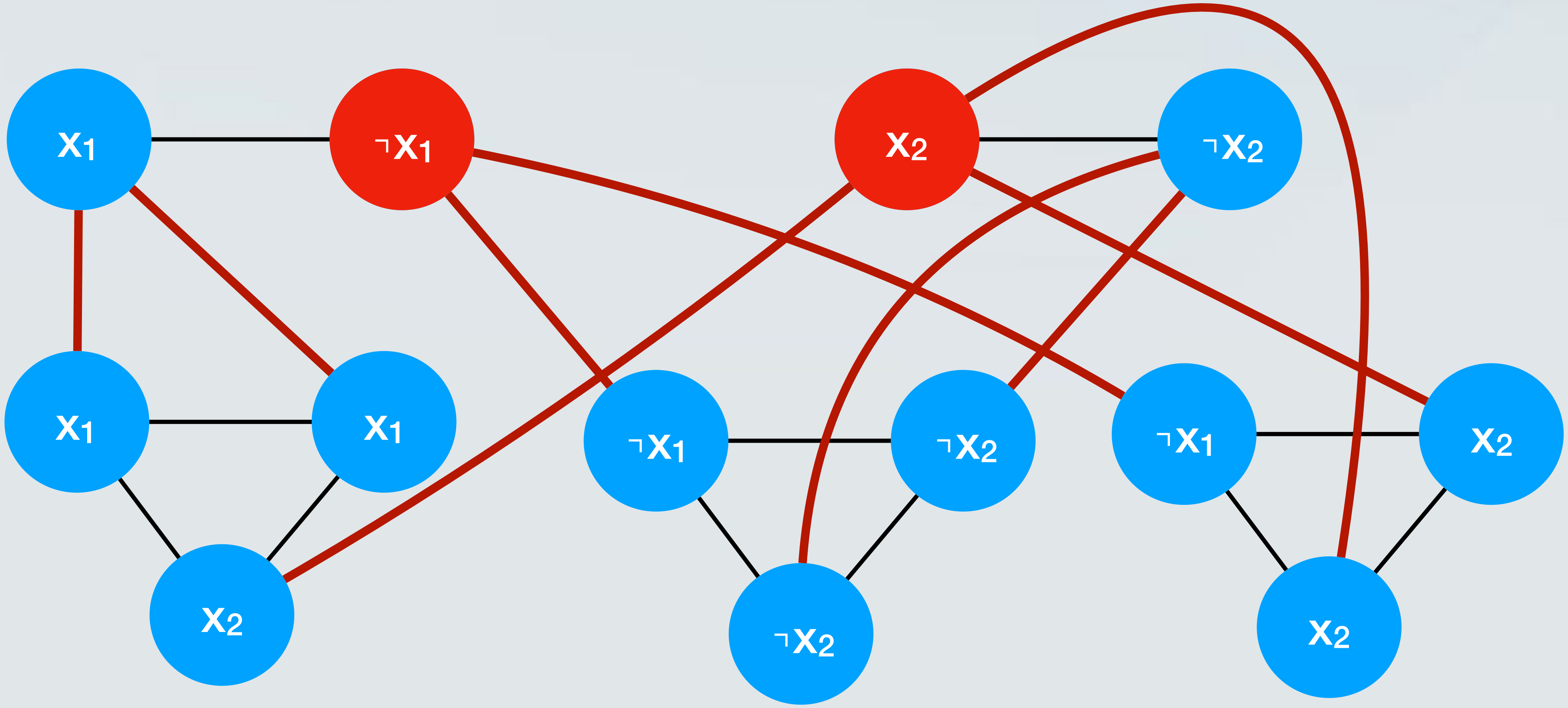
- For the nodes on the top: If $y_i = 1$, include node x_i in the vertex cover C, otherwise, include node $\neg x_i$.
  - Assume $y_1 = 0$, $y_2 = 1$.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Example

