# A Domino Game

- Consider these dominos, can you find a permutation of them, so the text on the upper part is exactly the same as the text on the lower part?
(You can use one domino more than once, but not put them upside down.)
  - Example:

| yx | xy | xy |
|---|---|---|
| x | xyyxy | y |

→

| xy | yx | yx | xy |
|---|---|---|---|
| xyyxy | x | x | y |

**xyyxyxxy**

- Set 1:

| ab | b | aba | aa |
|---|---|---|---|
| abab | a | b | a |

- Set 2:

| 0 | 101 | 010 | 0111 |
|---|---|---|---|
| 01 | 010 | 101 | 1 |

# Algorithms for Decision Support

## NP-Completeness (3/3)

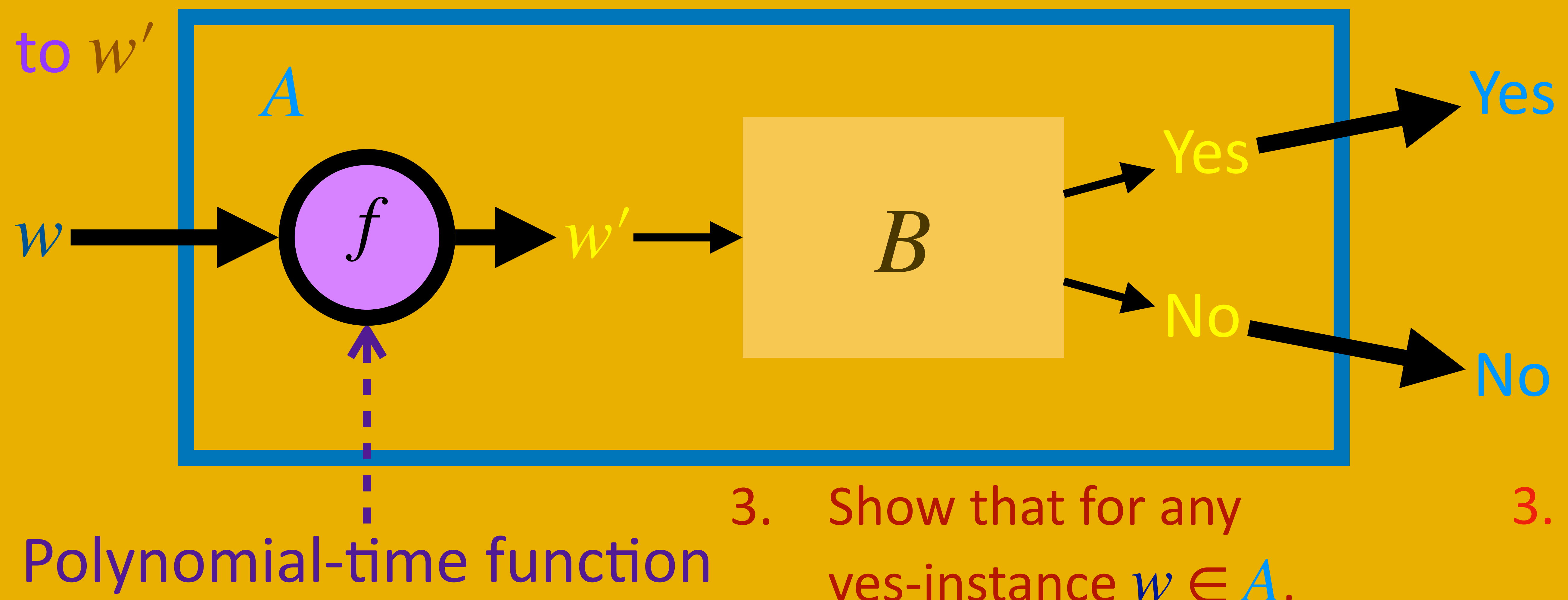
Optimization problems

# Polynomial-Time Reduce $A$ to $B$

- Problem $A$ with input $w$
  - Return yes if $w \in A$
  - Return no if $w \notin A$

- Problem $B$ with input $w'$
  - Return yes if $w' \in B$
  - Return no if $w' \notin B$

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time




2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$

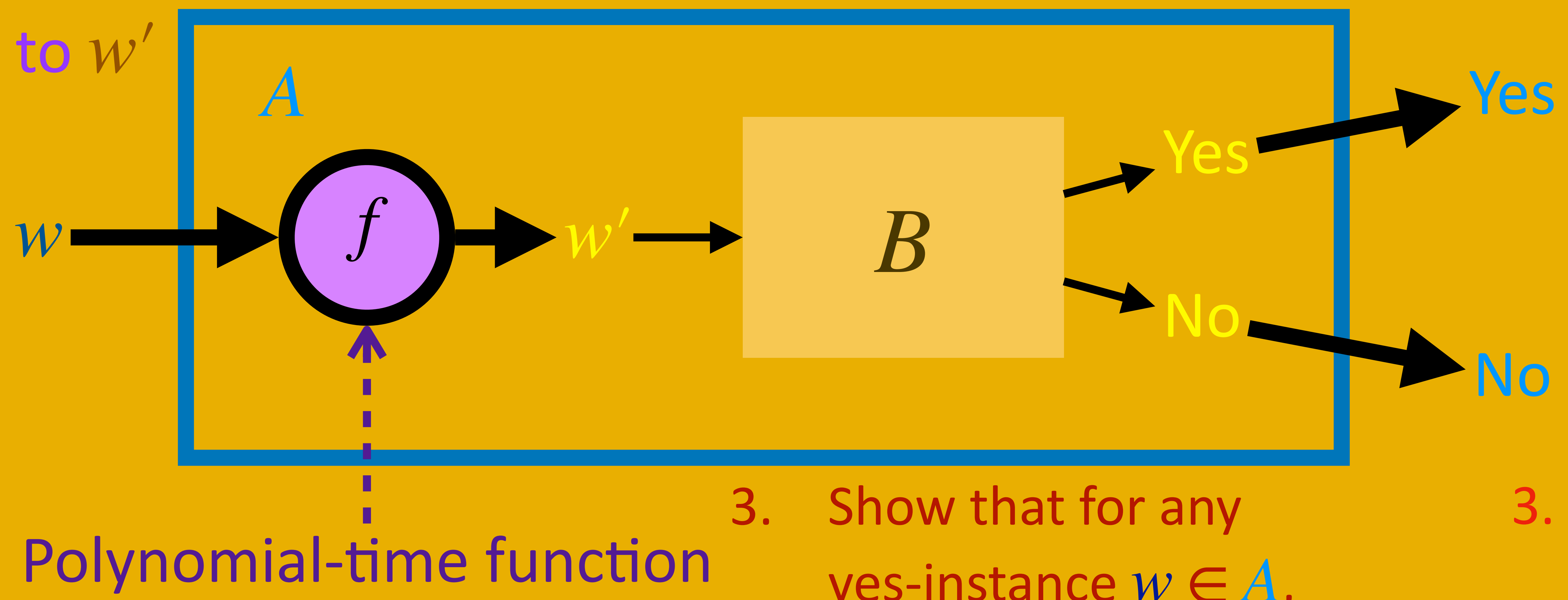
3. Show that for any yes-instance $w \in A$, the corresponding instance $w'$ is also a yes-instance of $B$

3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

# Instance Transformation

- Design a method to transform any instance $w$ of $A$ into an instance $w'$ of $B$
  - The transformation should be done in polynomial time

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time




2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$

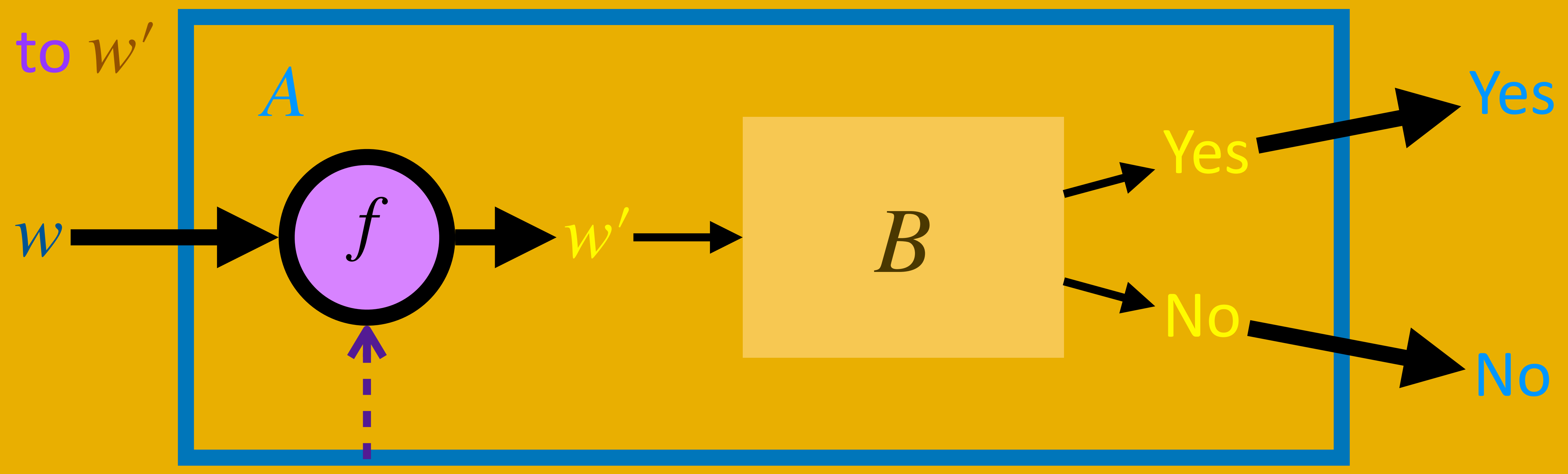
3. Show that for any yes-instance $w \in A$, the corresponding instance $w'$ is also a yes-instance of $B$

3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

# Show that the reduction *works*

- That is, $w$ is a yes-instance of $A$ if and only if $w'$ is a yes-instance to $B$
  - So we can rely on the yes/no answer of $w' \in B$ to decide if $w \in A$

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time




Polynomial-time function

2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$

3. Show that for any yes-instance $w \in A$, the corresponding instance $w'$ is also a yes-instance of $B$

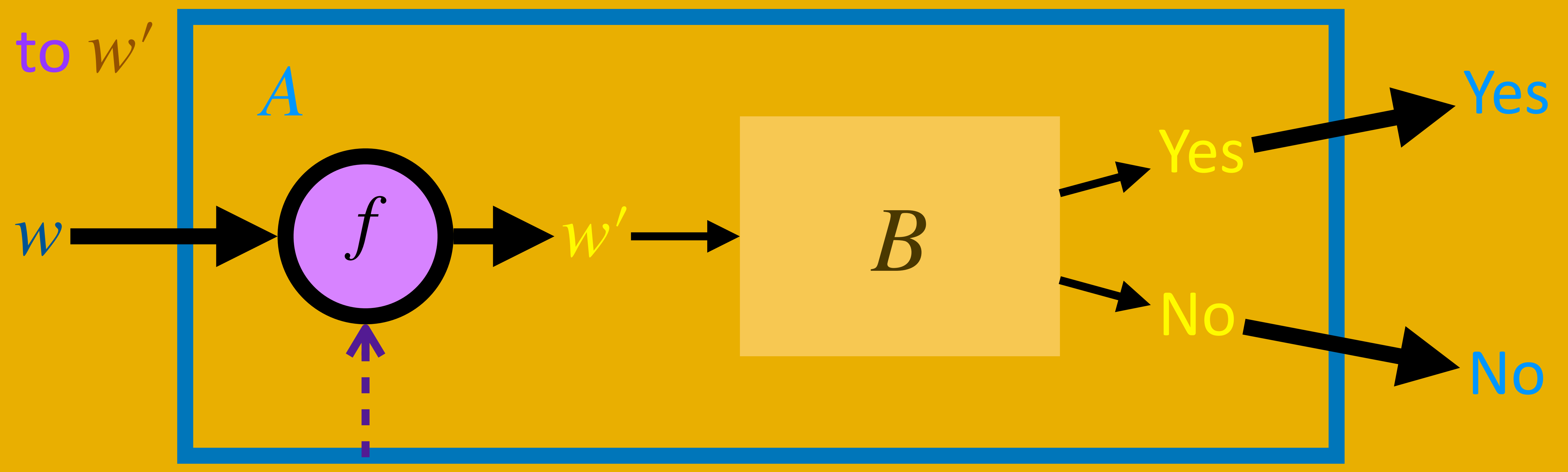
3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

# Polynomial-Time Reduce $A$ to $B$

- Problem $A$ with input $w$
  - Return yes if $w \in A$
  - Return no if $w \notin A$

- Problem $B$ with input $w'$
  - Return yes if $w' \in B$
  - Return no if $w' \notin B$

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time




Polynomial-time function

2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$

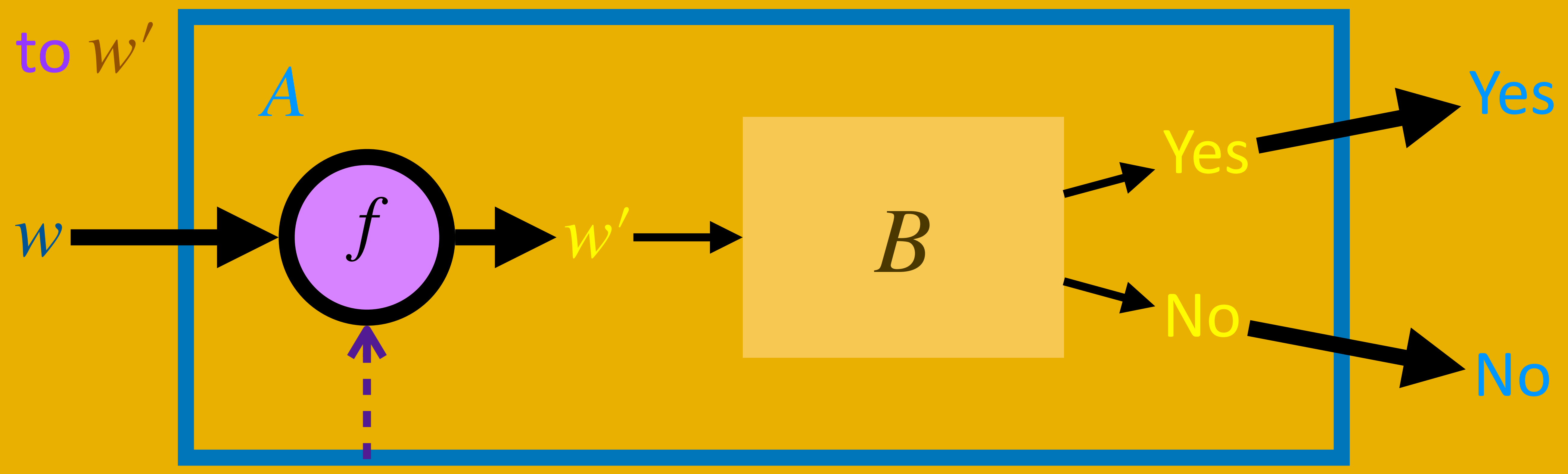
3. Show that for any yes-instance $w \in A$, the corresponding instance $w'$ is also a yes-instance of $B$

3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

# Instance Transformation

- Design a method to transform any instance $w$ of $A$ into an instance $w'$ of $B$
  - The transformation should be done in polynomial time

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time




Polynomial-time function

2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$

3. Show that for any yes-instance $w \in A$, the corresponding instance $w'$ is also a yes-instance of $B$

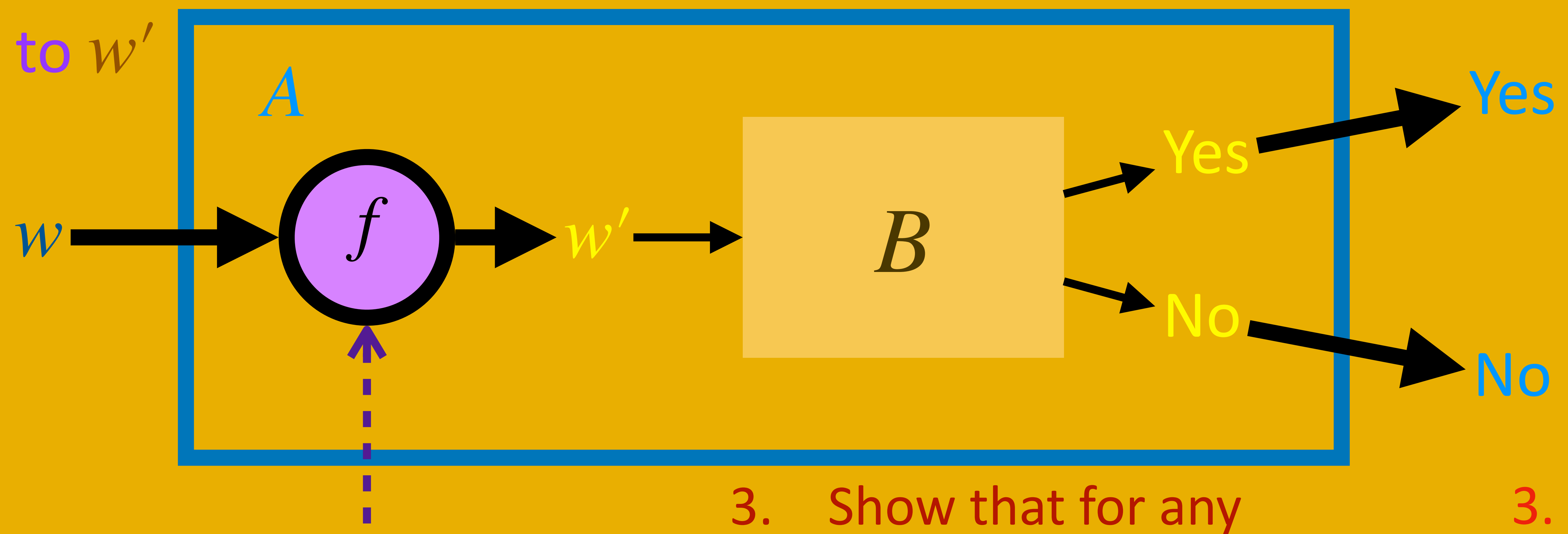
3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

# Show that the reduction *works*

- That is, $w$ is a yes-instance of $A$ if and only if $w'$ is a yes-instance to $B$
  - So we can rely on the yes/no answer of $w' \in B$ to decide if $w \in A$

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time

2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$




Polynomial-time function

3. Show that for any yes-instance $w \in A$, the corresponding instance $w'$ is also a yes-instance of $B$

3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

# Show that the reduction *works*

- That is, $w$ is a yes-instance of $A$ if and only if $w'$ is a yes-instance to $B$
  - So we can rely on the yes/no answer of $w' \in B$ to decide if $w \in A$

- Argue that:
  - If $w$ is a yes-instance of $A$, there is a solution $S_w$ to $w$
  - Using $S_w$, we can construct a solution $S_{w'}$ to $w'$
    - Argue by how we construct $w'$

➡ $w' \in B$

# Outline

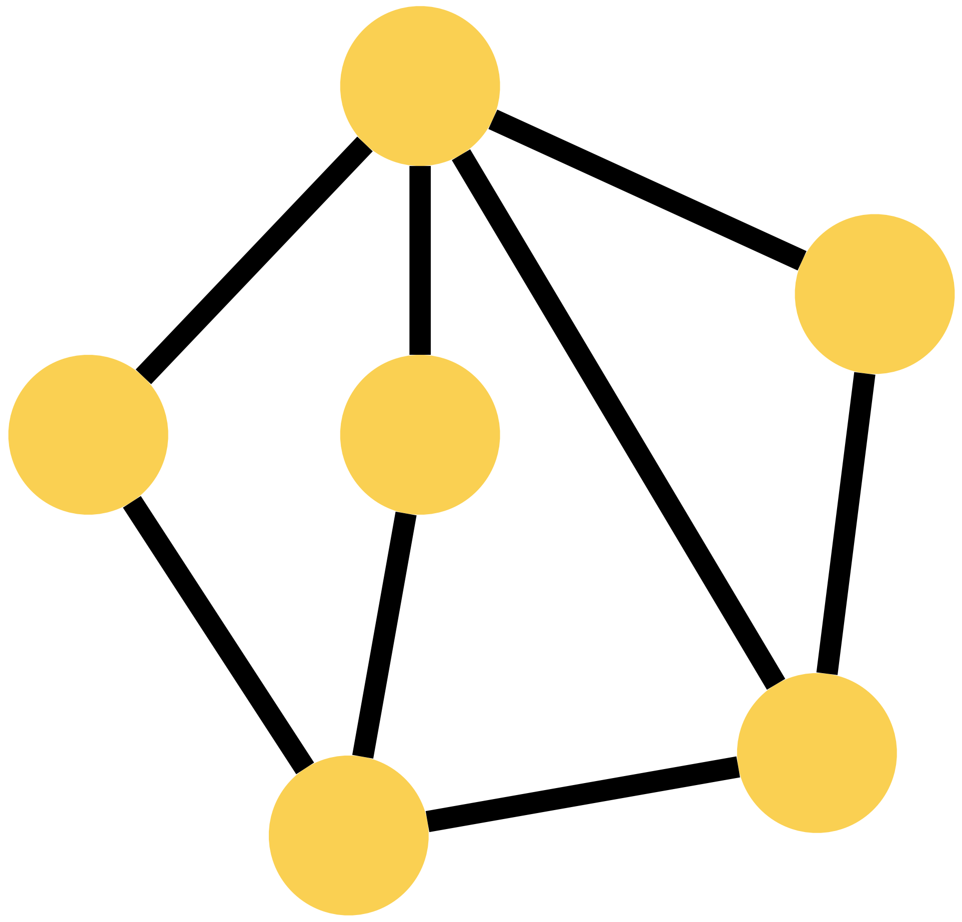
- More NP-Hardness proofs
  - 3SAT $\leq_p$ VERTEX-COVER
  - VERTEX-COVER $\leq_p$ INDEPENDENT SET
  - VERTEX-COVER $\leq_p$ FEEDBACK-VERTEX-SET
  - VERTEX-COVER $\leq_p$ Integer Linear Program
- Pseudo-polynomial time algorithms
- **NP** and **Co-NP**
- Turing undecidable languages

# Outline

- More NP-Hardness proofs
  - 3SAT $\leq_p$ VERTEX-COVER
  - VERTEX-COVER $\leq_p$ INDEPENDENT SET
  - VERTEX-COVER $\leq_p$ FEEDBACK-VERTEX-SET
  - VERTEX-COVER $\leq_p$ Integer Linear Program
- Pseudo-polynomial time algorithms
- **NP** and **Co-NP**
- Turing undecidable languages

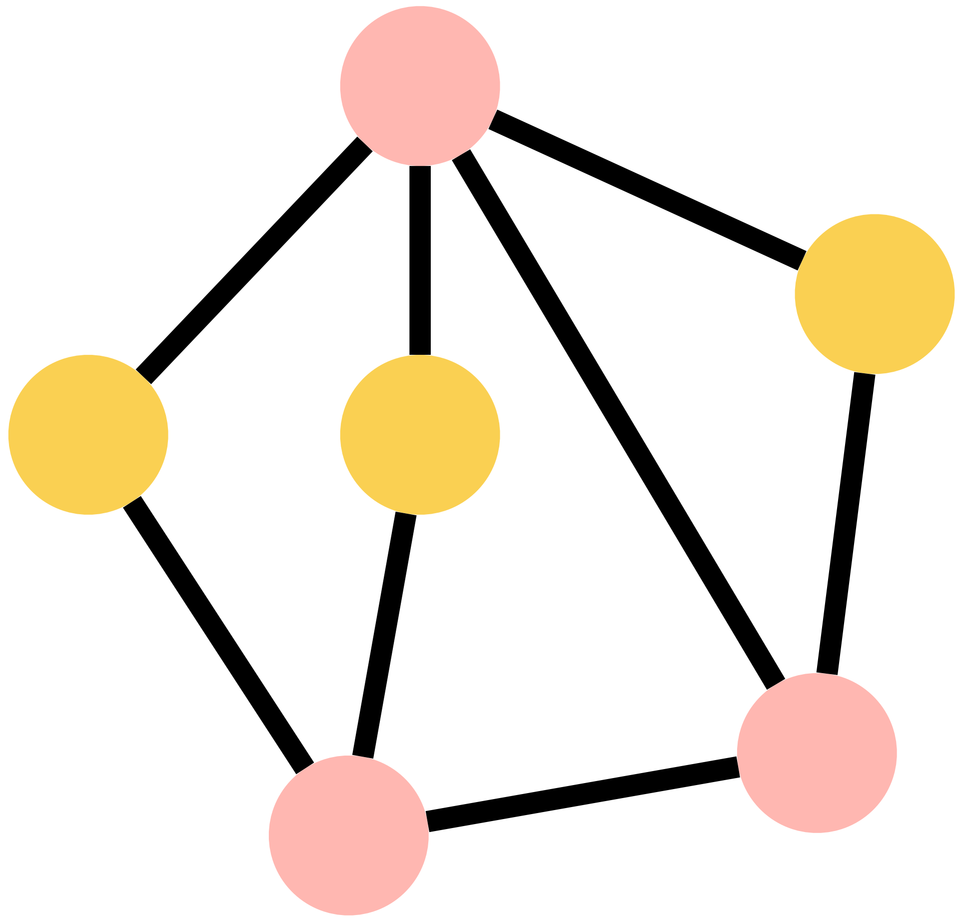
# Vertex Cover

- Given a graph $G = (V, E)$, a *vertex cover* is a subset $U$ of vertices such that for every edge $(u, v)$, $|\{u, v\} \cap U| \geq 1$
  - That is, every edge is *covered* by at least one of its endpoints
  - Removing all vertices in $U$ leaves no edge
    - When a vertex is removed, all the edges incident to it are also removed



# Vertex Cover

- Given a graph $G = (V, E)$, a *vertex cover* is a subset $U$ of vertices such that for every edge $(u, v)$, $|\{u, v\} \cap U| \geq 1$
  - That is, every edge is *covered* by at least one of its endpoints
  - Removing all vertices in $U$ leaves no edge
    - When a vertex is removed, all the edges incident to it are also removed



# Vertex Cover

- Given a graph $G = (V, E)$, a *vertex cover* is a subset $U$ of vertices such that for every edge $(u, v)$, $|\{u, v\} \cap U| \geq 1$
  - That is, every edge is *covered* by at least one of its endpoints
  - Removing all vertices in $U$ leaves no edge
    - When a vertex is removed, all the edges incident to it are also removed

# Vertex Cover

- Minimum vertex cover problem: Given a graph $G$, what is the size of the minimum vertex cover in $G$?

- Decision version: Given a graph $G$, is there a vertex cover of size at most $k$ in $G$?
  - An instance of VERTEX-COVER is $\langle \langle G \rangle, k \rangle$

New parameter!

- VERTEX-COVER is NP-complete

# 3SAT $\leq_p$ VERTEX-COVER

# 3SAT $\leq_p$ VERTEX-COVER

- 3SAT $= \{\langle\phi\rangle \mid \phi$ is a satisfiable 3cnf-formula$\}$
- VERTEX-COVER $= \{\langle G, k\rangle \mid$ graph $G$ has a vertex cover of size at most $k\}$

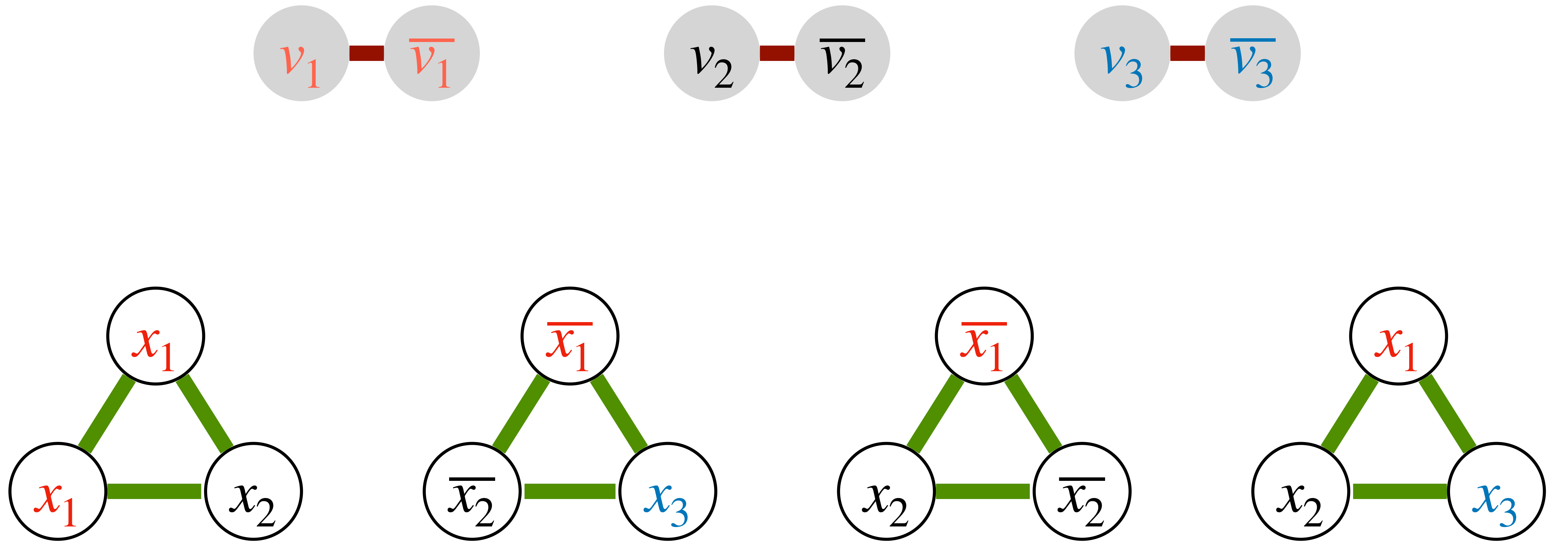
# 3SAT $\leq_p$ VERTEX-COVER

- Construction:
  - For each variable $x_i$, there are two vertices $v_i$ and $\overline{v_i}$ forming an edge in $G$
  - For each clause $(l_a, l_b, l_c)$, there is a triangle in $G$
  - For each clause $(l_a, l_b, l_c)$, there are three edges from $l_a, l_b,$ and $l_c$ to the corresponding variable vertex
  - $k = 2 \cdot$ number of clauses $+$ number of vertices

# 3SAT $\leq_p$ VERTEX-COVER

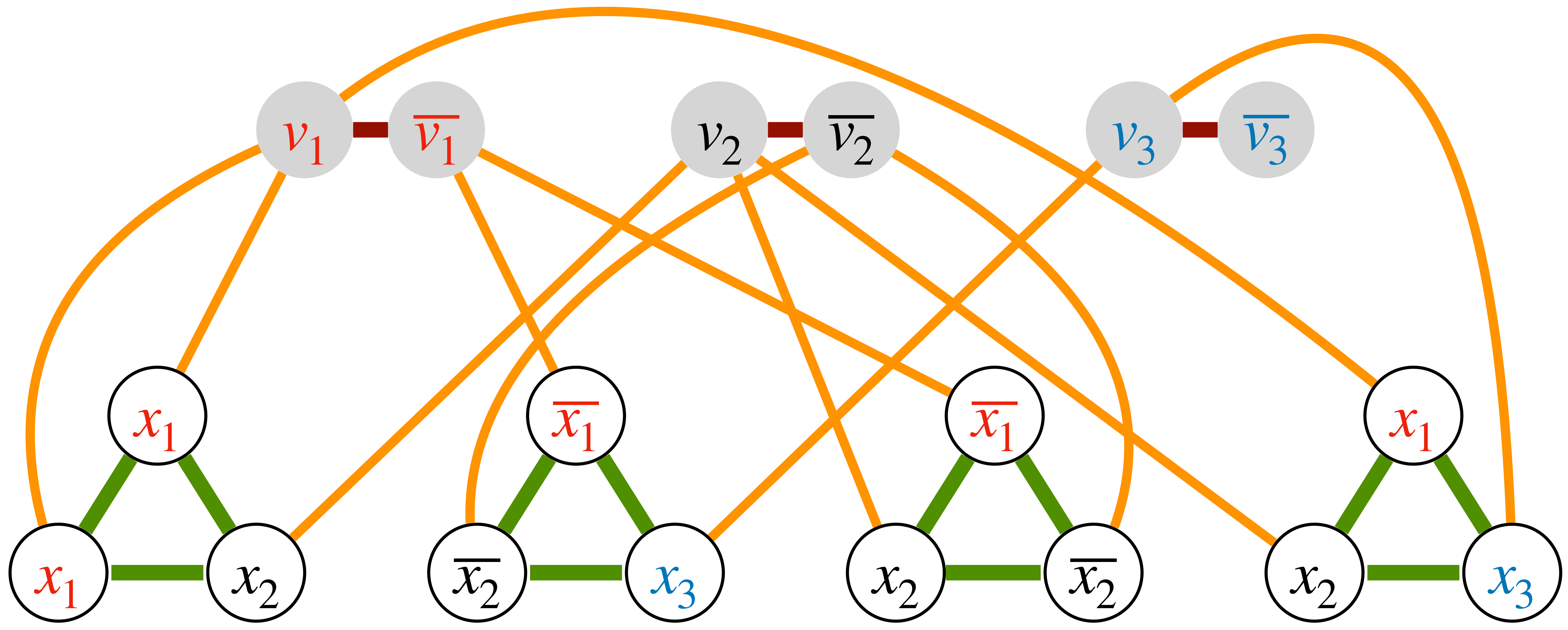
- Construction:
  - For each variable $x_i$, there are two vertices $v_i$ and $\overline{v_i}$ forming an edge in $G$
  - For each clause $(l_a, l_b, l_c)$, there is a triangle in $G$
  - For each clause $(l_a, l_b, l_c)$, there are three edges from $l_a, l_b,$ and $l_c$ to the corresponding variable vertex
  - $k = 2 \cdot$ number of clauses + number of vertices

$\phi = \{(x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_2}) \wedge (x_1 \vee x_2 \vee x_3)$



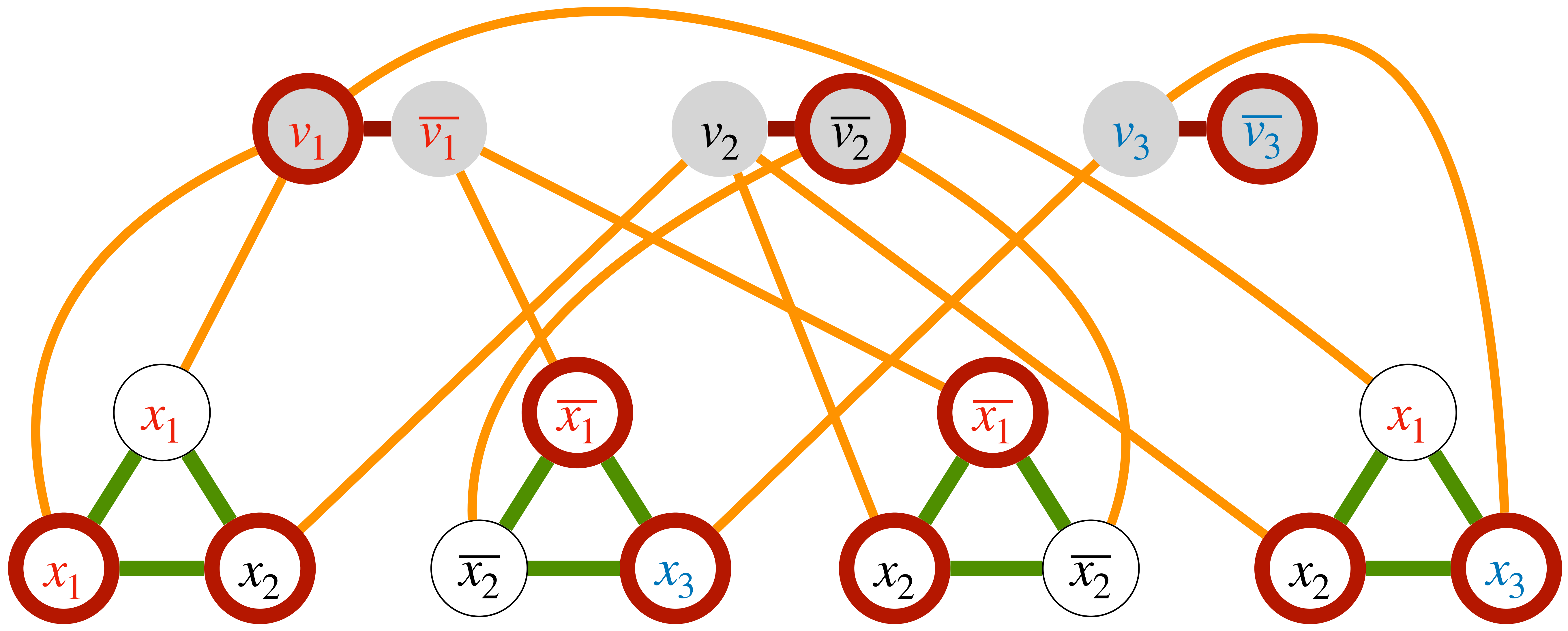
# 3SAT $\leq_p$ VERTEX-COVER

$$\phi = \{(x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_2}) \wedge (x_1 \vee x_2 \vee x_3)$$



# 3SAT $\leq_p$ VERTEX-COVER

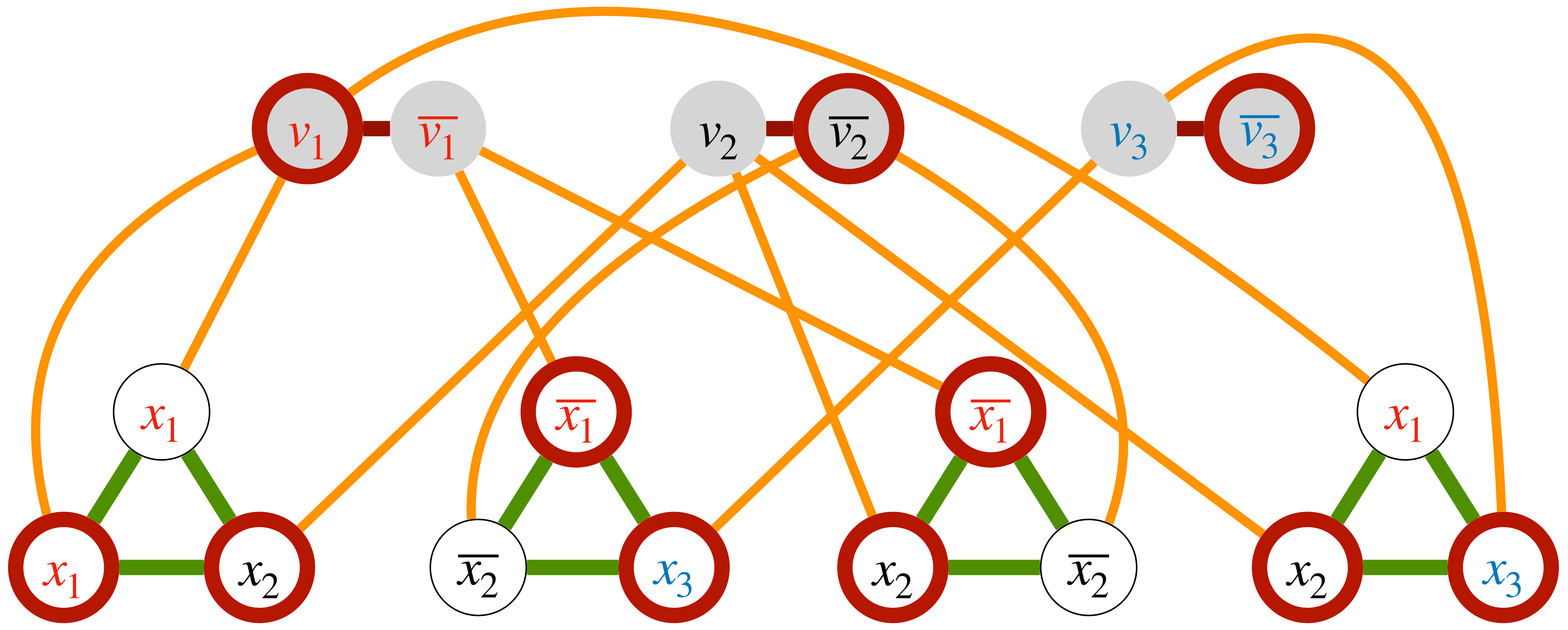
$\phi = \{(x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_2}) \wedge (x_1 \vee x_2 \vee x_3)$



# 3SAT $\leq_p$ VERTEX-COVER

- $\phi$ is satisfiable $\Rightarrow$ there is a truth assignment that makes $\phi$ TRUE
  - Pick the variable vertices corresponding to the true literals
    - Since the truth assignment is valid, every variable-variable edge is covered, and each clause has at least one variable-clause edge is covered
  - Pick the vertices in each clause that incident to the not covered variable-clause edges
    - There are at most two these vertices in each classes. They cover the clause edges
  - There are at most $\ell$+$2k$ picked vertices and they form a vertex cover

$$\phi = \{(\overset{\mathbf{T}}{x_1} \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overset{\mathbf{T}}{\overline{x_2}} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overset{\mathbf{T}}{\overline{x_2}}) \wedge (\overset{\mathbf{T}}{x_1} \vee x_2 \vee x_3)$$

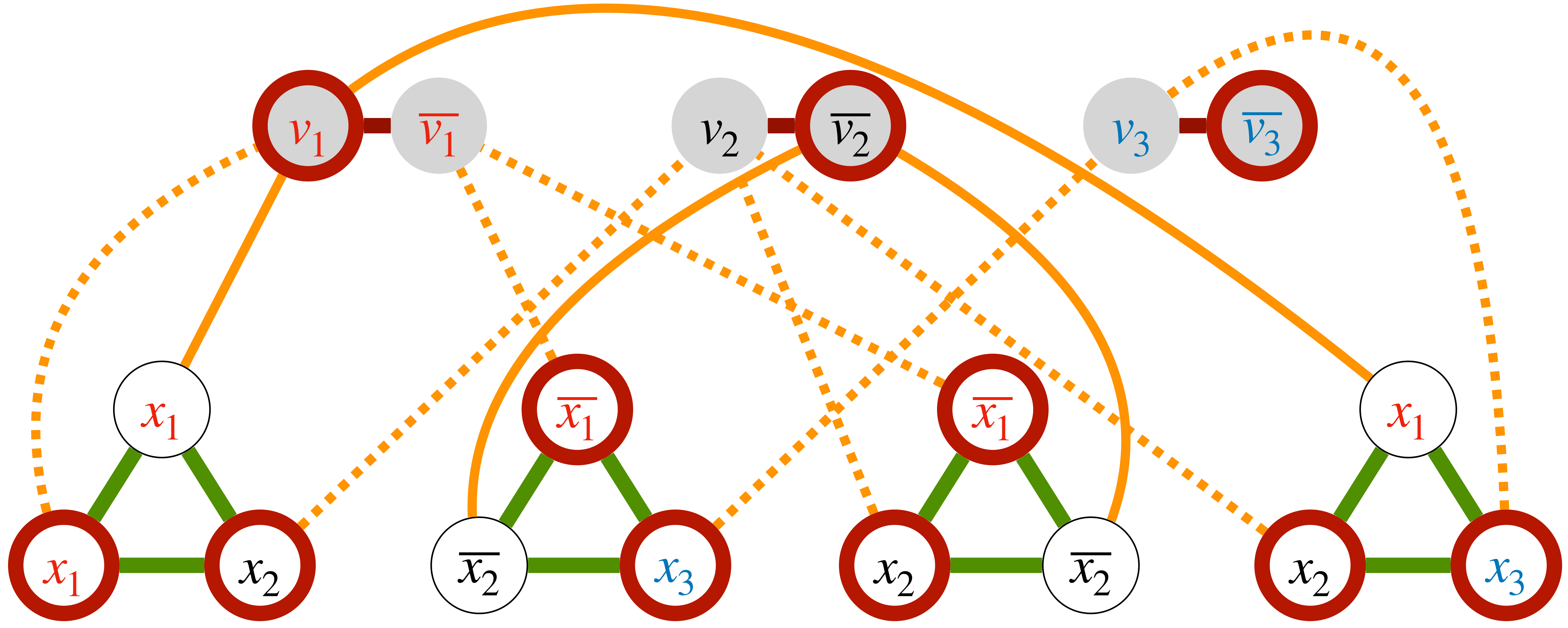


# 3SAT $\leq_p$ VERTEX-COVER

- $G$ has a vertex cover $C$ of cardinality $2k + \ell$ $\Rightarrow \phi$ is satisfiable
  - Since each triangle needs two vertices to cover it, each clause has at least two vertices picked ( $\geq 2k$)
    - They can only cover at most two variable-clause edges
  - Since the variable-variable edges need to be covered, at least one vertex in each pair of $v_i$ and $\overline{v_i}$ is in $C$ ( $\geq \ell$)
    - These vertices cover the variable-clause edges that are not yet covered
    - Pick the corresponding literals to be true, the covered variable-clause edges implies that every clause is true

T T T T

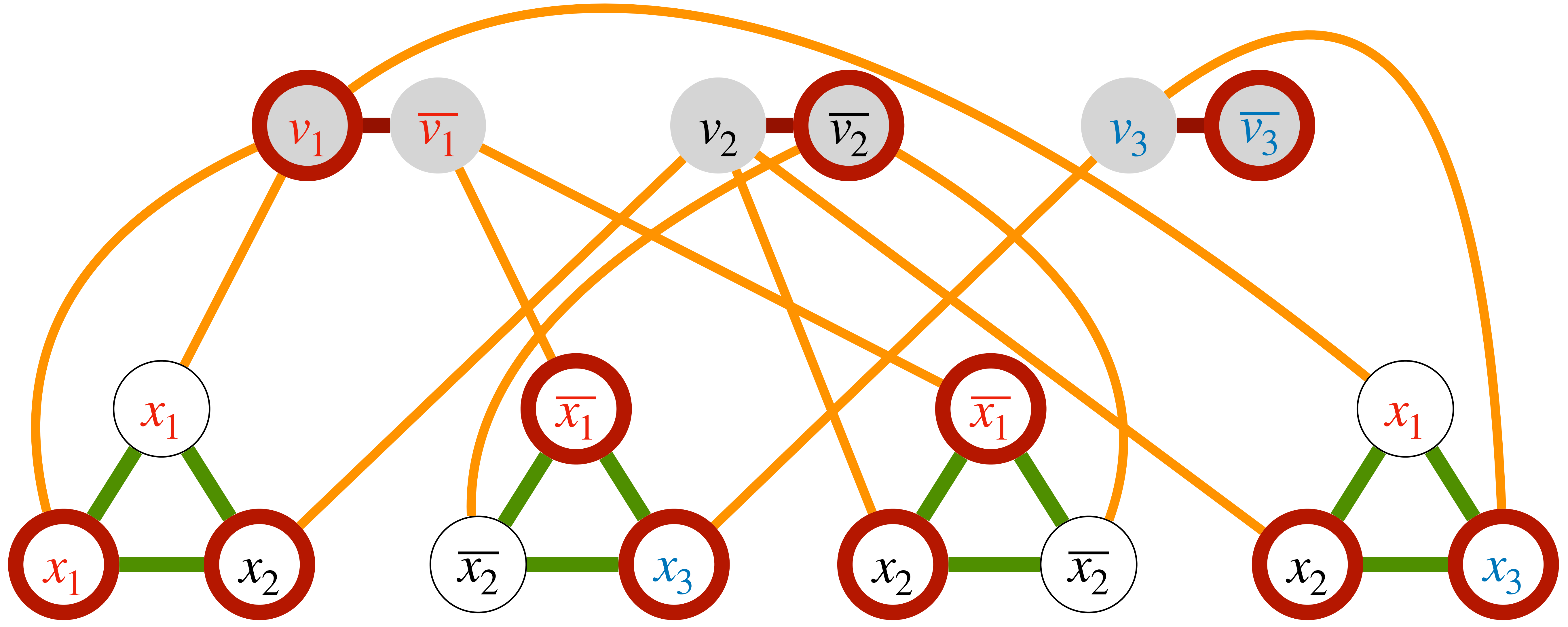
$$\phi = \{(x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_2}) \wedge (x_1 \vee x_2 \vee x_3)$$



# 3SAT $\leq_p$ VERTEX-COVER

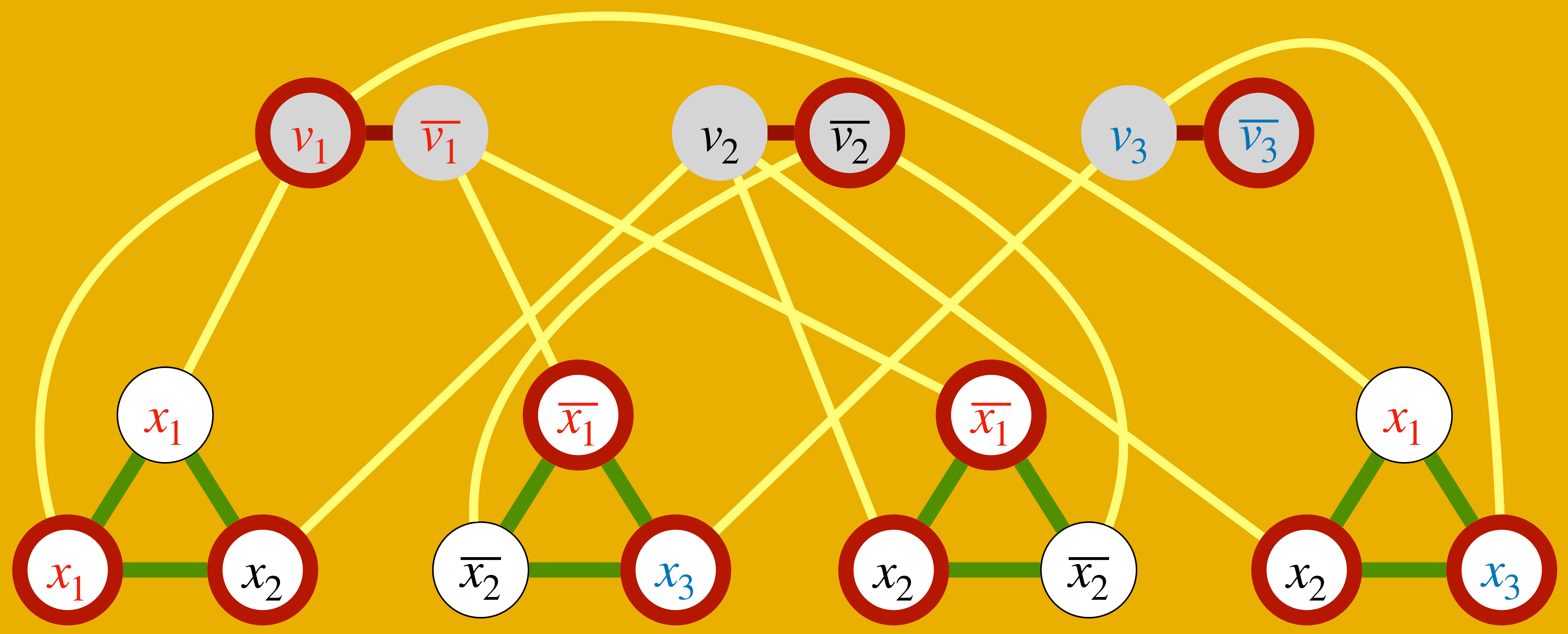
- $G$ has a vertex cover $C$ of cardinality $2k + \ell$ $\Rightarrow \phi$ is satisfiable
  - Since each triangle needs two vertices to cover it, each clause has at least two vertices picked ( $\geq 2k$)
    - They can only cover at most two variable-clause edges
  - Since the variable-variable edges need to be covered, at least one vertex in each pair of $v_i$ and $\overline{v_i}$ is in $C$ ( $\geq \ell$)
    - These vertices cover the variable-clause edges that are not yet covered
    - Pick the corresponding literals to be true, the covered variable-clause edges implies that every clause is true

$$\phi = \{(x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_2}) \wedge (x_1 \vee x_2 \vee x_3)$$