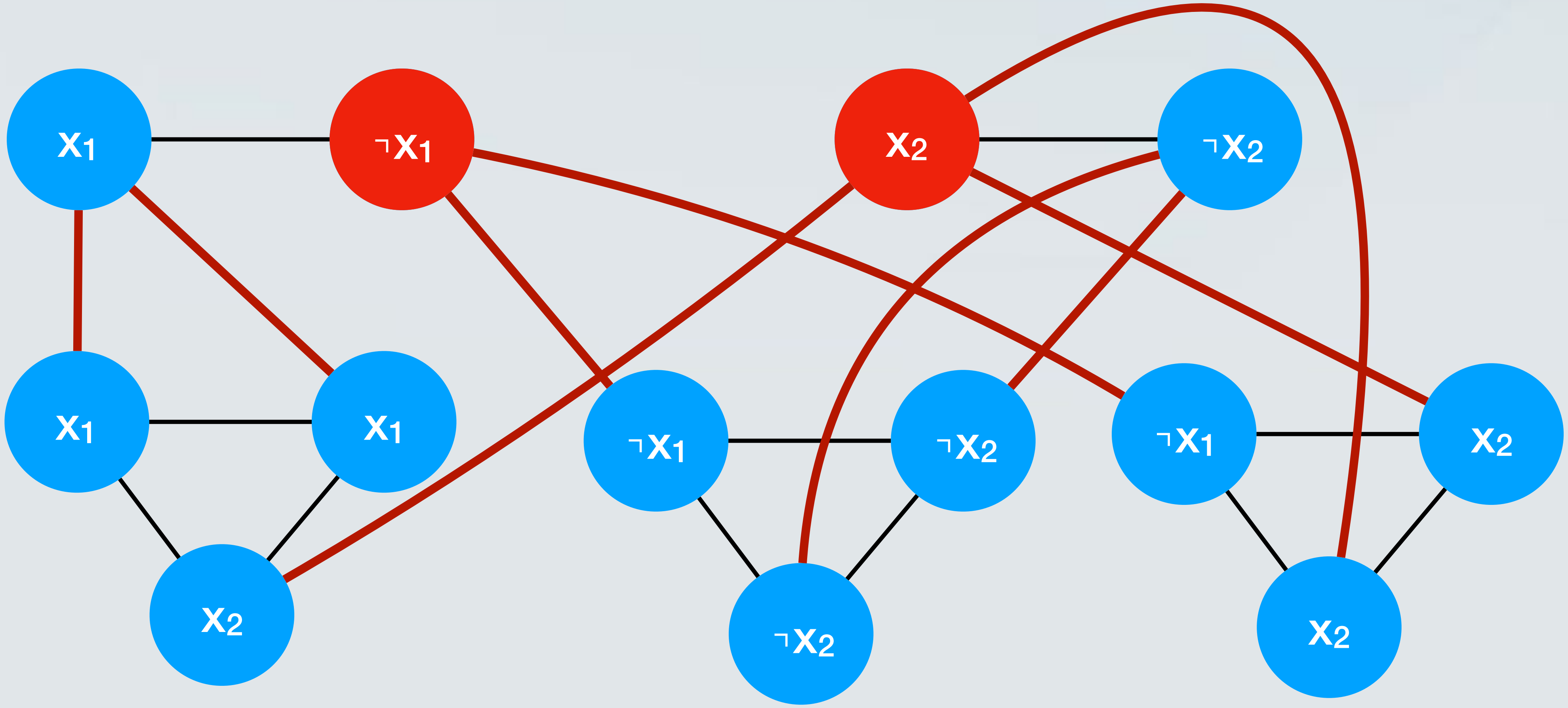
- For the nodes on the top: If $y_i = 1$, include node x_i in the vertex cover C, otherwise, include node $\neg x_i$.
  - Assume $y_1 = 0$, $y_2 = 1$.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Example

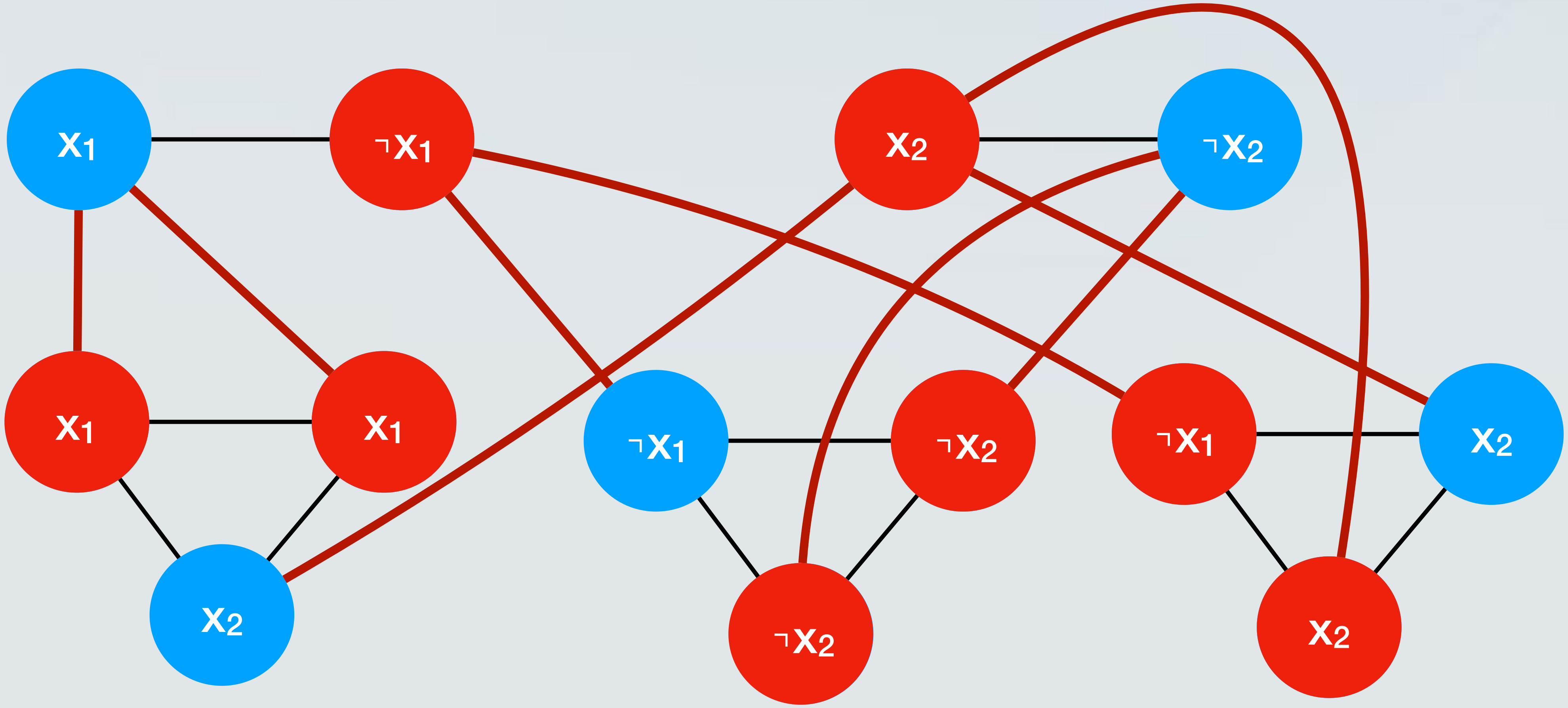
- For the nodes on the bottom: In each triangle, choose a note x_i that has been picked on the top and do not include it in the vertex cover. Include the other two nodes.
  - Assume $y_1 = 0$, $y_2 = 1$.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Example

- For the nodes on the bottom: In each triangle, choose a note x_i that has been picked on the top and do not include it in the vertex cover. Include the other two nodes.
  - Assume $y_1 = 0$, $y_2 = 1$.



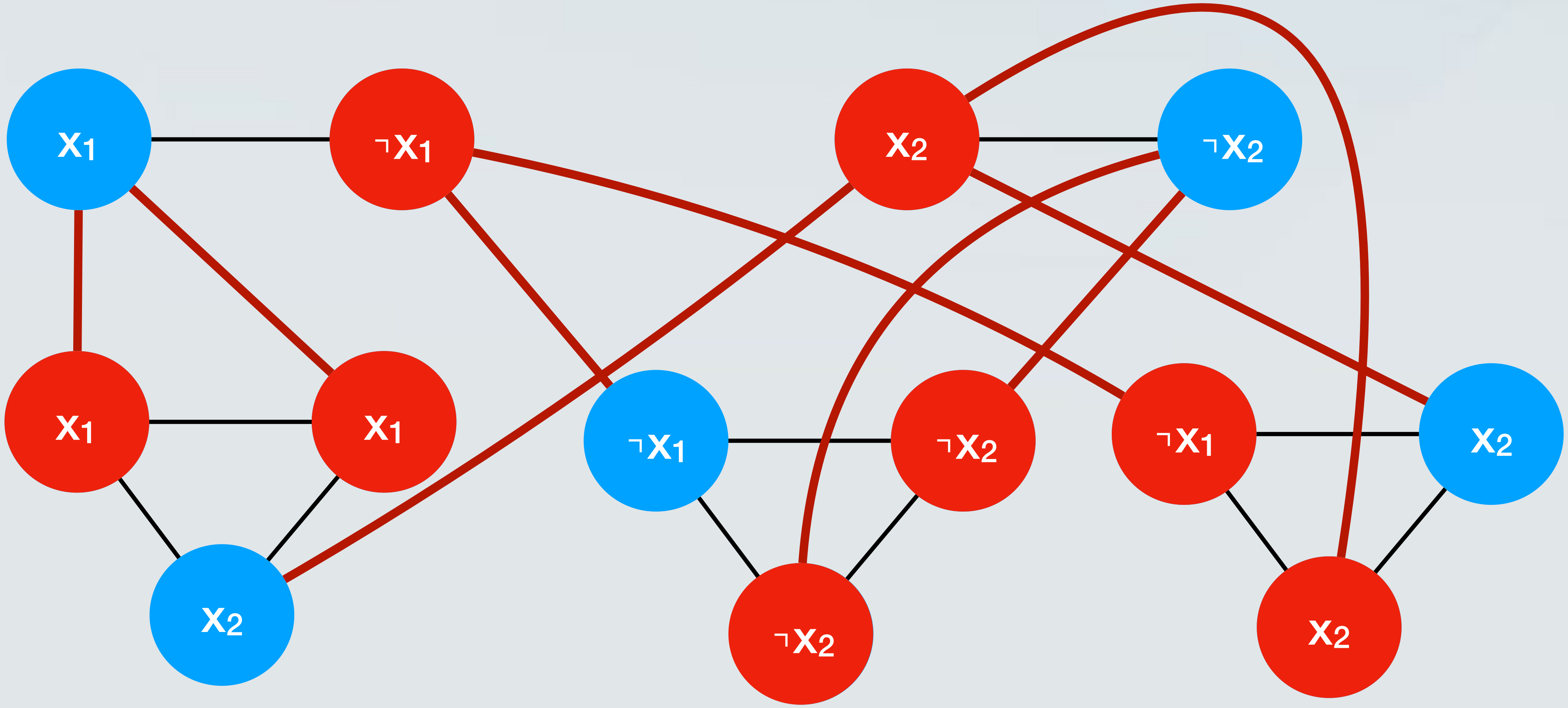
Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

One direction

- Claim: The set of nodes we have chosen is a vertex cover.
    - Every edge on the top is incident to either node x_i or node $\neg x_i$.
    - Every edge on the bottom is incident to some node in the set, since we select two out of three nodes.
    - Every edge between the top and to bottom is incident to some node.