# 3SAT $\leq_p$ VERTEX-COVER

- For each vertex-clause edge, its end points refer to the same literal
  - For each clause, there can be at most $2$ vertex-clause edge covered by clause vertices $\Rightarrow$ at least $1$ vertex-clause edge covered by a variable vertex $\Leftrightarrow$ there must be a true literal in this clause



# Outline

- More NP-Hardness proofs
  - 3SAT $\leq_p$ VERTEX-COVER
  - VERTEX-COVER $\leq_p$ INDEPENDENT SET
  - VERTEX-COVER $\leq_p$ FEEDBACK-VERTEX-SET
  - VERTEX-COVER $\leq_p$ Integer Linear Program
- Pseudo-polynomial time algorithms
- **NP** and **Co-NP**
- Turing undecidable languages

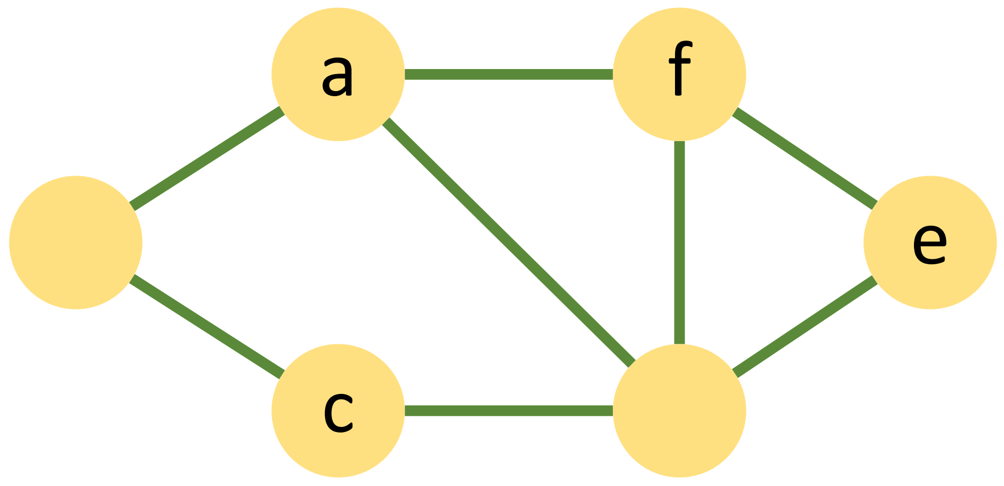
# Maximum Independent Set

# Maximum Independent Set

- Maximum INDEP-SET problem: Given a graph $G = (V, E)$, we want to find a subset of $V$ with maximum cardinality which forms an independent set

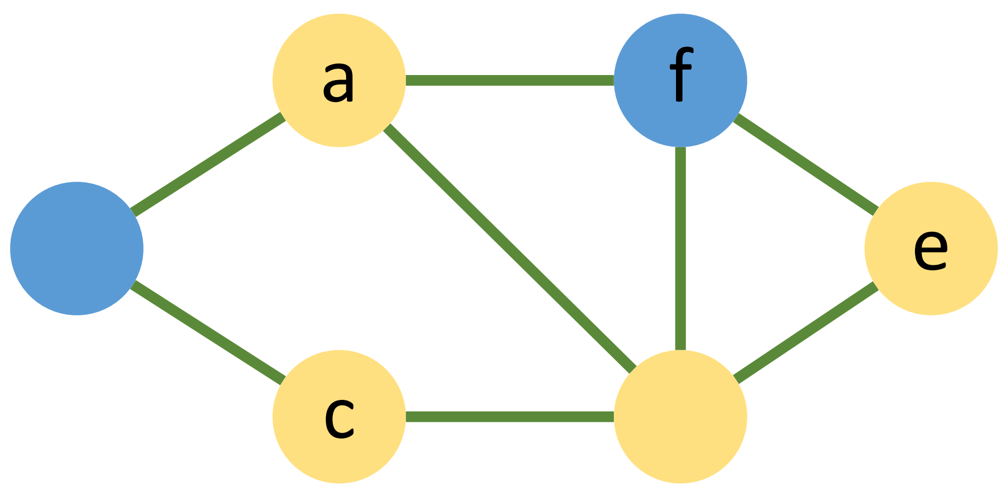
# Maximum Independent Set

- Maximum INDEP-SET problem: Given a graph $G = (V, E)$, we want to find a subset of $V$ with maximum cardinality which forms an independent set



# Maximum Independent Set

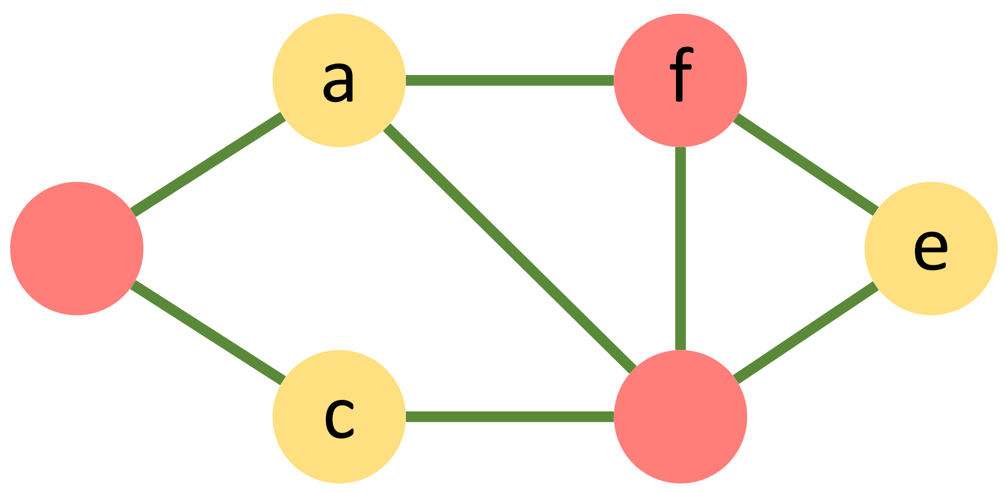
- Maximum INDEP-SET problem: Given a graph $G = (V, E)$, we want to find a subset of $V$ with maximum cardinality which forms an independent set




An independent set

# Maximum Independent Set

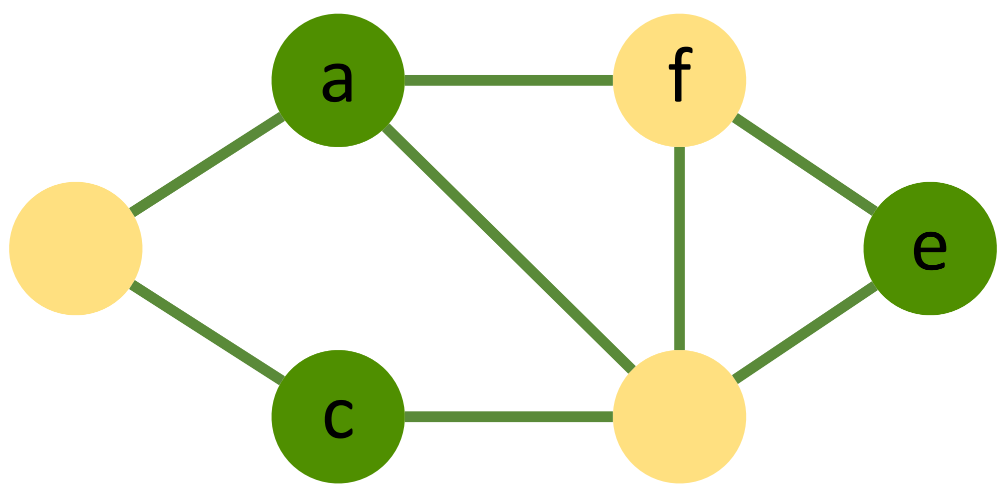
- Maximum INDEP-SET problem: Given a graph $G = (V, E)$, we want to find a subset of $V$ with maximum cardinality which forms an independent set




NOT an independent set

# Maximum Independent Set

- Maximum INDEP-SET problem: Given a graph $G=(V,E)$, we want to find a subset of $V$ with maximum cardinality which forms an independent set




Maximum independent set

# Maximum Independent Set is NP-hard

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise

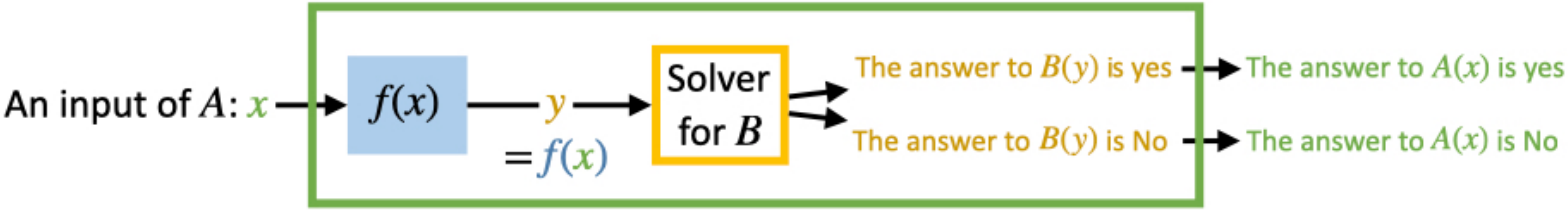
# Maximum Independent Set is NP-hard

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise



# Maximum Independent Set is NP-hard

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise

$(G', k')$ → IS Solver

There is an independent set in $G'$ with size $\leq k$

There is no independent set in $G'$ with size $\leq k$

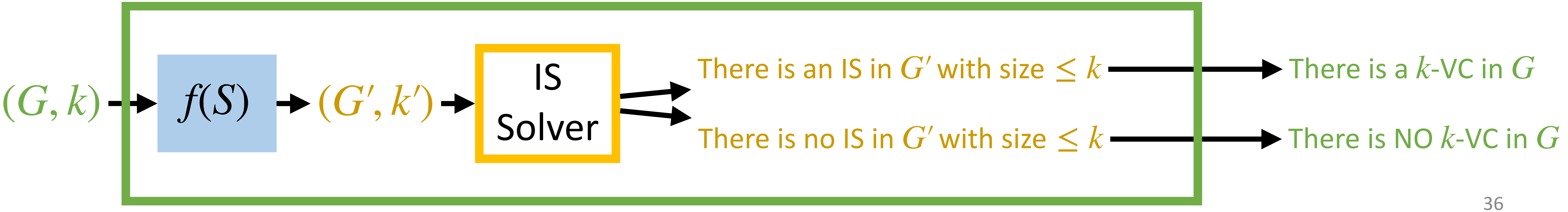
# Maximum Independent Set is NP-hard

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise



# Maximum Independent Set is NP-hard

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise

There is a subset $V \backslash W$

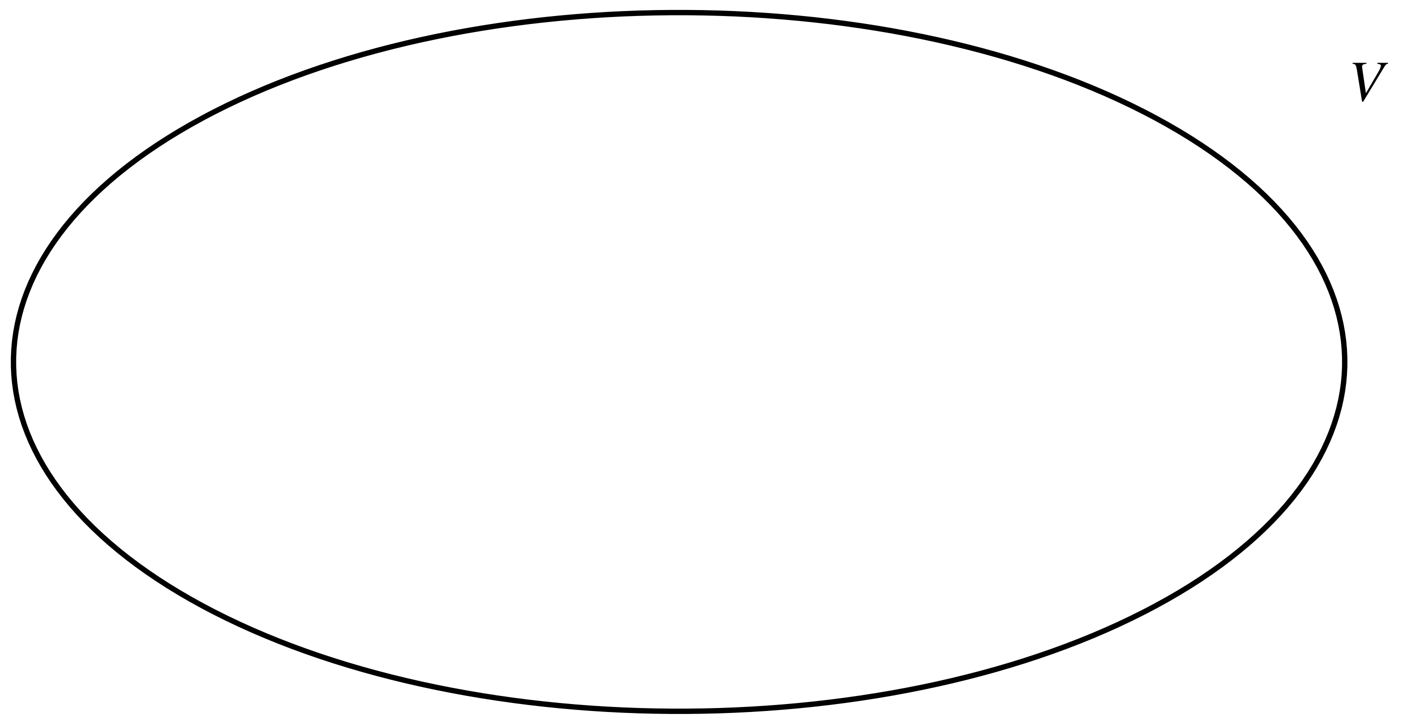
which is an vertex cover with size at most $k$

There is a subset $W$

which is an independent set with size at least $k$

# VC $\leq_p$ INDEP-Set

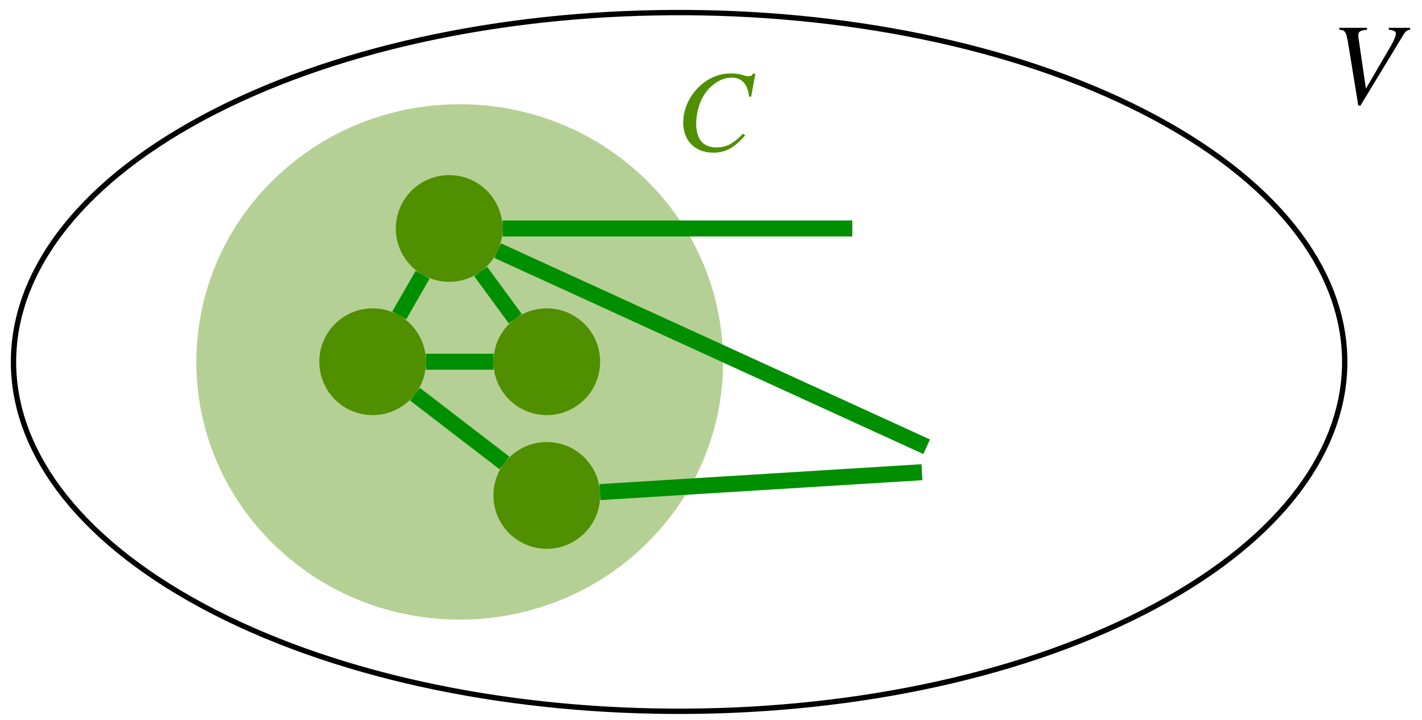
Observation: In graph $G$, for any of its vertex cover $C$, any pair of the vertices in $V \backslash C$ are not adjacent.



# VC $\leq_p$ INDEP-Set

Observation: In graph $G$, for any of its vertex cover $C$, any pair of the vertices in $V \backslash C$ are not adjacent.

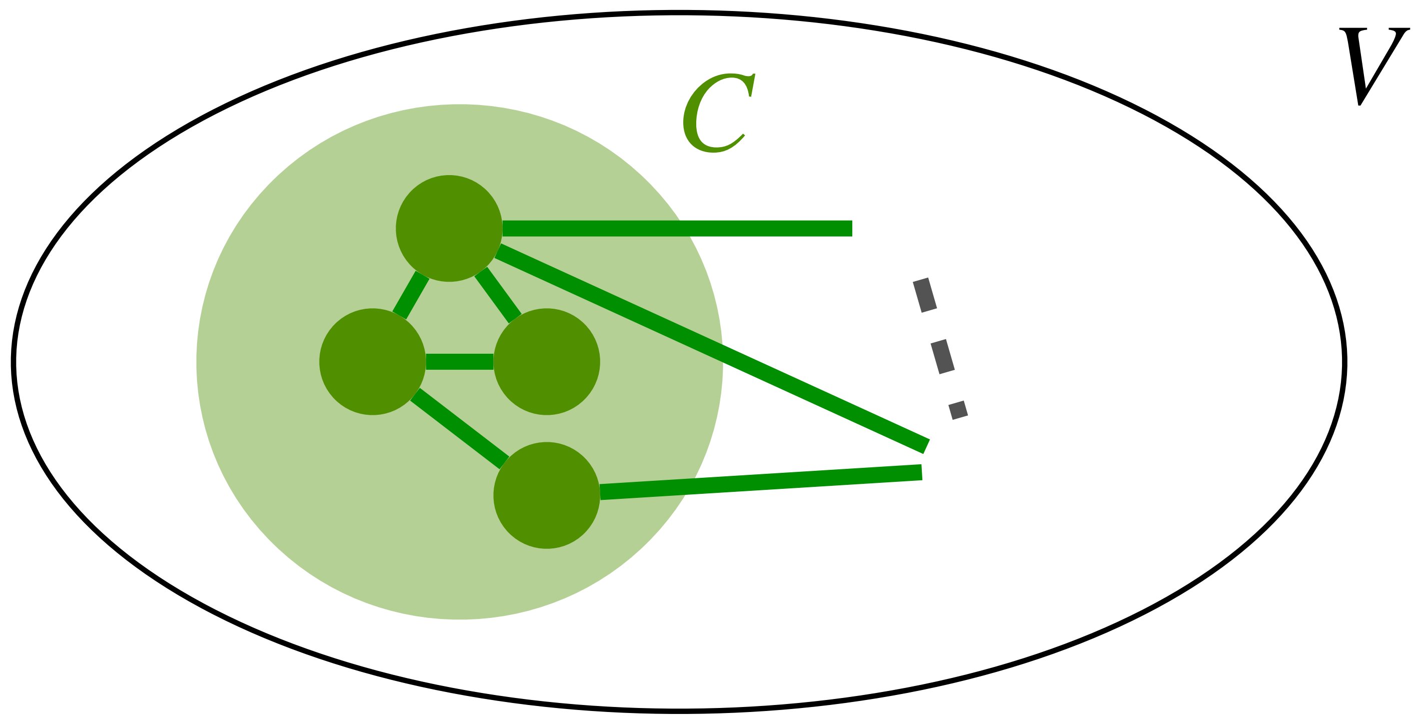
Vertex cover of size $k'$



# VC $\leq_p$ INDEP-Set

Observation: In graph $G$, for any of its vertex cover $C$, any pair of the vertices in $V \backslash C$ are not adjacent.

Vertex cover of size $k'$



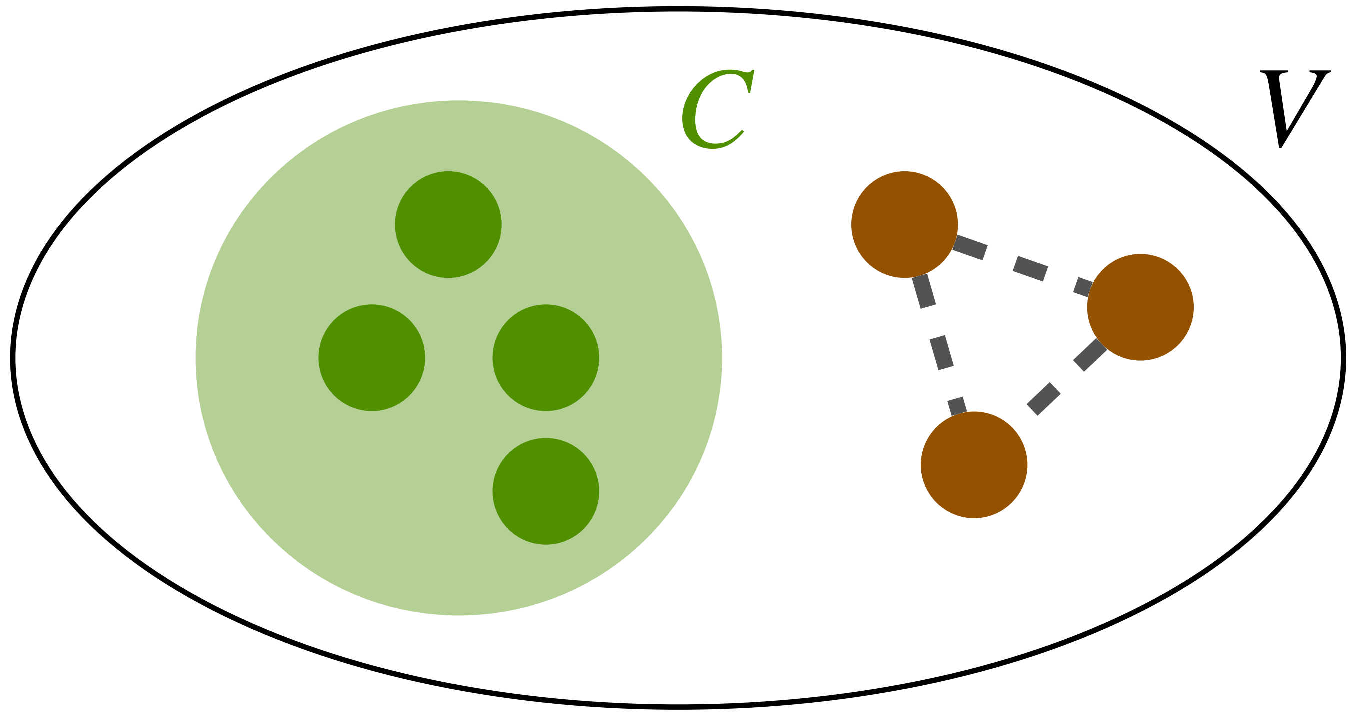
# VC $\leq_p$ INDEP-Set

Observation: In graph $G$, for any of its vertex cover $C$, any pair of the vertices in $V \backslash C$ are not adjacent.

There is a subset $V \backslash W$ which is an vertex cover with size at most $k$

There is a subset $W$ which is an independent set with size at least $k$

Vertex cover of size $k'$




Independent set of size $|V| - k'$

# VC $\leq_p$ INDEP-Set

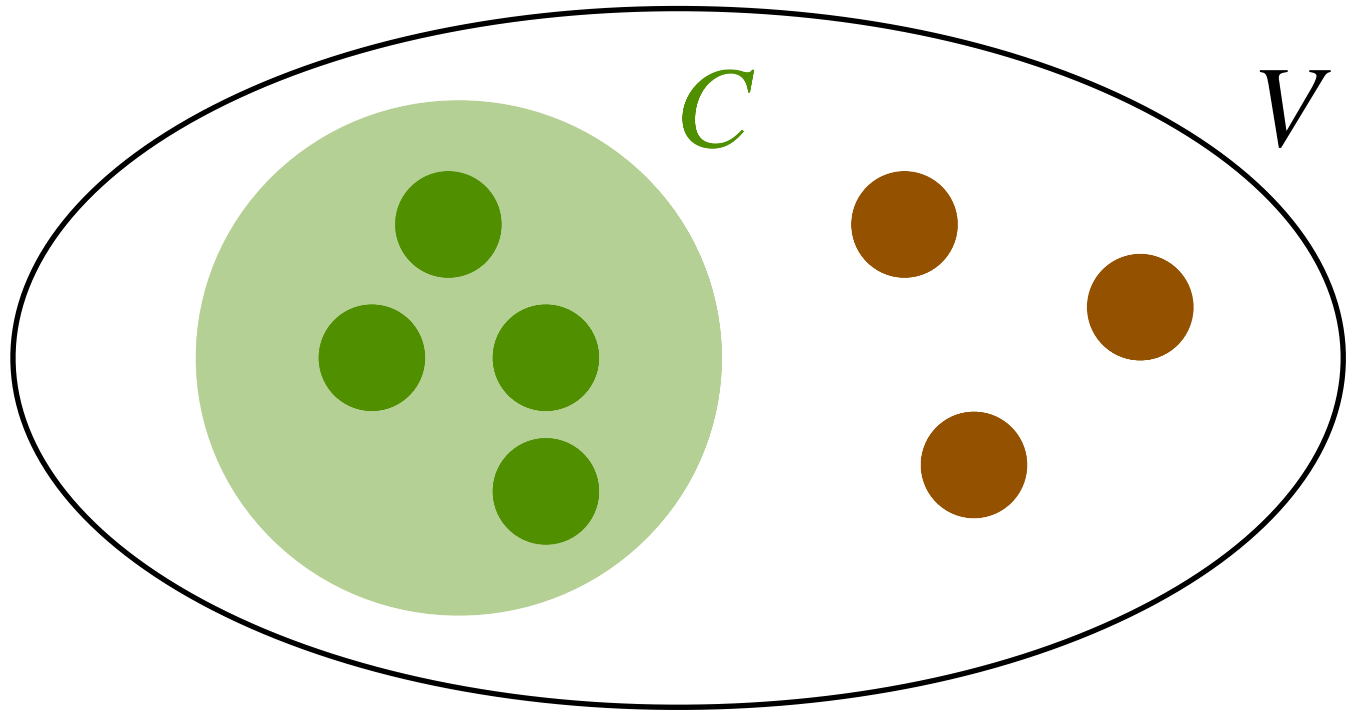
**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise

Vertex cover of size $k'$




Independent set of size $|V| - k'$

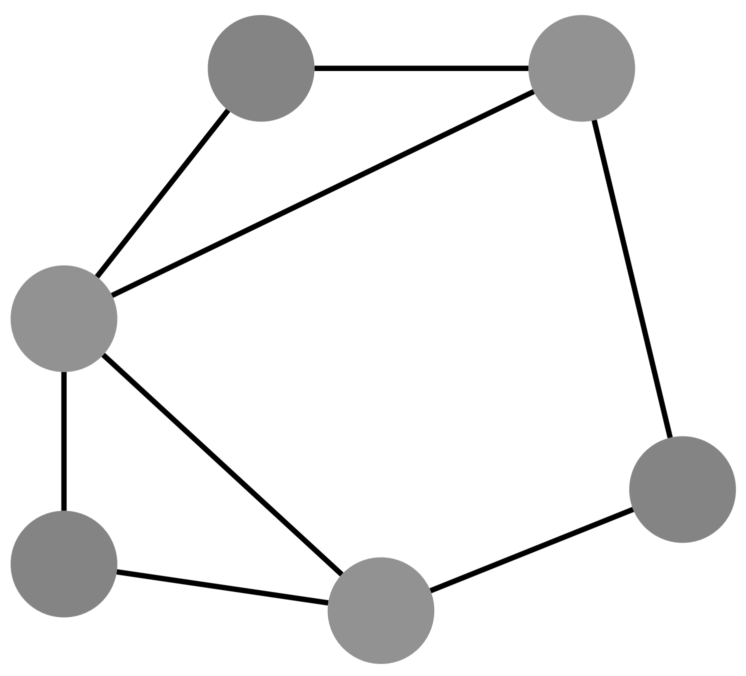
# VC $\leq_p$ INDEP-Set

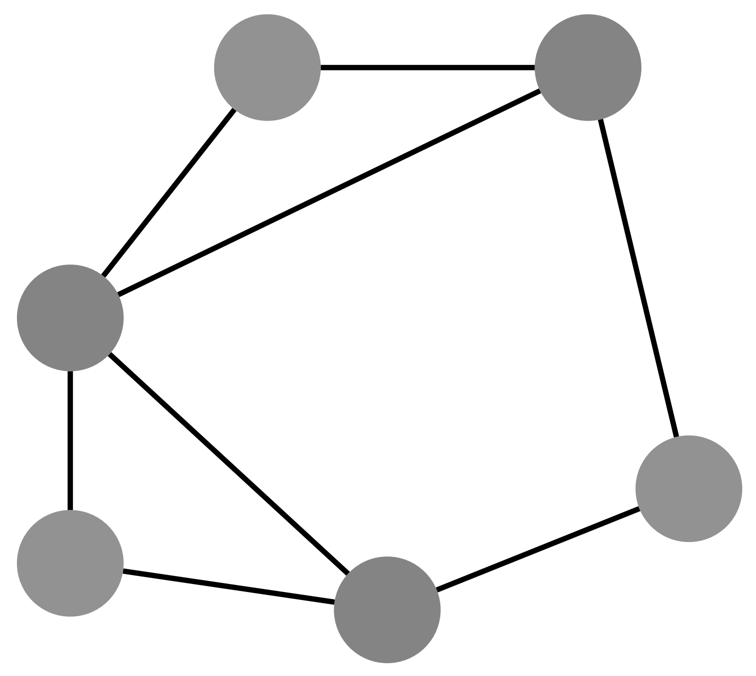
**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise





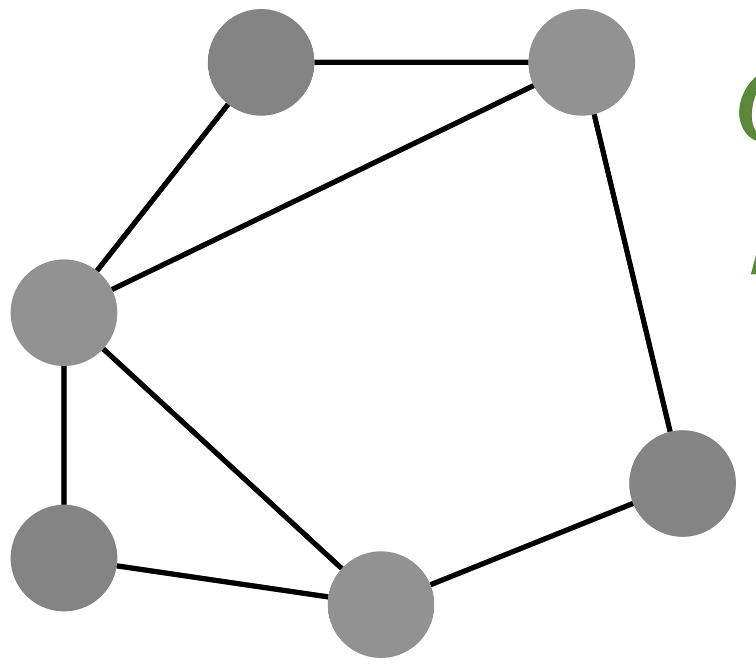
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise

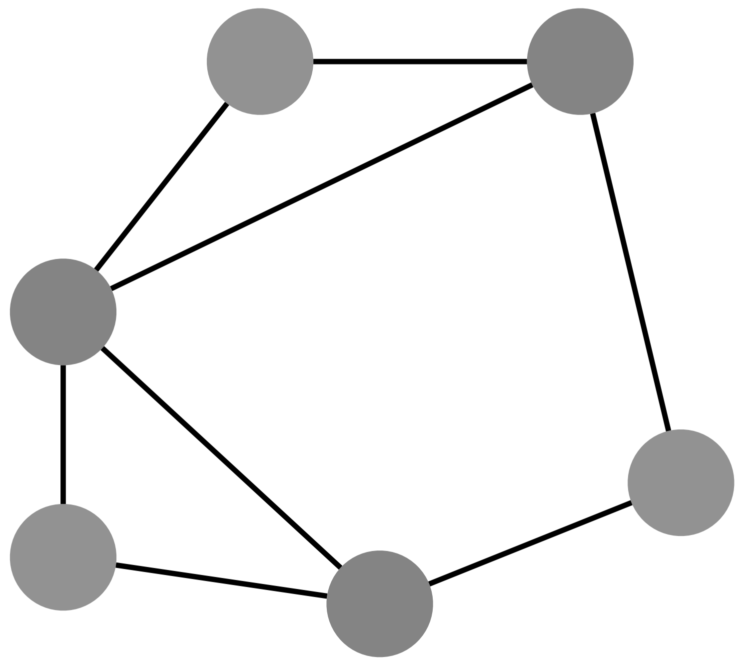


$G$

$k$

$G' = G$

$k' = |V| - k$



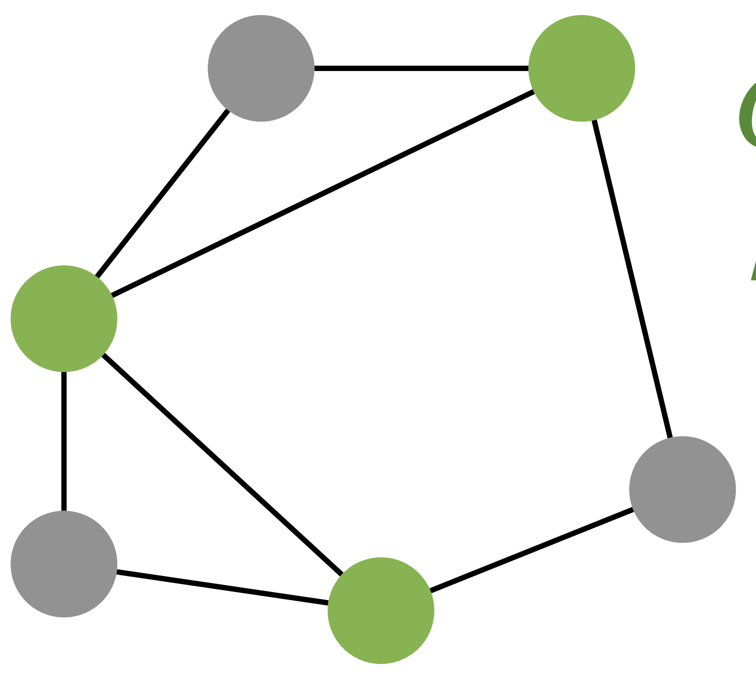
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise



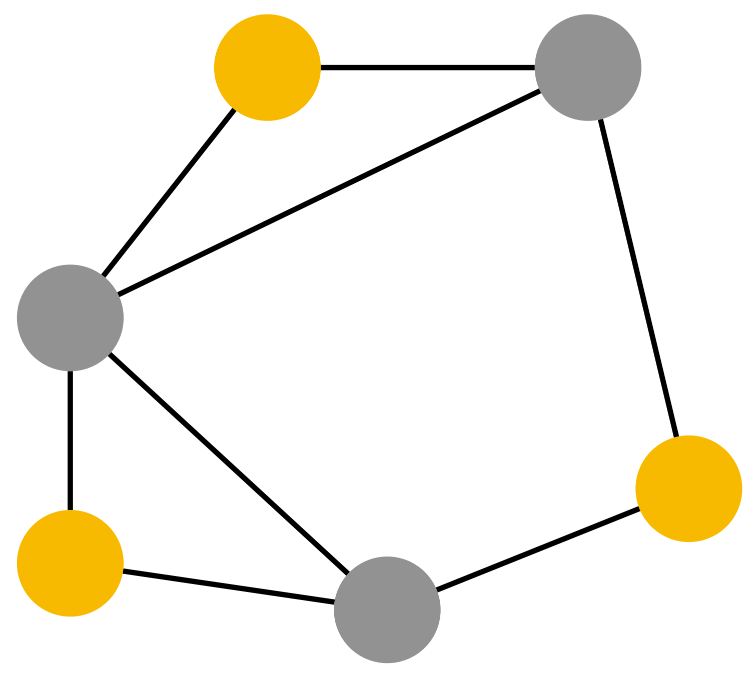
$G$

$k$

If there is a vertex cover of size $k$ in $G$

$G' = G$

$k' = |V| - k$



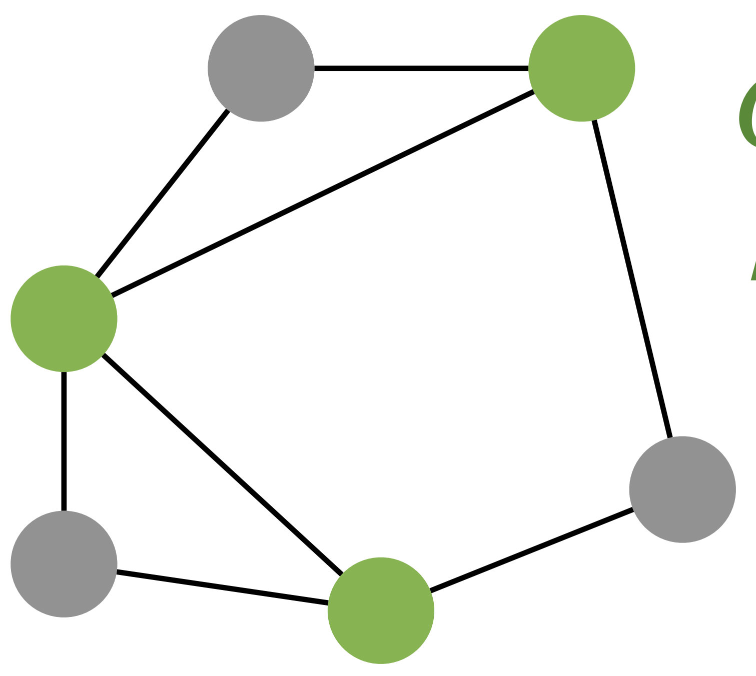
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise

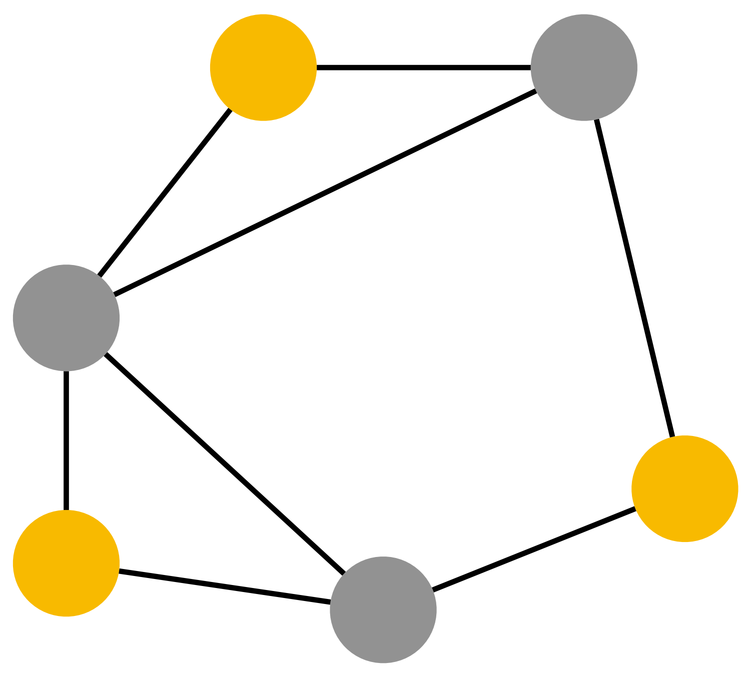


$G$
$k$

If there is a vertex cover of size $k$ in $G$

$\Rightarrow$ The other vertices

$G' = G$
$k' = |V| - k$



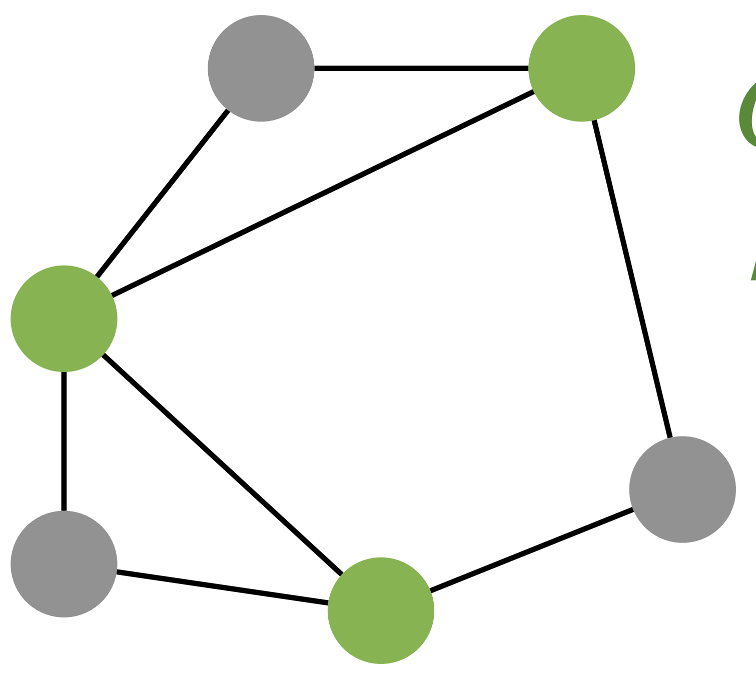
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

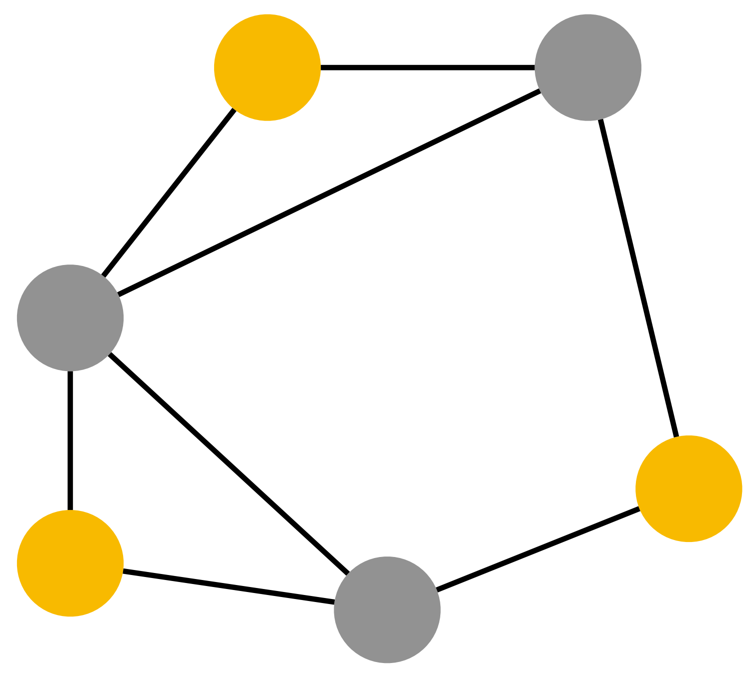
**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise



$G$
$k$

$G' = G$
$k' = |V| - k$



If there is a vertex cover of size $k$ in $G$

$\Rightarrow$ The other vertices

there is no edge between them

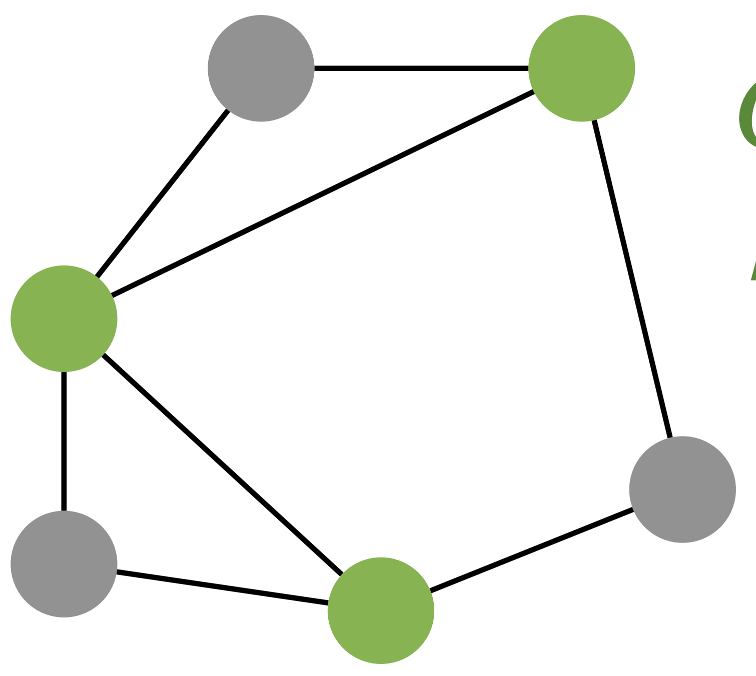
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

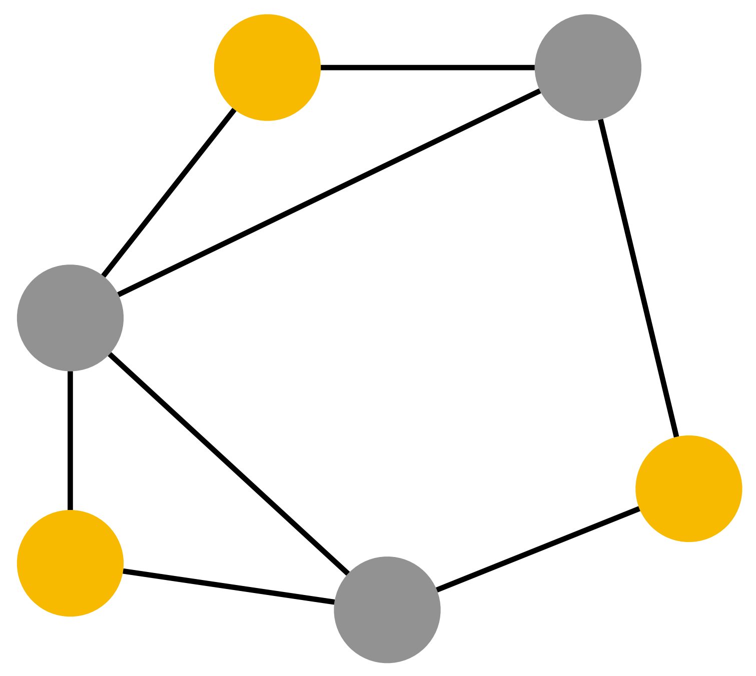
**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise



$G$
$k$

$G' = G$
$k' = |V| - k$



If there is a vertex cover of size $k$ in $G$
$\Rightarrow$ The other vertices form an independent set of size $|V| - k = k'$ in $G' = G$ since there is no edge between them