Example

- For the nodes on the bottom: In each triangle, choose a note x_i that has been picked on the top and do not include it in the vertex cover. Include the other two nodes.
    - Assume $y_1 = 0$, $y_2 = 1$.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

One direction

- Claim: The vertex cover has size $k = d + 2m$
  - Each variable is selected at the top (either as x_i or as $\neg x_i$).
  - For each clause, we select two nodes at the bottom.

Other direction

- If ϕ is not satisfiable => G does not have any vertex cover of size at most k.

Other direction

- If ϕ is not satisfiable => G does not have any vertex cover of size at most k.
- G has a vertex cover of size at most k. => ϕ is satisfiable.

Other direction

- G has a vertex cover of size at most k. => ϕ is satisfiable.

Other direction

- G has a vertex cover of size at most k. => ϕ is satisfiable.

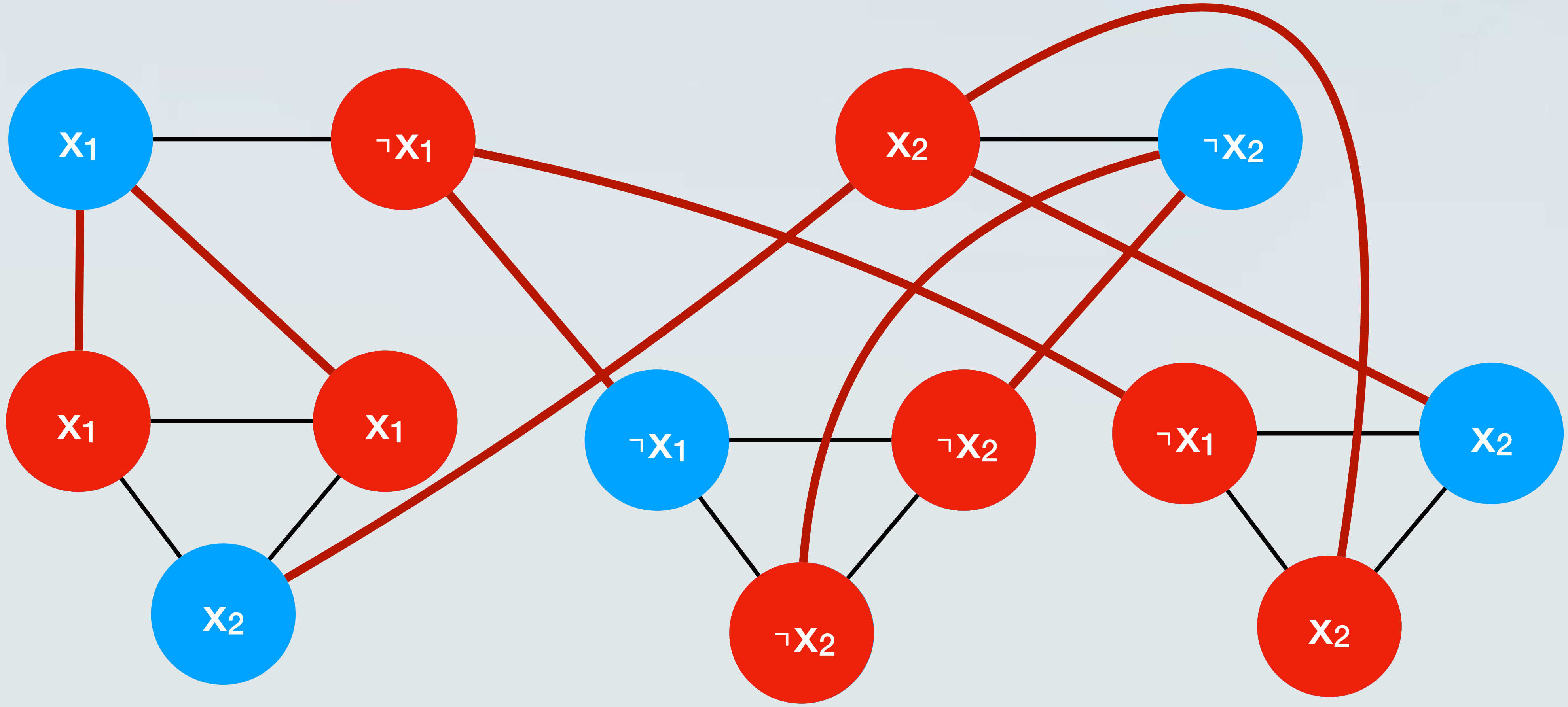
- Let C be a vertex cover of size $k = d + 2m$ in G.