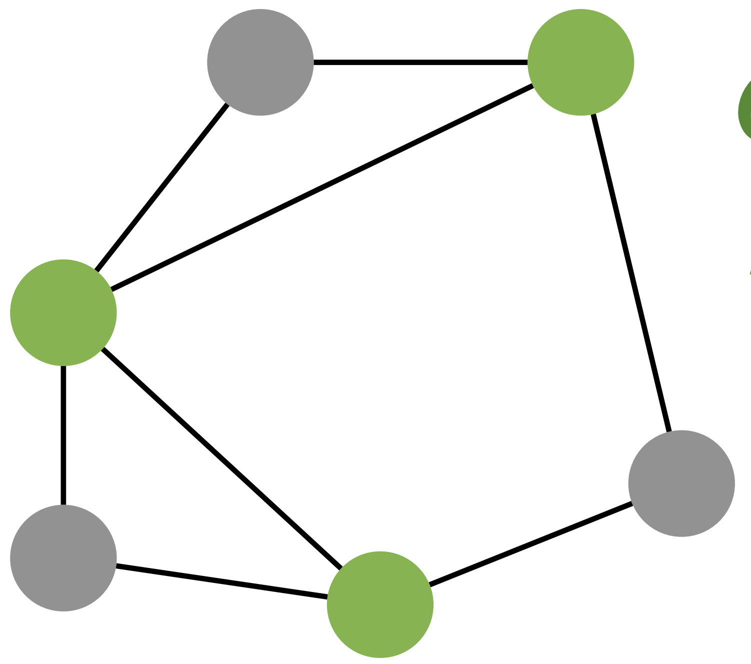
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

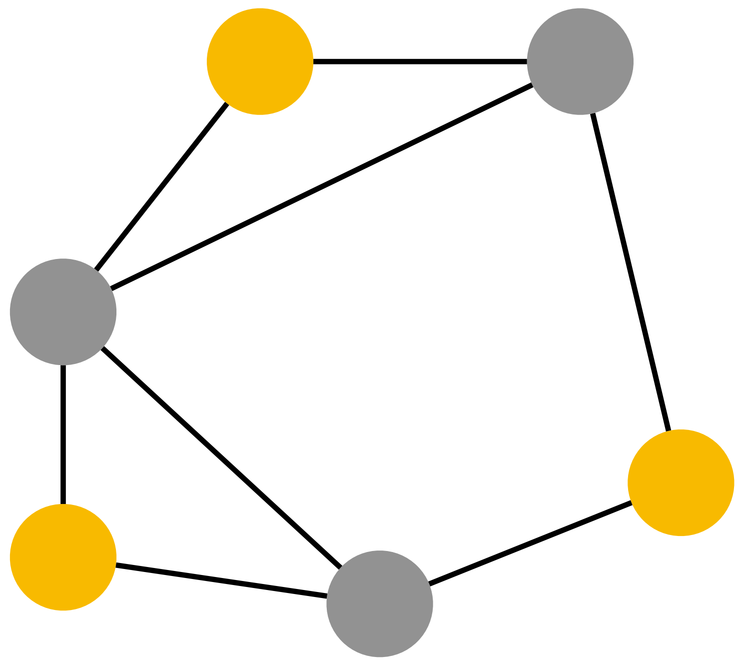
**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise



$G$
$k$

$G' = G$
$k' = |V| - k$



If there is a vertex cover of size $k$ in $G$
$\Rightarrow$ The other vertices form an independent set of size $|V| - k = k'$ in $G' = G$ since there is no edge between them

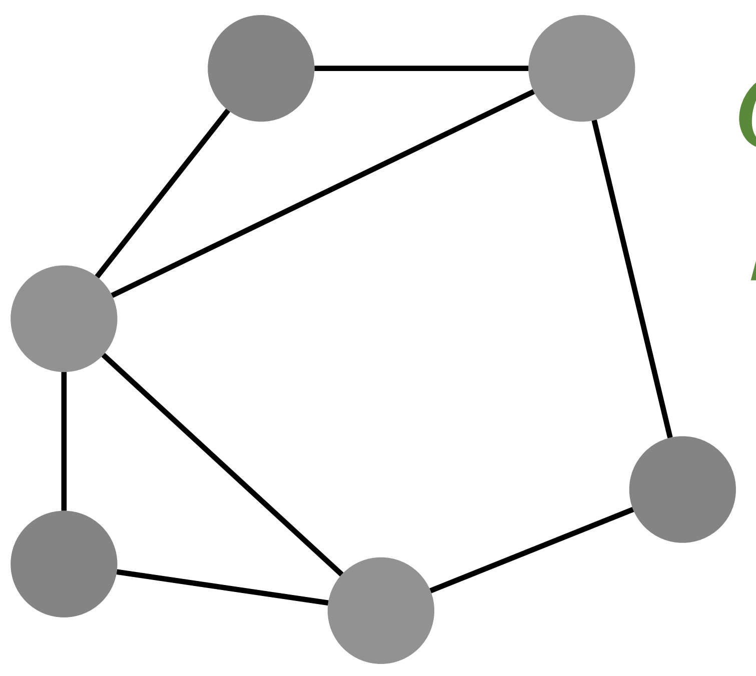
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

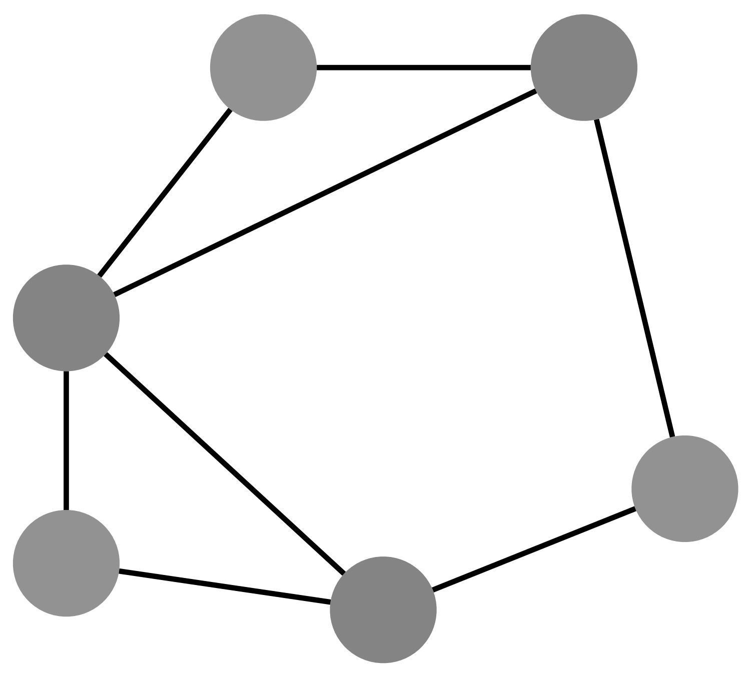
**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise



$G$
$k$

$G' = G$
$k' = |V| - k$



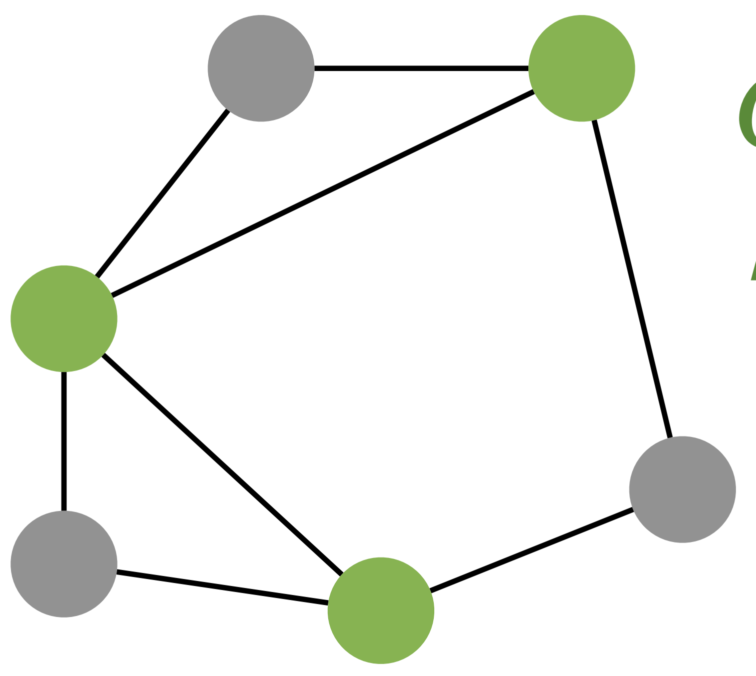
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

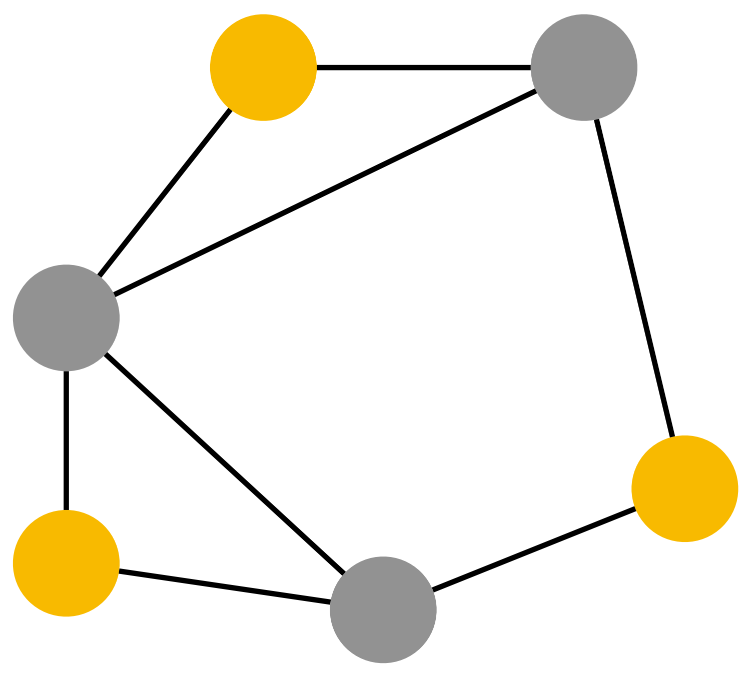
**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise



$G$
$k$

$G' = G$
$k' = |V| - k$



If there is an independent set of size $k'$ in $G'$

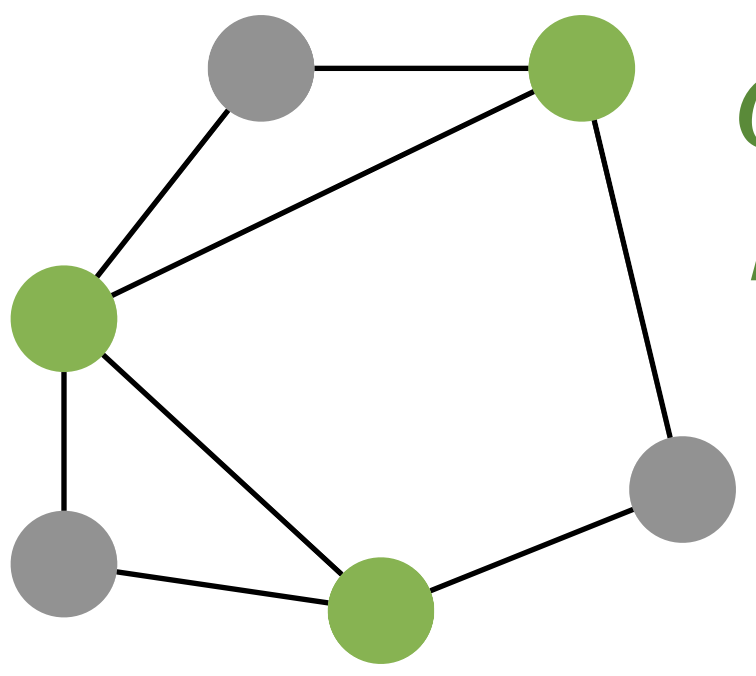
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

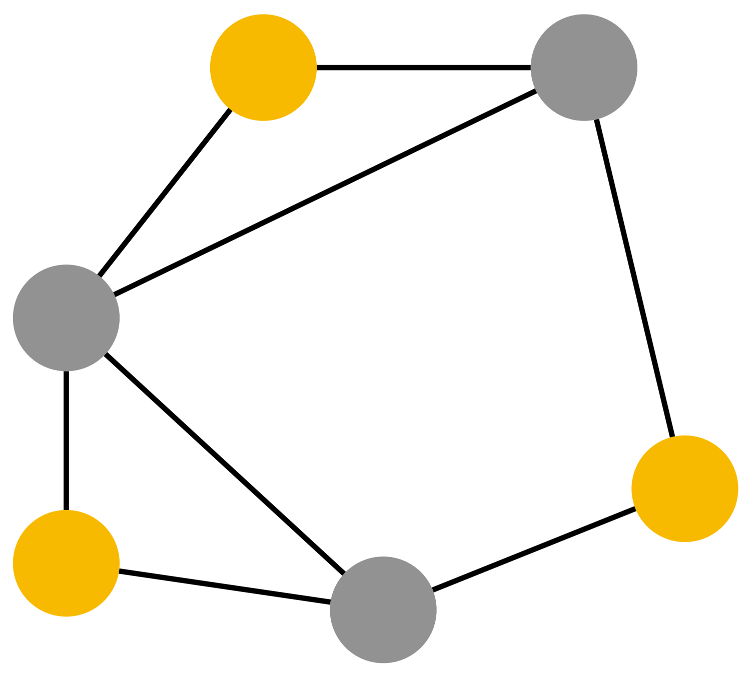
**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise



$G$
$k$

$G' = G$
$k' = |V| - k$



If there is an independent set of size $k'$ in $G'$
$\Rightarrow$ The other vertices

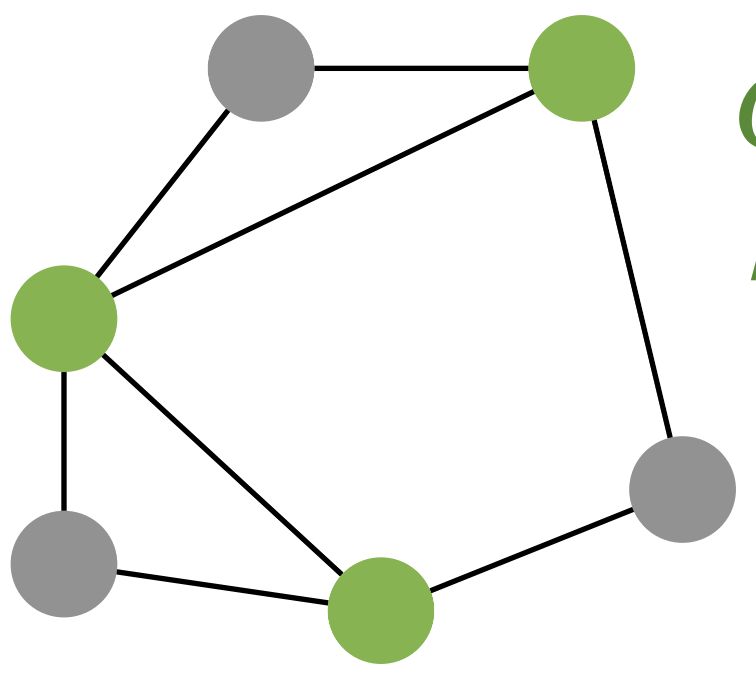
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise

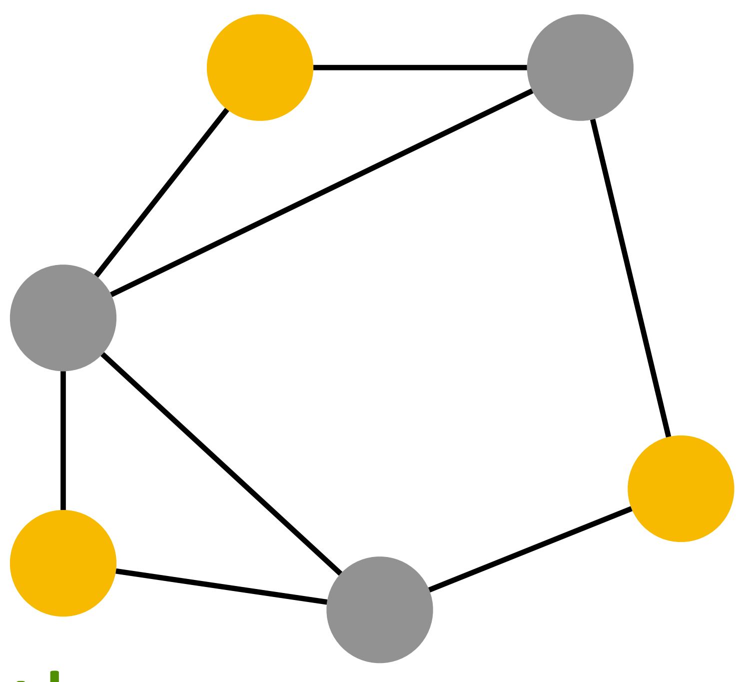


$G$

$k$

$G' = G$

$k' = |V| - k$



If there is an independent set of size $k'$ in $G'$

$\Rightarrow$ The other vertices

every edge must incident with one of them

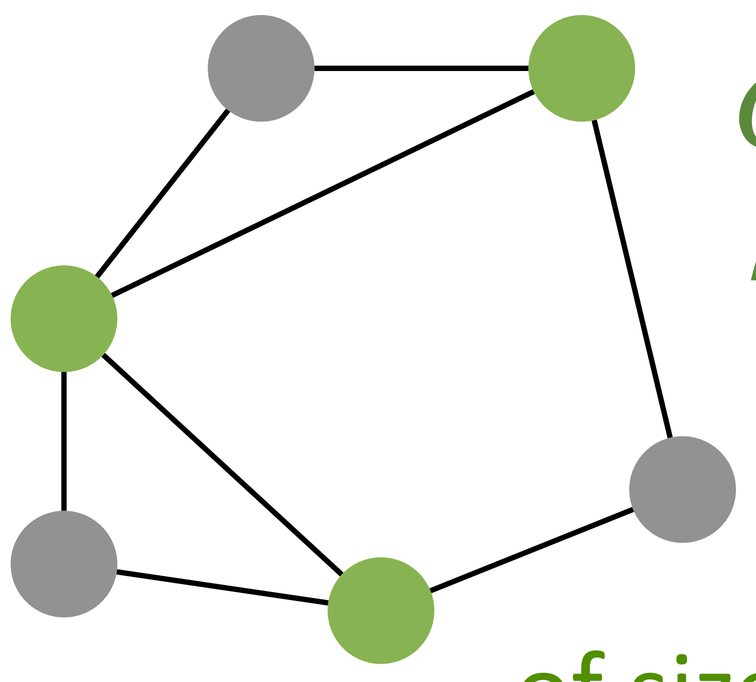
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise

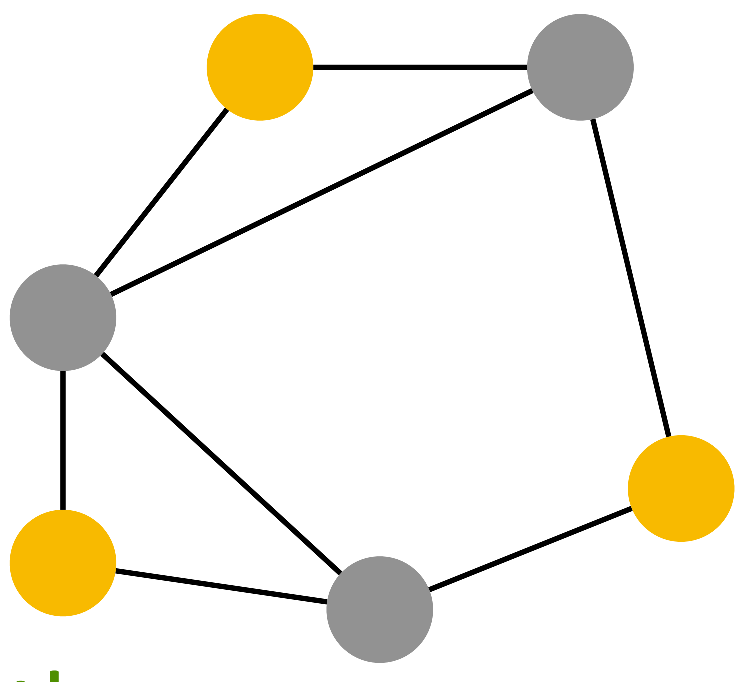


$G$

$k$

$G' = G$

$k' = |V| - k$



If there is an independent set of size $k'$ in $G'$

$\Rightarrow$ The other vertices form a vertex cover of size $|V| - k' = k$ in $G = G'$ since every edge must incident with one of them

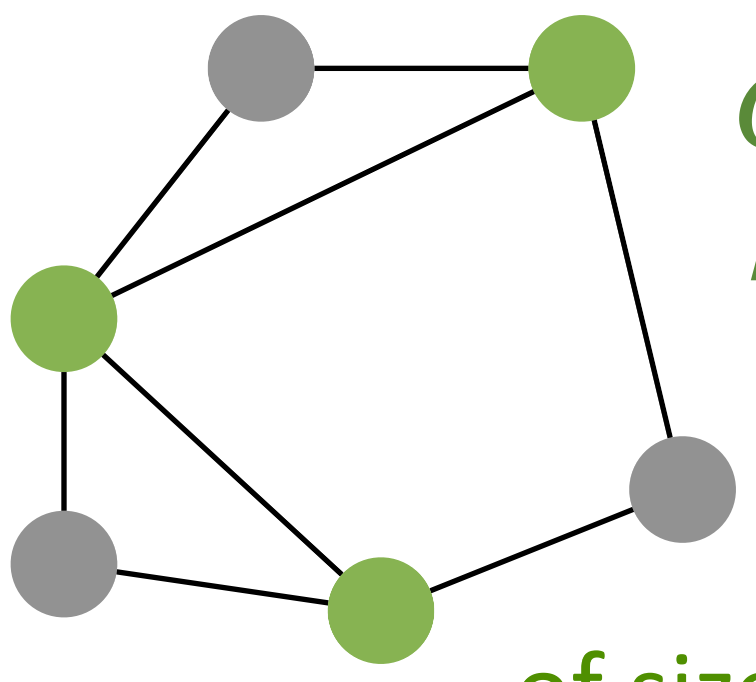
# VC $\leq_p$ INDEP-Set

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

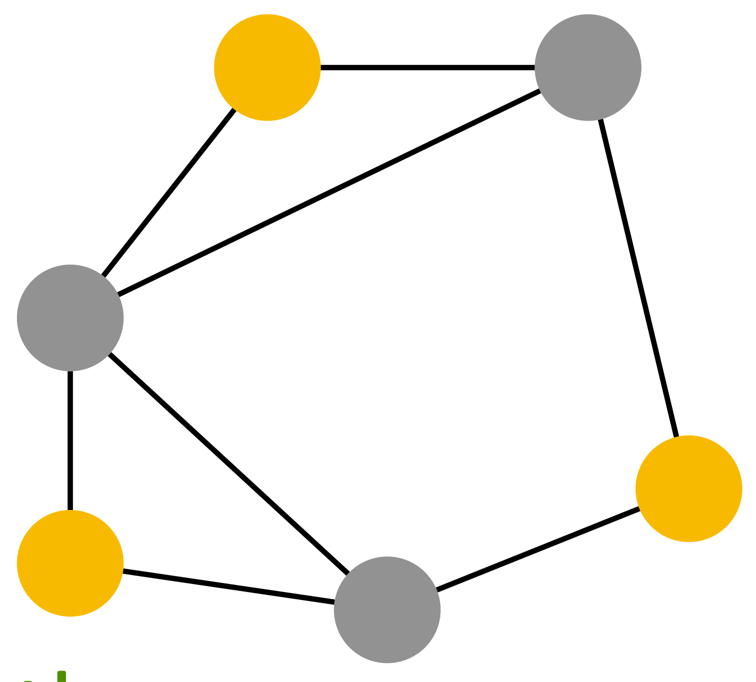
**Maximum INDEP-SET (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at least $k$ that forms an **independent set**
  - *no* otherwise



$G$
$k$

$G' = G$
$k' = |V| - k$



If there is an independent set of size $k'$ in $G'$
$\Rightarrow$ The other vertices form a vertex cover of size $|V| - k' = k$ in $G = G'$ since every edge must incident with one of them

# Outline

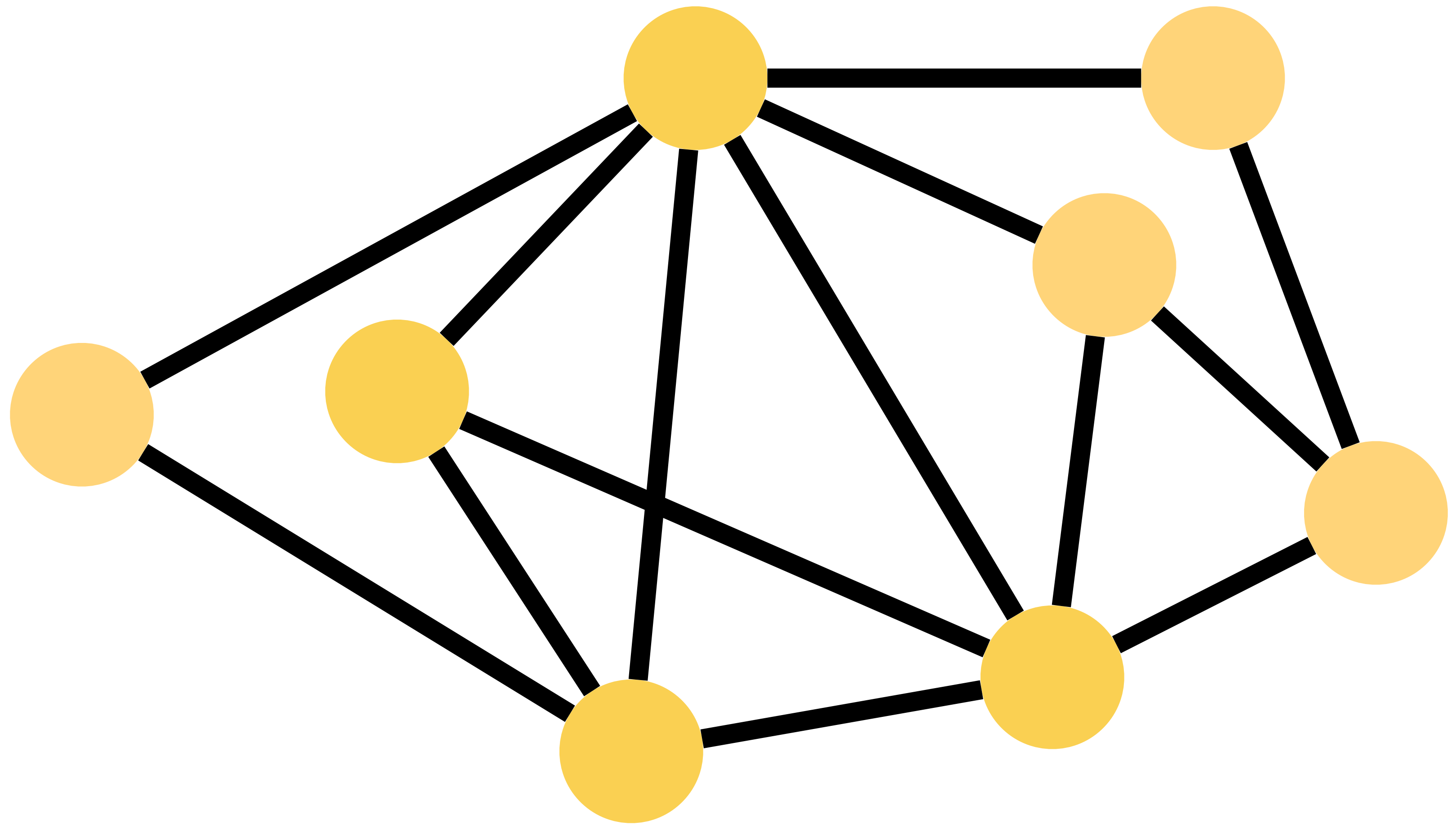
- More NP-Hardness proofs
  - 3SAT $\leq_p$ VERTEX-COVER
  - VERTEX-COVER $\leq_p$ INDEPENDENT SET
  - VERTEX-COVER $\leq_p$ FEEDBACK-VERTEX-SET
  - VERTEX-COVER $\leq_p$ Integer Linear Program
- Pseudo-polynomial time algorithms
- **NP** and **Co-NP**
- Turing undecidable languages

# Feedback Vertex Set (FVS)

- Given a graph $G = (V, E)$, a *feedback vertex set* is a subset $U$ of vertices such that removing the vertices in $U$ leaves a graph without cycles

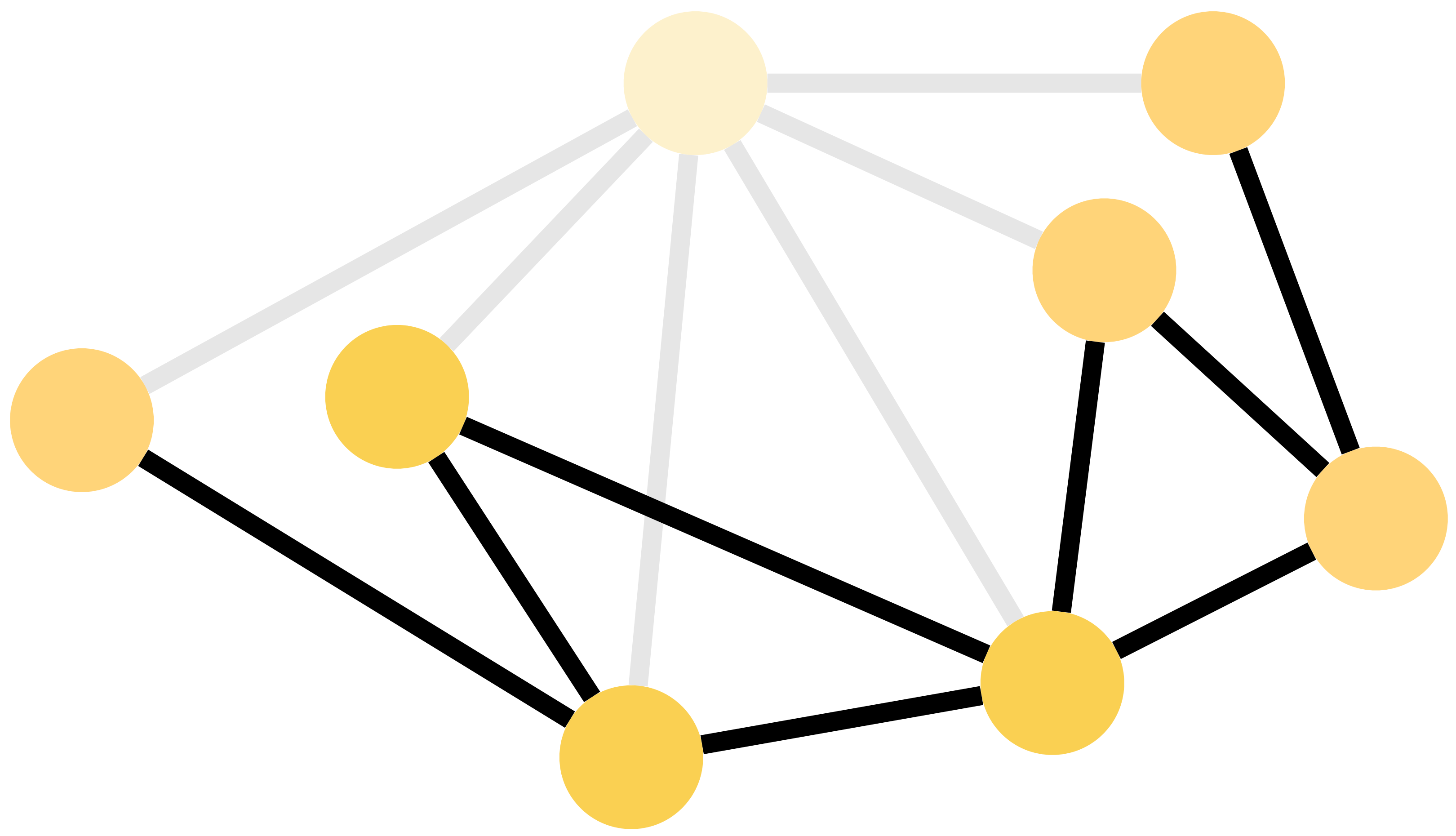
# Feedback Vertex Set (FVS)

- Given a graph $G = (V, E)$, a *feedback vertex set* is a subset $U$ of vertices such that removing the vertices in $U$ leaves a graph without cycles
  - When a vertex is removed, all the edges incident to it are also removed



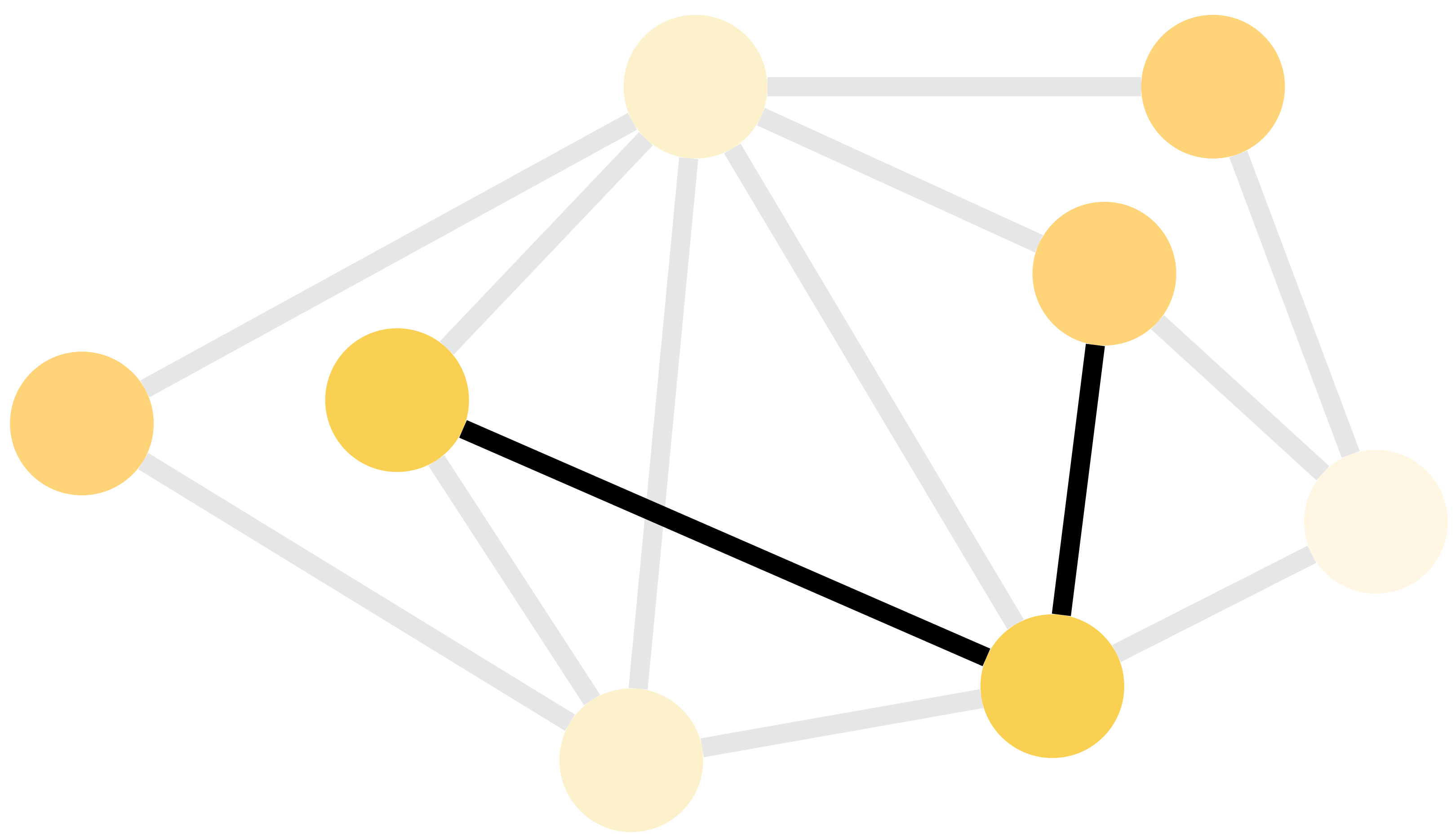
# Feedback Vertex Set (FVS)

- Given a graph $G = (V, E)$, a *feedback vertex set* is a subset $U$ of vertices such that removing the vertices in $U$ leaves a graph without cycles
  - When a vertex is removed, all the edges incident to it are also removed



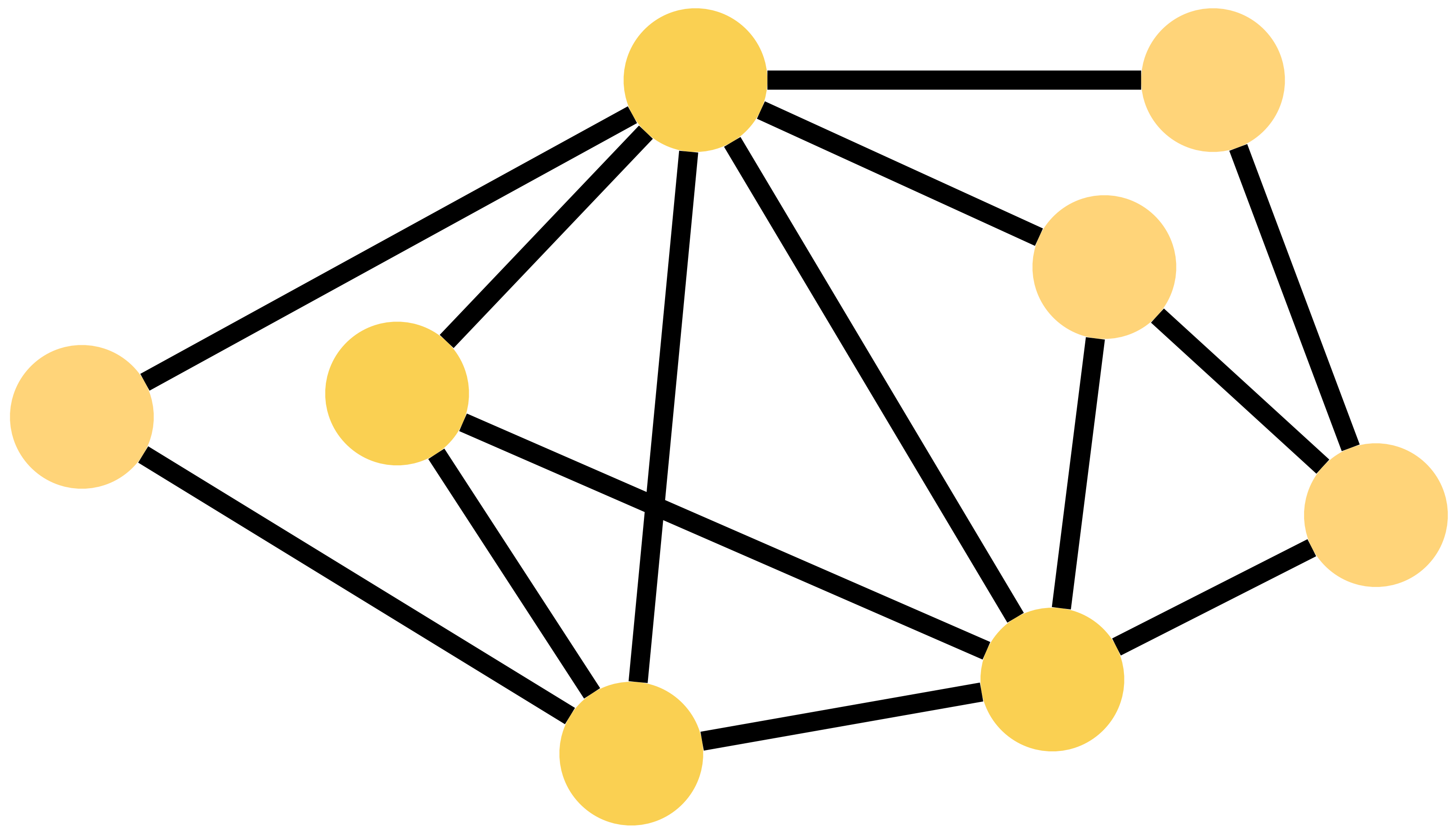
# Feedback Vertex Set (FVS)

- Given a graph $G = (V, E)$, a *feedback vertex set* is a subset $U$ of vertices such that removing the vertices in $U$ leaves a graph without cycles
  - When a vertex is removed, all the edges incident to it are also removed



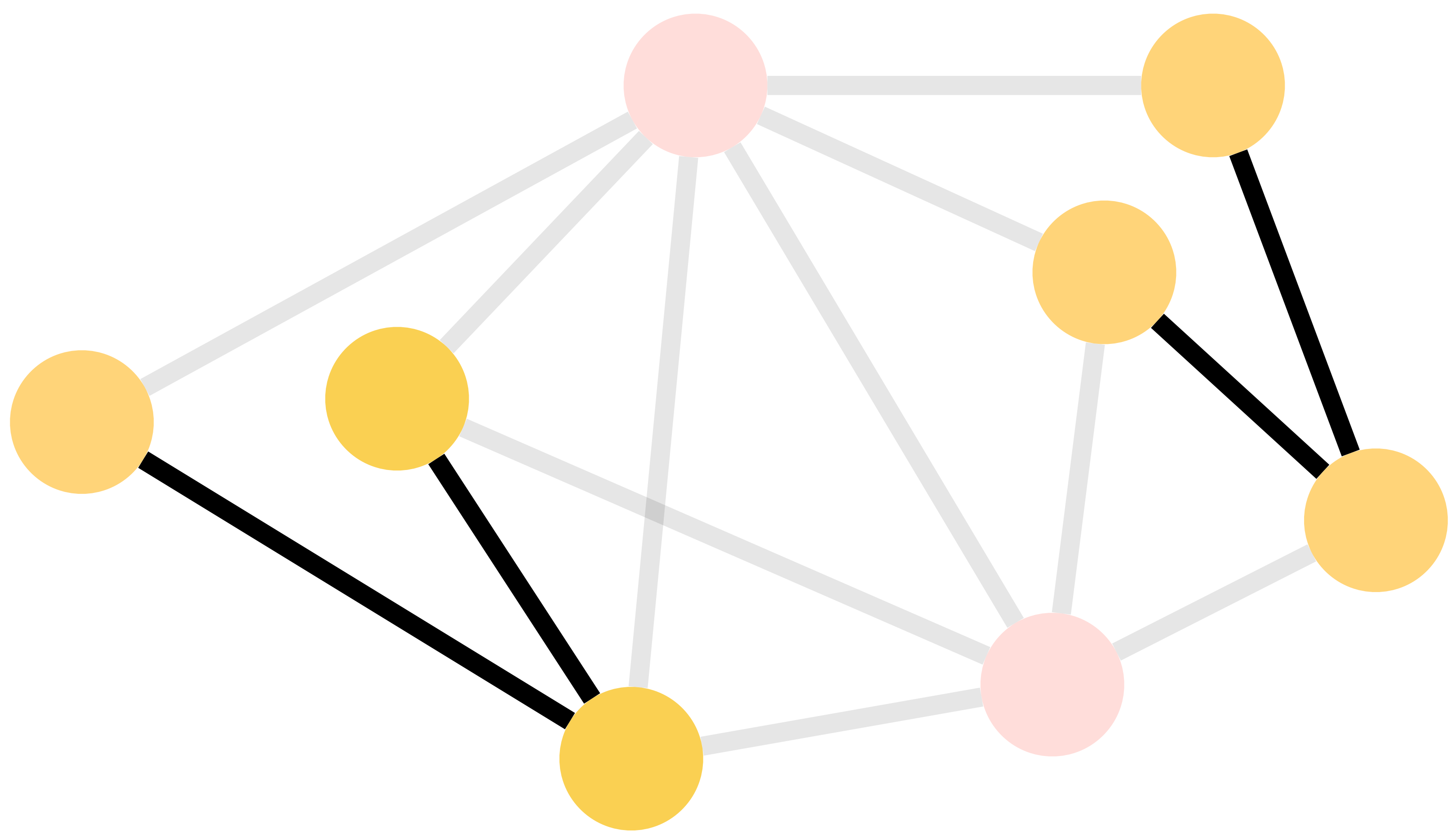
# Feedback Vertex Set (FVS)

- Minimum-FVS: Given a graph $G = (V, E)$, what is the size of its minimum feedback vertex set?



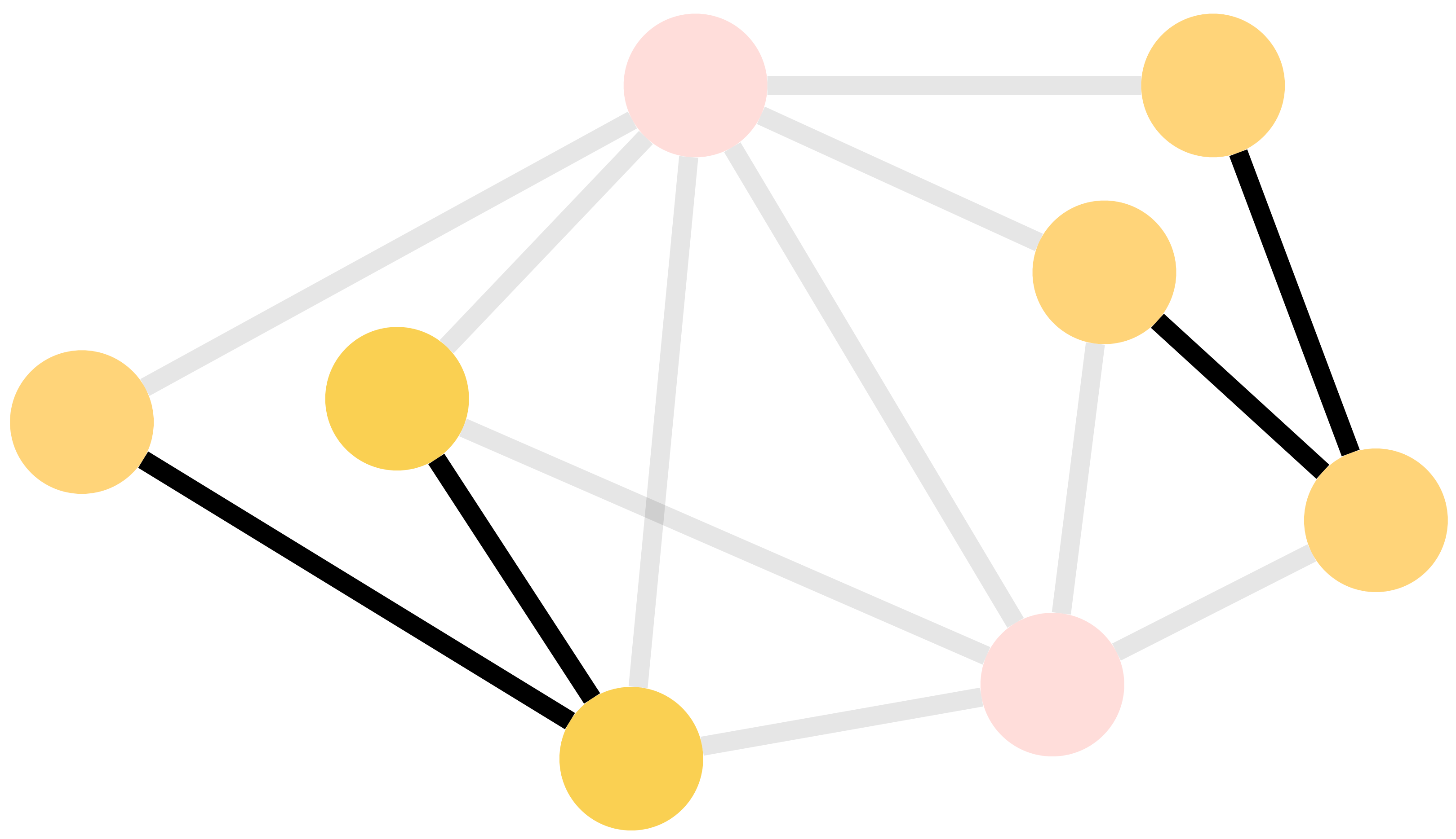
# Feedback Vertex Set (FVS)

- Minimum-FVS: Given a graph $G = (V, E)$, what is the size of its minimum feedback vertex set?



# Feedback Vertex Set (FVS)

- Minimum-FVS: Given a graph $G = (V, E)$, what is the size of its minimum feedback vertex set?
- Decision version of Minimum-FVS problem:
  - Given a graph $G = (V, E)$, is there a feedback vertex set with size at most $k$?



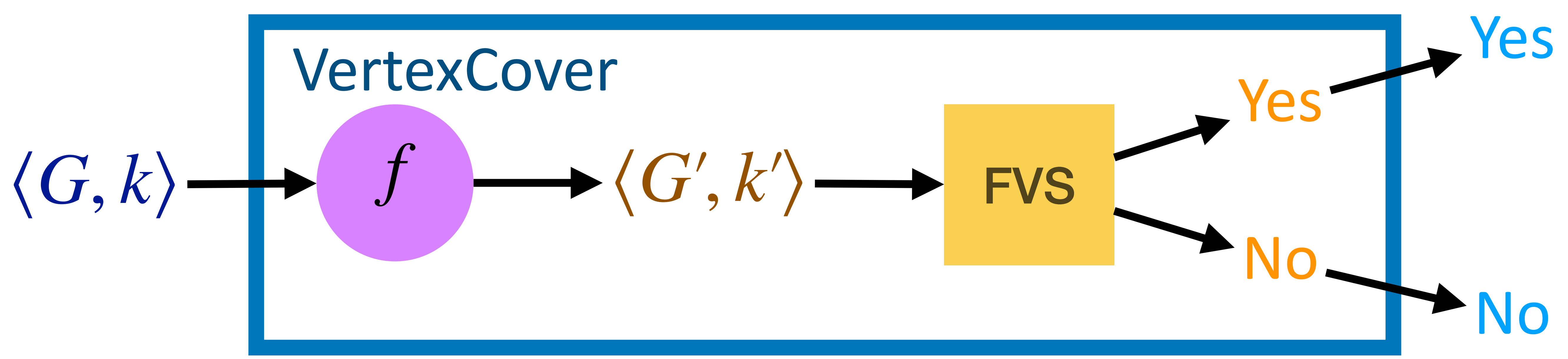
# Feedback Vertex Set (FVS)

- Minimum-FVS: Given a graph $G = (V, E)$, what is the size of its minimum feedback vertex set?
- Decision version of Minimum-FVS problem:
  - Given a graph $G = (V, E)$, is there a feedback vertex set with size at most $k$?

- Theorem: FVS is NP-complete

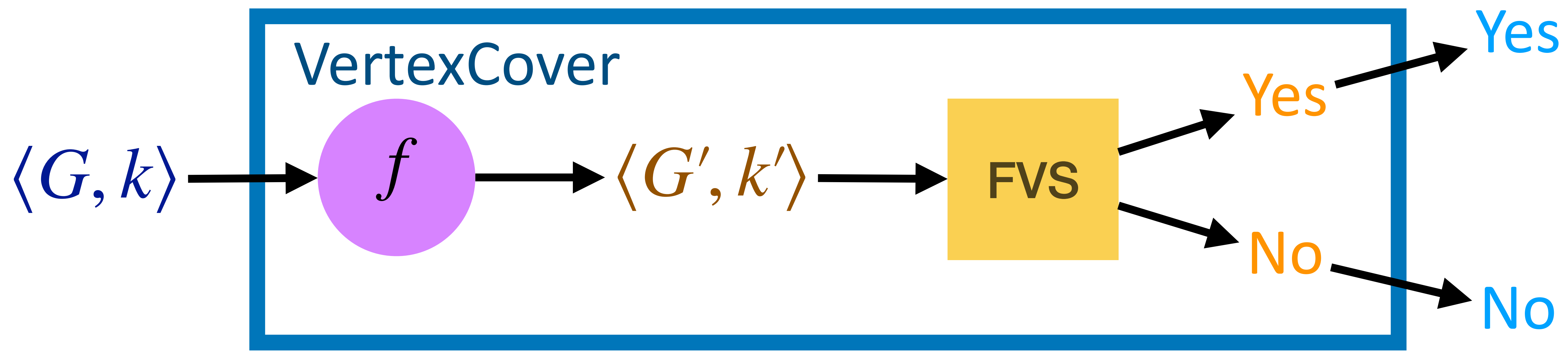
# Feedback Vertex Set (FVS)

- VertexCover $= \{\langle G, k\rangle \mid$ There is a vertex cover in $G$ with size at most $k\}$
- FVS $= \{\langle G', k'\rangle \mid$ There is a feedback vertex set in $G'$ with size at most $k'\}$



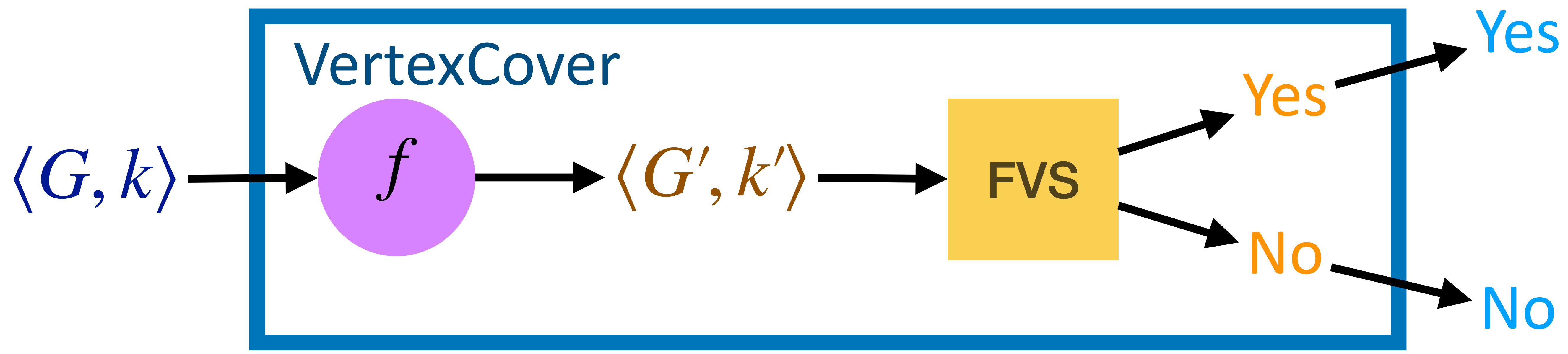
# Feedback Vertex Set (FVS)

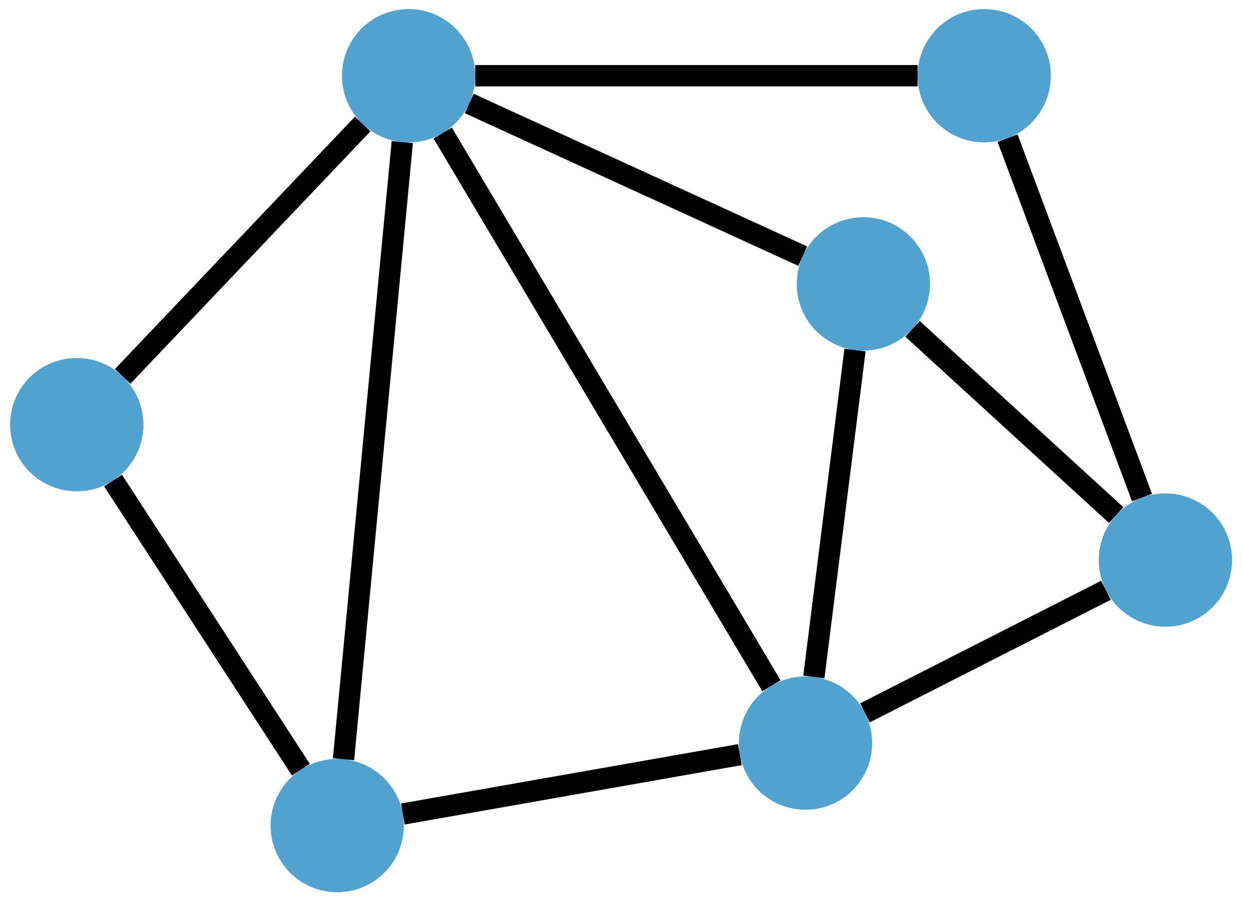
- VertexCover $= \{\langle G, k\rangle \mid$ There is a set of at most $k$ vertices in $G$ such that removing them leaves no edges$\}$
- FVS $= \{\langle G', k'\rangle \mid$ There is a set of at most $k'$ vertices in $G'$ such that removing them leaves no cycles$\}$

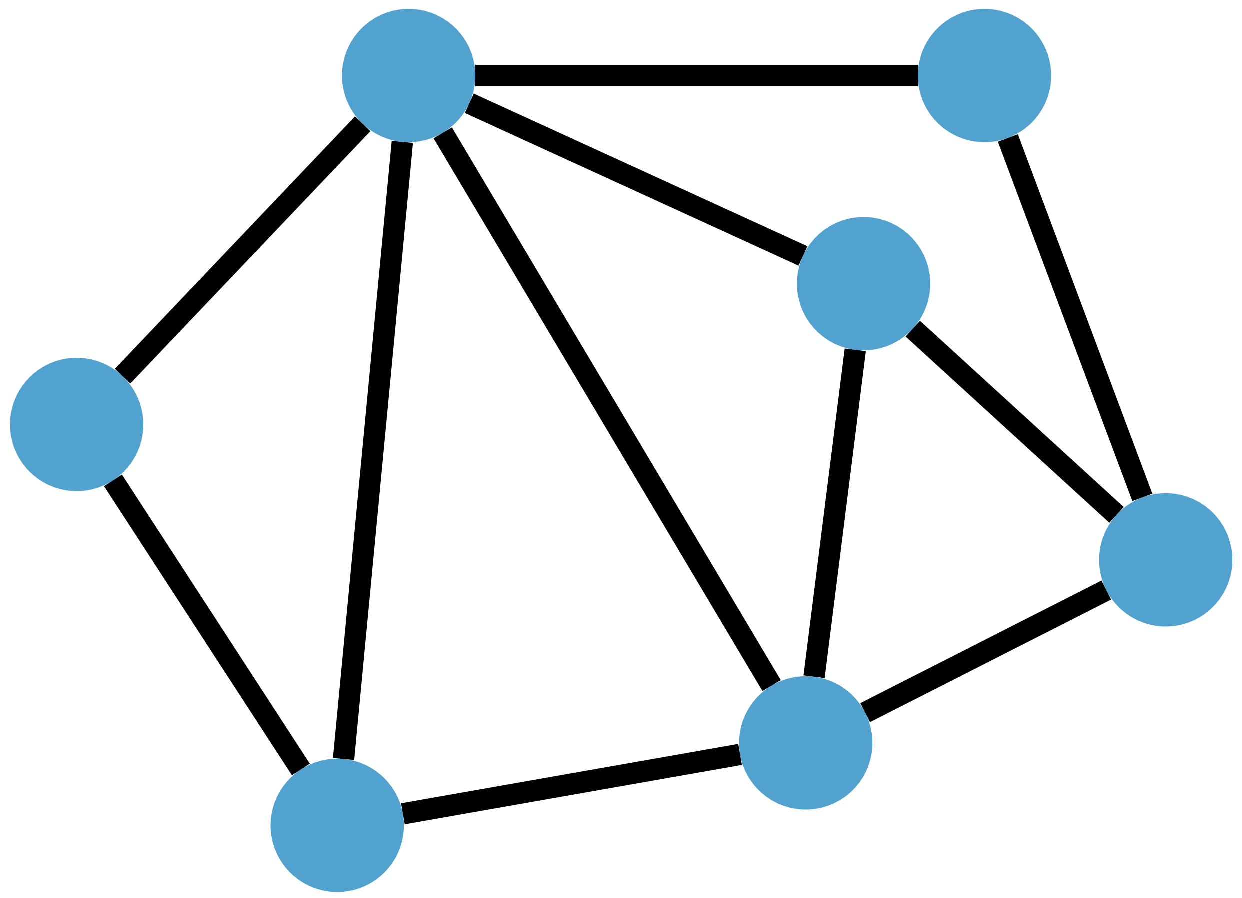


# Feedback Vertex Set (FVS)

- VertexCover $= \{\langle G, k\rangle \mid$ There is a set of at most $k$ vertices in $G$ such that removing them leaves no edges$\}$
- FVS $= \{\langle G', k'\rangle \mid$ There is a set of at most $k'$ vertices in $G'$ such that removing them leaves no cycles$\}$

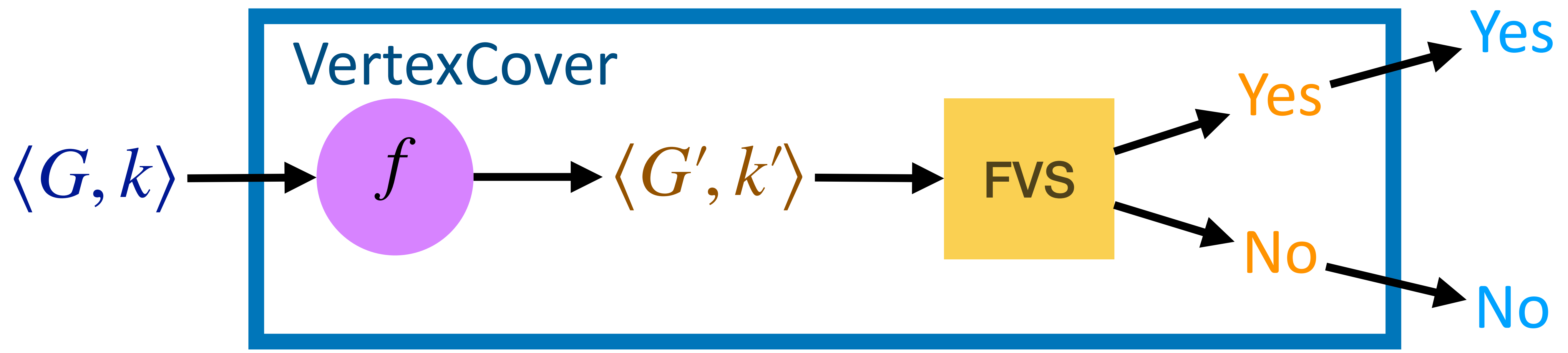


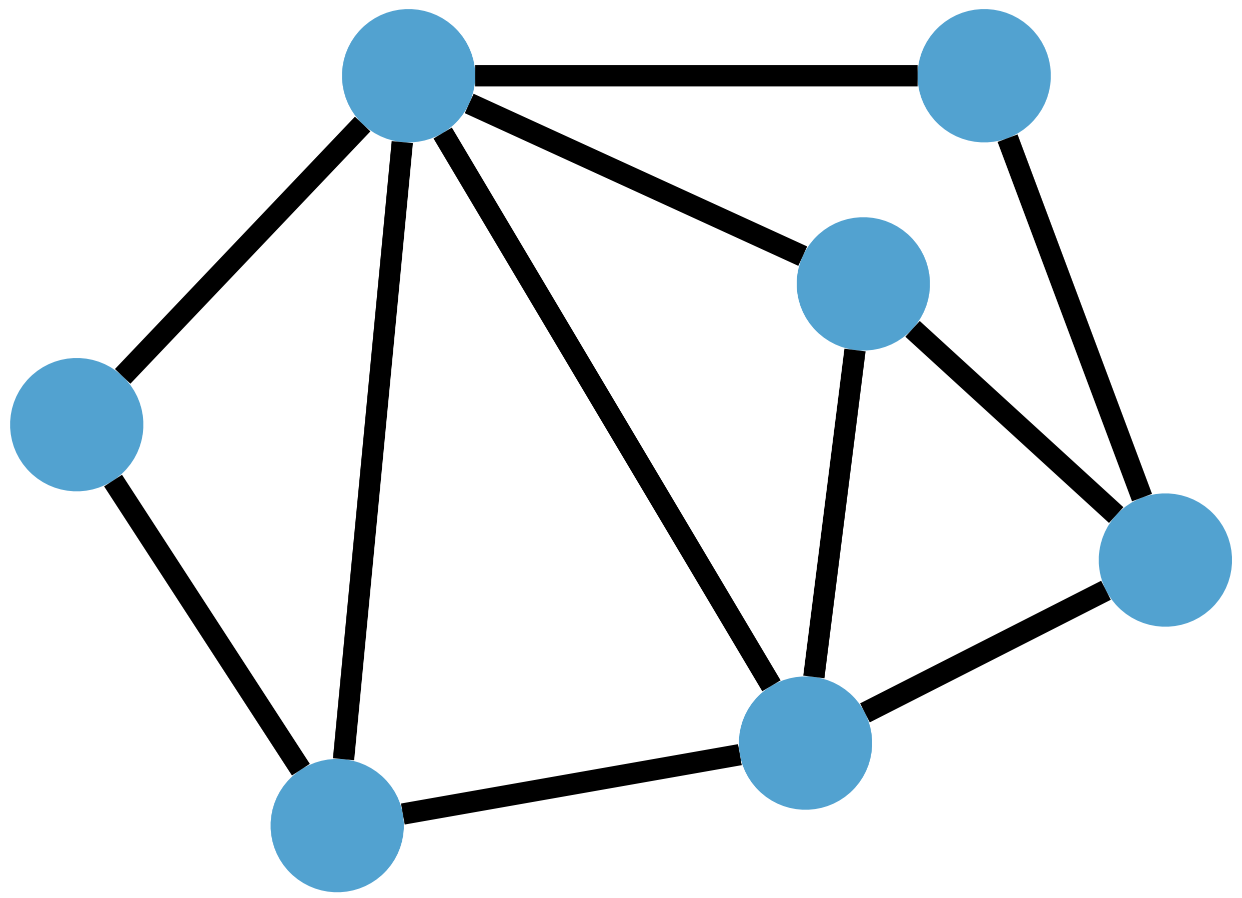



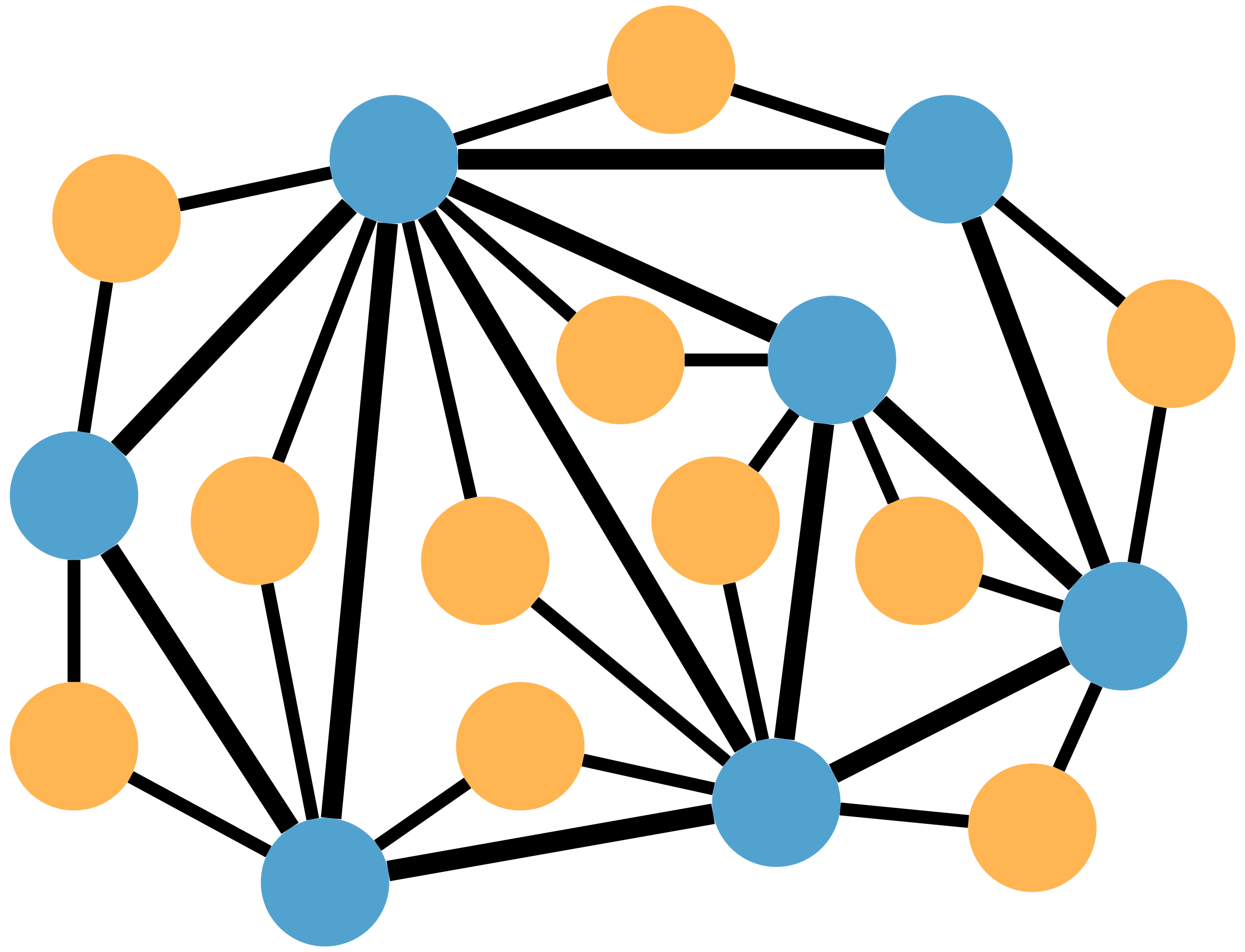


# Feedback Vertex Set (FVS)

- VertexCover $= \{\langle G, k\rangle \mid$ There is a set of at most $k$ vertices in $G$ such that removing them leaves no edges$\}$
- FVS $= \{\langle G', k'\rangle \mid$ There is a set of at most $k'$ vertices in $G'$ such that removing them leaves no cycles$\}$

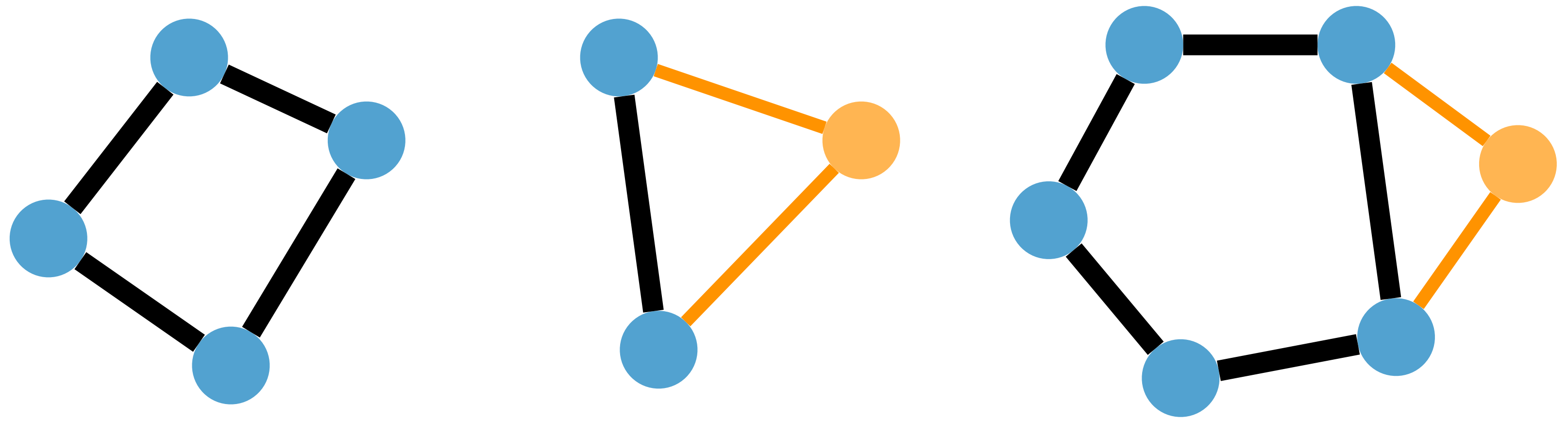


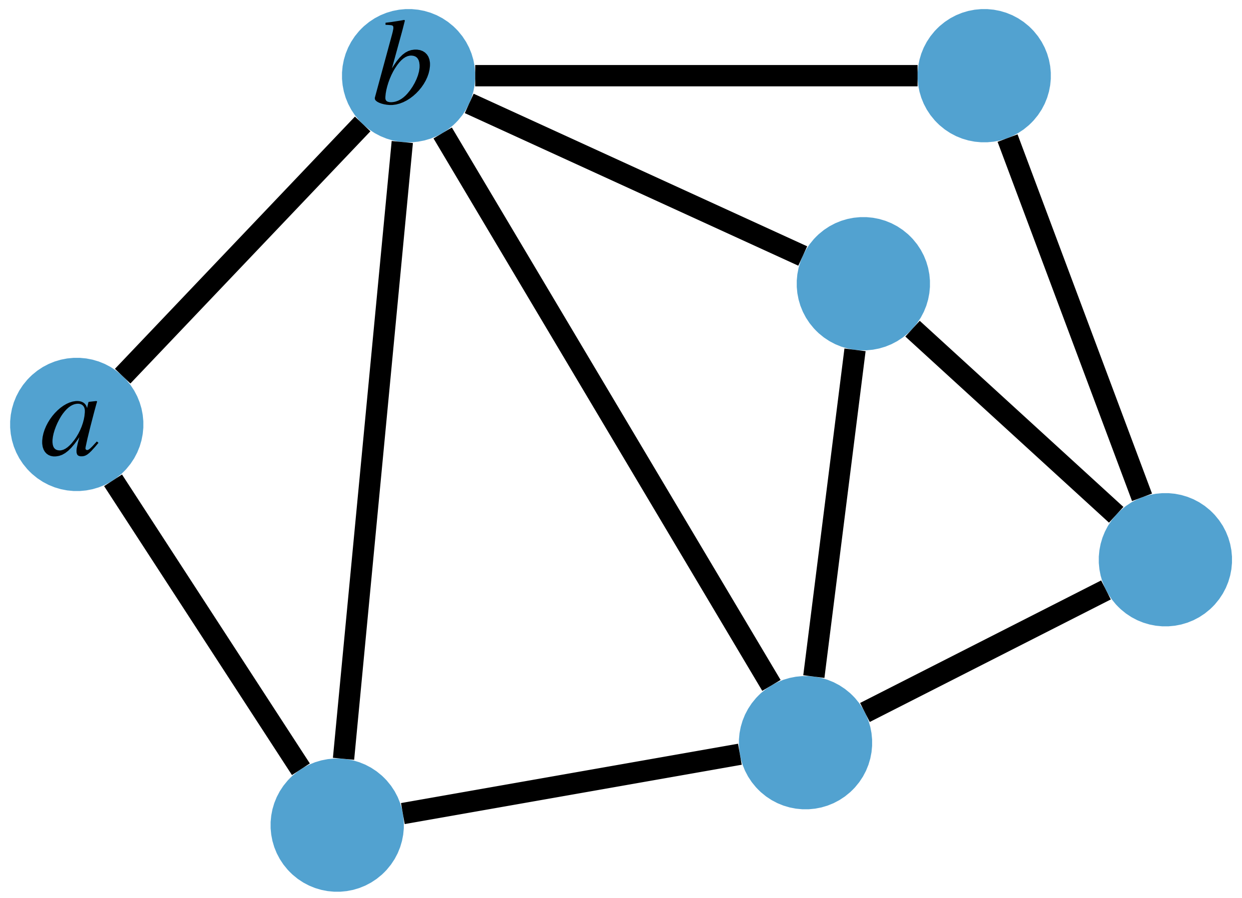



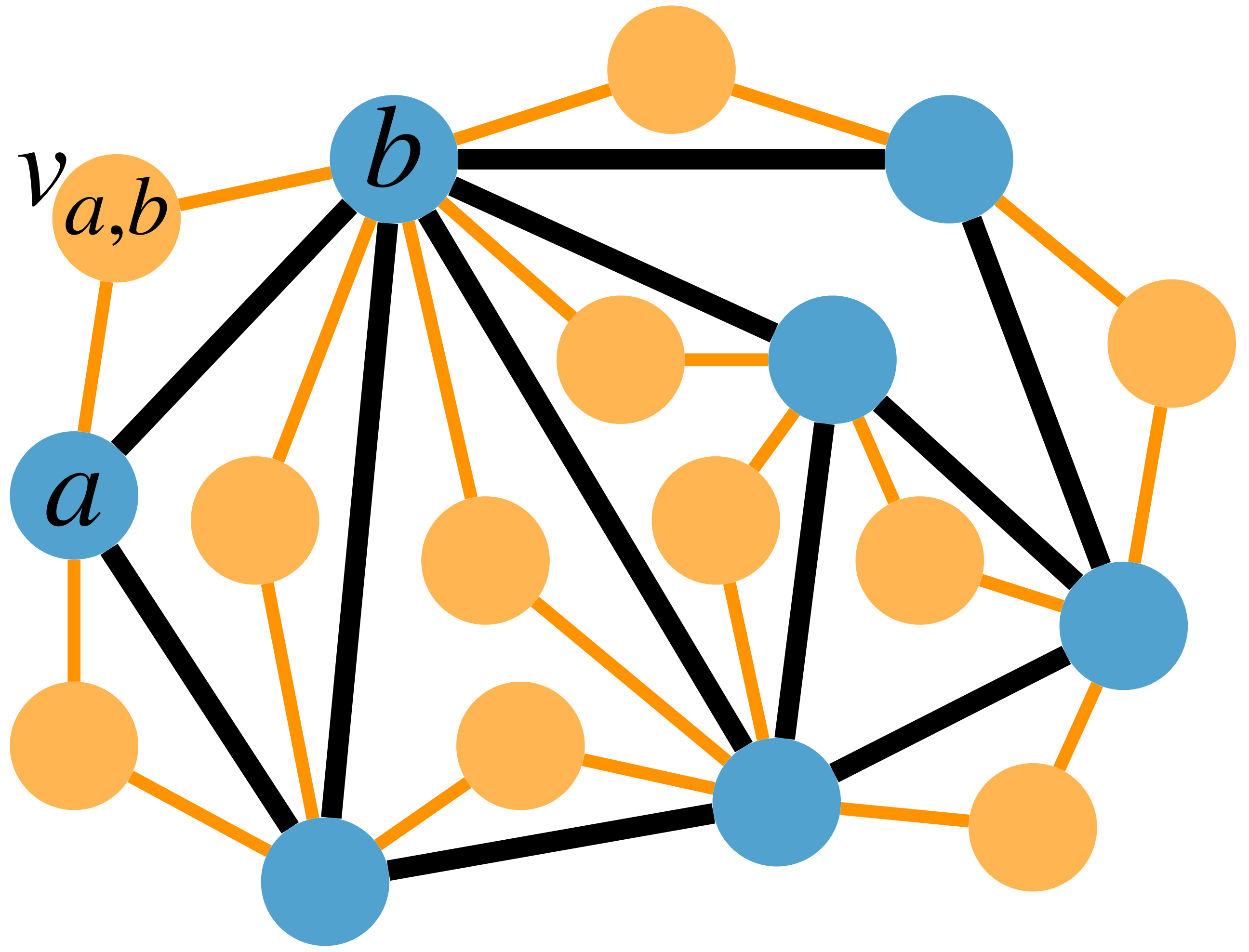


# Feedback Vertex Set (FVS)

- VertexCover $= \{\langle G, k\rangle \mid$ There is a set of at most $k$ vertices in $G$ such that removing them leaves no edges$\}$
- FVS $= \{\langle G', k'\rangle \mid$ There is a set of at most $k'$ vertices in $G'$ such that removing them leaves no cycles$\}$





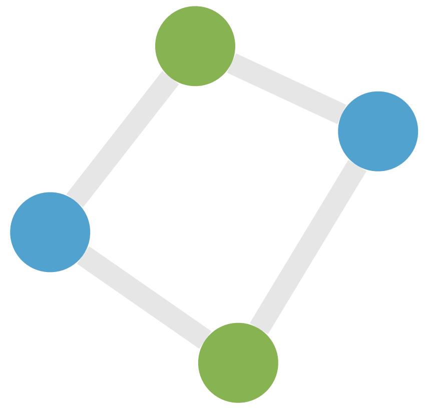

$G' = (V', E')$

$V' = V \cup V_E$

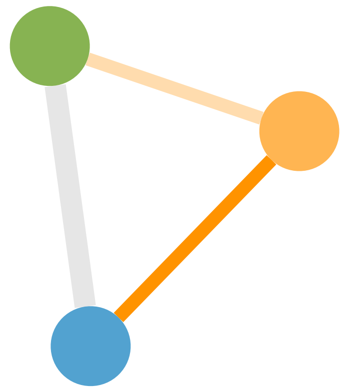
$E' = E \cup E_{V,E}$

# Feedback Vertex Set (FVS)

After removing the size-$k$ vertex cover from $G'$:



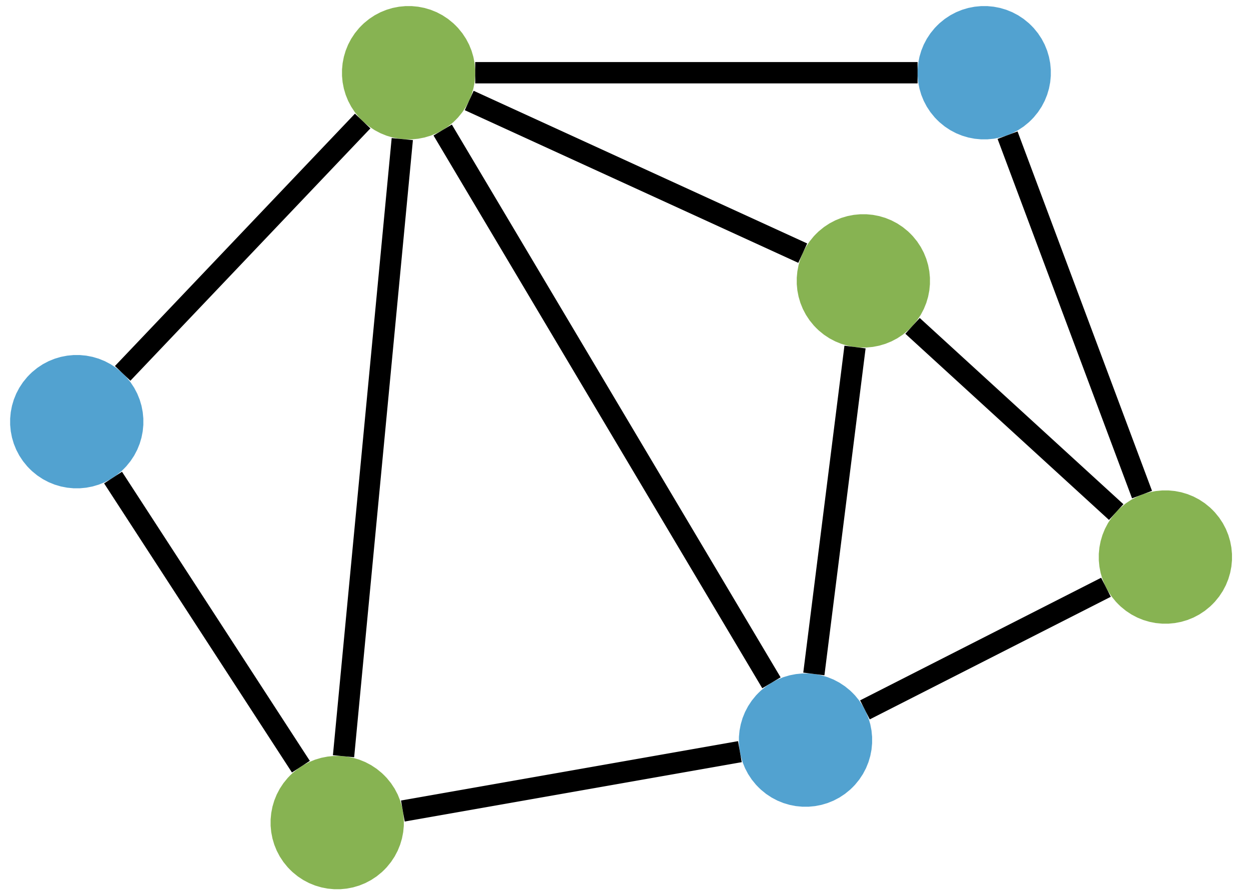
There is no edge between the remaining $V$ vertices

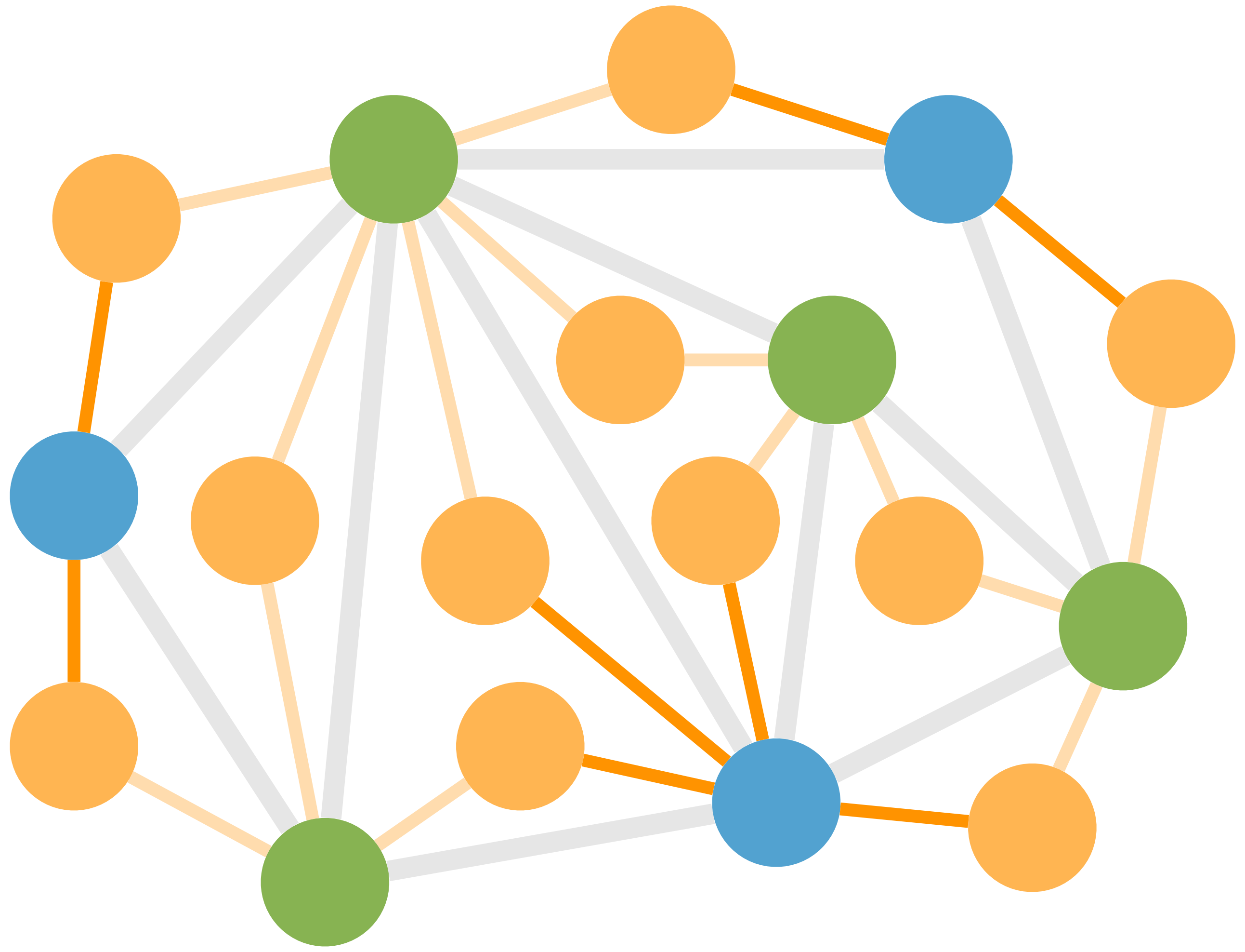


The $E_{V,E}$ vertices have degree at most $1$

The size-$k$ vertex cover

The size-$k$ vertex cover is a FVS of $G'$





$G' = (V', E')$

$V' = V \cup V_E$

$E' = E \cup E_{V,E}$

# Feedback Vertex Set (FVS)

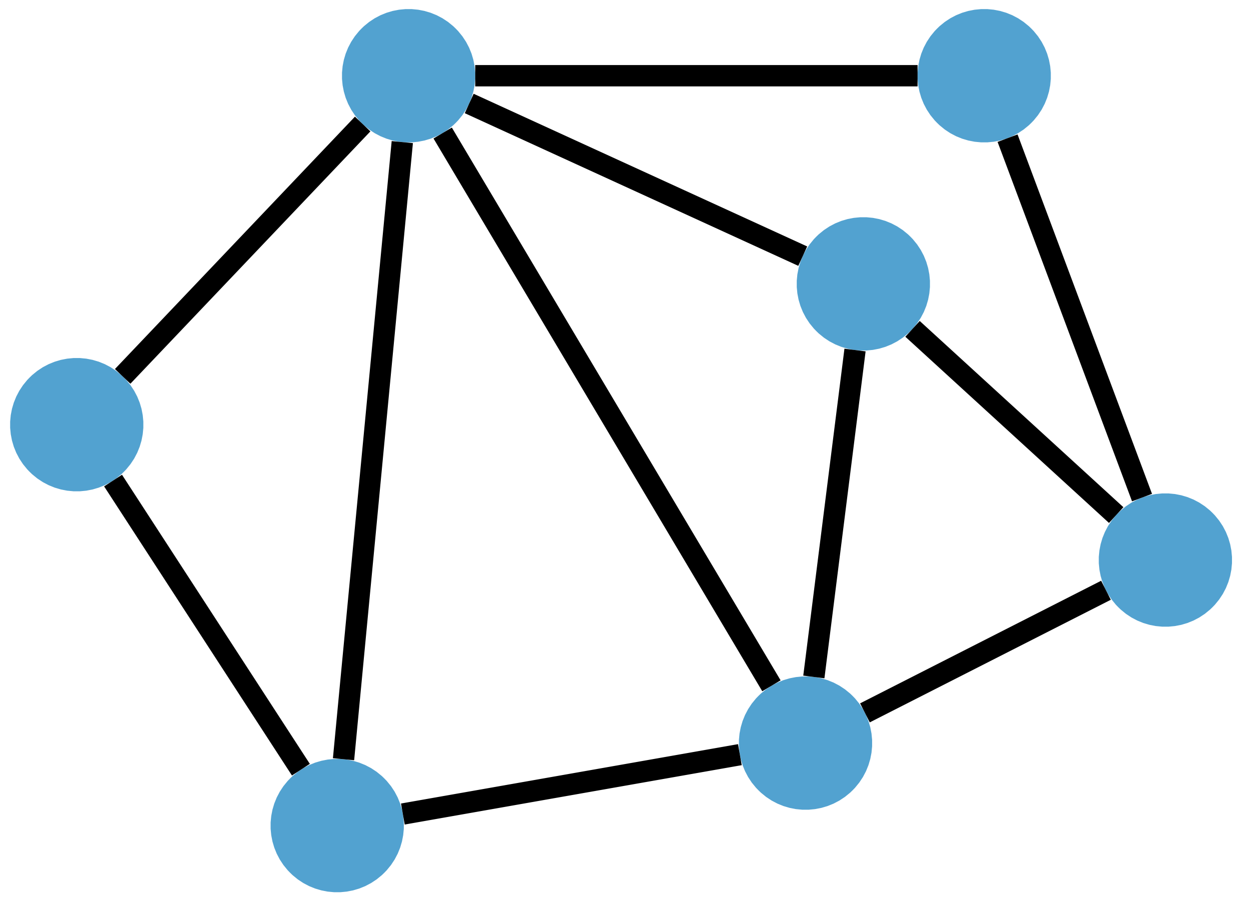
The size-$k$ FVS is a vertex cover of $G$ ⬅ The size-$k$ FVS of $G'$

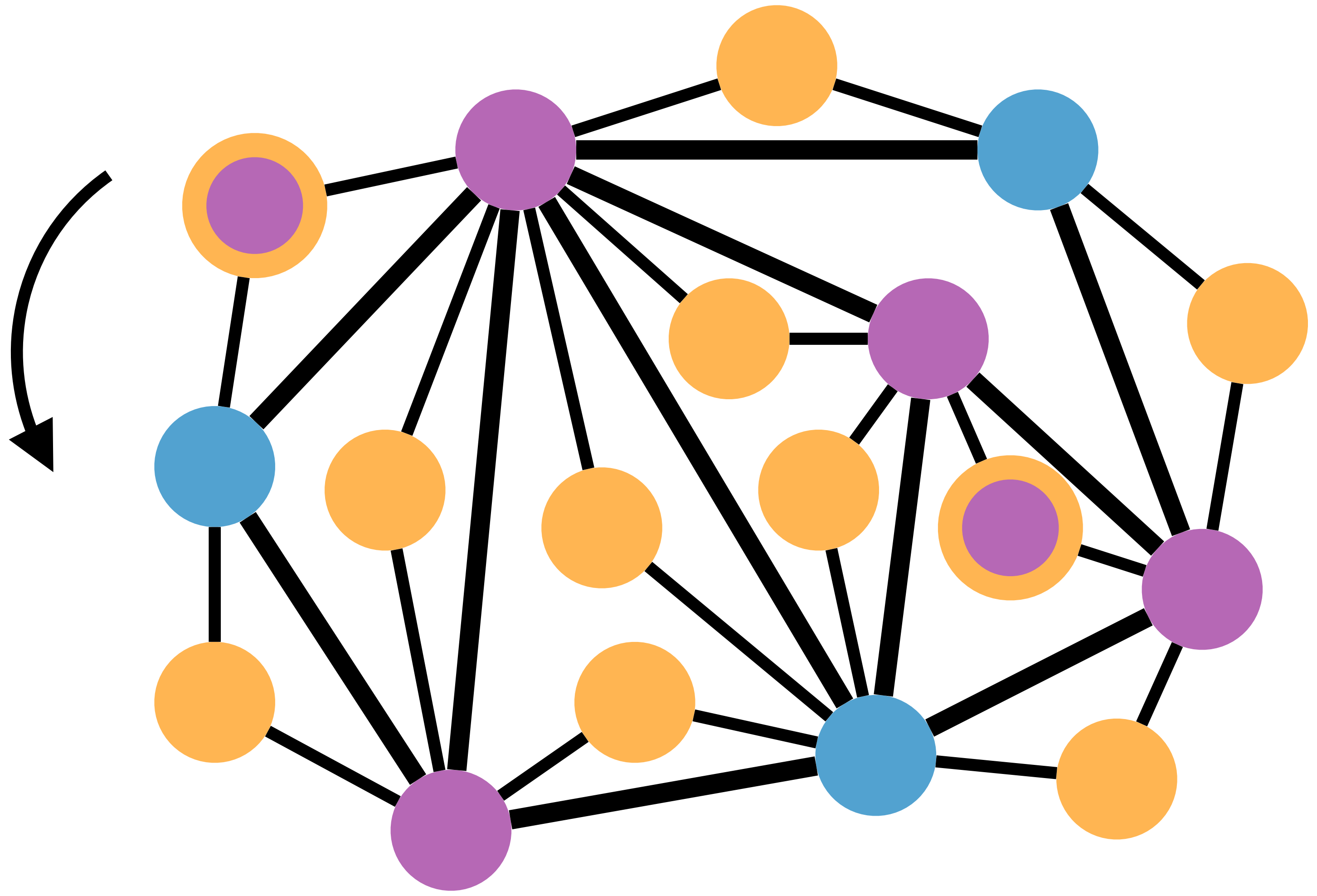
Construct a set $F'$:

Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$





# Feedback Vertex Set (FVS)

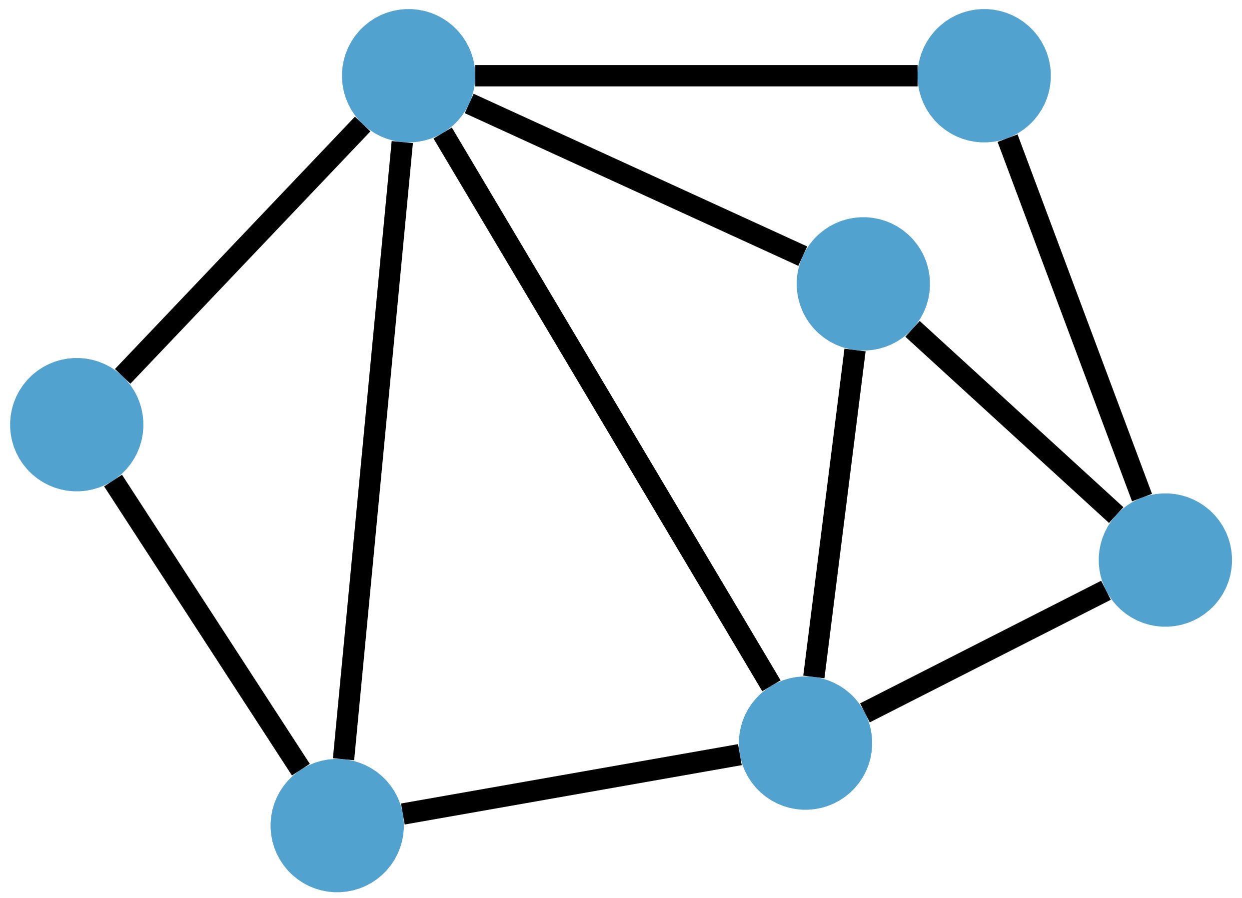
The size-$k$ FVS is a vertex cover of $G$ ⬅ The size-$k$ FVS of $G'$