Other direction

- G has a vertex cover of size at most k. => ϕ is satisfiable.
- Let C be a vertex cover of size k = d + 2m in G.
- Since it is a vertex cover, it must include at least two out of three nodes in each “clause gadget” at the bottom.

Example

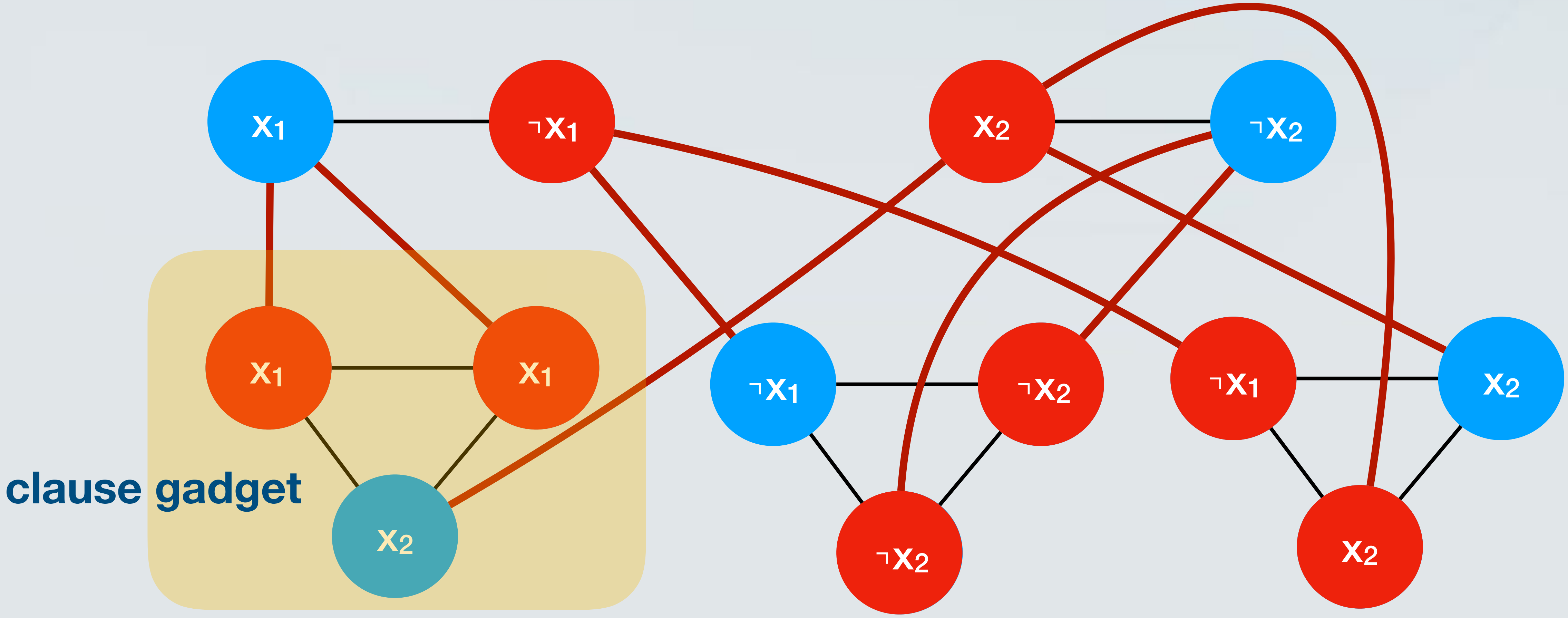
- Since it is a vertex cover, it must include at least two out of three nodes in each "clause gadget" at the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Example

- Since it is a vertex cover, it must include at least two out of three nodes in each "clause gadget" at the bottom.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Other direction

- G has a vertex cover of size at most k. => ϕ is satisfiable.
- Let C be a vertex cover of size $k = d + 2m$ in G.
- Since it is a vertex cover, it must include at least two out of three nodes in each “clause gadget” at the bottom.

Other direction

- G has a vertex cover of size at most k. => ϕ is satisfiable.
- Let C be a vertex cover of size $k = d + 2m$ in G.
- Since it is a vertex cover, it must include at least two out of three nodes in each “clause gadget” at the bottom.
  - This means that at least 2m nodes of C are at the bottom.