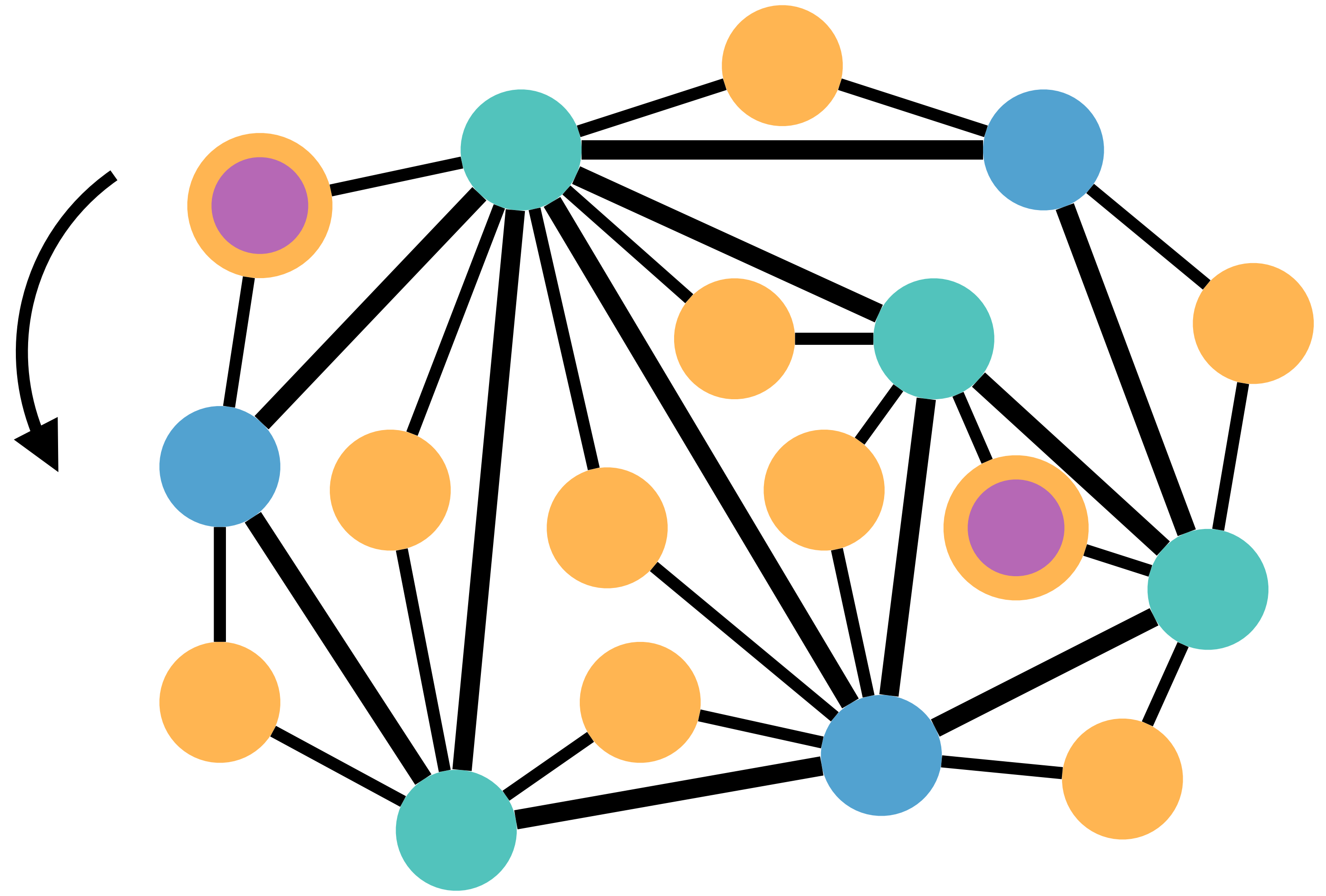
Construct a set $F'$:

Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$





# Feedback Vertex Set (FVS)

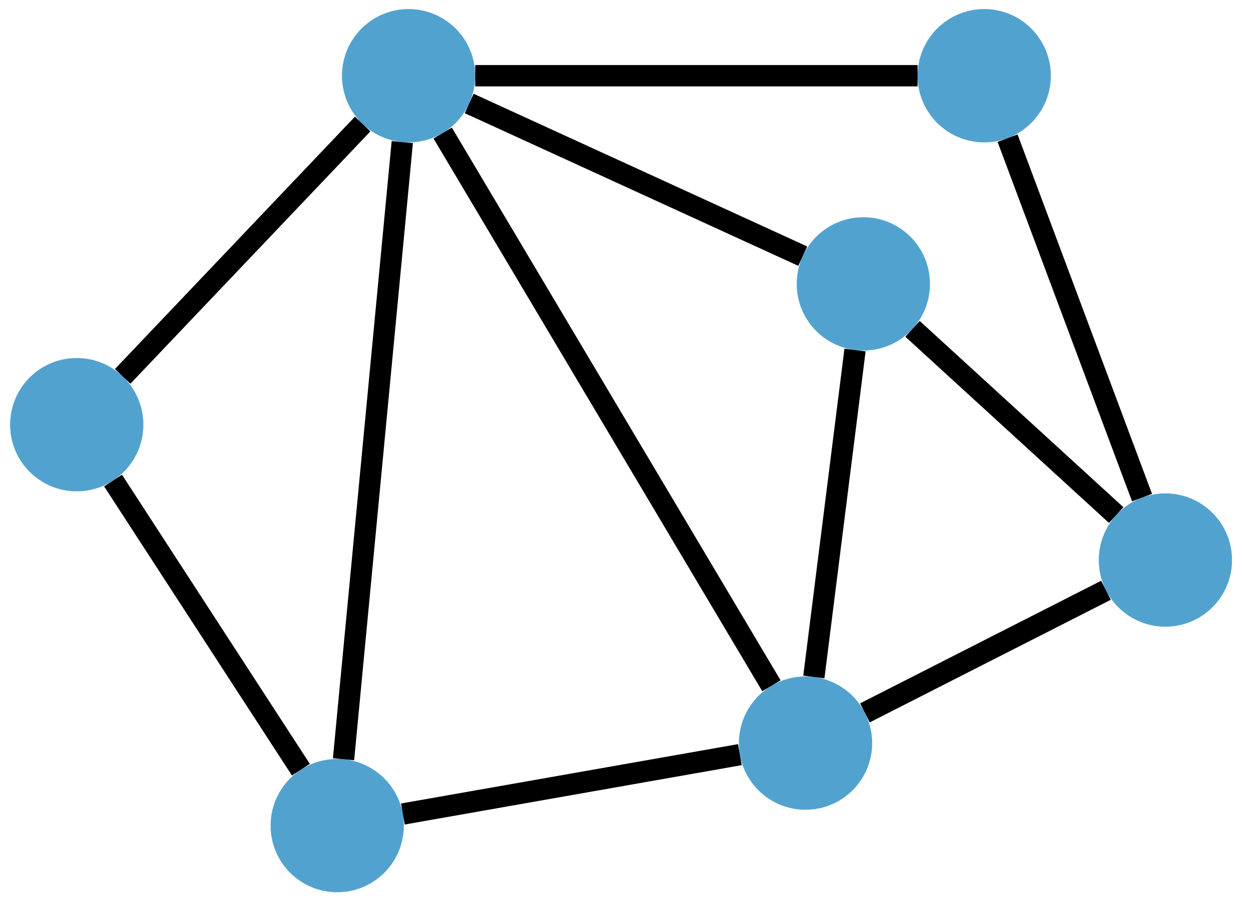
The size-$k$ FVS is a vertex cover of $G$ ⟸ The size-$k$ FVS of $G'$

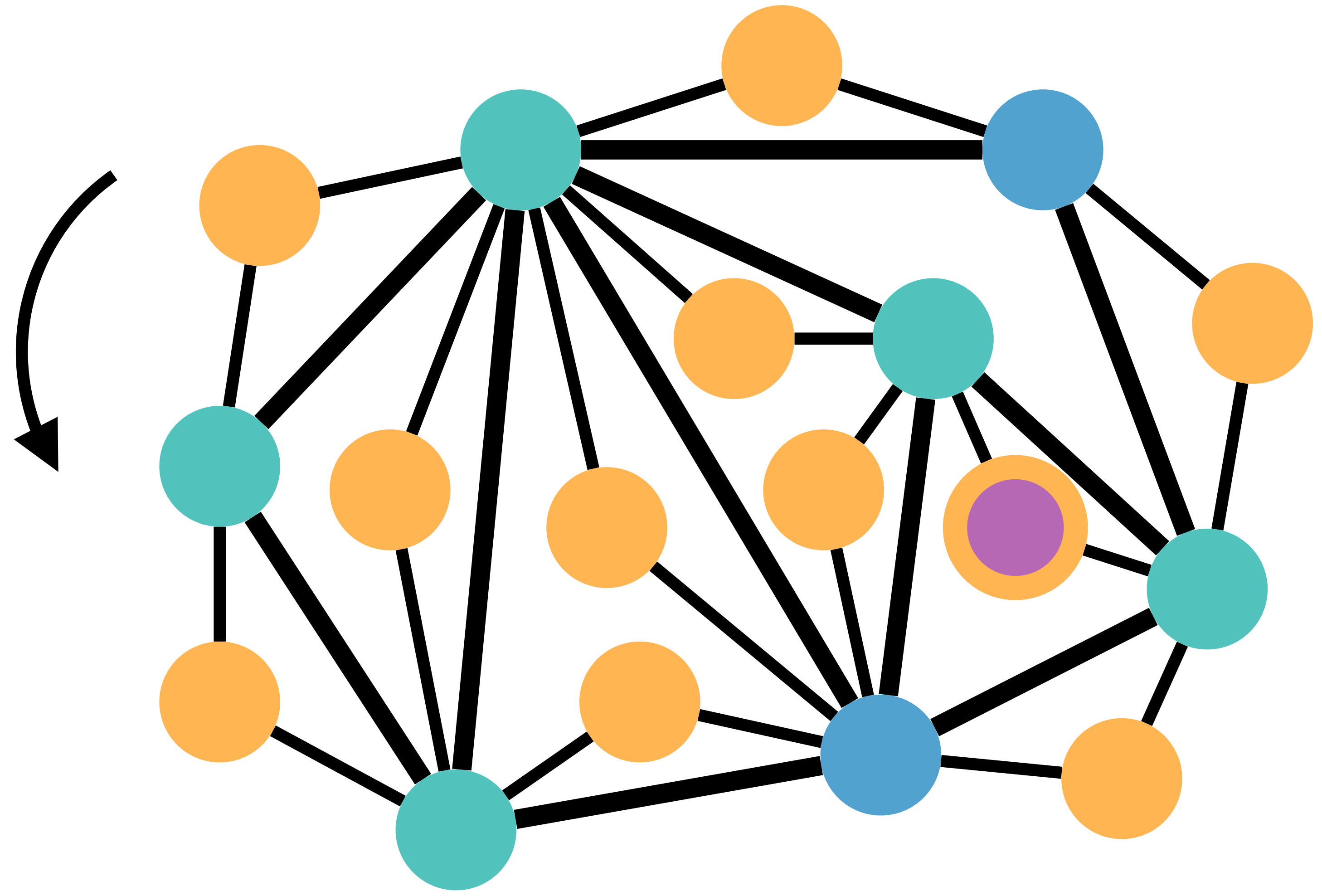
Construct a set $F'$:

Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$





# Feedback Vertex Set (FVS)

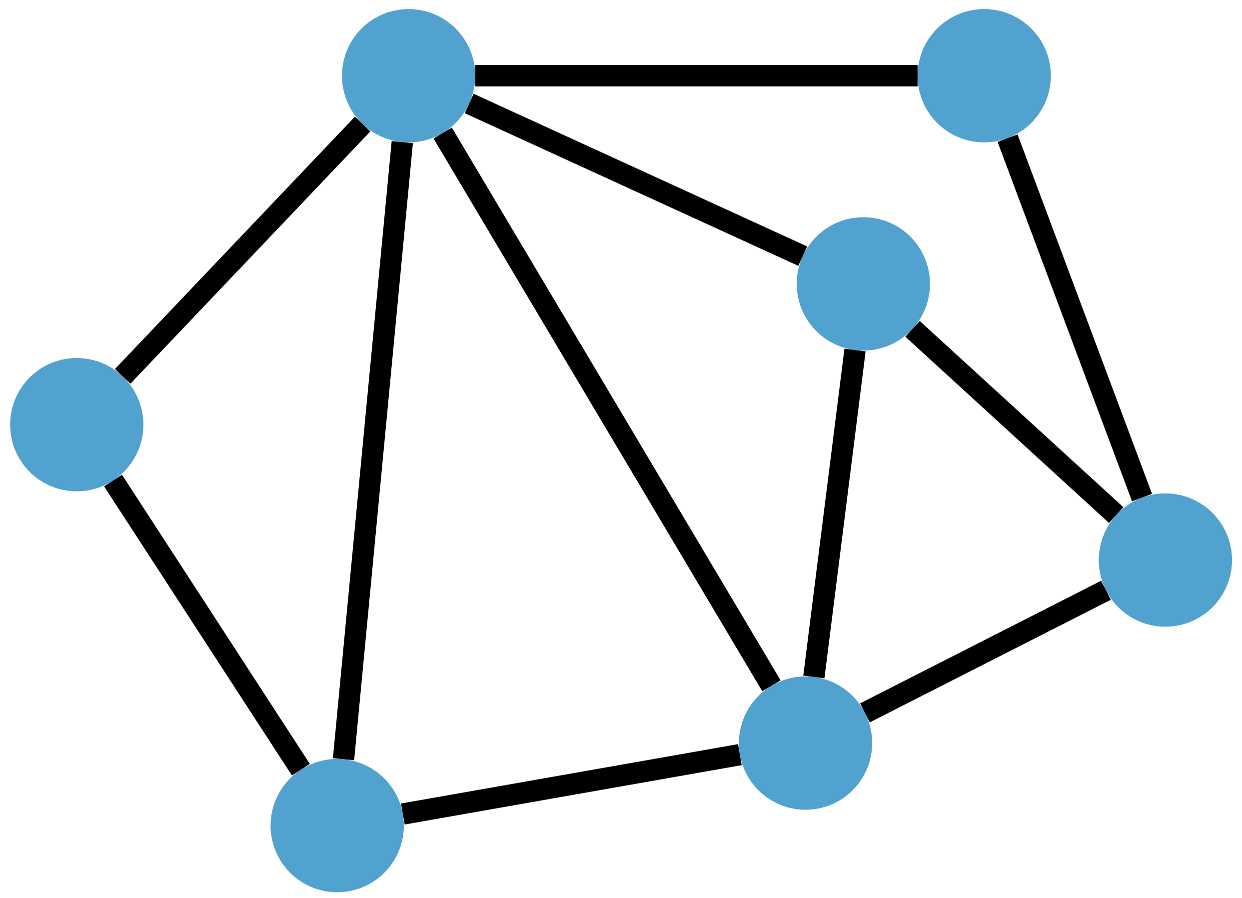
The size-$k$ FVS is a vertex cover of $G$ ⬅ The size-$k$ FVS of $G'$

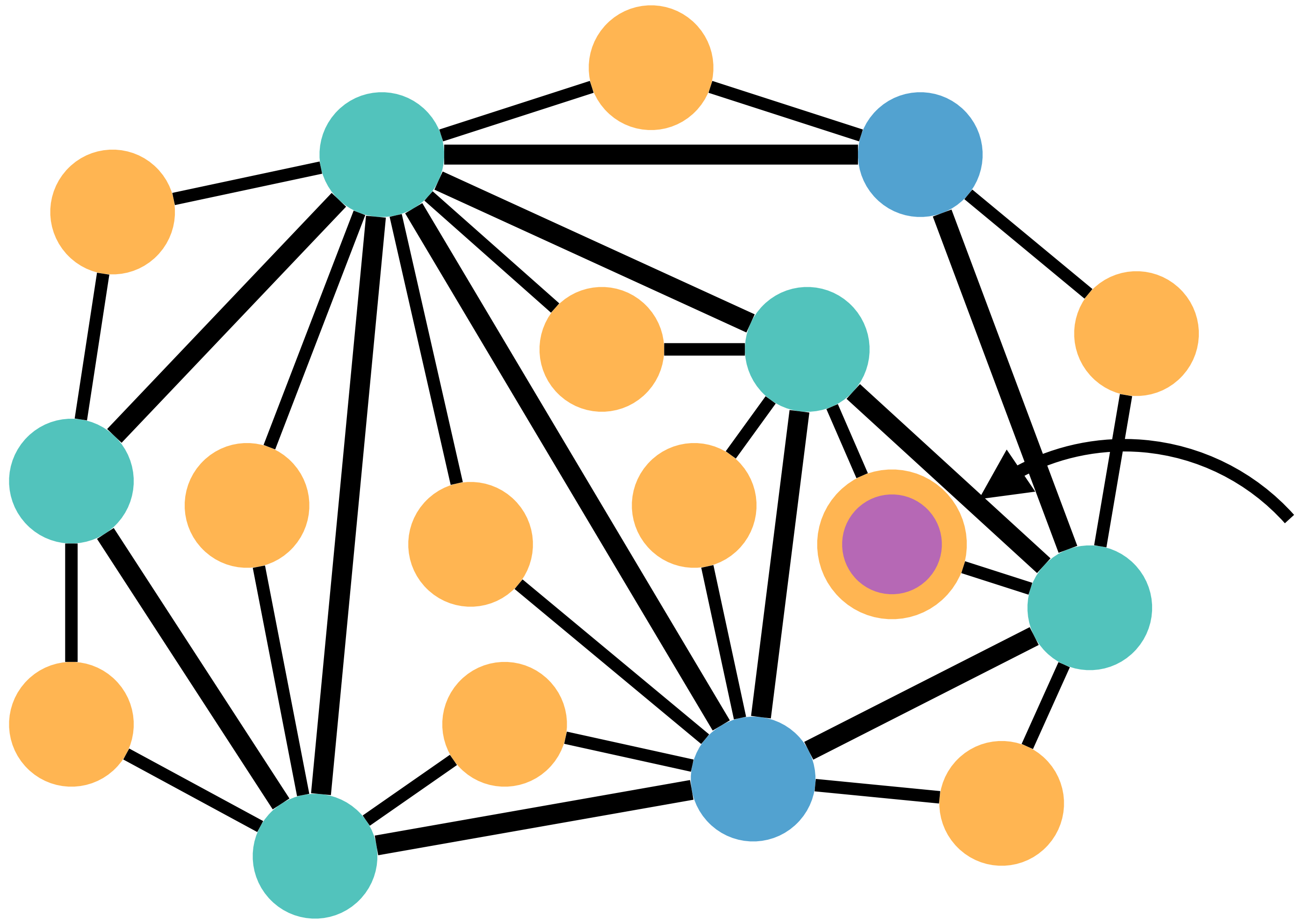
Construct a set $F'$:

Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$





# Feedback Vertex Set (FVS)

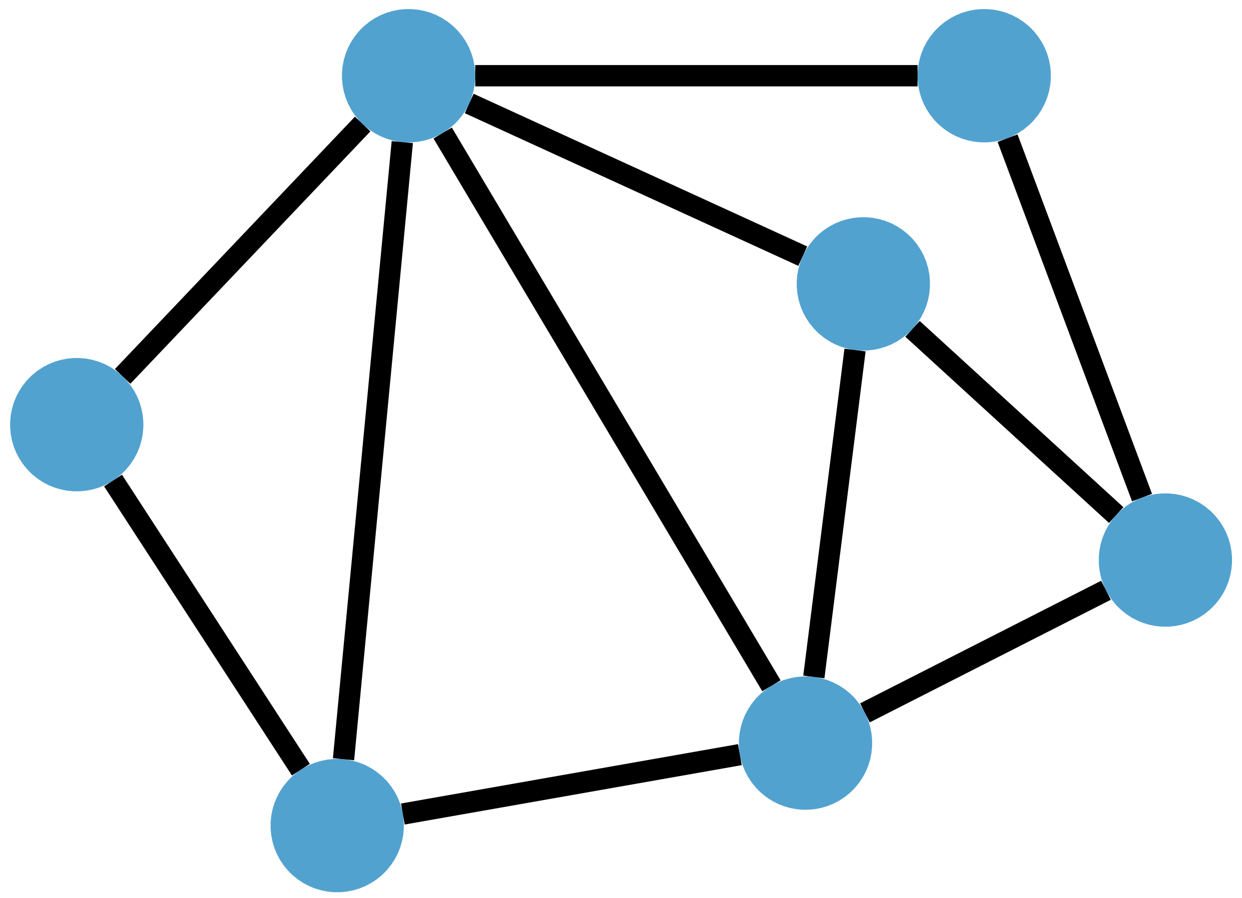
$F'$ is a vertex cover of $G$ ⇐ The size-$k$ FVS of $G'$

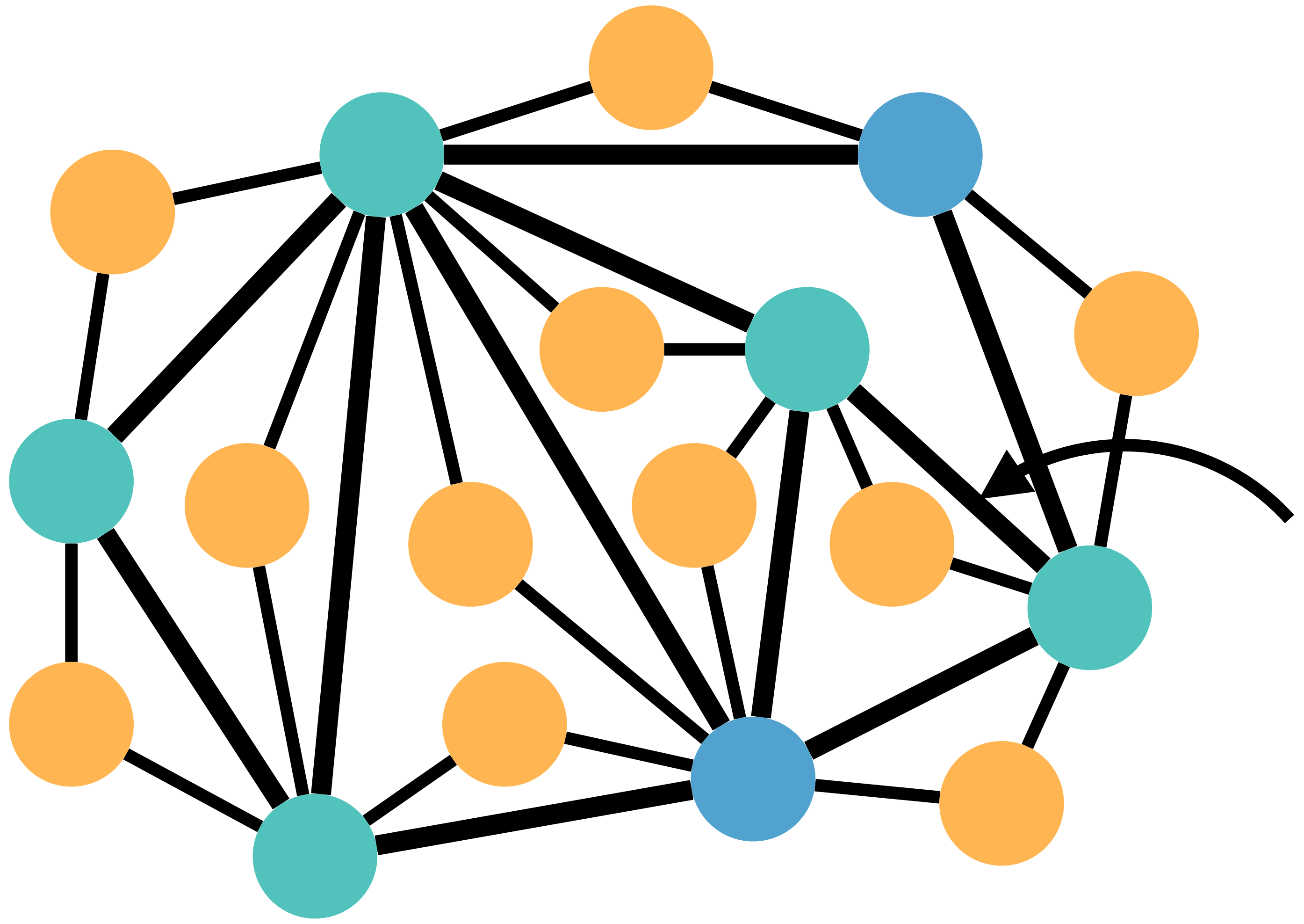
Construct a set $F'$:

Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$

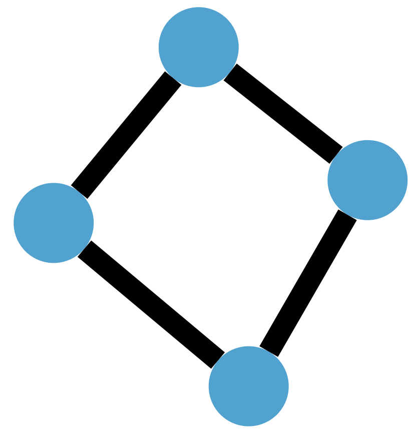




# Feedback Vertex Set (FVS)

$F'$ is a vertex cover of $G$

The size-$k$ FVS of $G'$



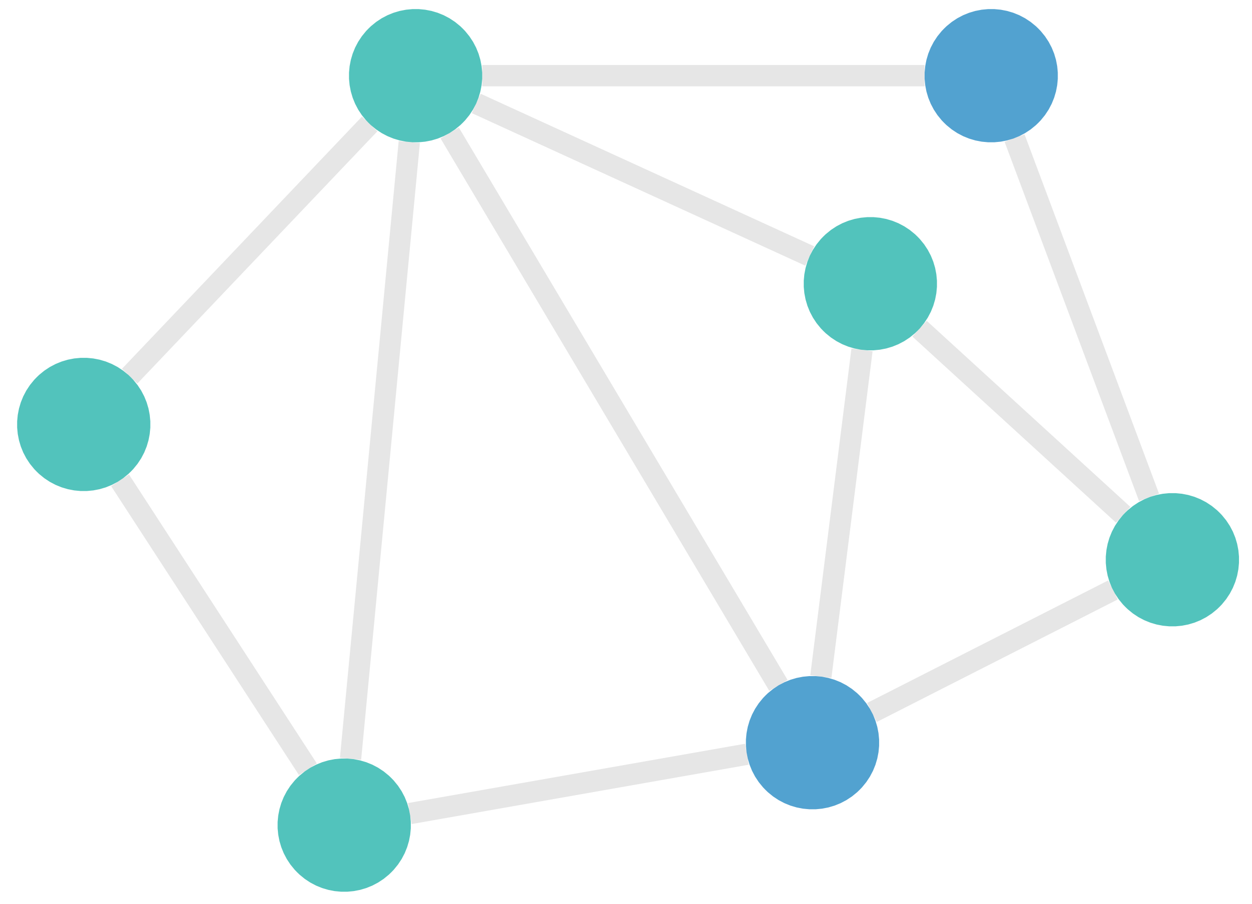
Construct a set $F'$:

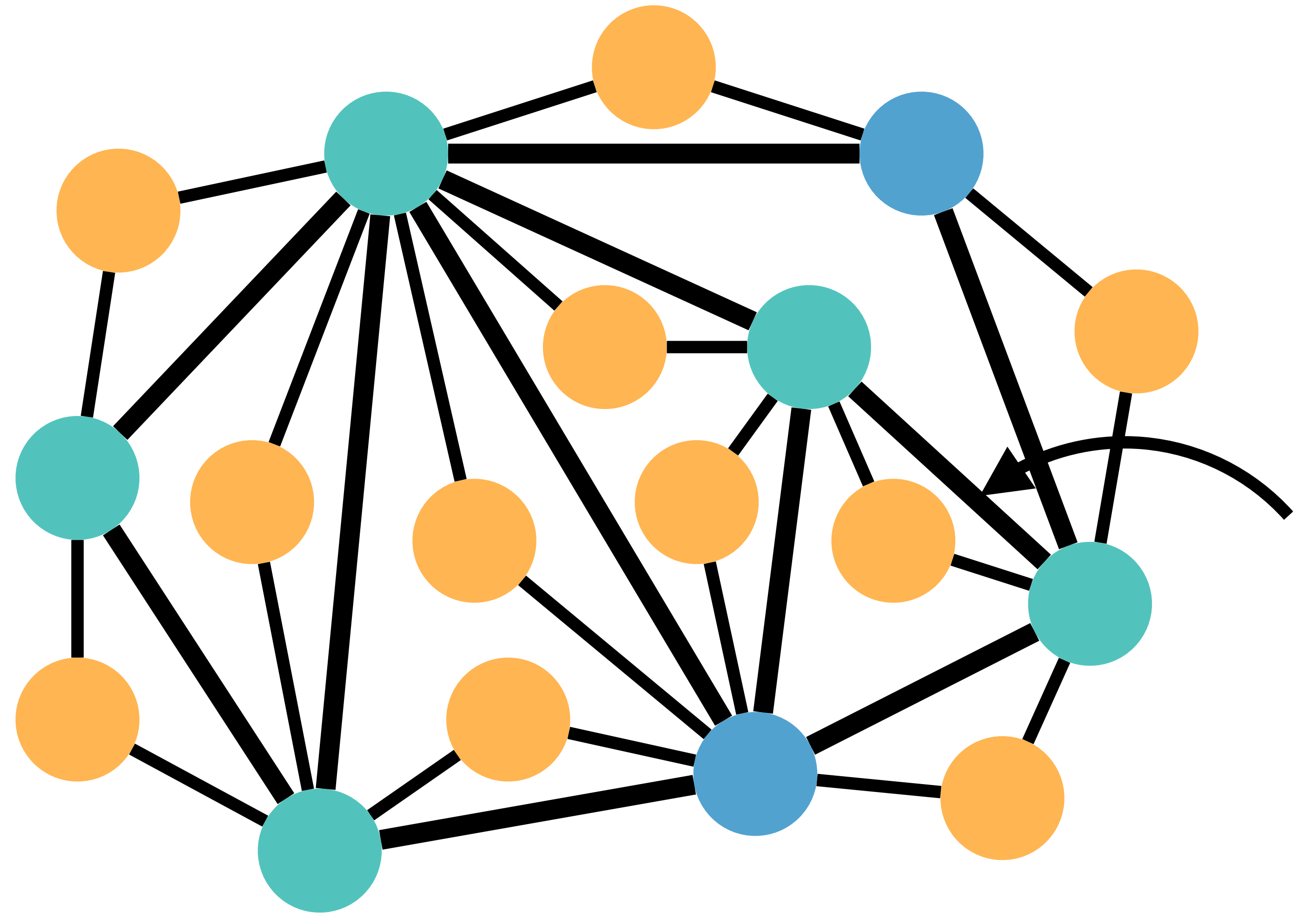
Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$

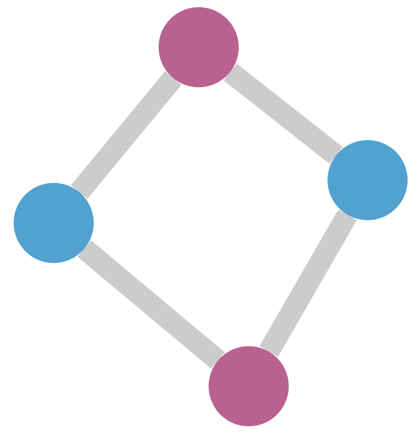
$\Rightarrow$ $F'$ is a FVS with size at most $k'$





# Feedback Vertex Set (FVS)

$F'$ is a vertex cover of $G$ ⇐ The size-$k$ FVS of $G'$



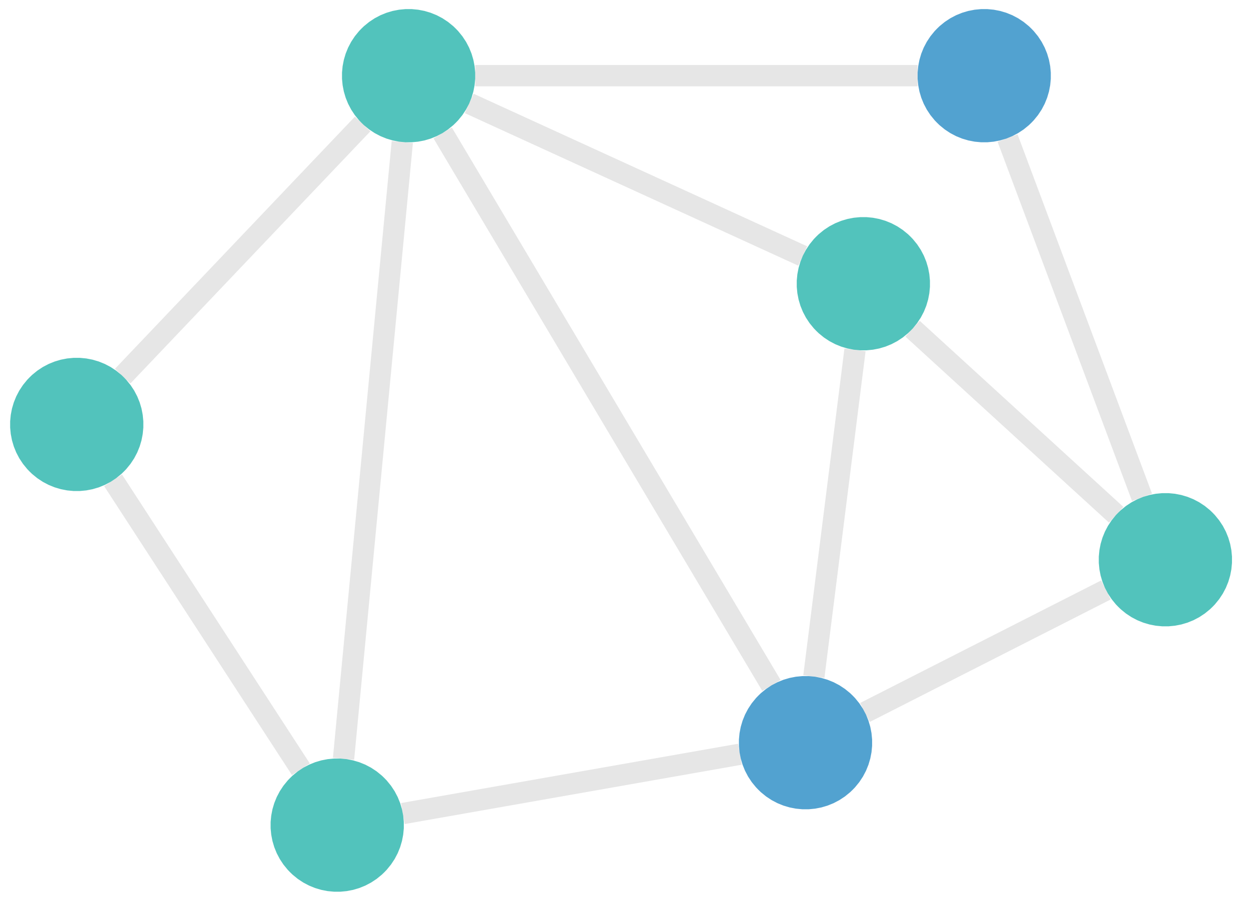
Construct a set $F'$:

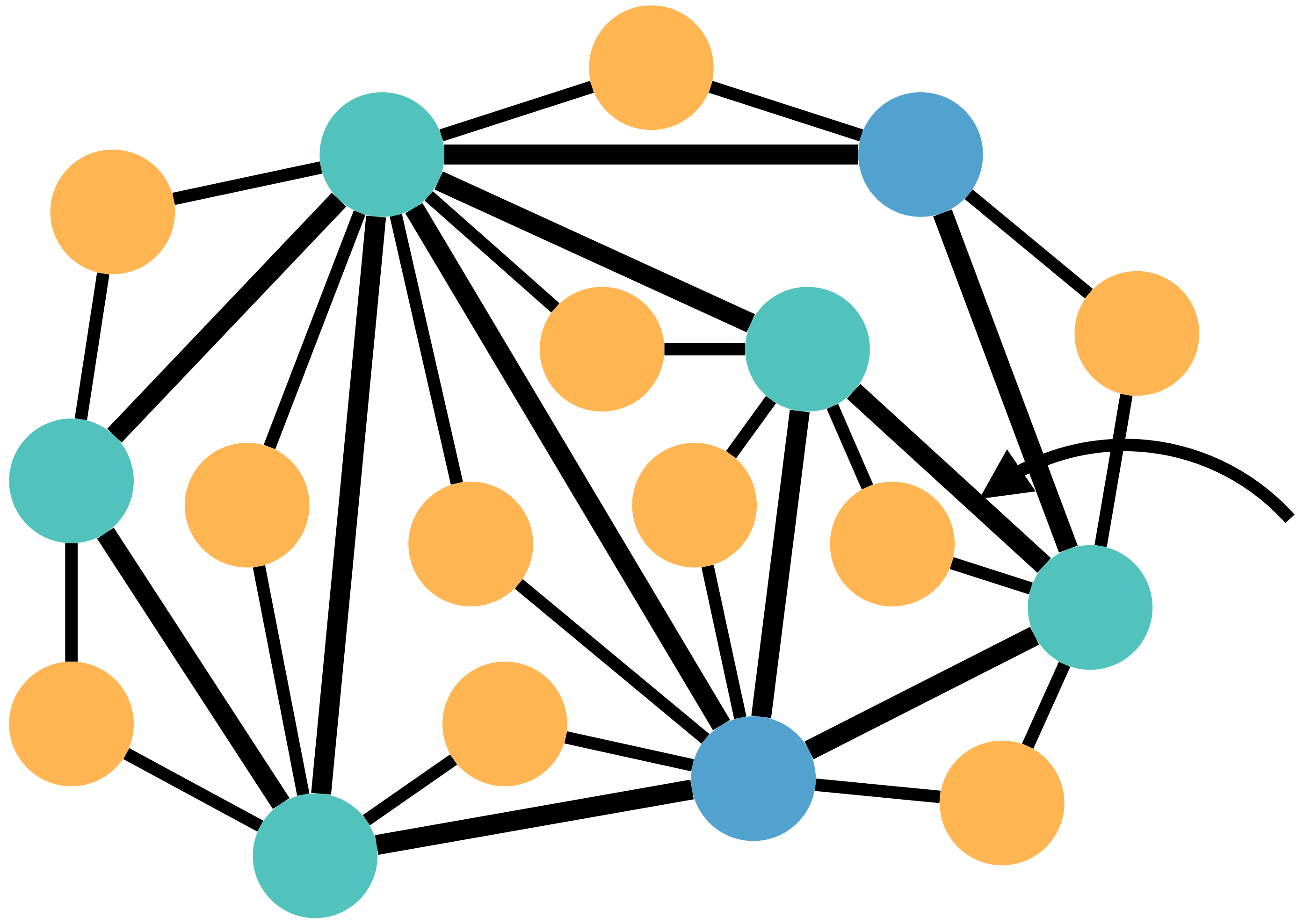
Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$

$\Rightarrow$ $F'$ is a FVS with size at most $k'$

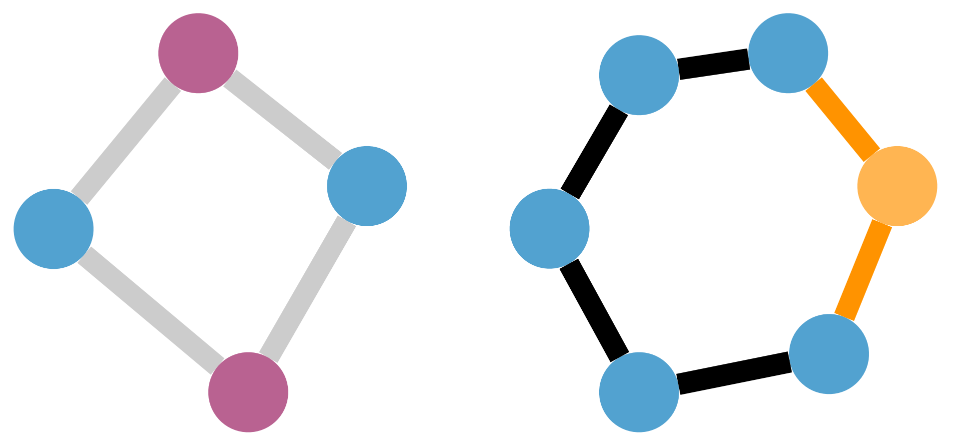




# Feedback Vertex Set (FVS)

$F'$ is a vertex cover of $G$

The size-$k$ FVS of $G'$



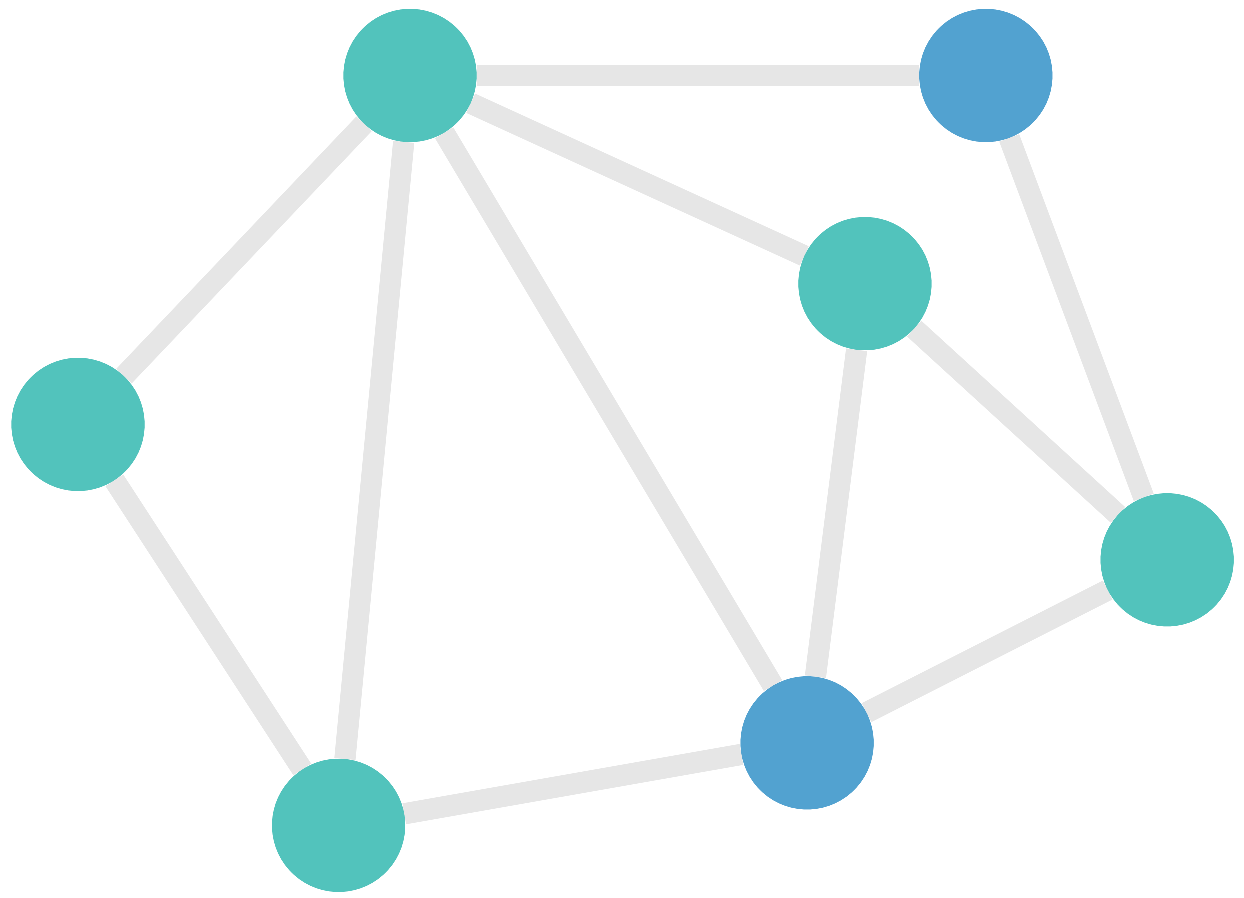
Construct a set $F'$:

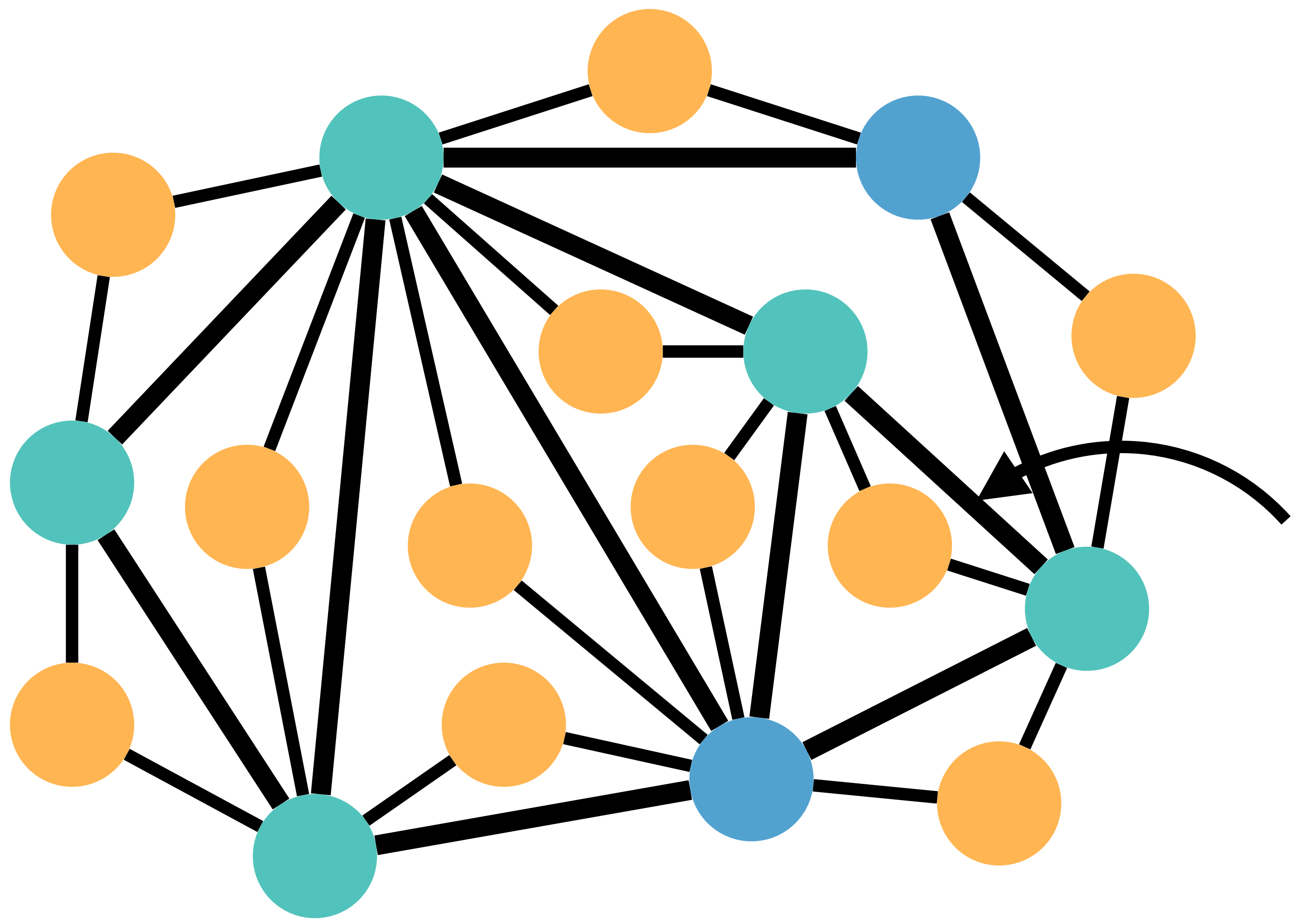
Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$