Other direction

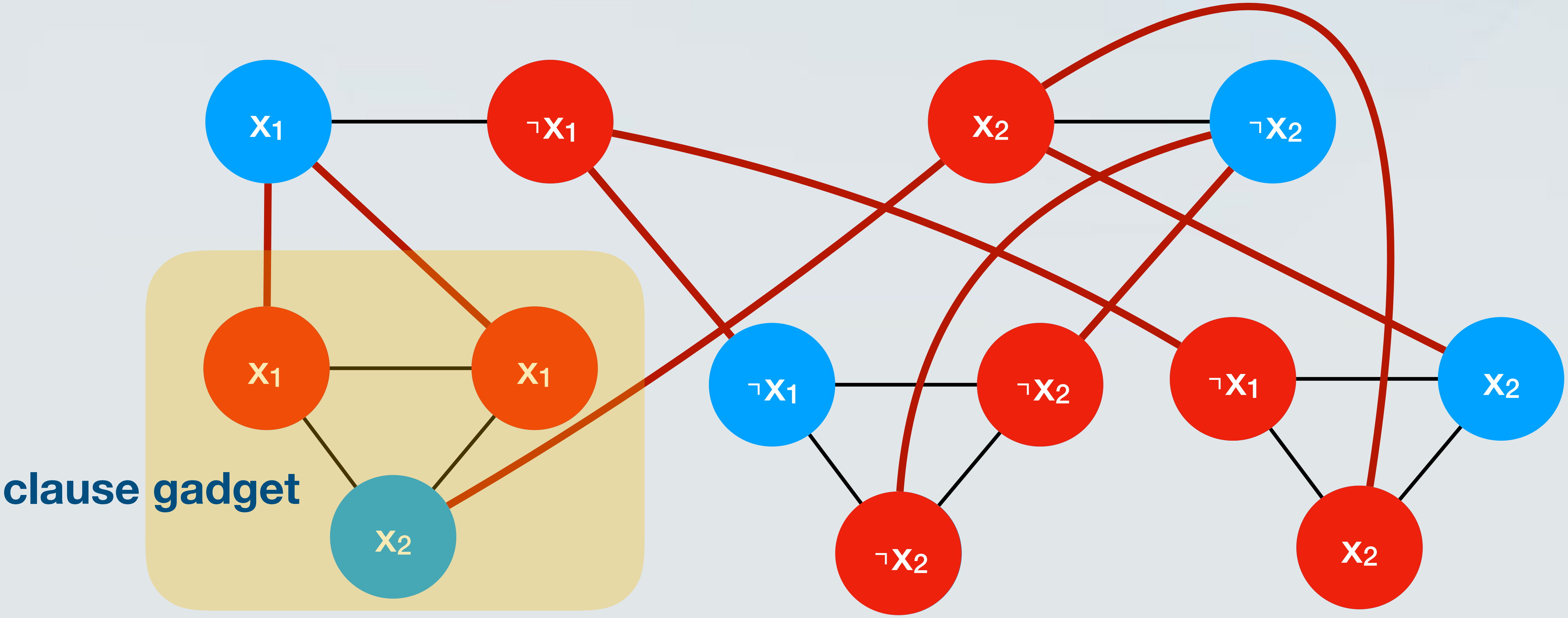
- G has a vertex cover of size at most k. => ϕ is satisfiable.
- Let C be a vertex cover of size $k = d + 2m$ in G.
- Since it is a vertex cover, it must include at least two out of three nodes in each "clause gadget" at the bottom.
  - This means that at least 2m nodes of C are at the bottom.
  - This means that at most d nodes of C are at the top.

Other direction

- This means that at most d nodes of C are at the top.
- To satisfy the edges at the top, in each "variable gadget", at least one node must be included in C.

Example

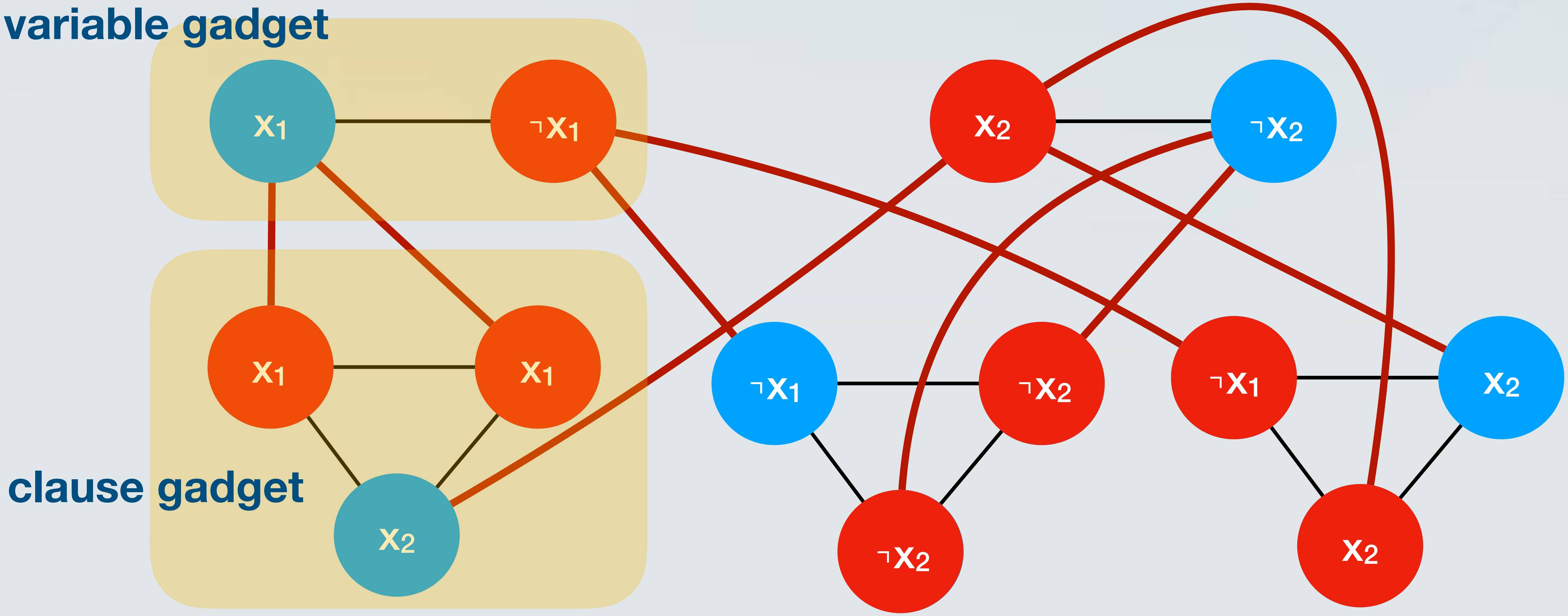
- To satisfy the edges at the top, in each “variable gadget”, at least one node must be included in C.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Example

- To satisfy the edges at the top, in each "variable gadget", at least one node must be included in C.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Other direction

- This means that at most d nodes of C are at the top.
- To satisfy the edges between the top and the bottom, in each “variable gadget”, at least one node must be included in C.

Other direction

- This means that at most d nodes of C are at the top.
- To satisfy the edges between the top and the bottom, in each “variable gadget”, at least one node must be included in C.
- From the two statements above, in each “variable gadget”, exactly one node must be included in C.

Satisfying the formula

Satisfying the formula

- Consider the truth assignment corresponding to the nodes of the vertex cover C on the top (in the variable gadgets).

Satisfying the formula

- Consider the truth assignment corresponding to the nodes of the vertex cover C on the top (in the variable gadgets).

- Note that we either choose x_i or $\neg x_i$ to be 1, but not both.

Satisfying the formula

- Consider the truth assignment corresponding to the nodes of the vertex cover C on the top (in the variable gadgets).

- Note that we either choose x_i or $\neg x_i$ to be 1, but not both.

    - From the statement “in each “variable gadget”, exactly one node must be included in C”.

Satisfying the formula

- Consider the truth assignment corresponding to the nodes of the vertex cover C on the top (in the variable gadgets).
- Note that we either choose x_i or $\neg x_i$ to be 1, but not both.
    - From the statement “in each “variable gadget”, exactly one node must be included in C”.
- Since all “cross” edges are covered, there must be one endpoint on the top (in the “variable gadget”) that is in C.

Satisfying the formula

- Consider the truth assignment corresponding to the nodes of the vertex cover C on the top (in the variable gadgets).

- Note that we either choose x_i or $\neg x_i$ to be 1, but not both.

  - From the statement “in each “variable gadget”, exactly one node must be included in C”.

- Since all “cross” edges are covered, there must be one endpoint on the top (in the “variable gadget”) that is in C.

  - This means that there is one variable of the clause that is set to 1.

Satisfying the formula

- Consider the truth assignment corresponding to the nodes of the vertex cover C on the top (in the variable gadgets).

- Note that we either choose x_i or $\neg x_i$ to be 1, but not both.

    - From the statement “in each “variable gadget”, exactly one node must be included in C”.

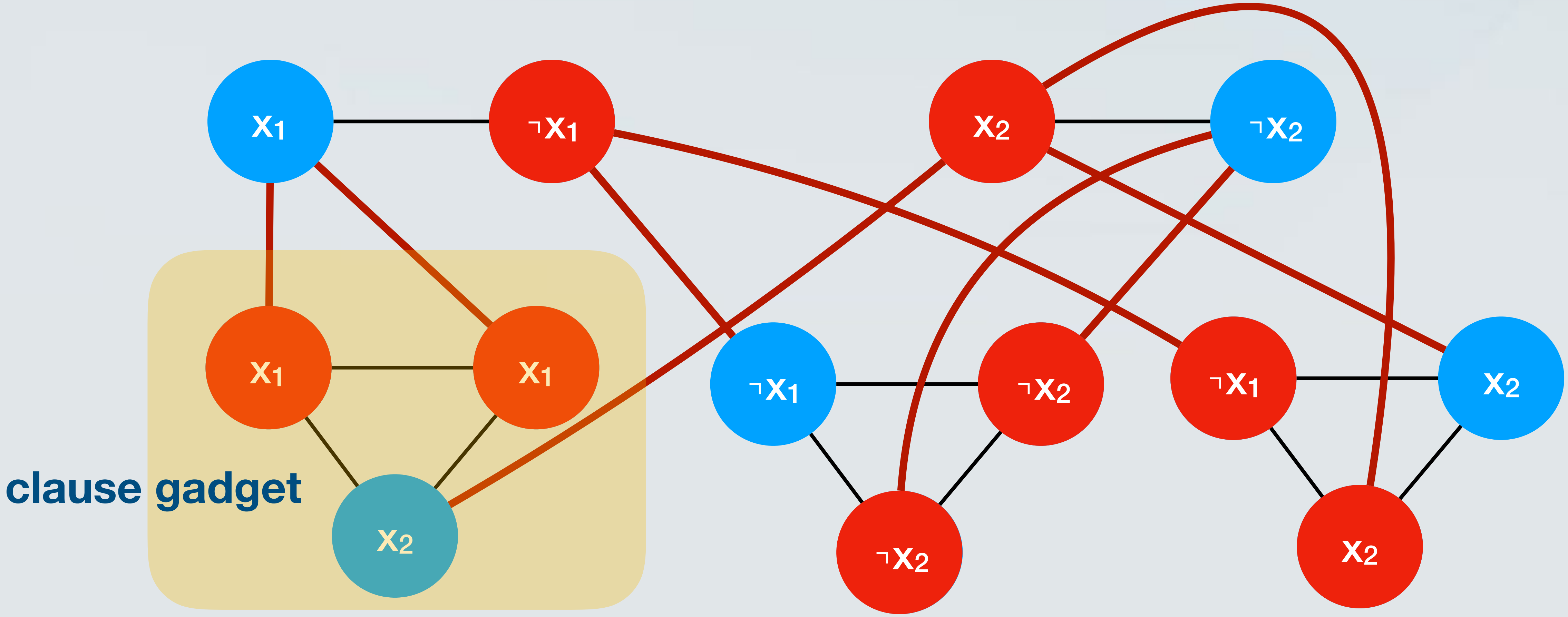
- Since all “cross” edges are covered, there must be one endpoint on the top (in the “variable gadget”) that is in C.