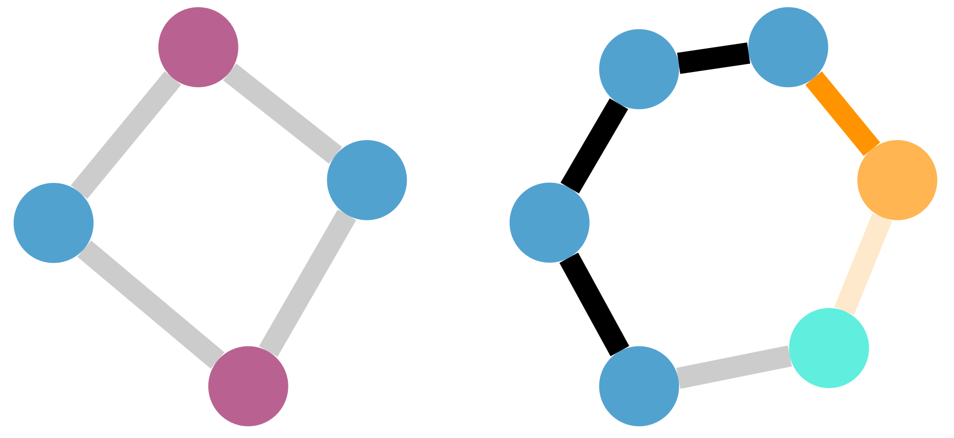
$\Rightarrow$ $F'$ is a FVS with size at most $k'$





# Feedback Vertex Set (FVS)

$F'$ is a vertex cover of $G$ ⇐ The size-$k$ FVS of $G'$



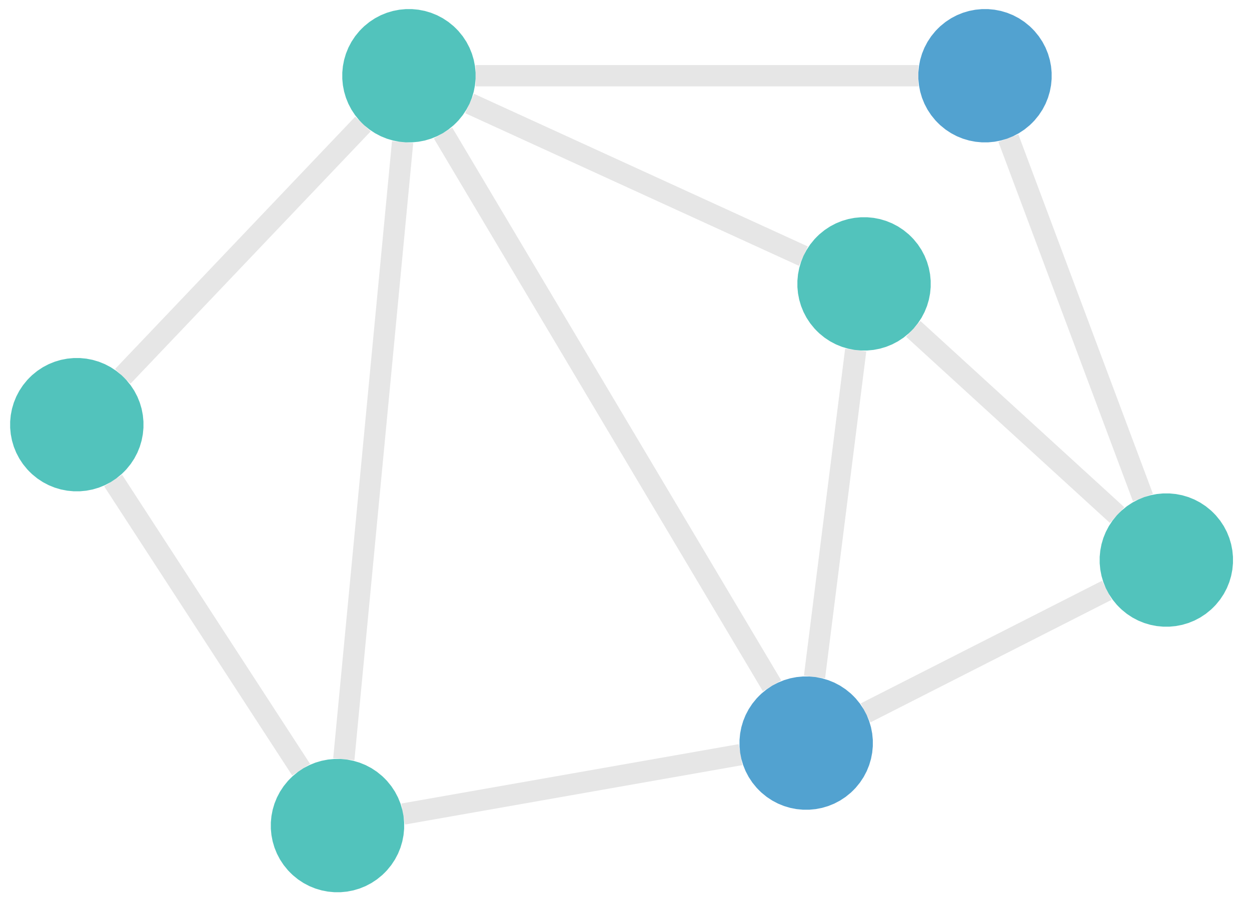
Construct a set $F'$:

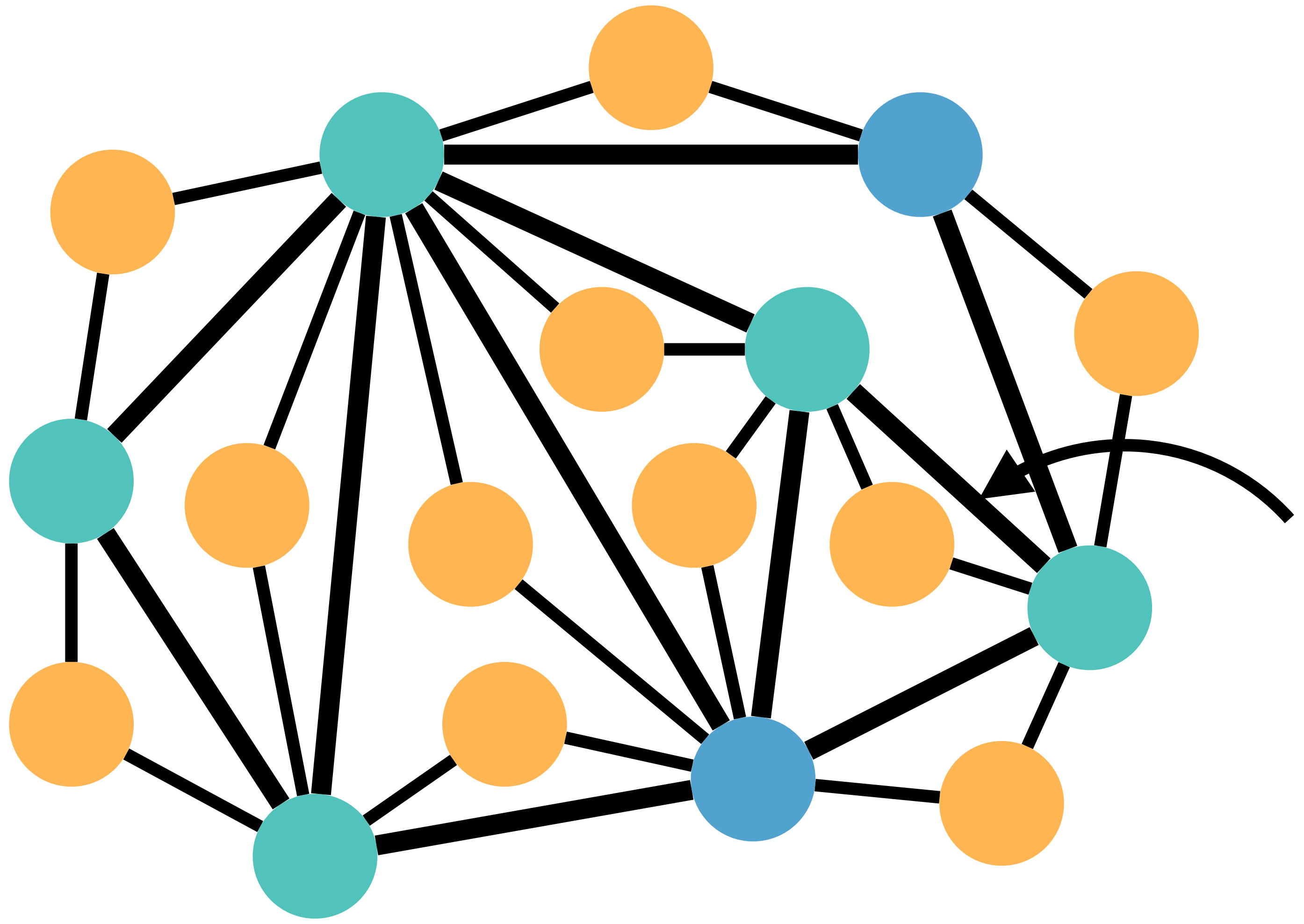
Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$

$\Rightarrow$ $F'$ is a FVS with size at most $k'$





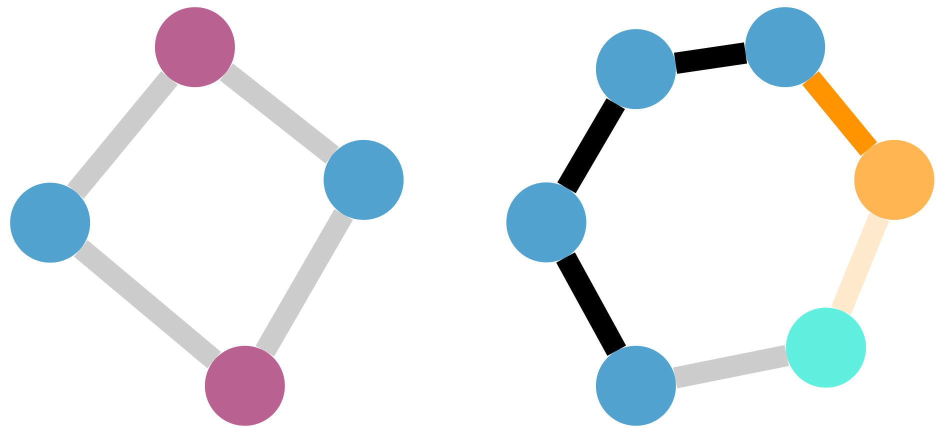
# Feedback Vertex Set (FVS)

$F'$ is a vertex cover of $G$

After removing $F'$ from $G'$, $G'$ has no cycle

$\Rightarrow$ After removing $F'$ from $G$, $G$ has no edge

The size-$k$ FVS of $G'$



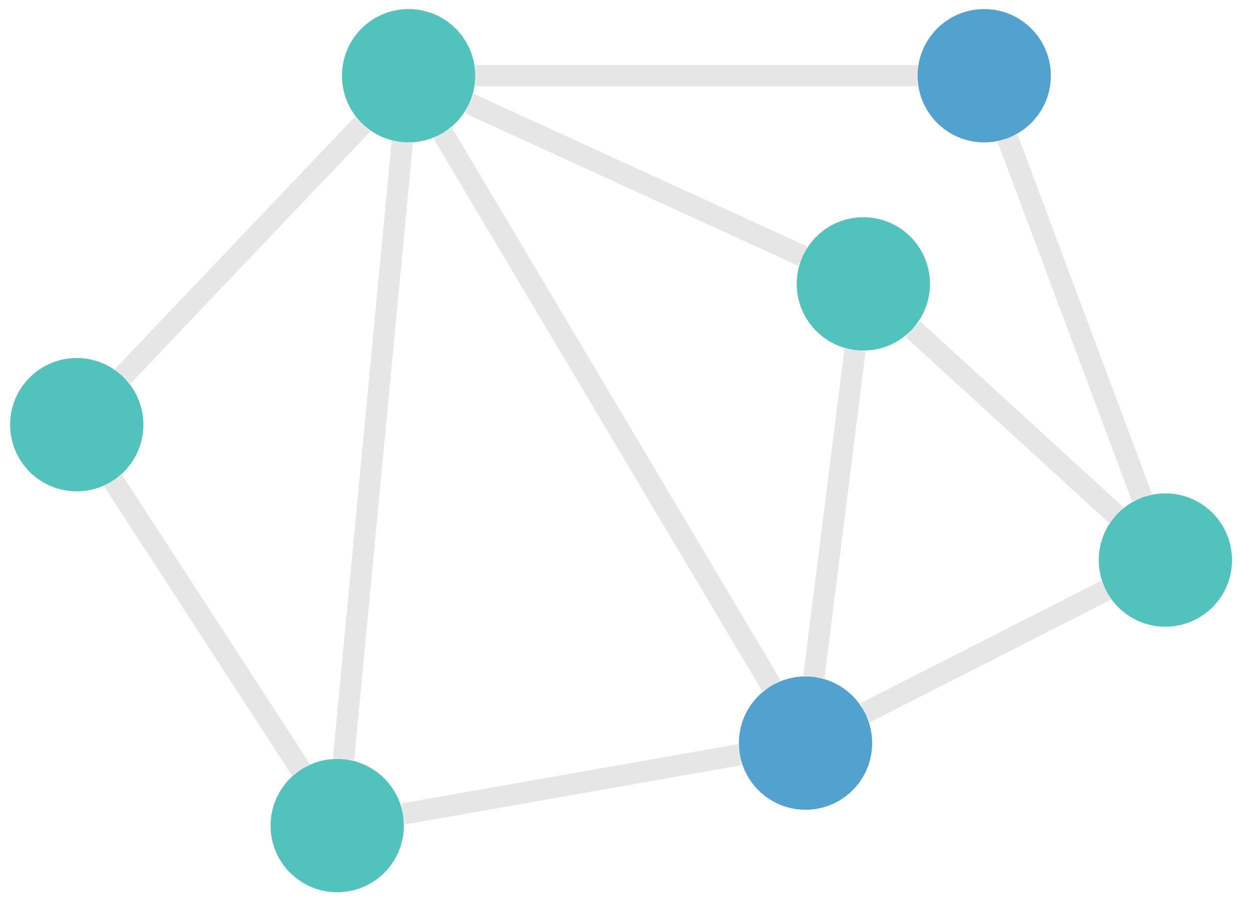
Construct a set $F'$:

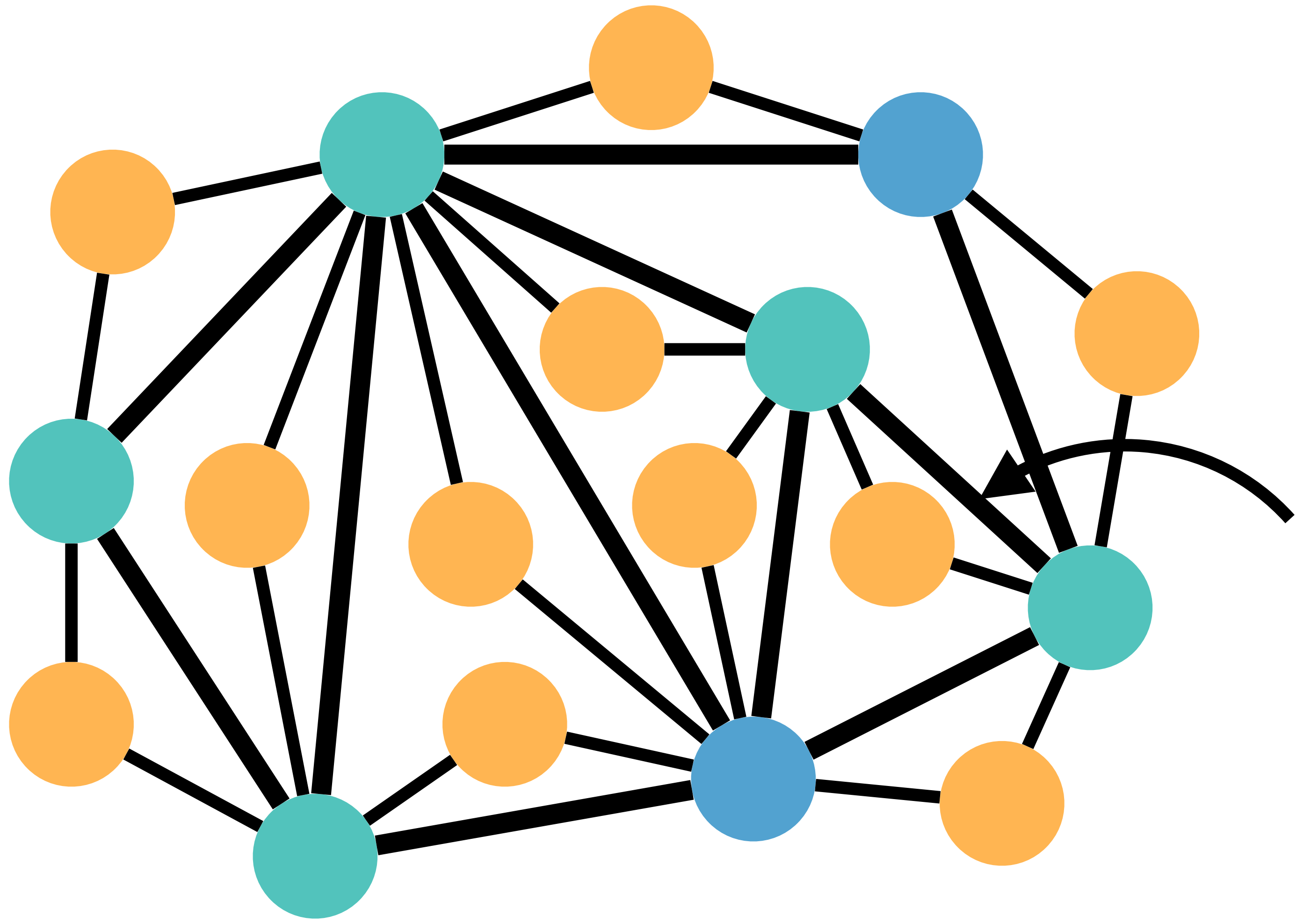
Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$

$\Rightarrow$ $F'$ is a FVS with size at most $k'$





# Feedback Vertex Set (FVS)

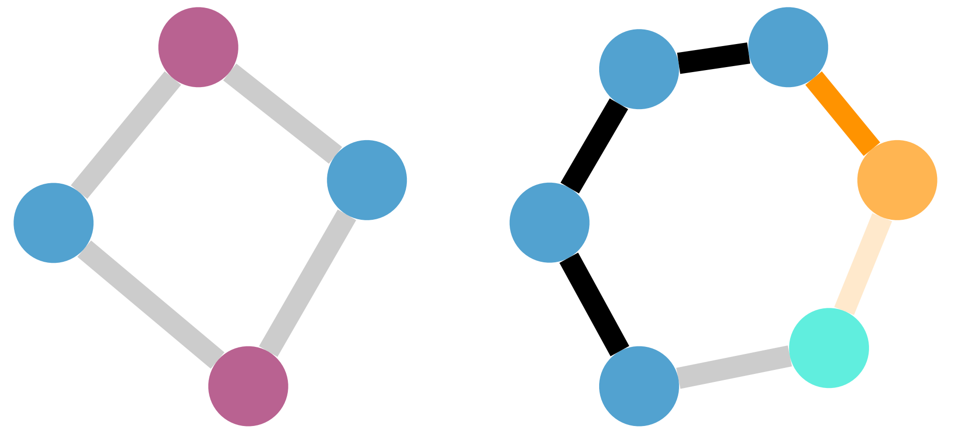
$F'$ is a vertex cover of $G$

After removing $F'$ from $G'$, $G'$ has no cycle

$\Rightarrow$ After removing $F'$ from $G$, $G$ has no edge

Otherwise, there is a cycle in $G'$ since no $u_{i,j}$ is in $F'$)

The size-$k$ FVS of $G'$



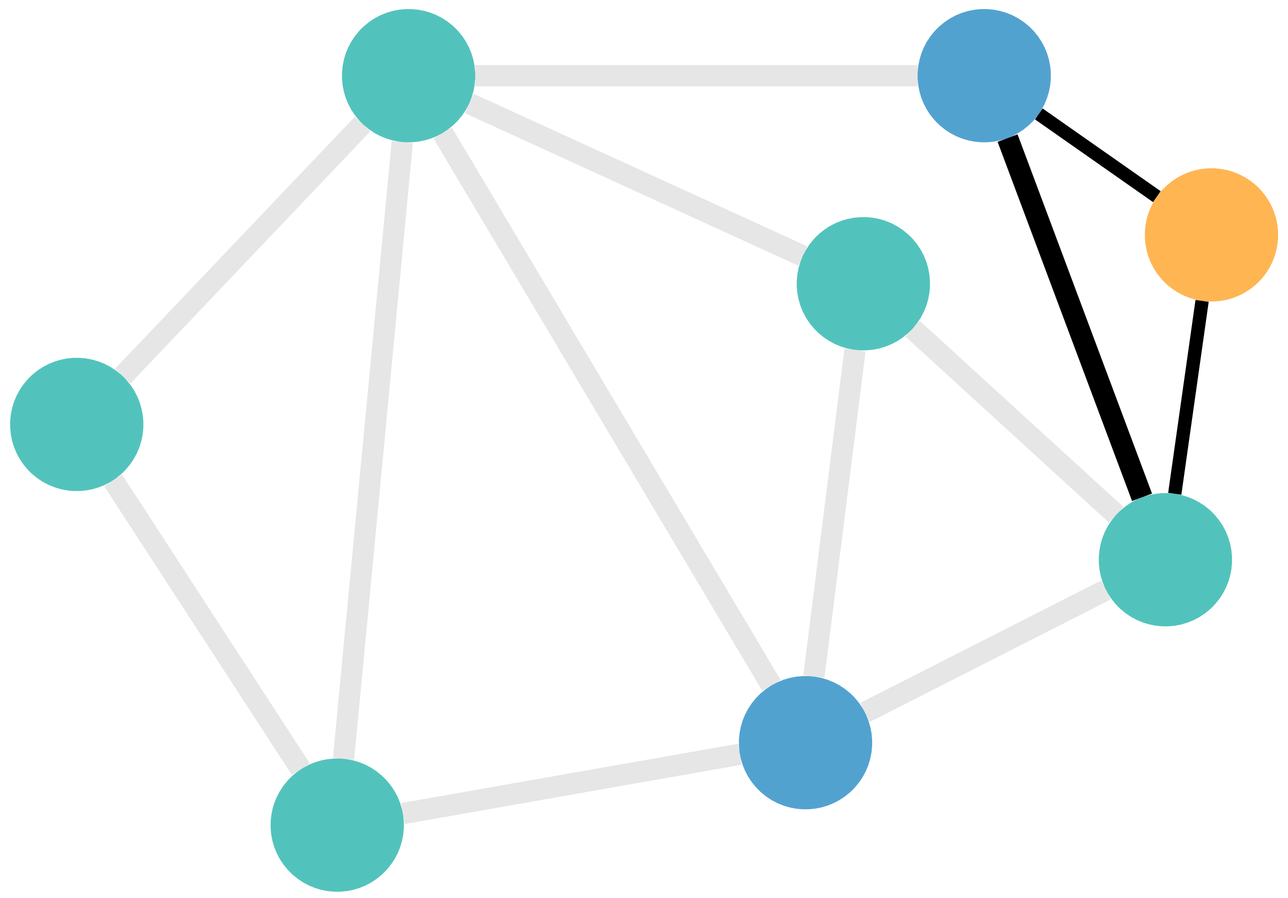
Construct a set $F'$:

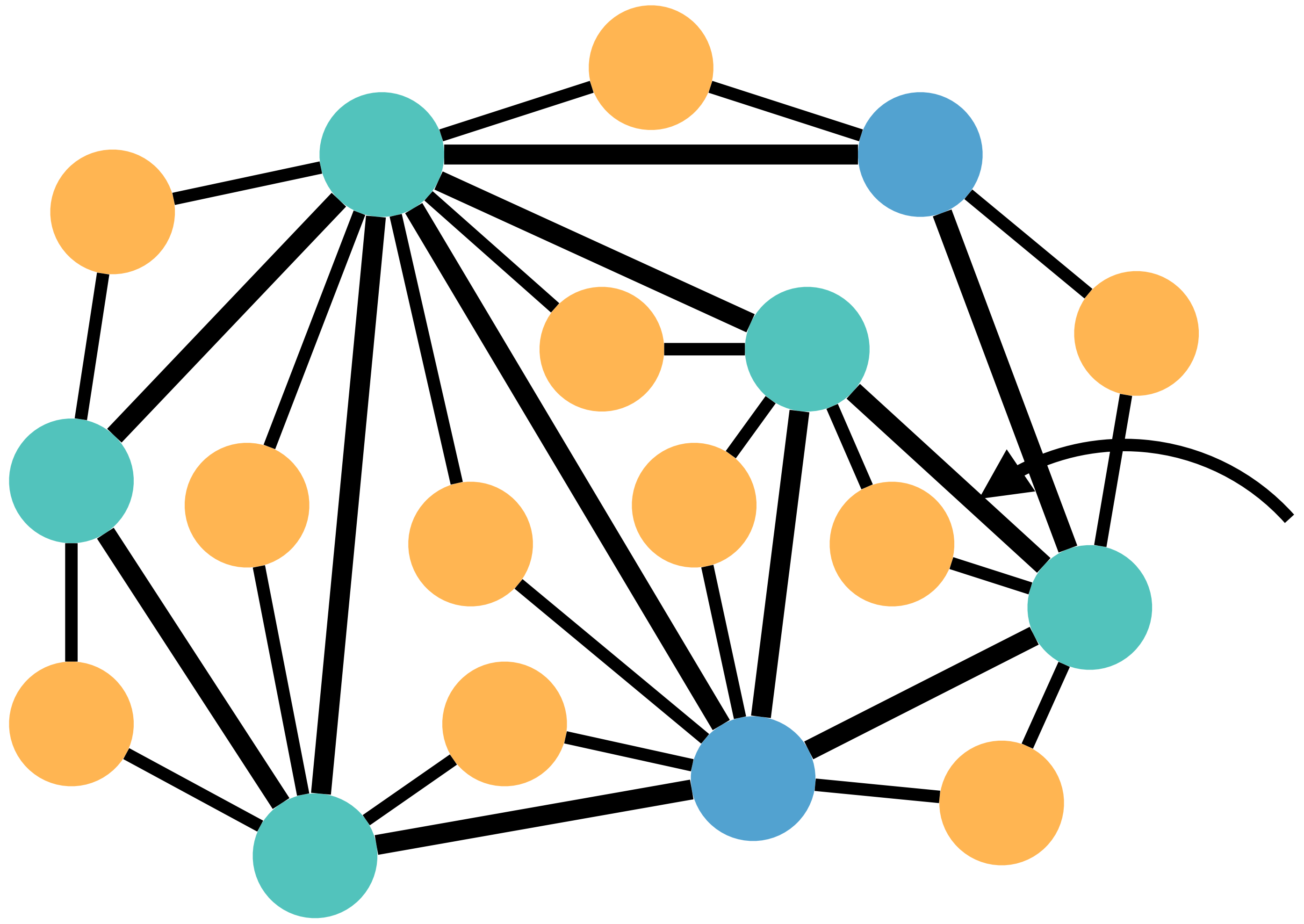
Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$

$\Rightarrow$ $F'$ is a FVS with size at most $k'$





# Feedback Vertex Set (FVS)

$F'$ is a vertex cover of $G$

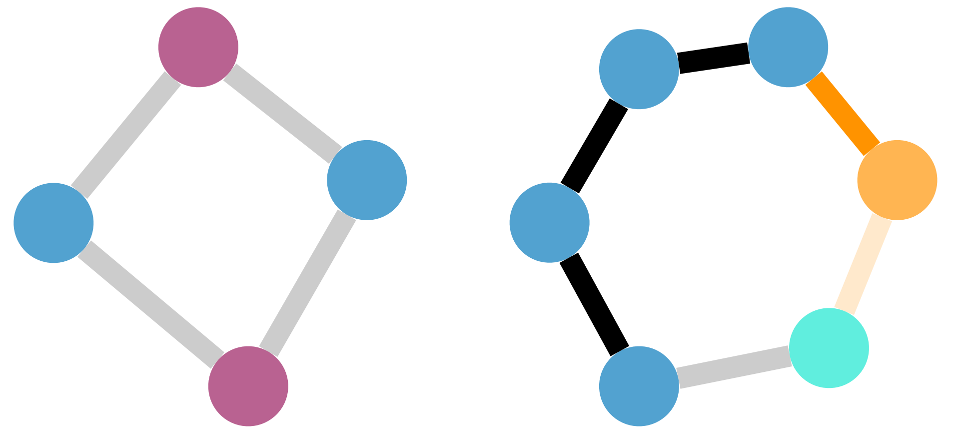
After removing $F'$ from $G'$, $G'$ has no cycle

$\Rightarrow$ After removing $F'$ from $G$, $G$ has no edge

Otherwise, there is a cycle in $G'$ since no $u_{i,j}$ is in $F'$)

$\Rightarrow$ $F'$ is a vertex cover with size at most $k$ in $G$

The size-$k$ FVS of $G'$



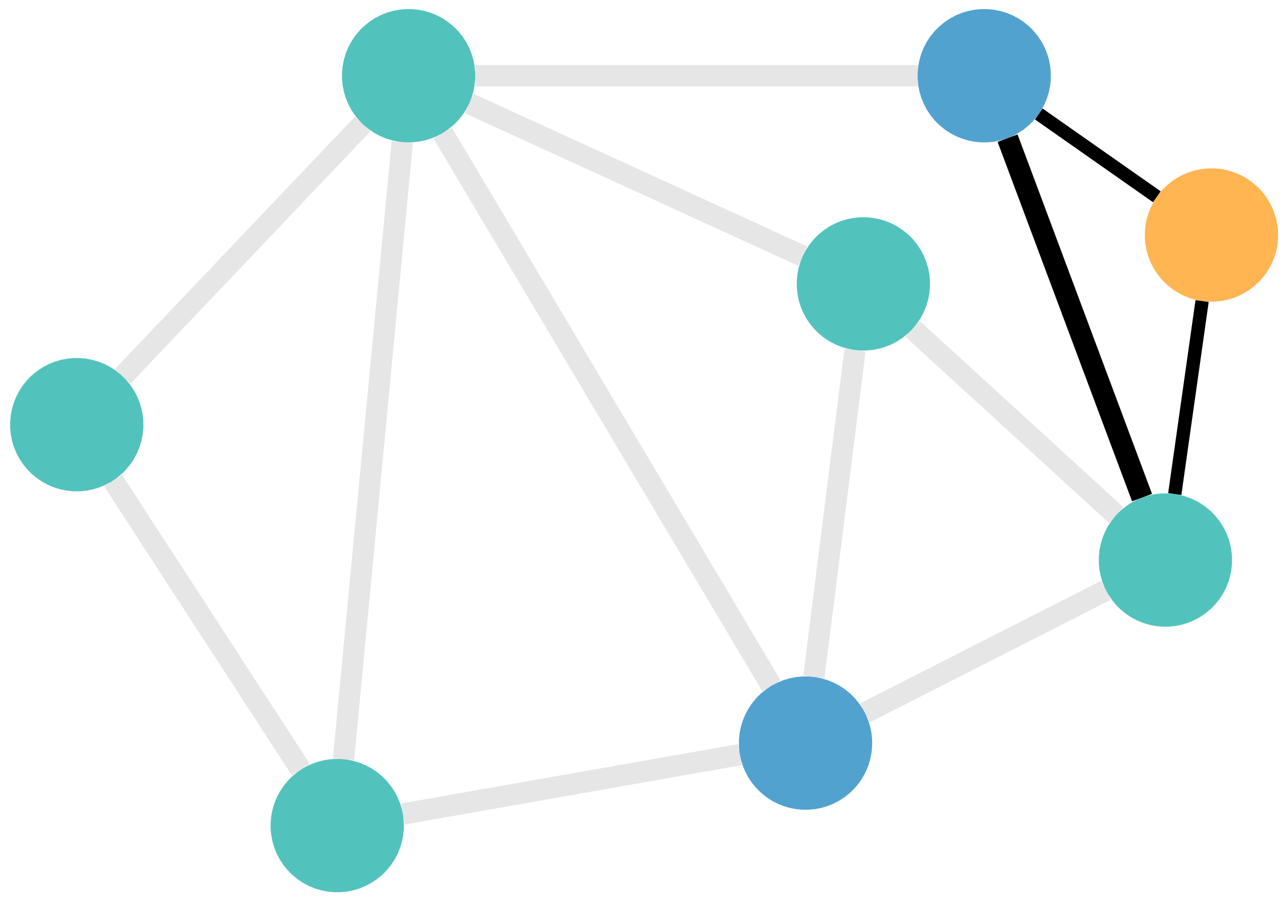
Construct a set $F'$:

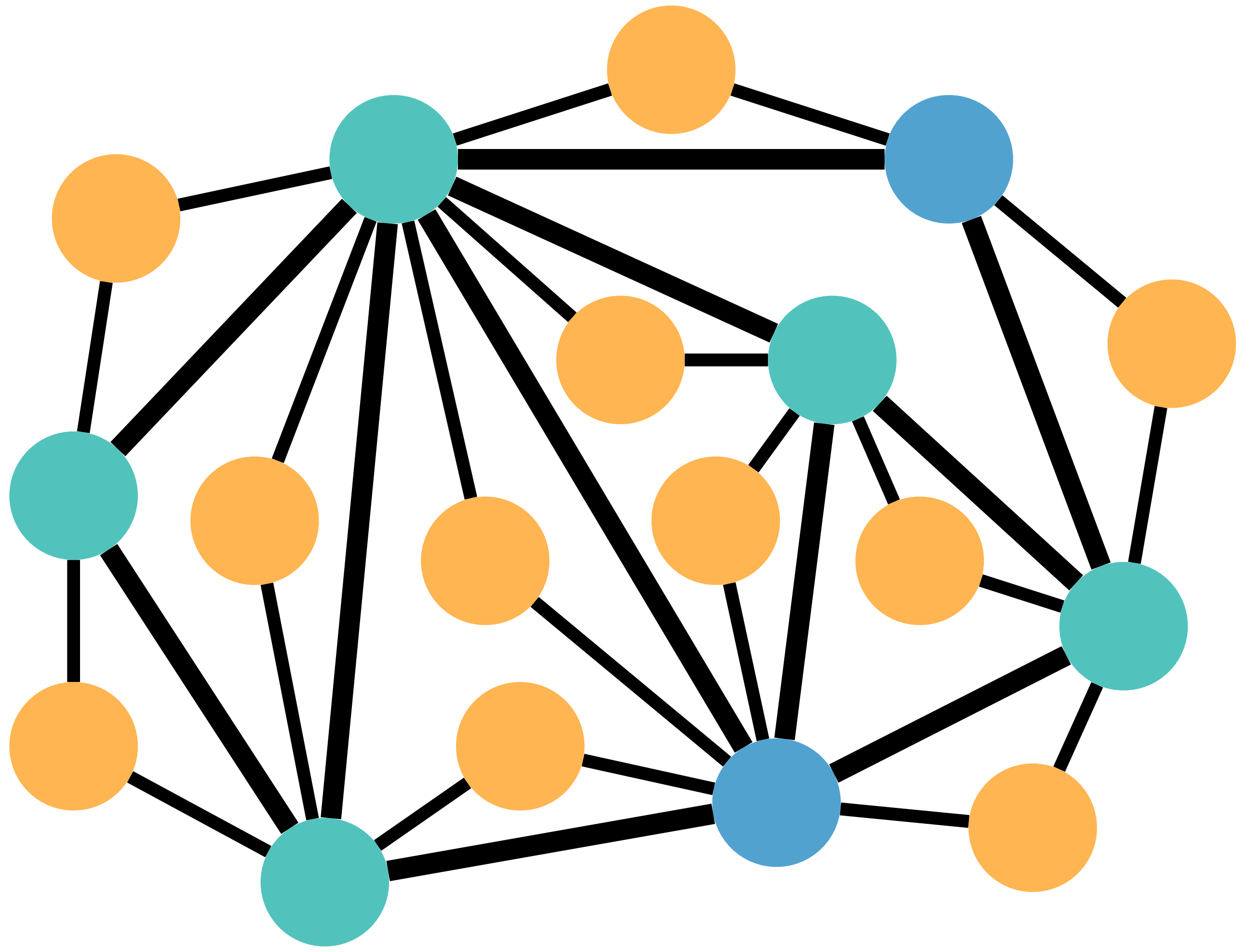
Keep all $u_i$ in the FVS

If some $u_{i,j}$ is in the FVS, replace it by $u_i$ or $u_j$

If both $u_i$ are in $u_j$ the FVS, remove $u_{i,j}$

$\Rightarrow$ $F'$ is a FVS with size at most $k'$





# Feedback Vertex Set (FVS)

- **FVS** $= \{\langle G, k\rangle \mid$ There is a set of at most $k$ vertices in $G$ such that removing them leaves no cycles$\}$

- Theorem: FVS is NP-complete

<proof> To prove that FVS is in NP, we use a size-$k$ feedback vertex set $U$ as the certificate. The verifier should check $U$ it is a proper subset of the vertices in $G$, and if $G$ is cycle-free after removing all edges incident to the vertices in $U$. The later can be done by running a breadth-first-search on the resulting graph. The checking time is in polynomial of the size of $G$.

# Feedback Vertex Set (FVS)

To prove the NP-hardness, we show that VERTEX-COVER $\leq_p$ FVS. For any instance of VERTEX-COVER, $G = (V, E)$ and $k$, we construct an instance of FVS, $G' = (V', E')$ and $k'$ as follows. For each vertex $v_i \in V$, there is a corresponding vertex $u_i \in V'$. More over, for each edge $(v_i, v_j) \in E$, there is a corresponding vertex $u_{i,j} \in V'$.

For each edge $(v_i, v_j) \in E$, we construct three edges in $E'$: $(u_i, u_j)$, $(u_i, u_{i,j})$, and $(u_j, u_{i,j})$.

We set $k' = k$.

The construction takes constant time to each element in $V$ or $E$ and can be done in polynomial-time.

# Feedback Vertex Set (FVS)

Now we prove that the reduction works. Suppose that there is a size-$k$ vertex cover $C$ of $G$. First observe that there are two types of cycles in $G'$: 1) cycles containing no $u_{i,j}$ vertices, and 2) cycles containing at least one $u_{i,j}$ vertex.

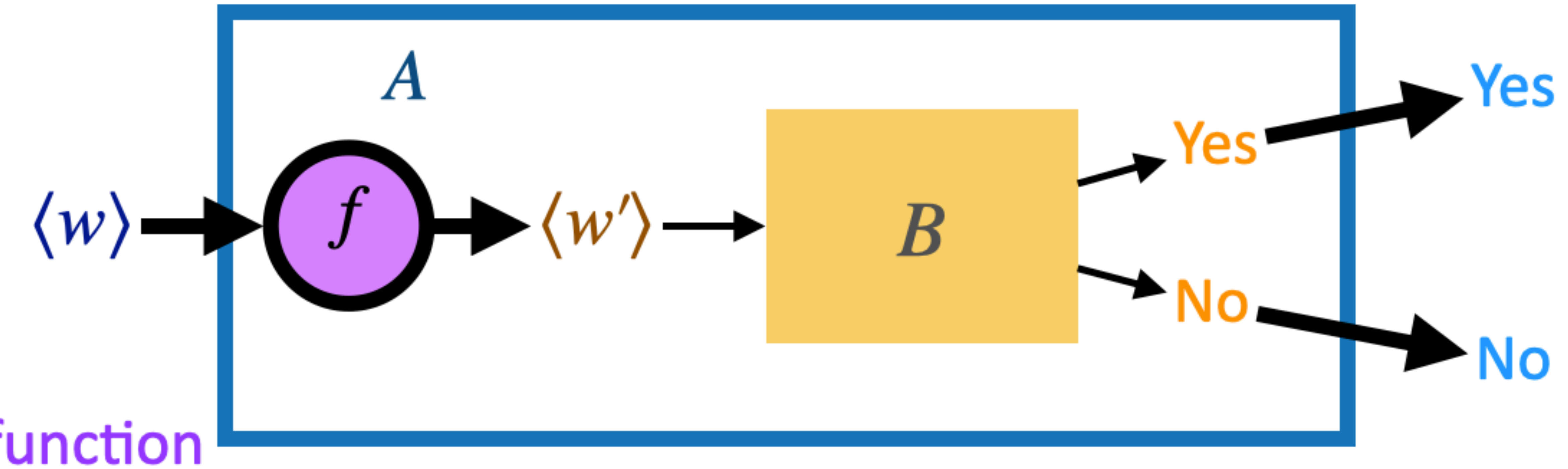
Consider removing all vertices in $C$ from $V'$, there is no edge between any two vertices in $V'$, $u_i$ and $u_j$. Therefore, there are no type-1 cycles in the remaining graph.

Furthermore, because every vertex u_{i,j} only adjacent to $u_i'$ and $u_j'$, the degree of $u_{i,j}$ is at most 1 after removing vertices in $C$. Thus, there are no type-2 cycles left. Hence, $C$ is a size-$k$ feedback vertex set of $G'$, and $\langle G', k' \rangle$ is a yes-instance of FVS.

# Feedback Vertex Set (FVS)

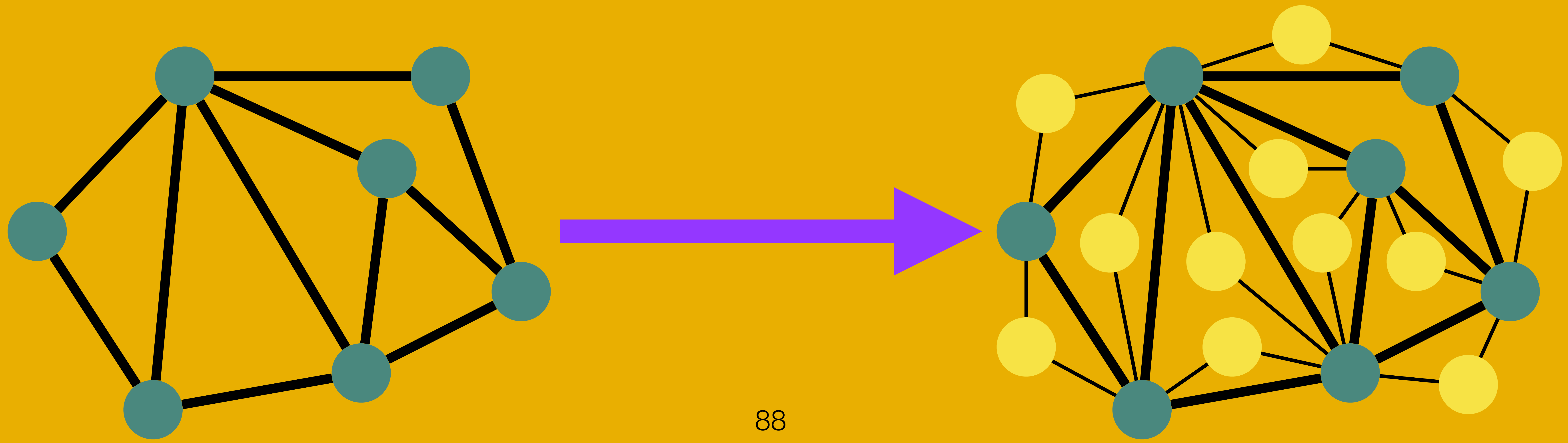
For the other direction, suppose that there is a size-$k'$ feedback vertex set $F$ of $G'$. We make a feedback vertex set $F'$ of $G'$ with size at most $k'$ as follows. For all vertices $u_i$ in $F$, we add them into $F'$. If there is a vertex $u_{i,j}$ in $F$, we replace it by $u_i$ or $u_j$, which was not in $F$, in $F'$. If both $u_i$ and $u_j$ are already in $F$, we simply remove $u_{i,j}$. Any cycle $C$ that only contains vertices $u_i$'s is broken by $C \cup F'$ since $C \cup F \subseteq C \cup F'$. Any cycle that contains an $u_{ij}$ vertex is broken by $u_i$ or $u_j$. Therefore, $F'$ is a feasible feedback vertex set with size at most $k'$.