    - This means that there is one variable of the clause that is set to 1.

    - Thus the clause is satisfied.

Example

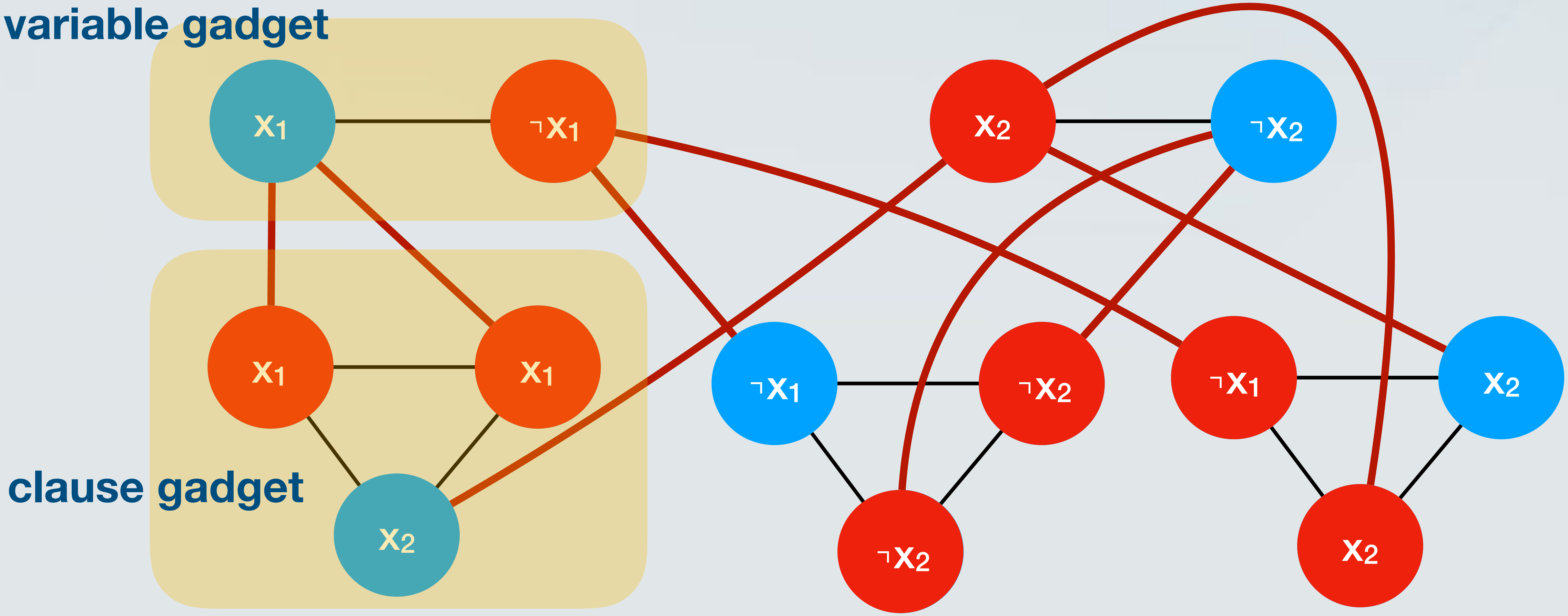
- To satisfy the edges at the top, in each "variable gadget", at least one node must be included in C.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Example

- To satisfy the edges at the top, in each "variable gadget", at least one node must be included in C.



Running example: $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$