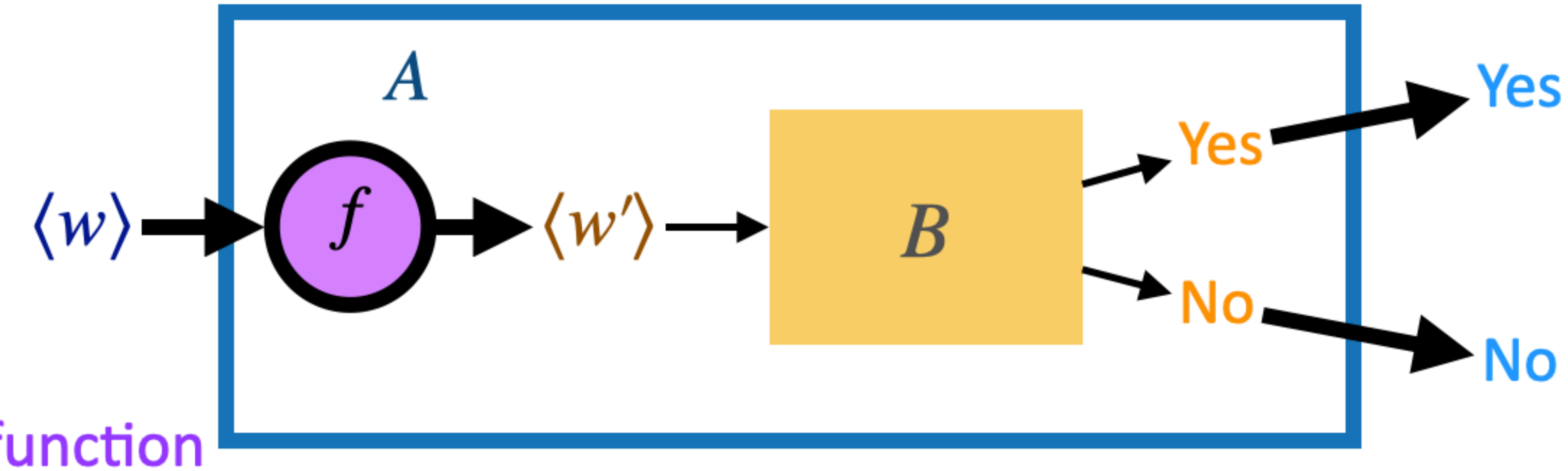
Now, we argue that the vertices in $F'$ form a vertex cover in $G$. Since there is no vertex $u_{i,j}$ in $F'$, removing all vertices in $F'$ leaves no edge between any pair of $u_i$ and $u_j$. Otherwise, there is a cycle $(u_i, u_j, u_{i,j})$, and it contradicts to the fact that $F'$ is a feasible vertex set. Thus, $F'$ is a vertex cover in $G$. That is, $G$ is a yes-instance of the VERTEX-COVER problem ■




1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time
2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$
3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

$\langle G, k \rangle$ $\qquad$ $\langle G', k \rangle$





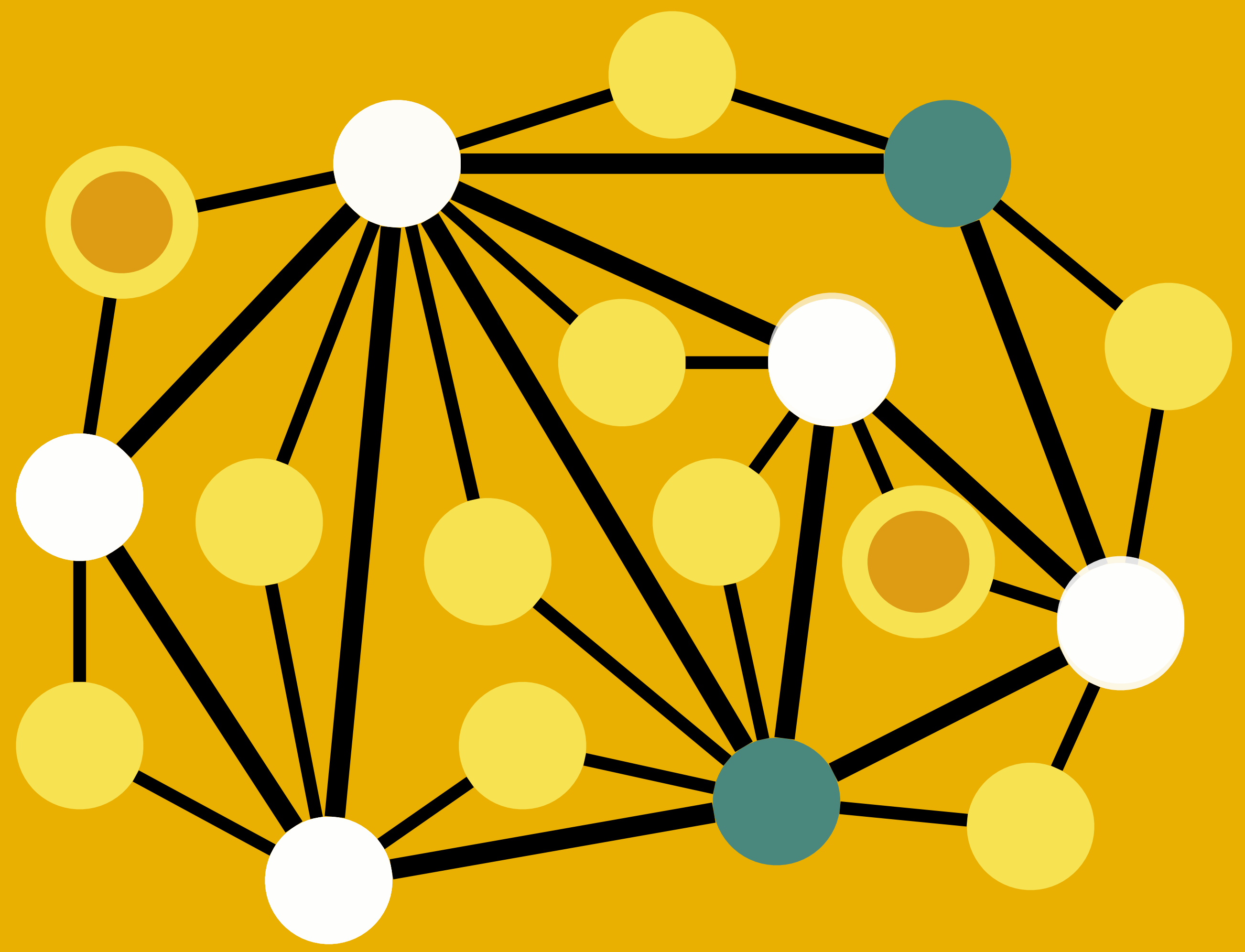

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time

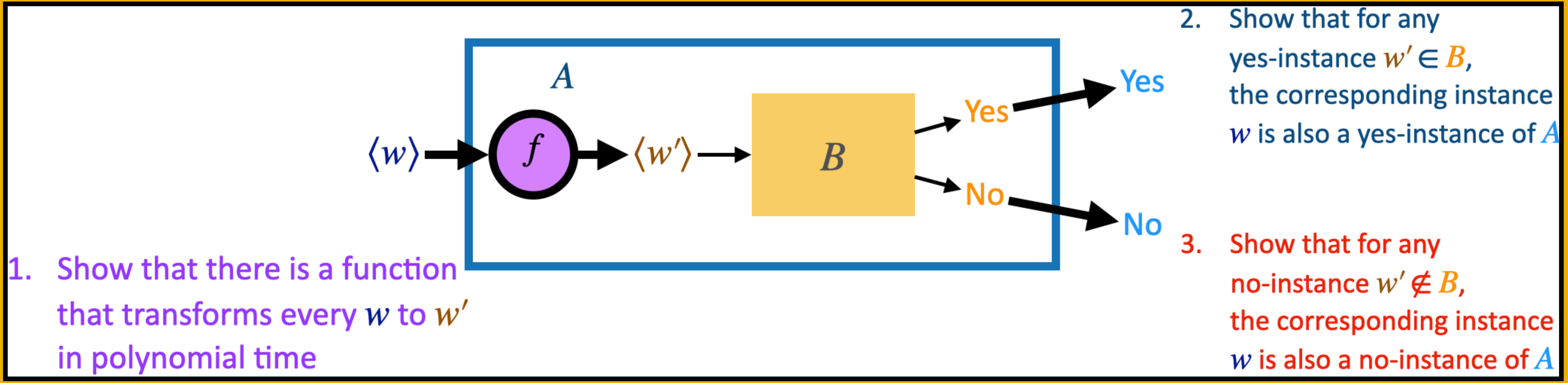
2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$

3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

If $\langle G', k \rangle$ is a yes-instance

$G'$ has a size-$k$ feedback vertex set

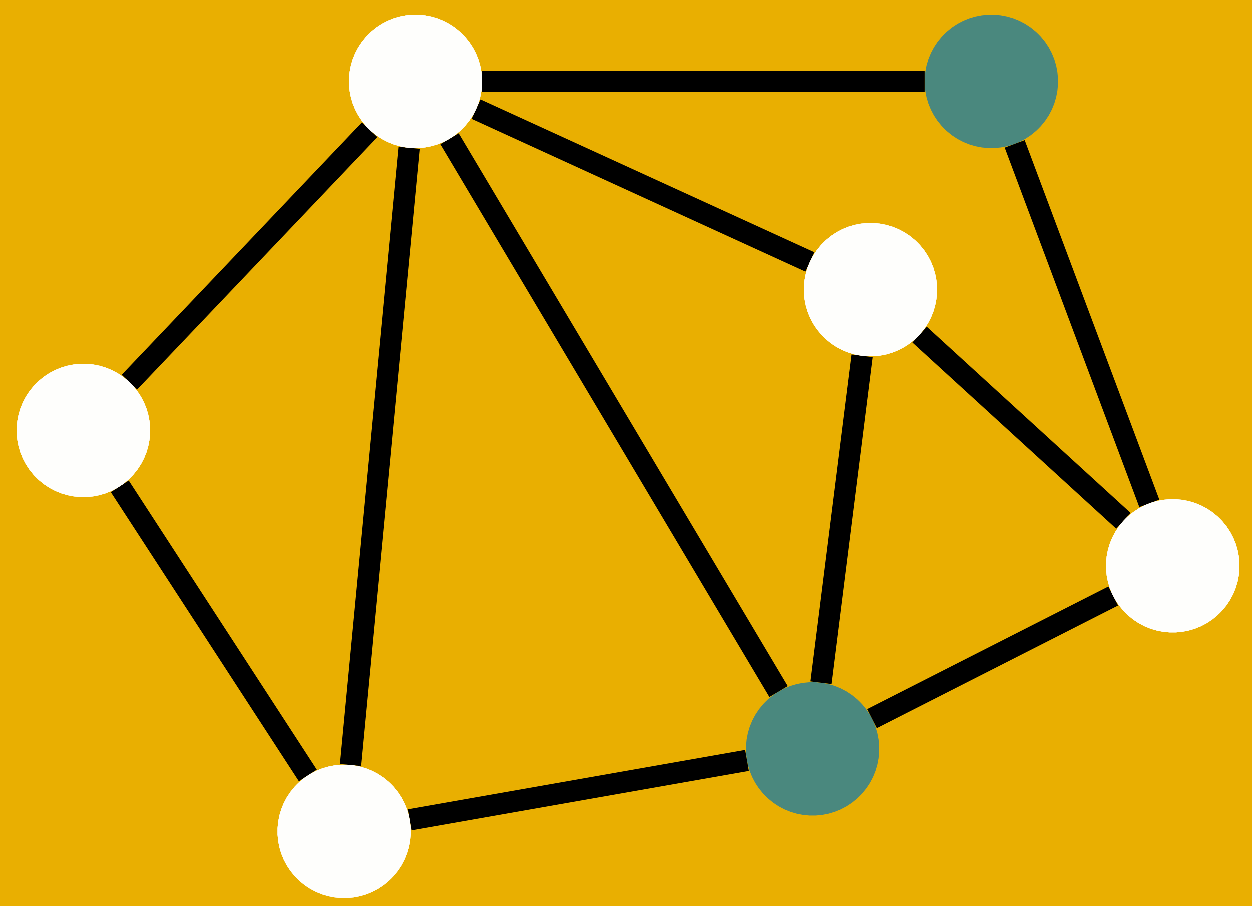


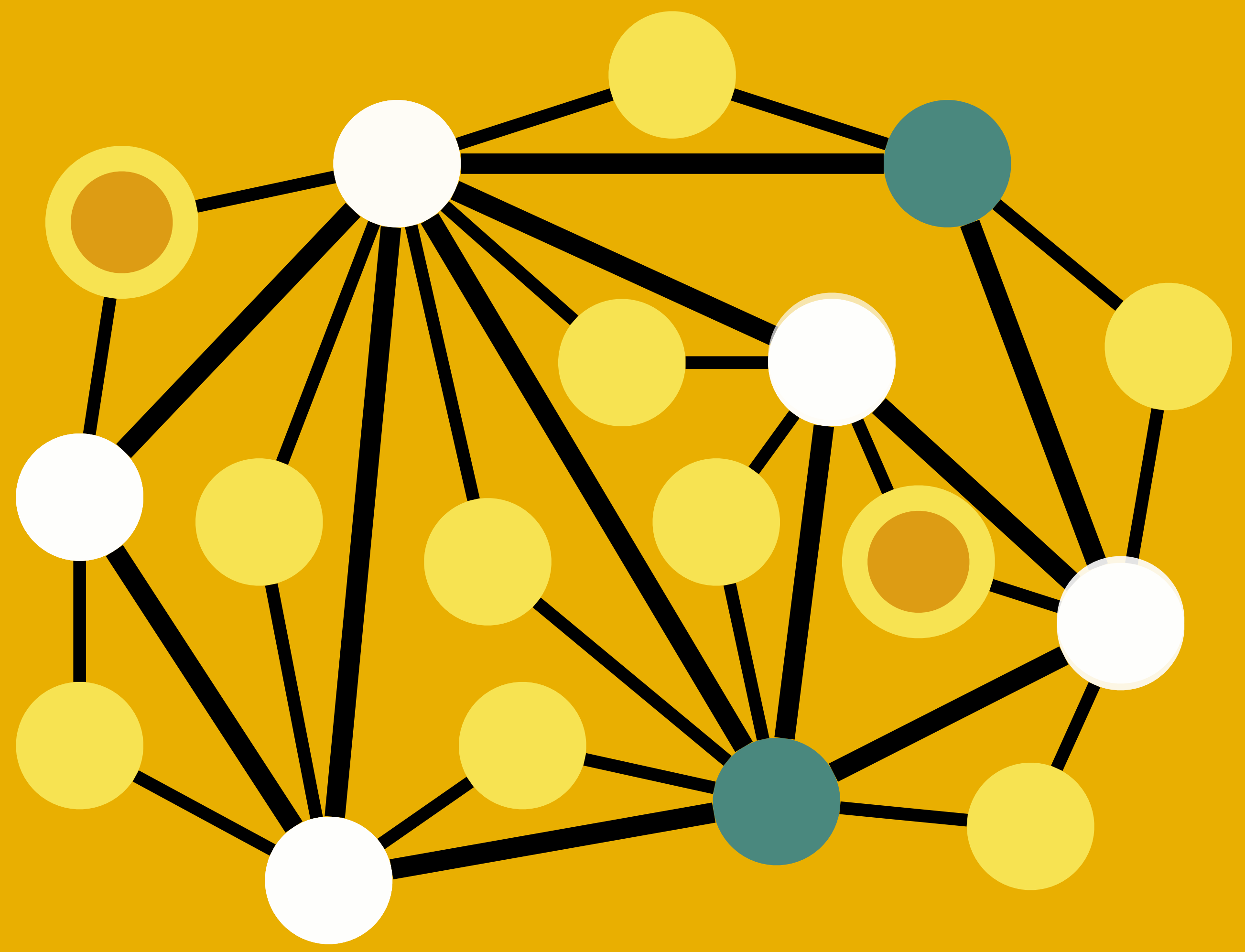


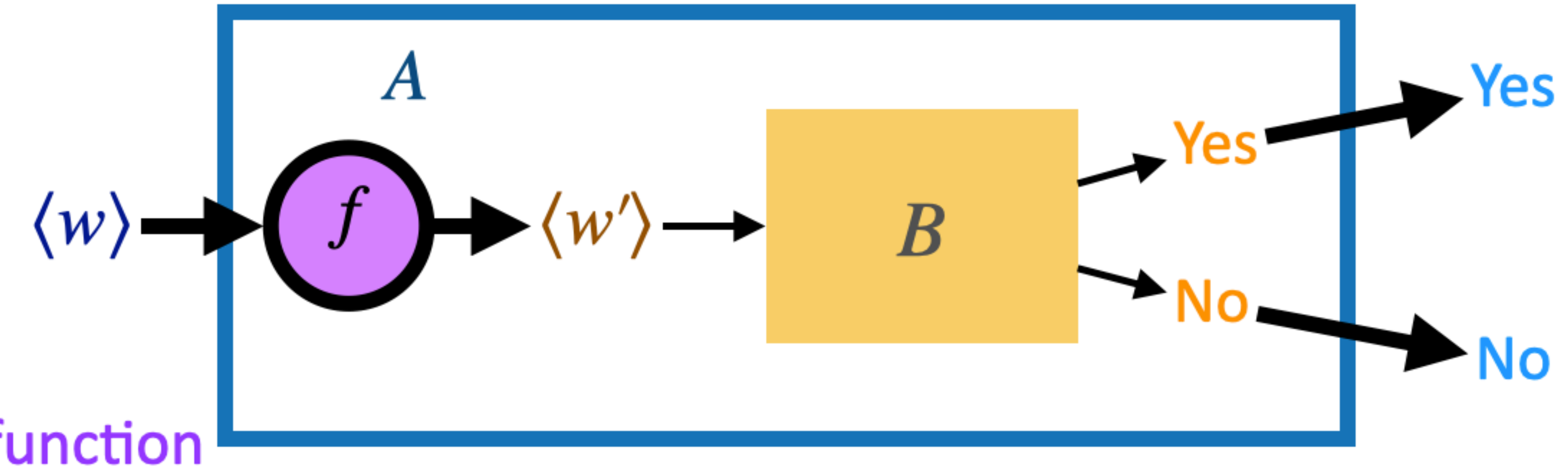

There is a feedback vertex set $F'$ with size at most $k$ and only contains the vertices in $G$

If $\langle G', k \rangle$ is a yes-instance

$G'$ has a size-$k$ feedback vertex set





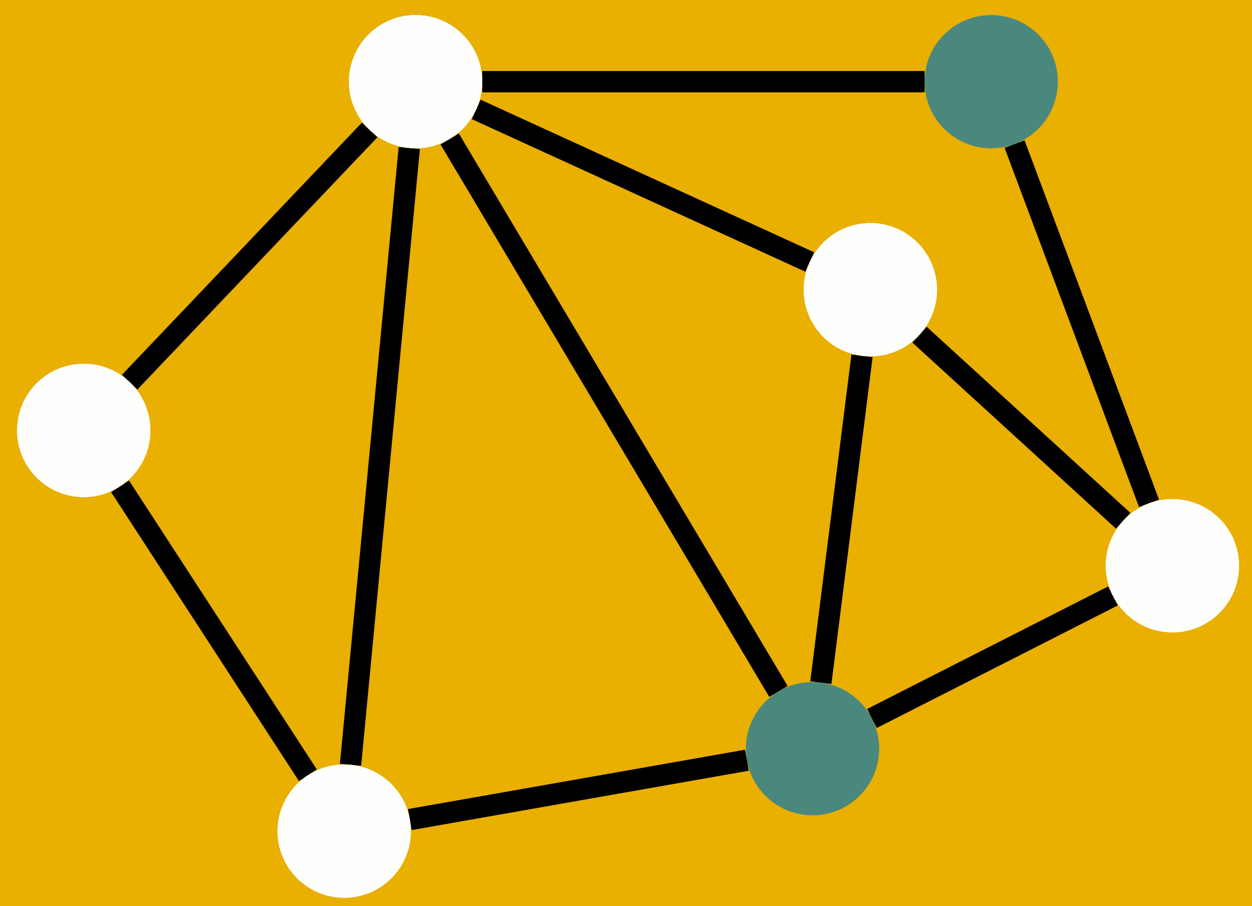


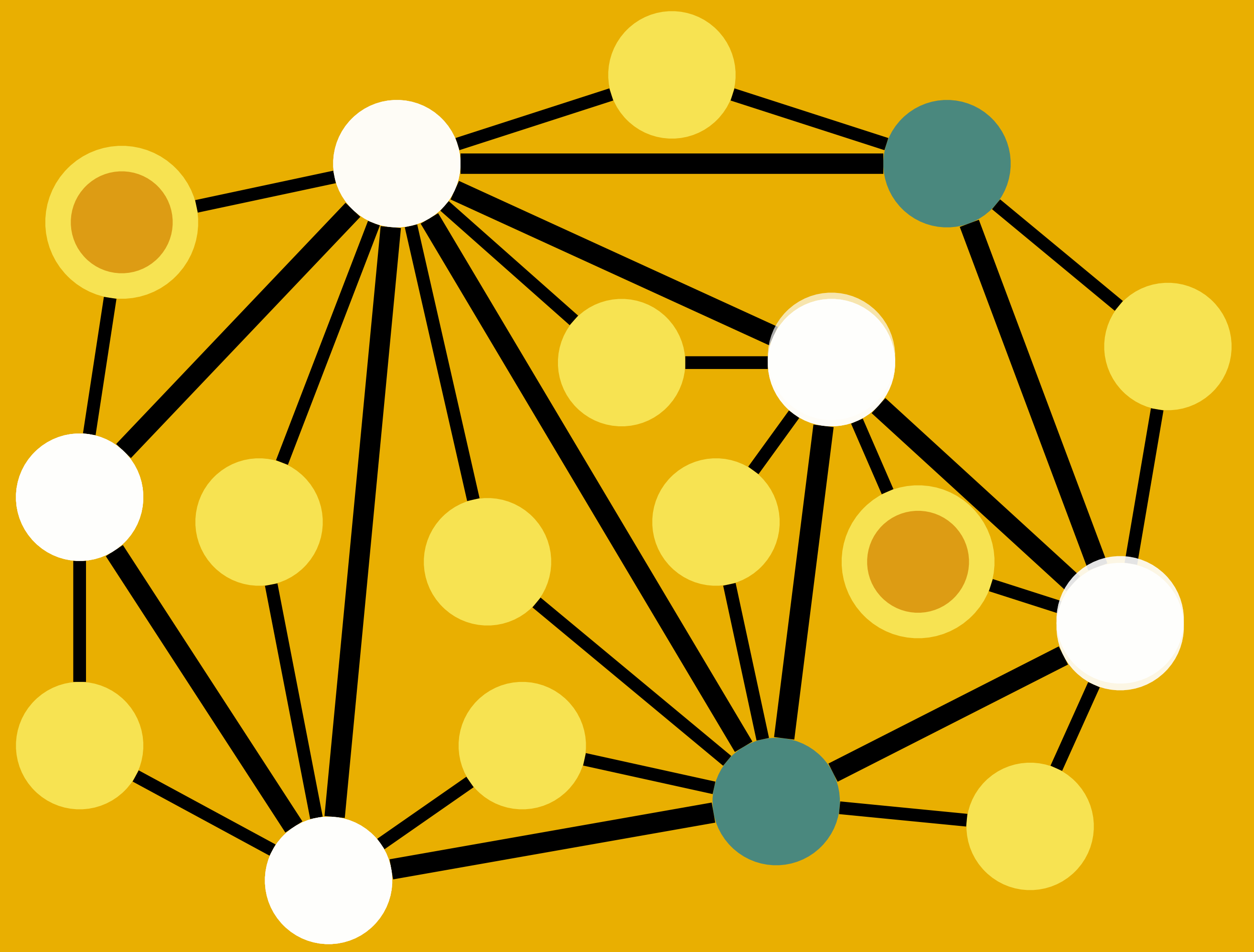

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time
2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$
3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

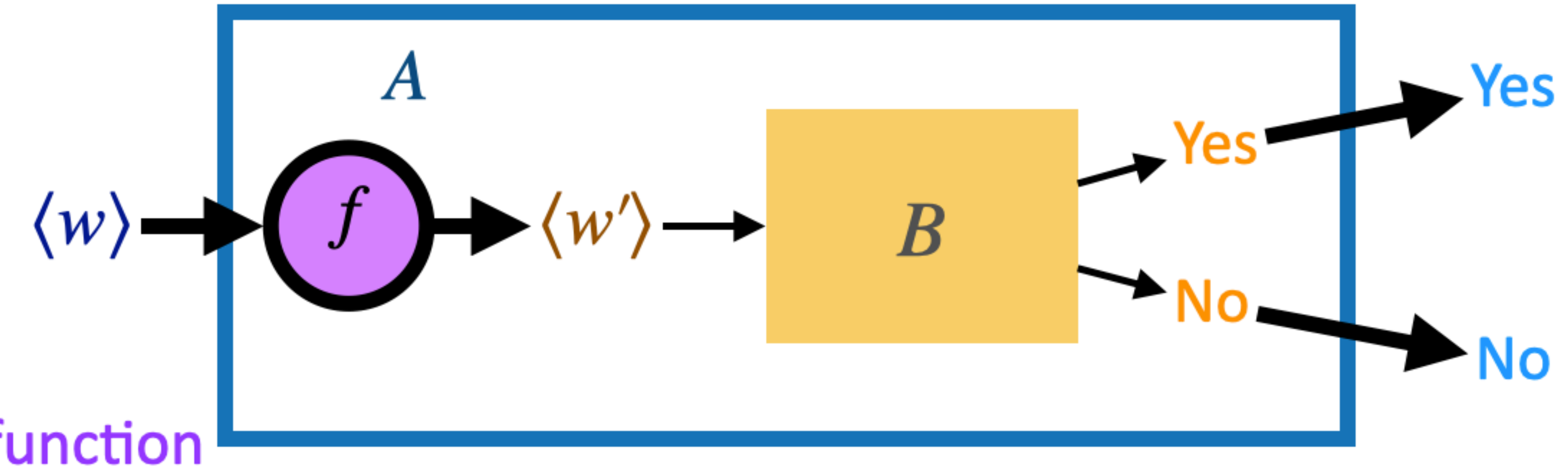
$F'$ is a vertex cover in $G$ with size at most $k$ (since …)

There is a feedback vertex set $F'$ with size at most $k$ and only contains the vertices in $G$

If $\langle G', k \rangle$ is a yes-instance $G'$ has a size-$k$ feedback vertex set





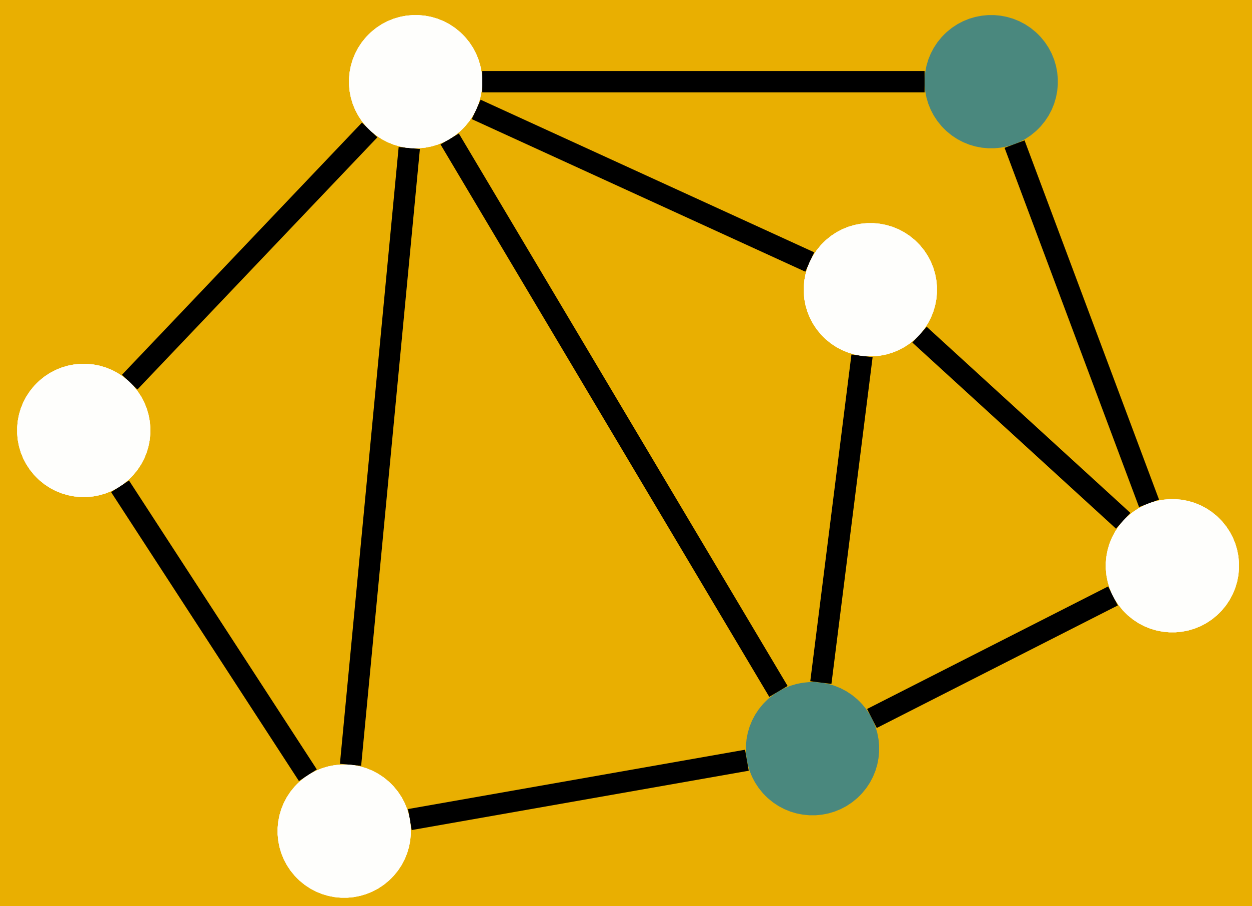


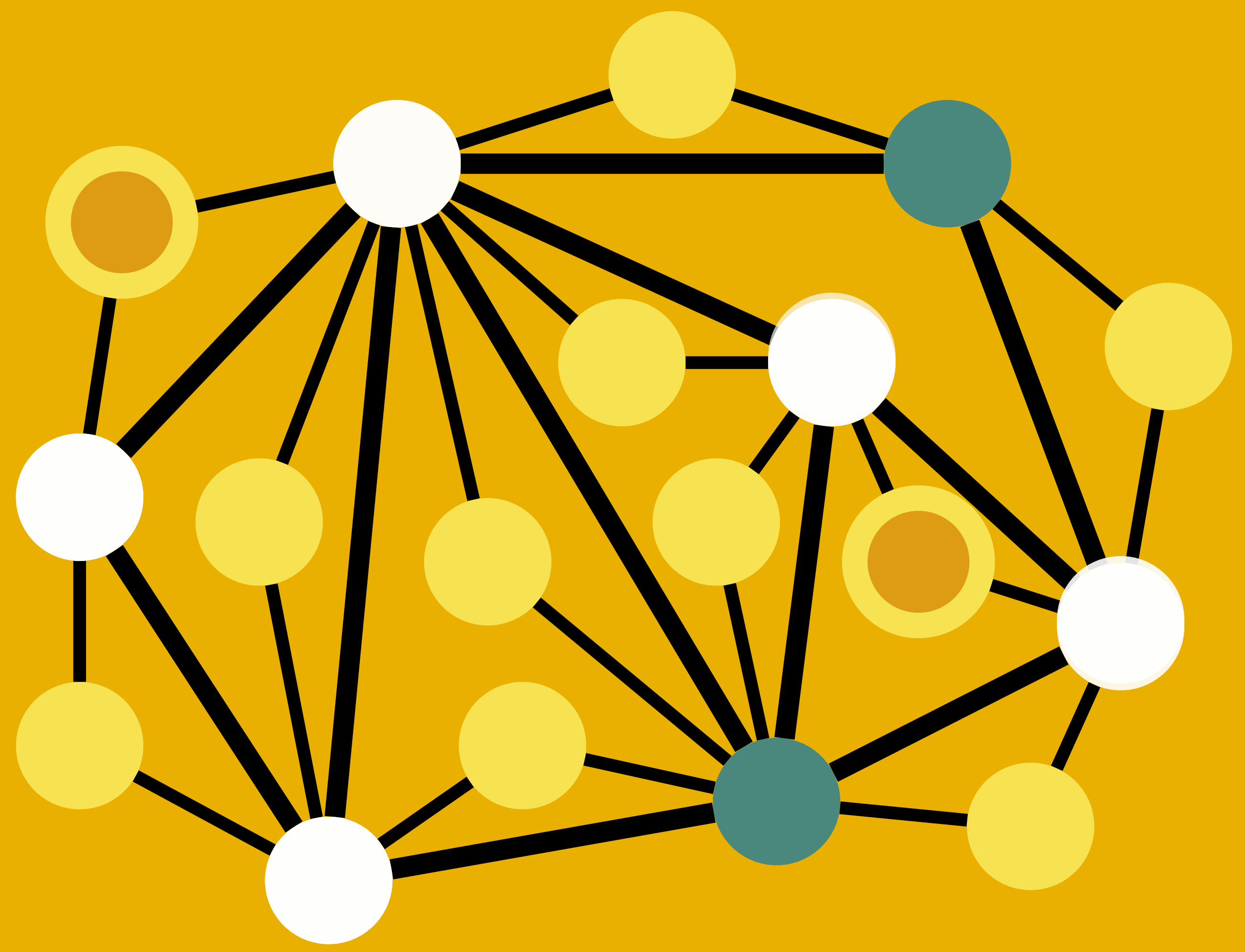

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time
2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$
3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

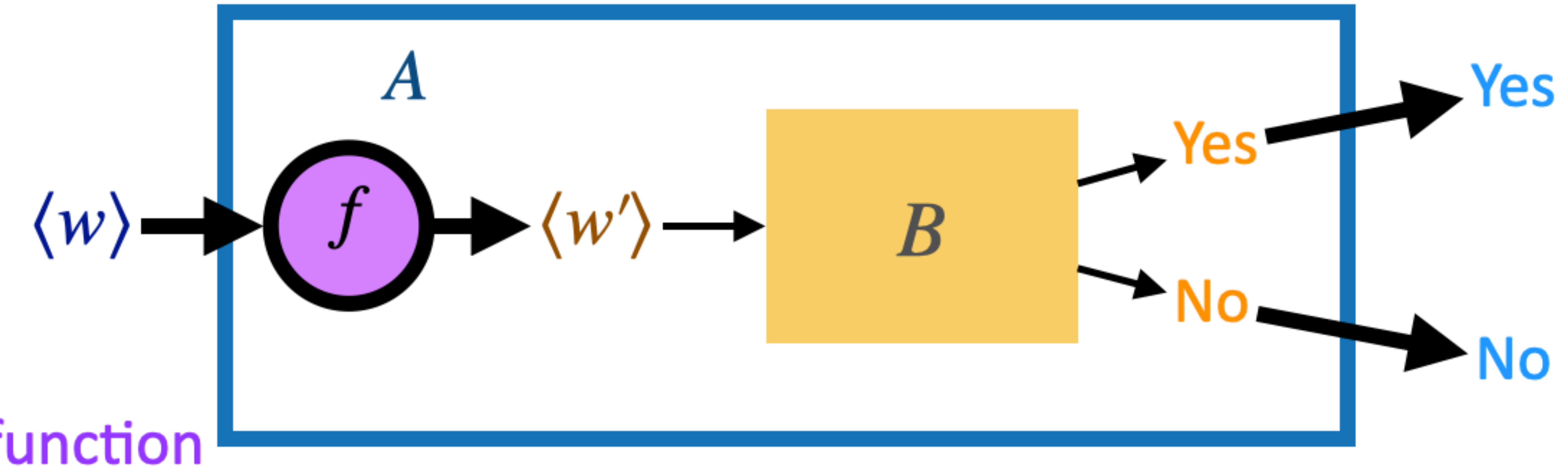
$F'$ is a vertex cover in $G$ with size at most $k$ (since …)
Therefore, $\langle G, k \rangle$ is a yes-instance

There is a feedback vertex set $F'$ with size at most $k$ and only contains the vertices in $G$

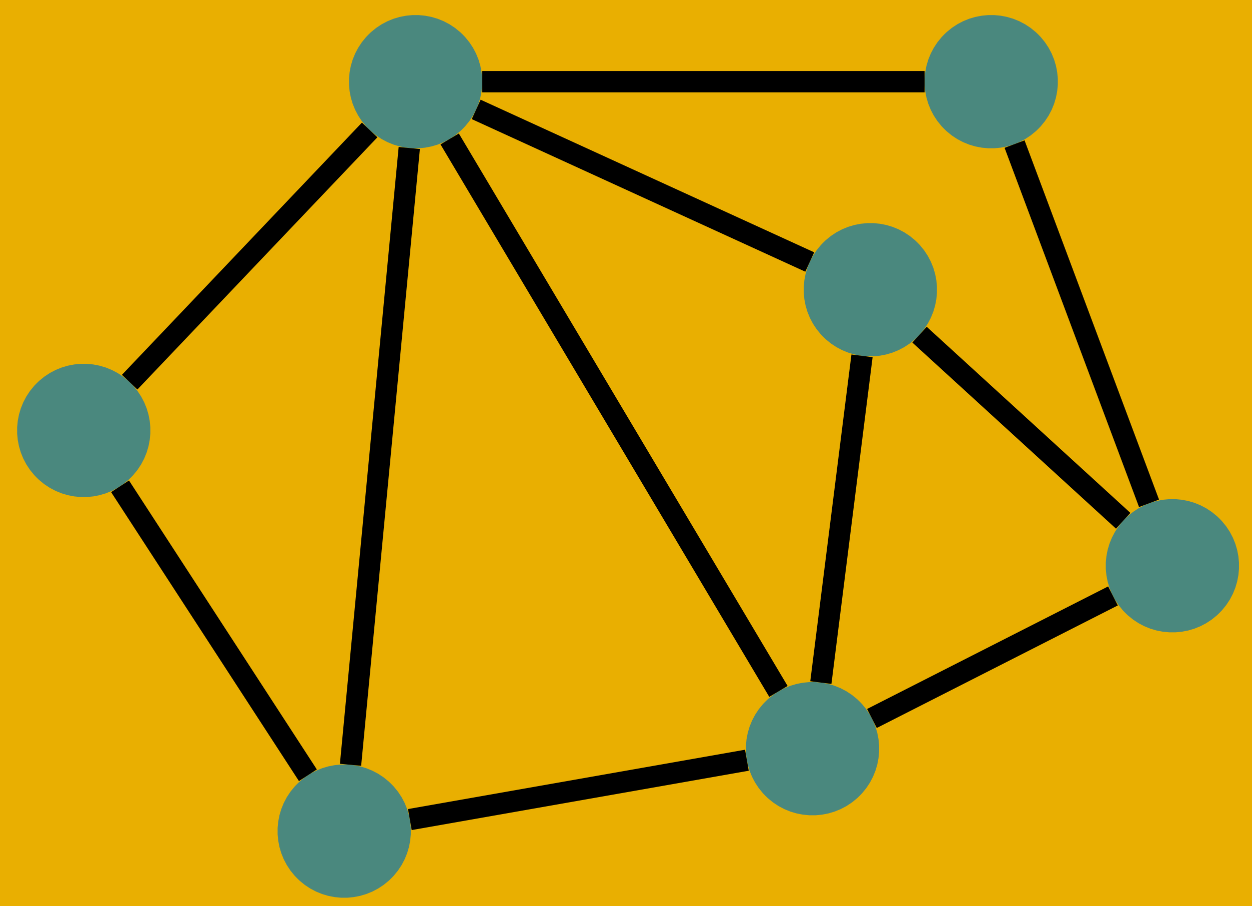
If $\langle G', k \rangle$ is a yes-instance
$G'$ has a size-$k$ feedback vertex set

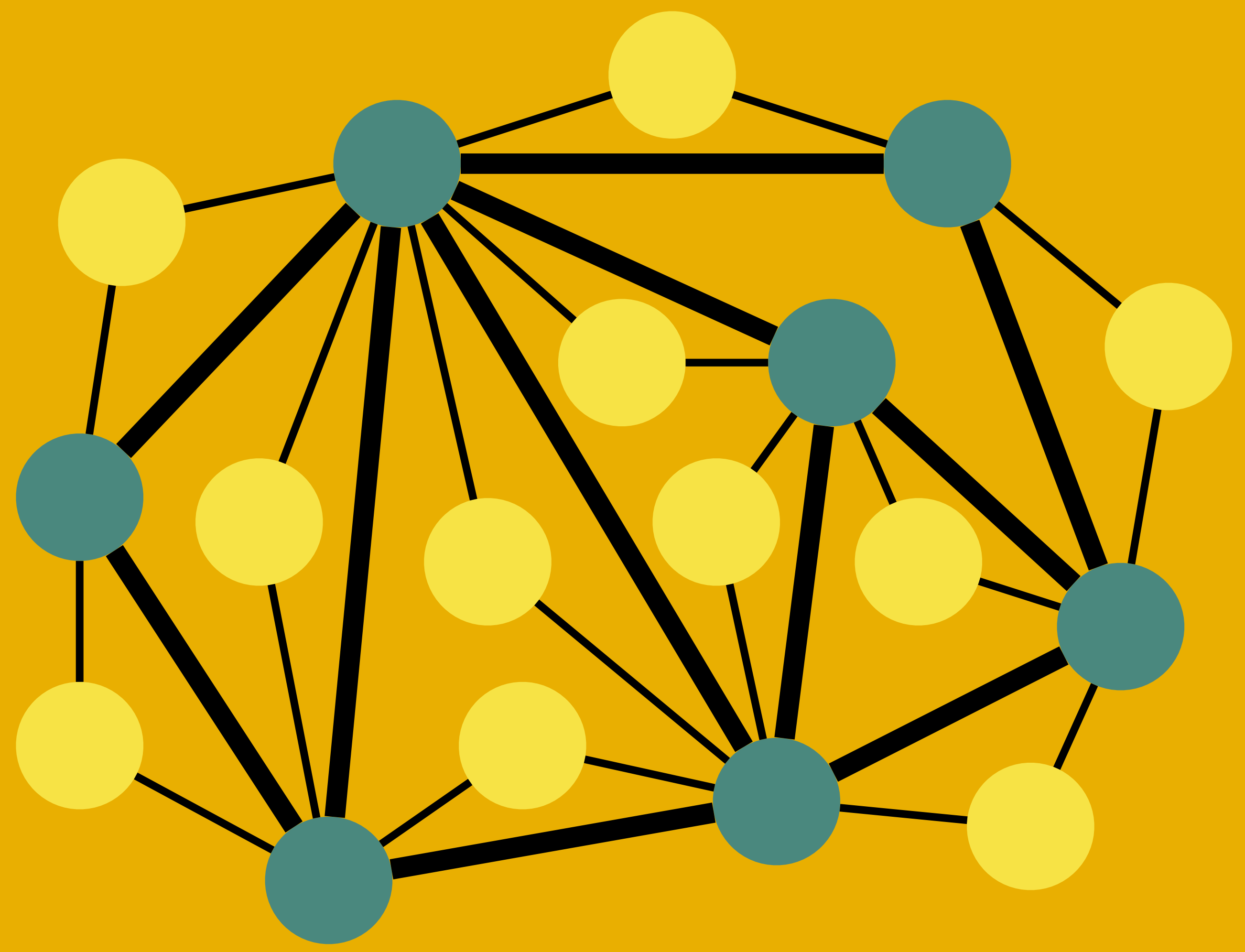


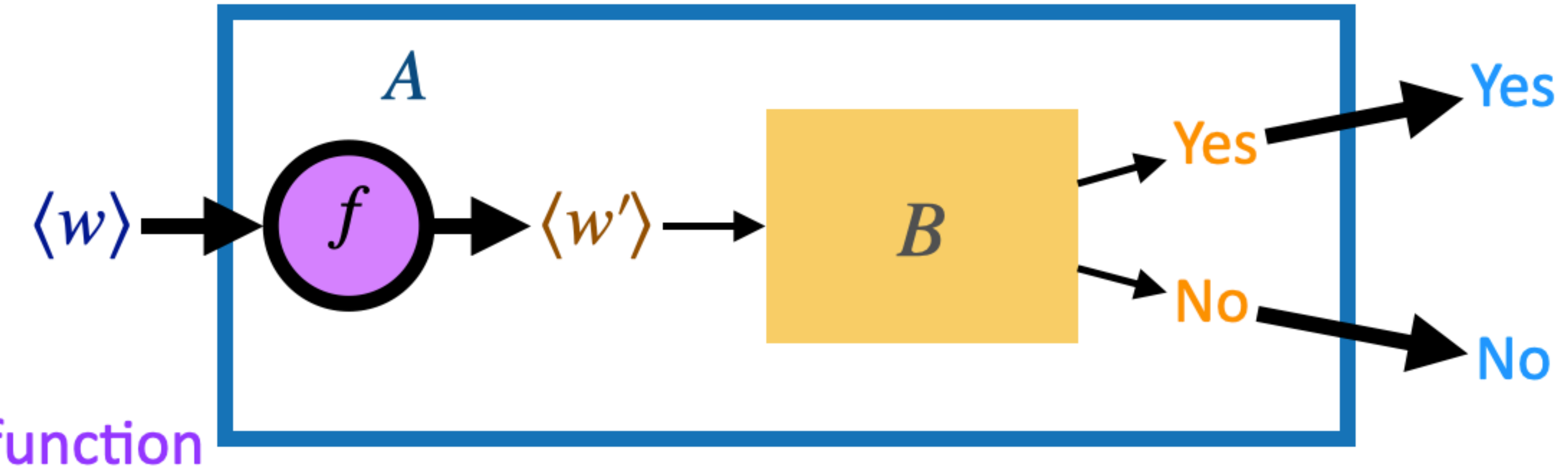





1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time
2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$
3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$





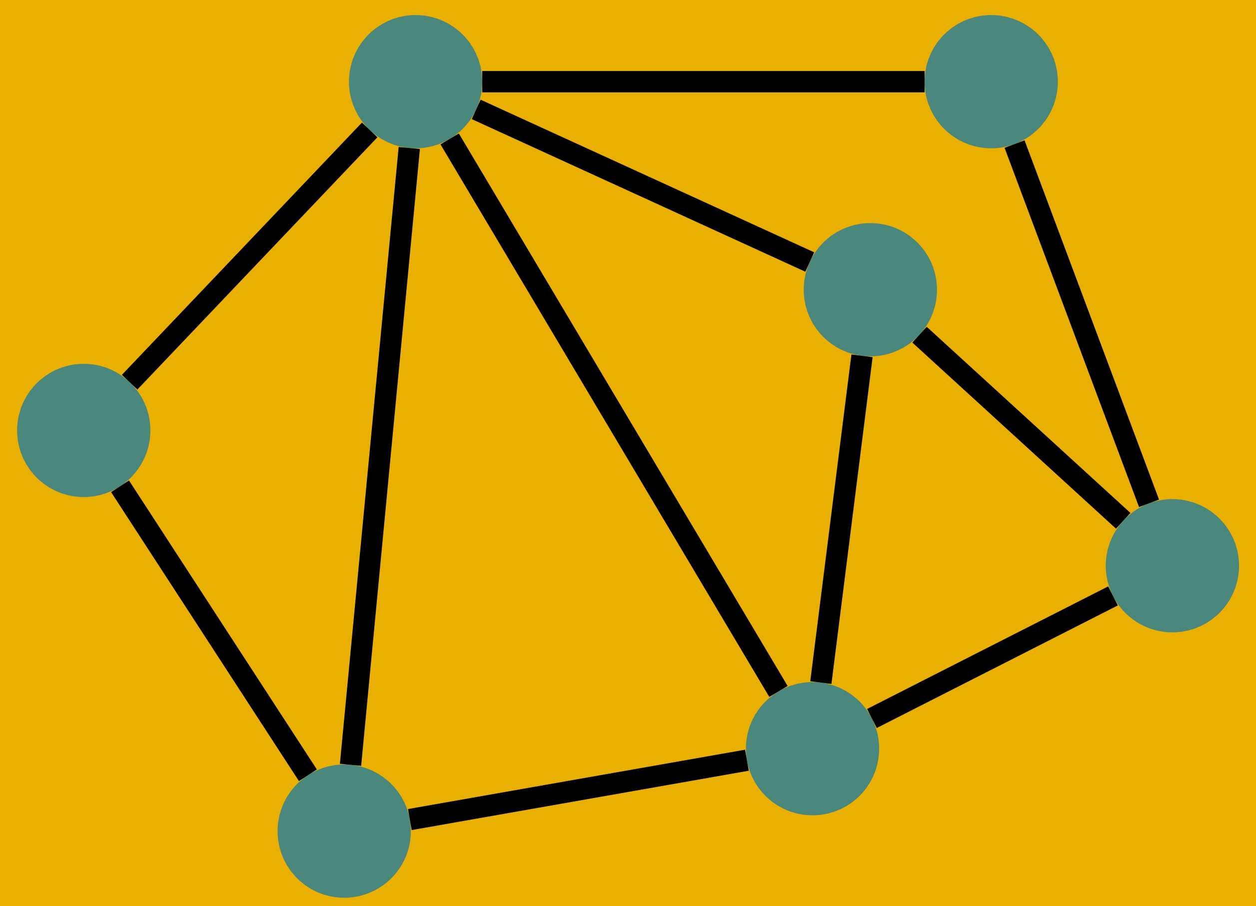


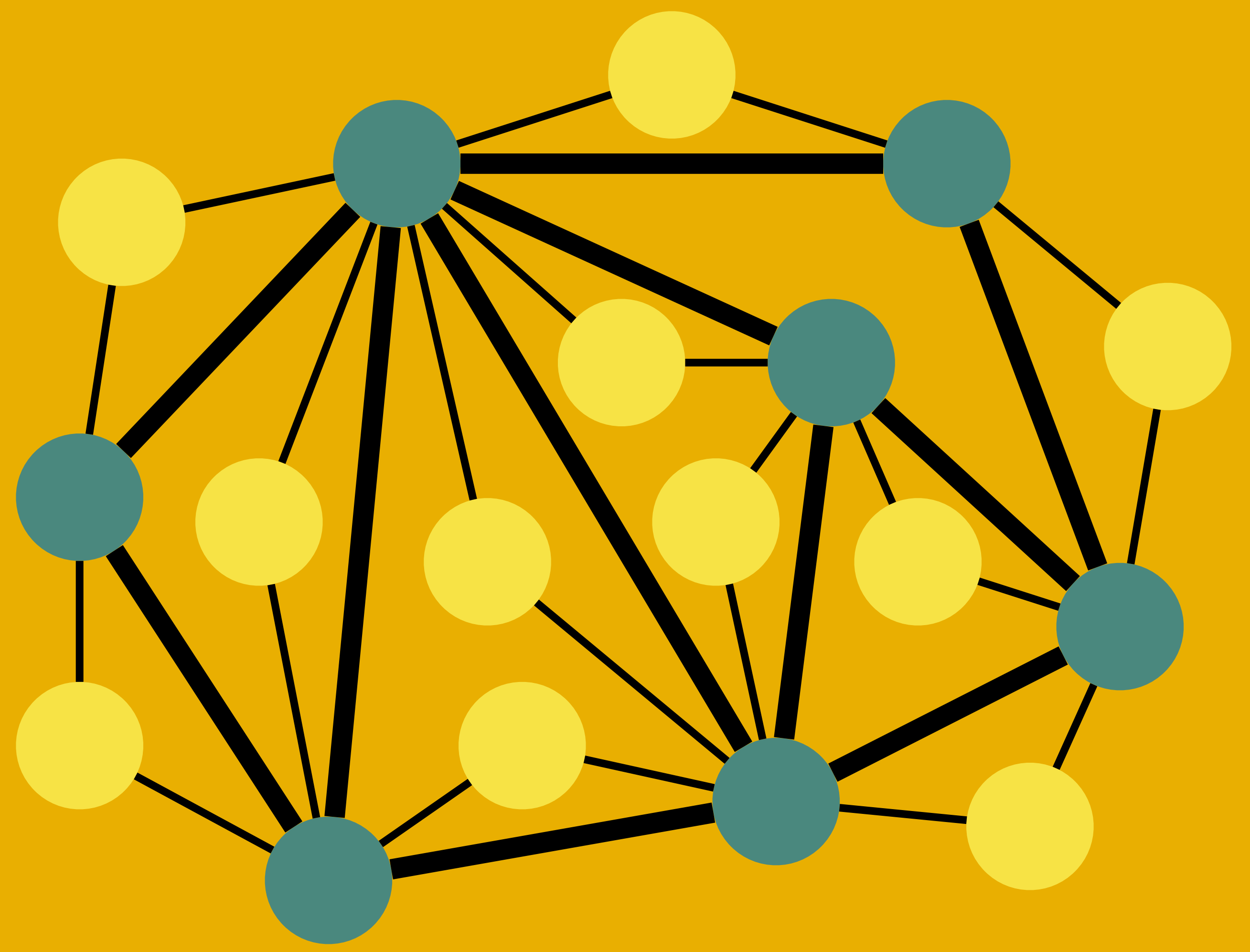

2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$

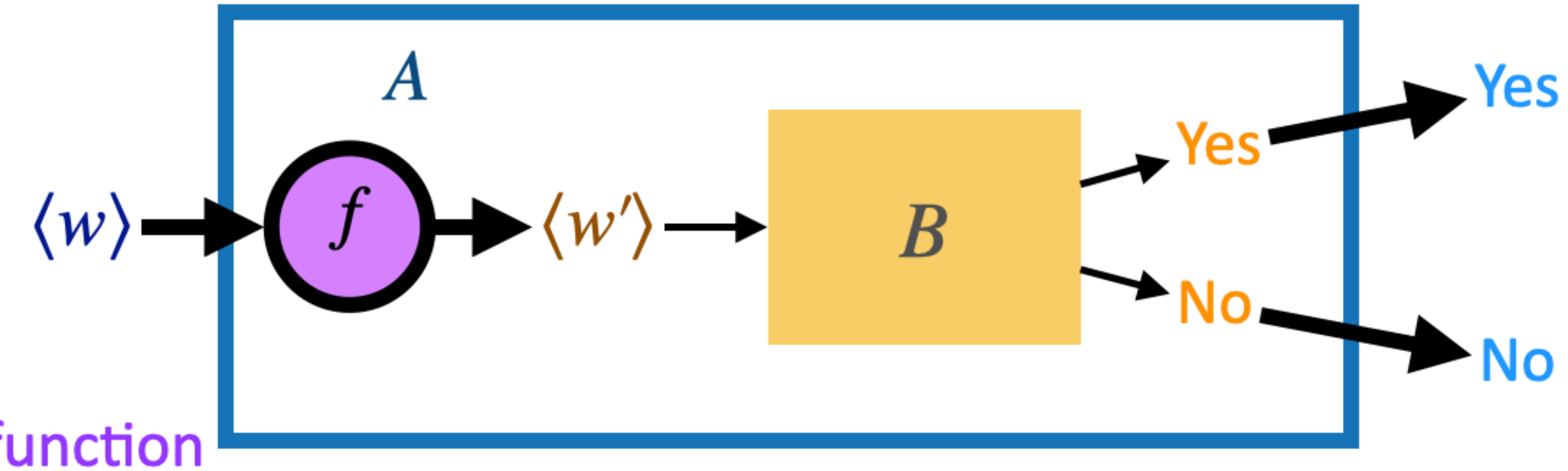
3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time

If $\langle G, k \rangle$ is a yes-instance



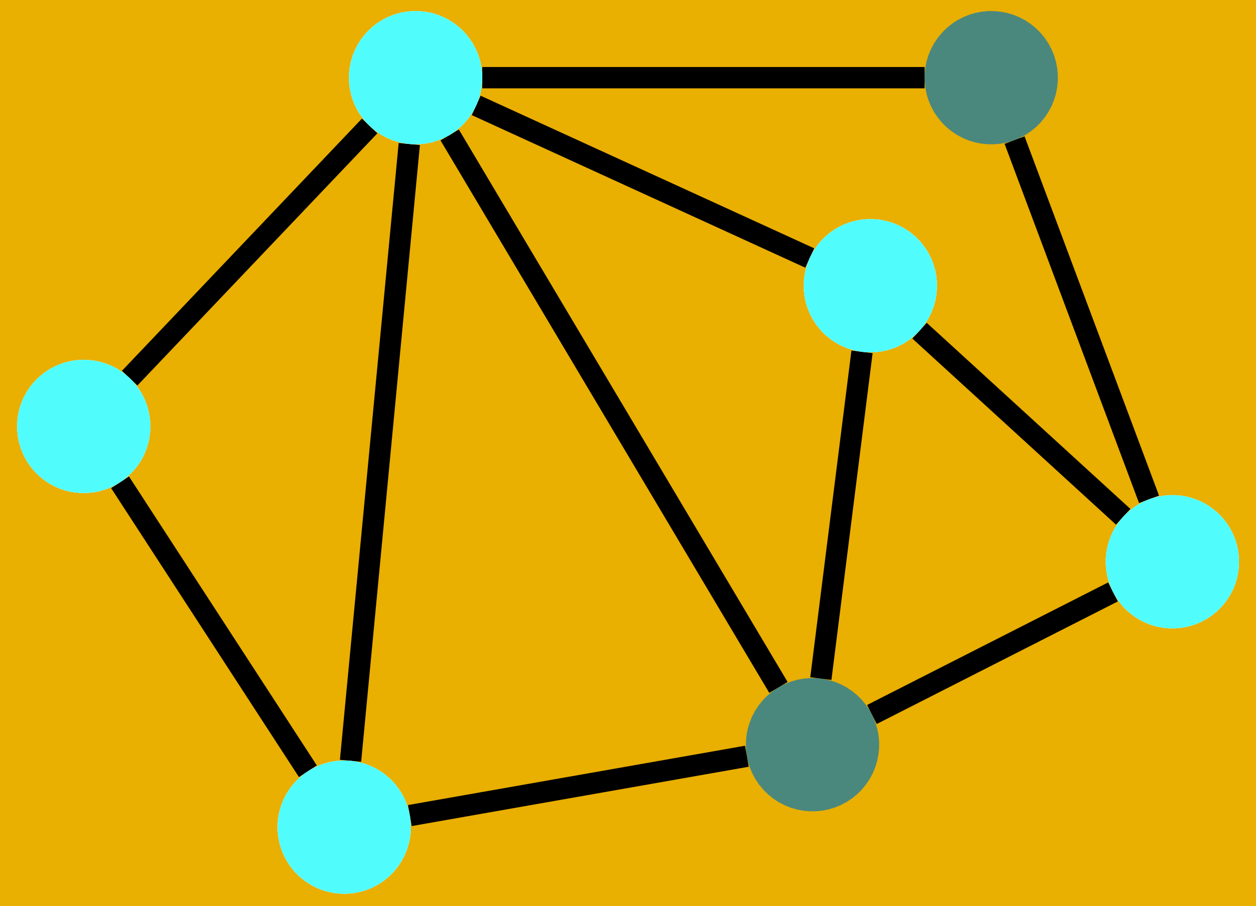


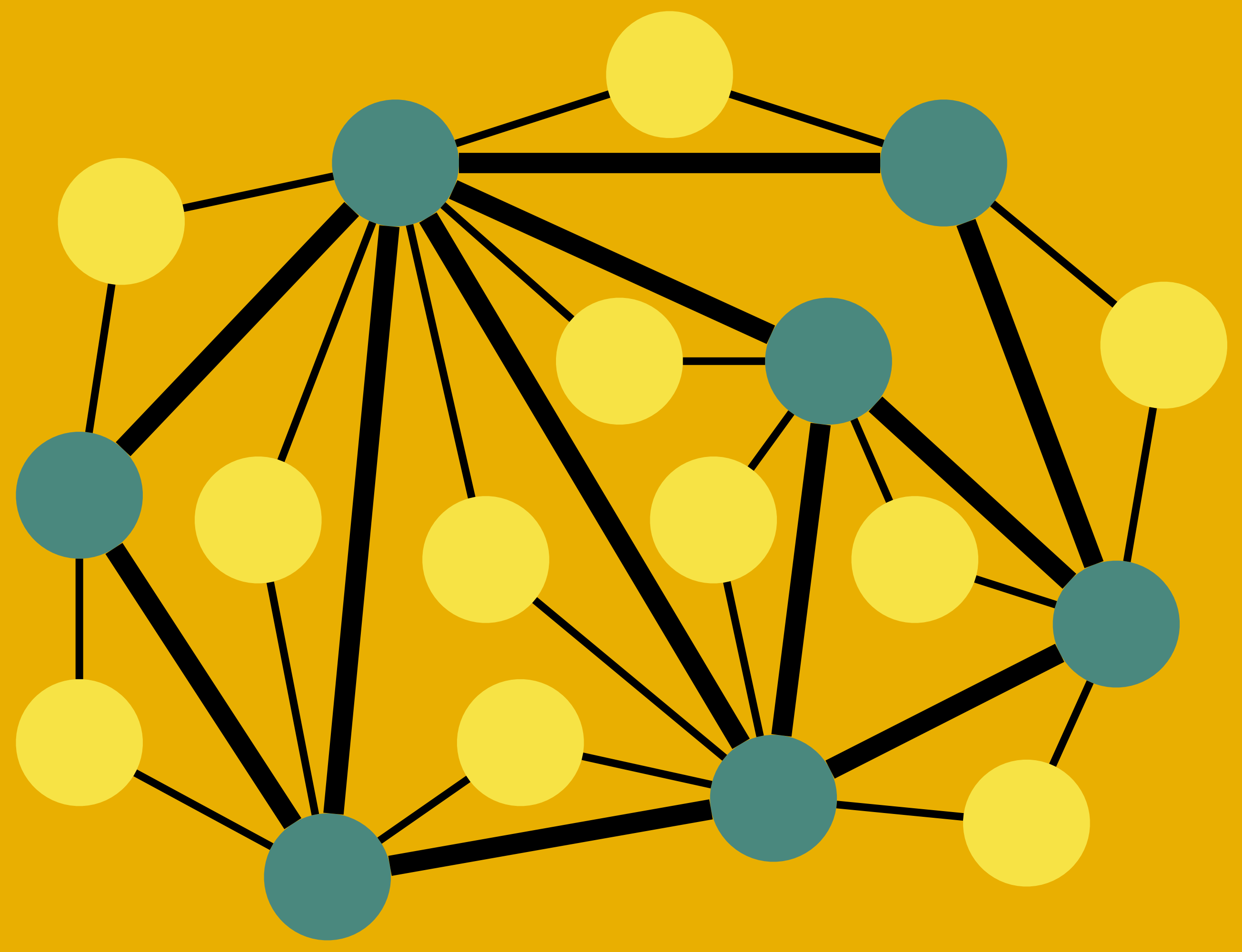


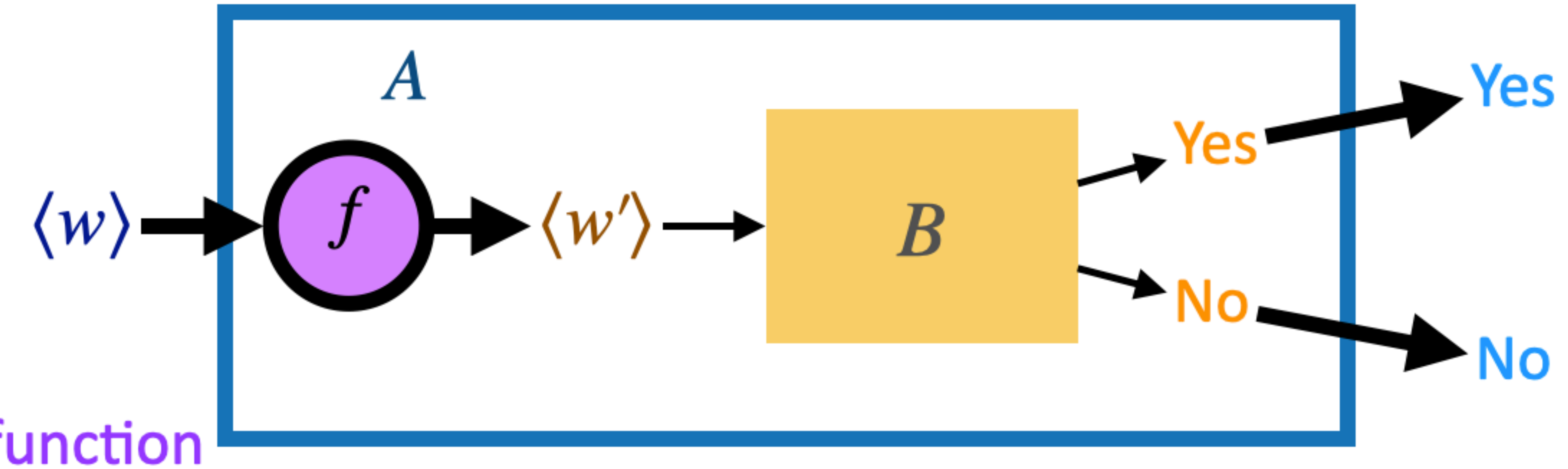

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time
2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$
3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

If $\langle G, k \rangle$ is a yes-instance

$G$ has a size-$k$ vertex cover $C$





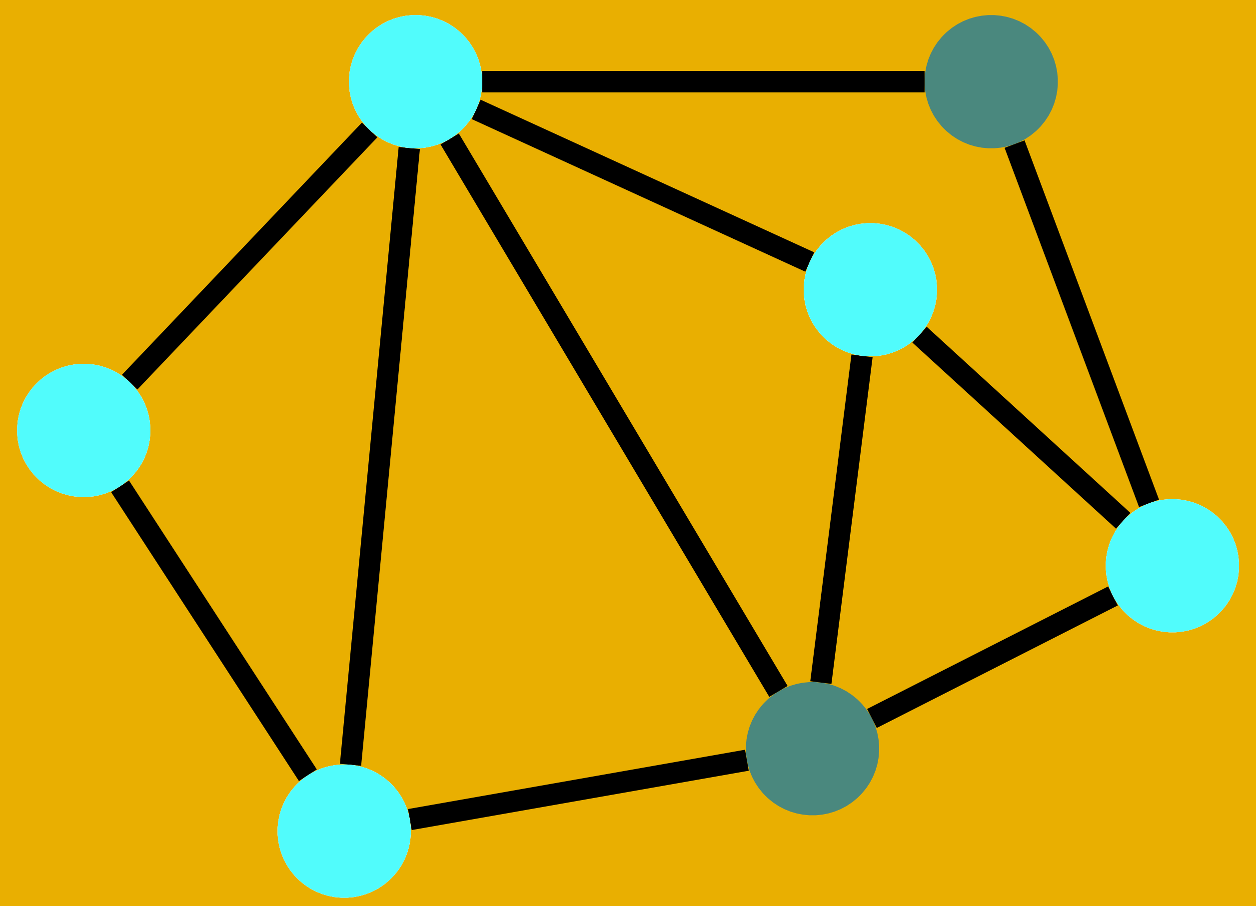


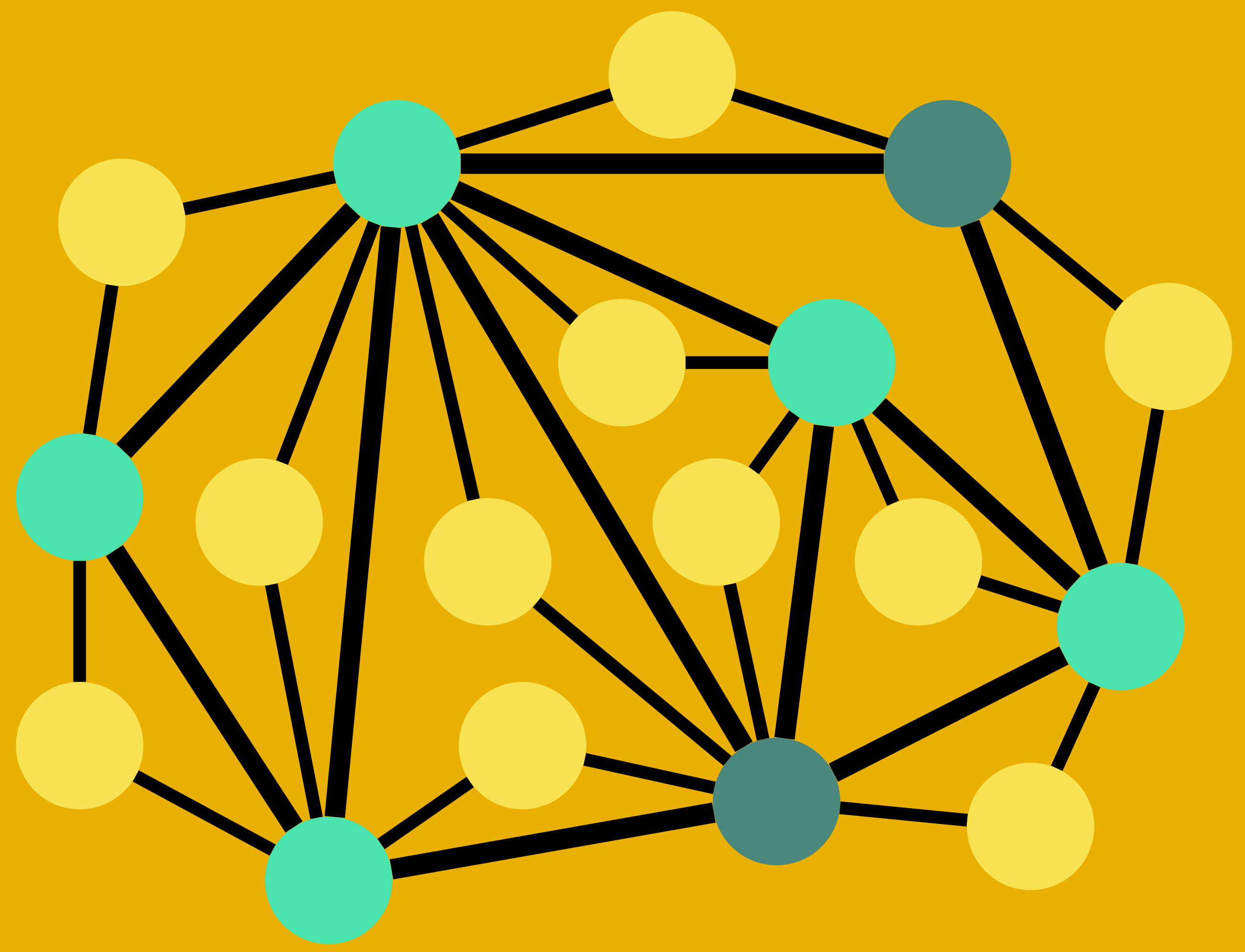

1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time
2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$
3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

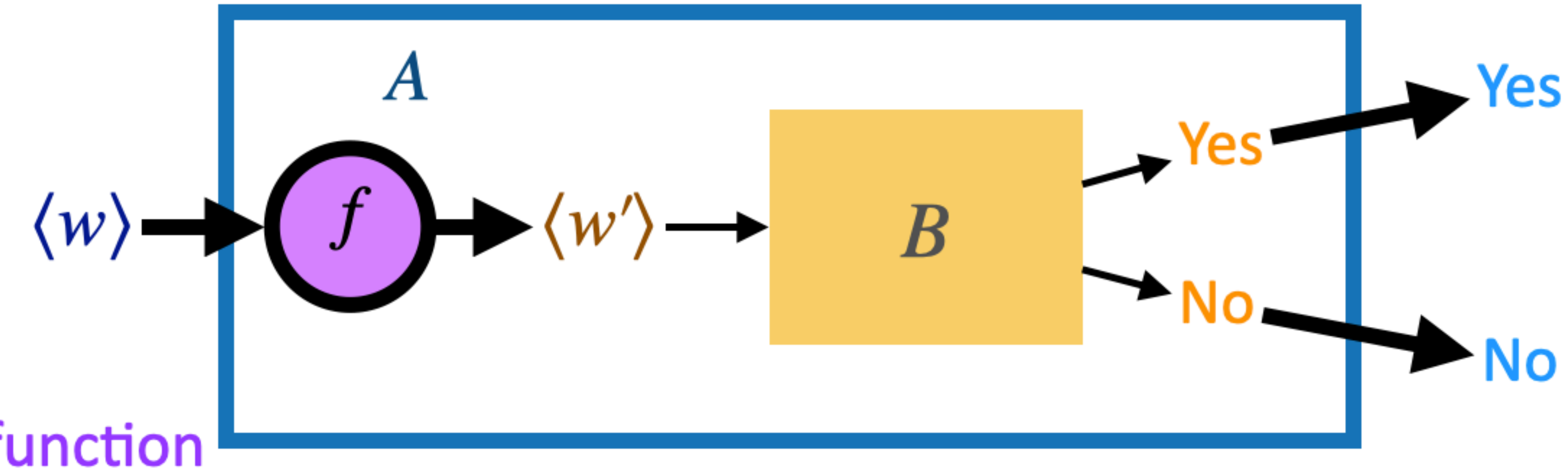
If $\langle G, k \rangle$ is a yes-instance

$G$ has a size-$k$ vertex cover $C$

$C$ is a feedback vertex set in $G'$ (since…)



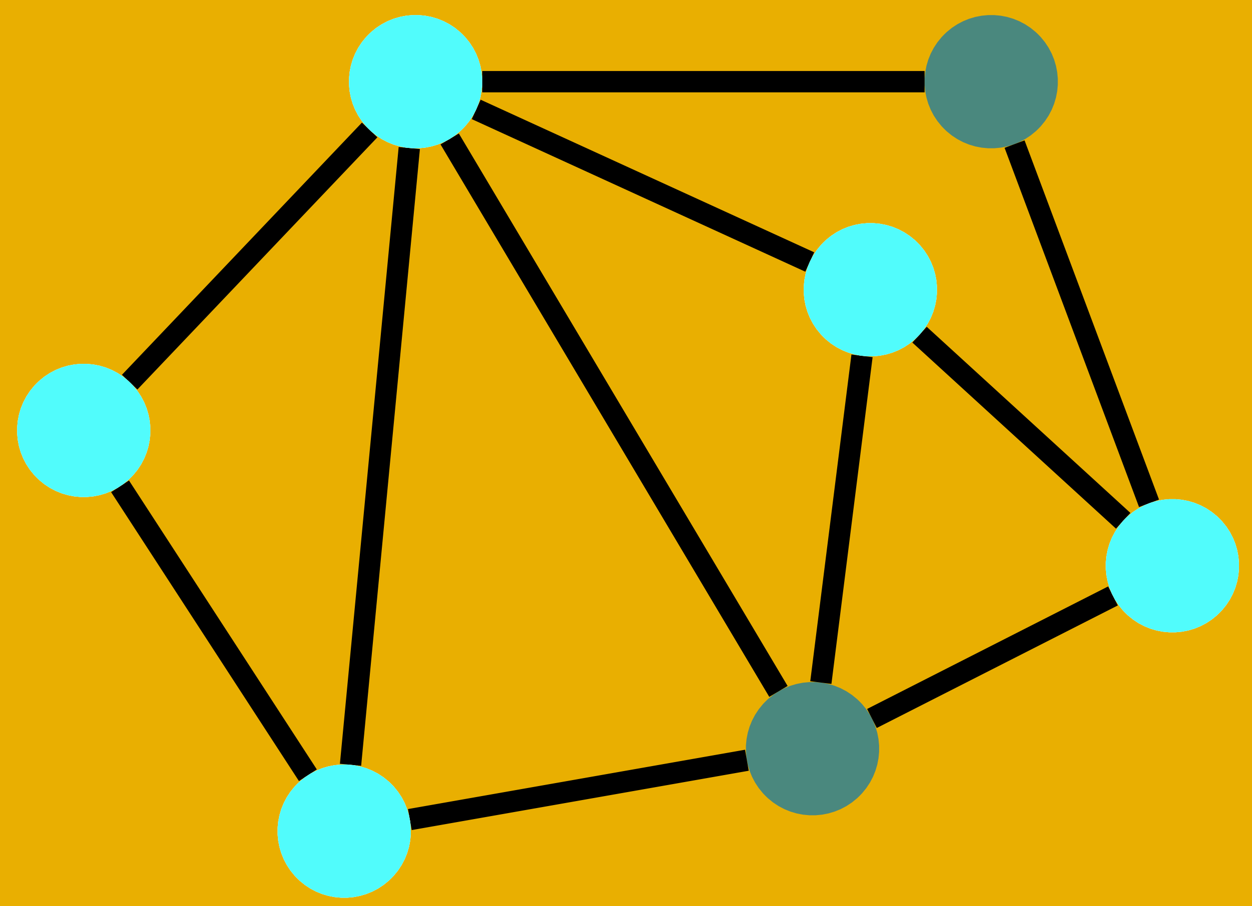





1. Show that there is a function that transforms every $w$ to $w'$ in polynomial time
2. Show that for any yes-instance $w' \in B$, the corresponding instance $w$ is also a yes-instance of $A$
3. Show that for any no-instance $w' \notin B$, the corresponding instance $w$ is also a no-instance of $A$

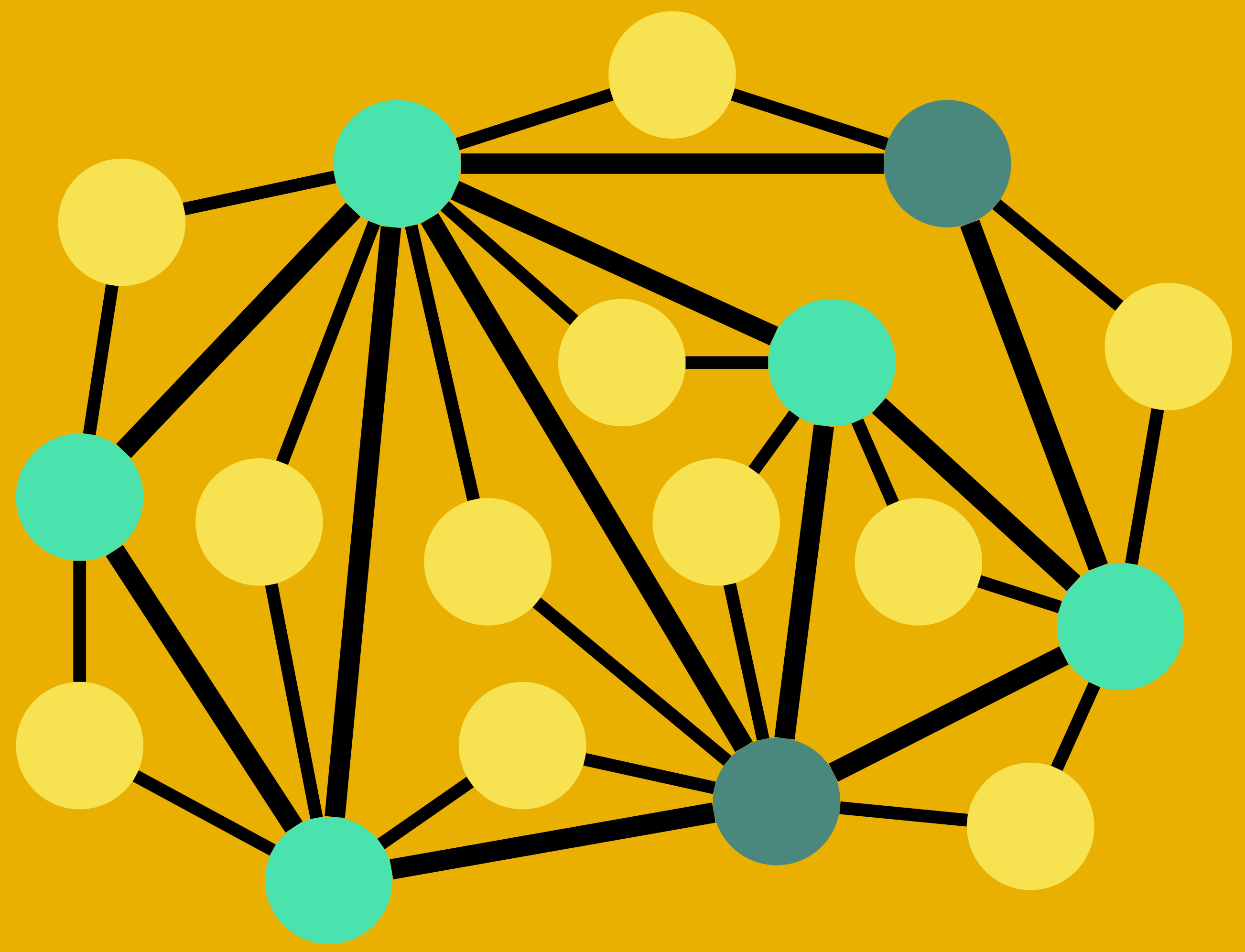
If $\langle G, k \rangle$ is a yes-instance

$G$ has a size-$k$ vertex cover $C$



$C$ is a feedback vertex set in $G'$ (since…)

Therefore, $\langle G', k \rangle$ is a yes-instance



# Outline

- More NP-Hardness proofs
  - 3SAT $\leq_p$ VERTEX-COVER
  - VERTEX-COVER $\leq_p$ INDEPENDENT SET
  - VERTEX-COVER $\leq_p$ FEEDBACK-VERTEX-SET
  - VERTEX-COVER $\leq_p$ Integer Linear Program
- Pseudo-polynomial time algorithms
- **NP** and **Co-NP**
- Turing undecidable languages

# ILP

**Minimum Vertex Cover problem (decision version)**

- Input: a graph $G = (V, E)$ and a integer $k$
- Output:
  - *yes* if there is a subset of vertices with cardinality at most $k$ that **all edges are covered by this subset of vertices**
  - *no* otherwise

**Integer linear programming**

- Input:
  - $\Sigma_i c_i x_i \leq k$, with $A\overrightarrow{x} \geq \overrightarrow{b}$ and $x_i \in \{0,1\}$
- Output:
  - *yes* if there is an assignment of $x_i$s such that the objective value is at most $k$ and constraints are satisfied
  - *no* otherwise

$\Sigma_i x_i \leq k$

for all edge $(u, v)$ in $E$, $x_u + x_v \geq 1$

# NP-complete Problems Map

# Outline

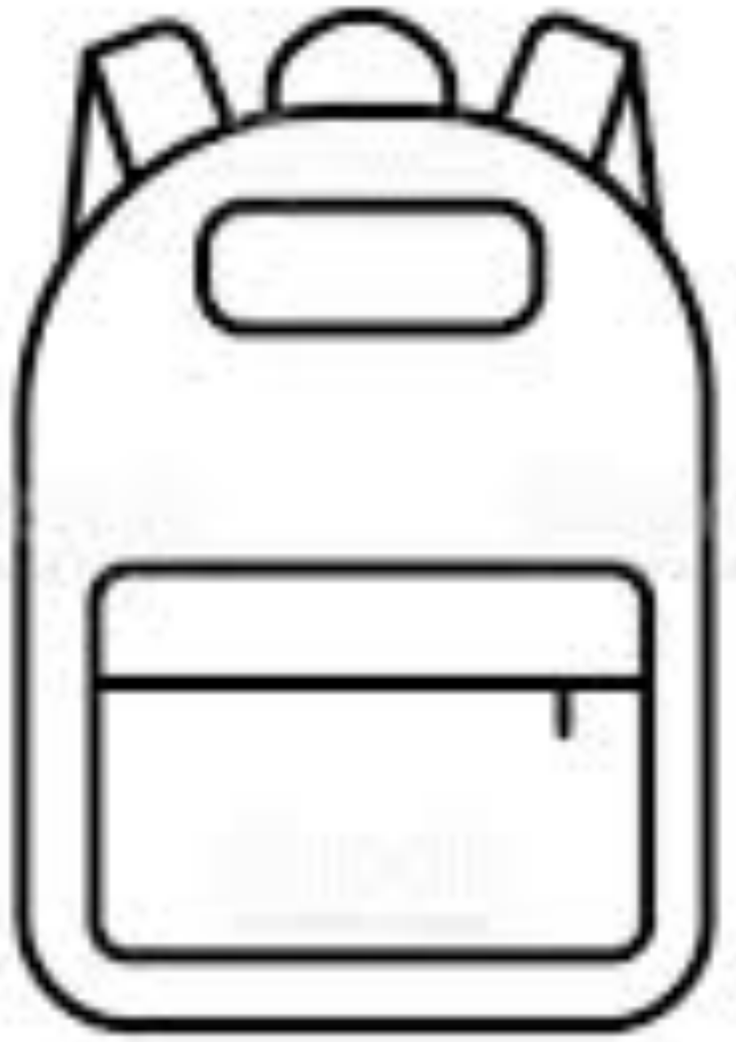
- More NP-Hardness proofs
  - 3SAT $\leq_p$ VERTEX-COVER
  - VERTEX-COVER $\leq_p$ INDEPENDENT SET
  - VERTEX-COVER $\leq_p$ FEEDBACK-VERTEX-SET
  - VERTEX-COVER $\leq_p$ Integer Linear Program
- Pseudo-polynomial time algorithms
- **NP** and **Co-NP**
- Turing undecidable languages

# Strong and Weak NP-complete

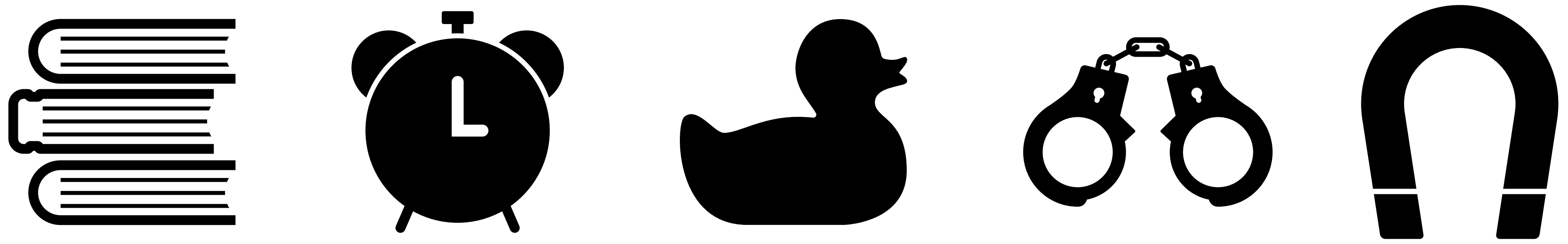
- KANPSACK problem: Give a set $S$ of items, each with an integer value $v_i$ and integer weight $w_i$. Also give integers $B$ and $V$. Is there a subset of $S$ of weight no more than $B$ with total value at least $V$?

# Strong and Weak NP-complete

- KANPSACK problem: Give a set $S$ of items, each with an integer value $v_i$ and integer weight $w_i$. Also give integers $B$ and $V$. Is there a subset of $S$ of weight no more than $B$ with total value at least $V$?



Weight capacity: 100



| | | | | | |
|---|---|---|---|---|---|
| value | 100 | 80 | 68 | 42 | 25 |
| weight | 50 | 45 | 10 | 15 | 20 |

# Strong and Weak NP-complete

- KANPSACK problem: Give a set $S$ of items, each with an integer value $v_i$ and integer weight $w_i$. Also give integers $B$ and $V$. Is there a subset of $S$ of weight no more than $B$ with total value at least $V$?

- KNAPSACK is NP-complete

# Strong and Weak NP-complete

- KANPSACK problem: Give a set $S$ of items, each with an integer value $v_i$ and integer weight $w_i$. Also give integers $B$ and $V$. Is there a subset of $S$ of weight no more than $B$ with total value at least $V$?

- KNAPSACK is NP-complete

- Using dynamic programming, it can be solved in $O(nB)$ time.
  - $W(j, w) := \max\{\Sigma_{i \in S} v_i \,|\, S \subseteq \{1,2,\cdots,j\}, \Sigma_{i \in S} w_i \leq w\}$
  - $W(j+1,w) = \max\{W(j,w), W(j, w - w_{j+1}) + v_{j+1}\}$

# Strong and Weak NP-complete

- KANPSACK problem: Give a set $S$ of items, each with an integer value $v_i$ and integer weight $w_i$. Also give integers $B$ and $V$. Is there a subset of $S$ of weight no more than $B$ with total value at least $V$?

- KNAPSACK is NP-complete

- Using dynamic programming, it can be solved in $O(nB)$ time.
  - $W(j, w) := \max\{\Sigma_{i \in S} v_i \,|\, S \subseteq \{1,2,\cdots,j\}, \Sigma_{i \in S} w_i \leq w\}$
  - $W(j+1, w) = \max\{W(j,w), W(j, w - w_{j+1}) + v_{j+1}\}$

- Have we just shown that P = NP?

# Strong and Weak NP-complete

- **Weak NP-complete** (**NP-complete in the ordinary sense**):

# Strong and Weak NP-complete

- **Weak NP-complete** (**NP-complete in the ordinary sense**):
    - Problem is NP-complete if numbers are given in binary, but polynomial time solvable when numbers are given in unary encoding.

# Strong and Weak NP-complete

- **Weak NP-complete** (**NP-complete in the ordinary sense**):
  - Problem is NP-complete if numbers are given in binary, but polynomial time solvable when numbers are given in unary encoding.
  - Algorithms are known which solve them in time bounded by a polynomial in the numeric value of the input instead of in the length of the input length.

# Strong and Weak NP-complete

- **Weak NP-complete** (**NP-complete in the ordinary sense**):
    - Problem is NP-complete if numbers are given in binary, but polynomial time solvable when numbers are given in unary encoding.
    - Algorithms are known which solve them in time bounded by a polynomial in the numeric value of the input instead of in the length of the input length.
    - The problem is solvable in **pseudo-polynomial** time

# Strong and Weak NP-complete

- **Weak NP-complete** (**NP-complete in the ordinary sense**):
  - Problem is NP-complete if numbers are given in binary, but polynomial time solvable when numbers are given in unary encoding.
  - Algorithms are known which solve them in time bounded by a polynomial in the numeric value of the input instead of in the length of the input length.
  - The problem is solvable in **pseudo-polynomial** time

- **Strong NP-complete** (**NP-complete in the strong sense**):
  - Problem is NP-complete if numbers are given in unary encoding
  - Problem is NP-complete even when the numerical parameters are bounded by a polynomial in the input size

# Strong and Weak NP-complete

- **Weak NP-complete** (**NP-complete in the ordinary sense**):
  - Problem is NP-complete if numbers are given in binary, but polynomial time solvable when numbers are given in unary encoding.
  - Algorithms are known which solve them in time bounded by a polynomial in the numeric value of the input instead of in the length of the input length.
  - The problem is solvable in **pseudo-polynomial** time

- **Strong NP-complete** (**NP-complete in the strong sense**):
  - Problem is NP-complete if numbers are given in unary encoding
  - Problem is NP-complete even when the numerical parameters are bounded by a polynomial in the input size
  - Ex: SAT, 3SAT, VertexCover, FVS, IS, CLIQUE, ILP, BinPacking…

# Showing that a problem is strongly NP-hard

- You need to:

  1. Reduce it from a strongly NP-complete problem, and

  2. Ensure that the **magnitudes** of the numerical parameters generated during the reduction are bounded by a polynomial of input size

# Outline

- More NP-Hardness proofs
  - 3SAT $\leq_p$ VERTEX-COVER
  - VERTEX-COVER $\leq_p$ INDEPENDENT SET
  - VERTEX-COVER $\leq_p$ FEEDBACK-VERTEX-SET
  - VERTEX-COVER $\leq_p$ Integer Linear Program
- Pseudo-polynomial time algorithms
- **NP** and **Co-NP**
- Turing undecidable languages

# The Class **NP**

- Definition: **NP** is the class of languages that are decidable in polynomial time on a nondeterministic Turing machine.

- Definition: **NP** is the class of languages that are polynomial time *verifiable*.

# The Class **co-NP**

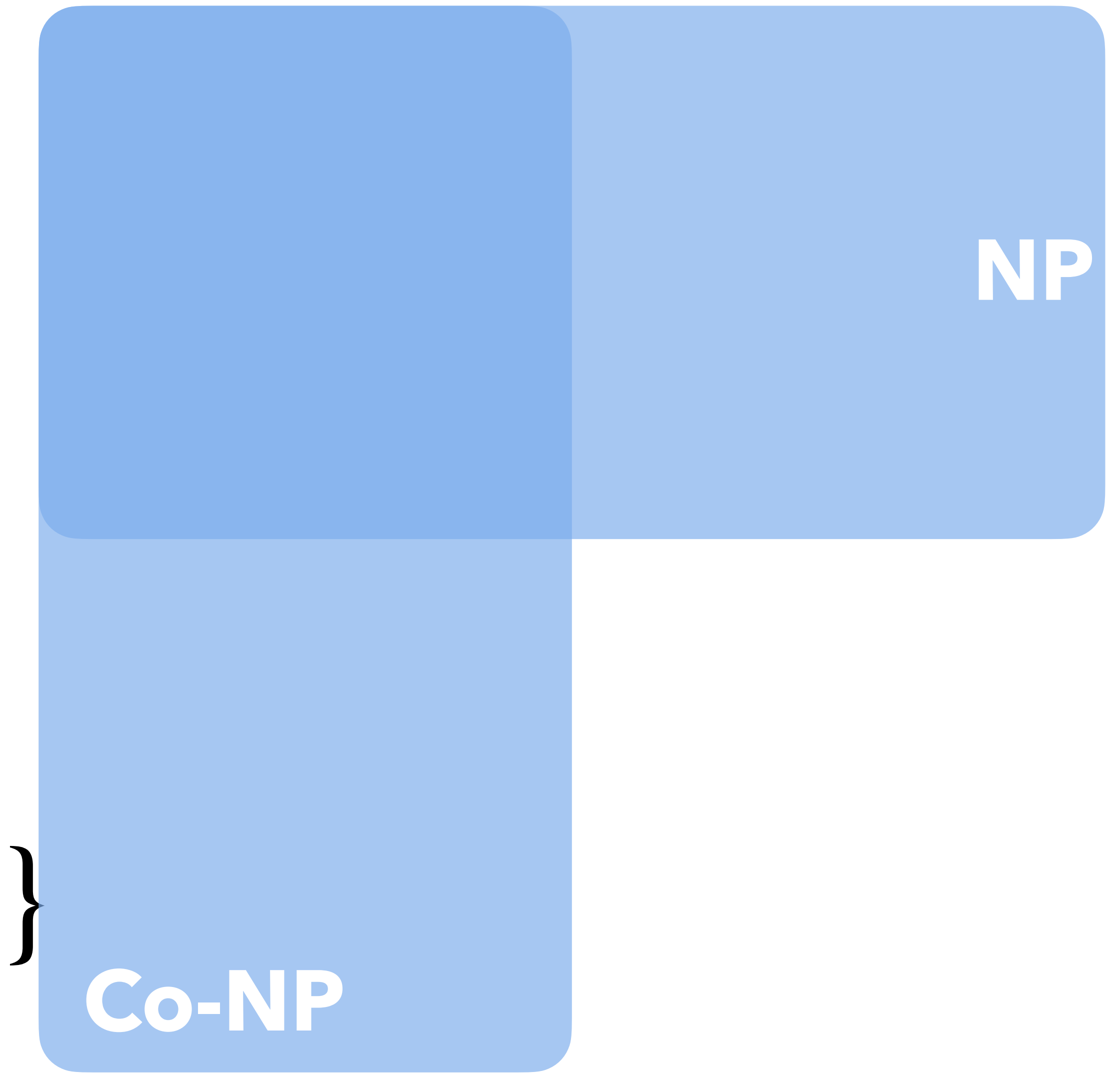
- Definition: **co-NP** is the class of languages that any no-instance are polynomial time *verifiable*.

# The Class **co-NP**

- Definition: **co-NP** is the class of languages that any no-instance are polynomial time *verifiable*.
- Definition: A language $L$ is in **co-NP** if $\overline{L} \in$ NP.
  - $\overline{L}$: complement language of $L$
    - NOT-HAMILTONIAN $= \{\langle G \rangle \,|\, G$ has no Hamiltonian cycle$\}$
    - UNSATISFIABLE $= \{\langle \phi \rangle \,|\,$ All truth assignments make $\phi$ false$\}$
- $P \subseteq$ NP $\cap$ co-NP

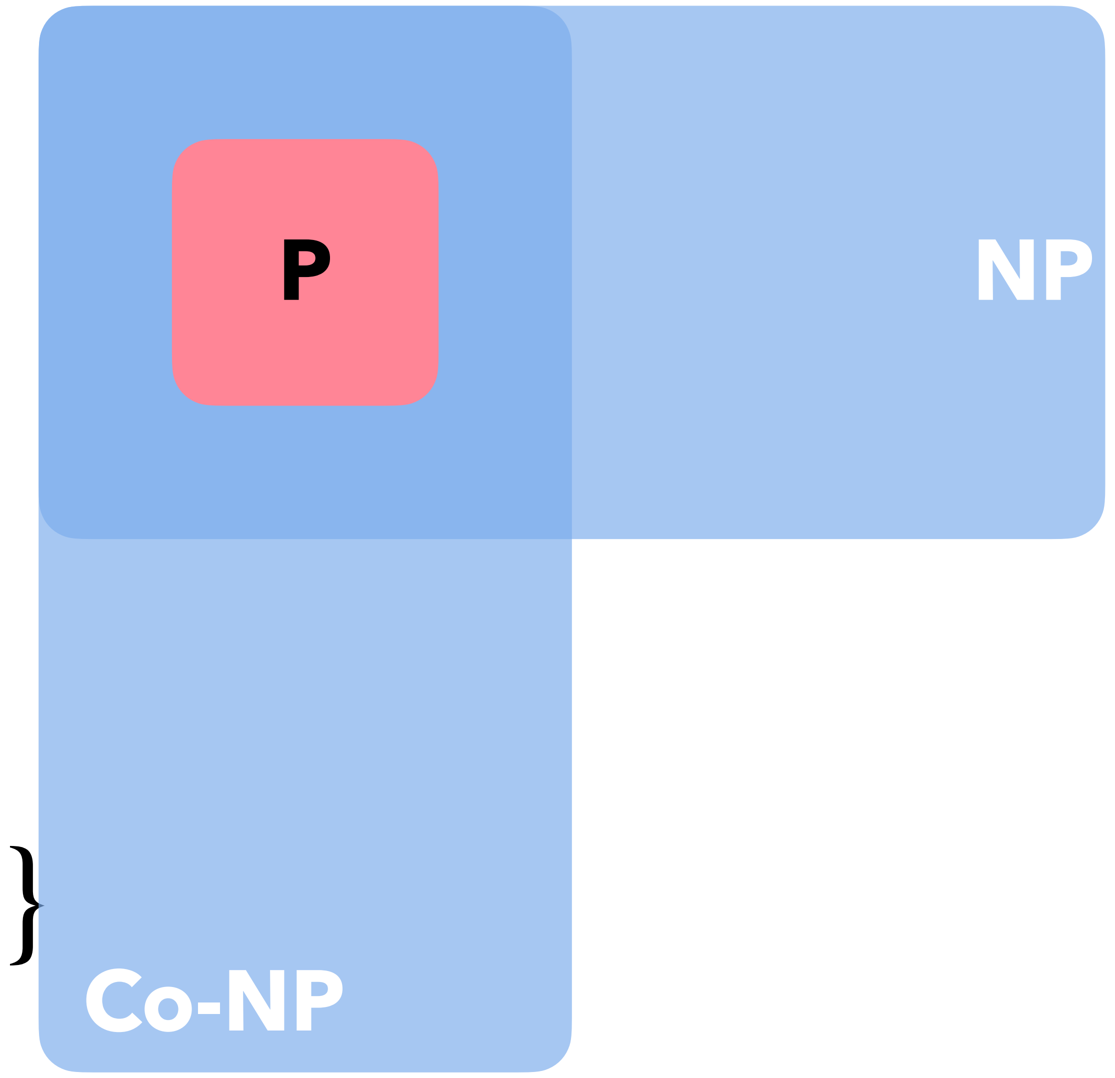
# The Class **co-NP**

- Definition: **co-NP** is the class of languages that any no-instance are polynomial time *verifiable*.
- Definition: A language $L$ is in **co-NP** if $\overline{L} \in$ NP.
  - $\overline{L}$: complement language of $L$
    - NOT-HAMILTONIAN $= \{\langle G \rangle \mid G$ has no Hamiltonian cycle$\}$
    - UNSATISFIABLE $= \{\langle \phi \rangle \mid$ All truth assignments make $\phi$ false$\}$
- $P \subseteq$ NP $\cap$ co-NP



# The Class **co-NP**

- Definition: **co-NP** is the class of languages that any no-instance are polynomial time *verifiable*.
- Definition: A language $L$ is in **co-NP** if $\overline{L} \in$ NP.
  - $\overline{L}$: complement language of $L$
    - NOT-HAMILTONIAN $= \{\langle G \rangle \mid G$ has no Hamiltonian cycle$\}$
    - UNSATISFIABLE $= \{\langle \phi \rangle \mid$ All truth assignments make $\phi$ false$\}$
- $P \subseteq$ NP $\cap$ co-NP



# A more natural example for NP and coNP

- INTEGER_FACTORISATION $= \{\langle n, k\rangle \,|\, n$ has a prime factor less than $k\}$ is in NP and co-NP:
  - In NP: A certificate is two numbers $c$ and $p < k$ where $p$ is a prime* such that $cp = n$
  - In co-NP: A certificate is the prime factorization of $n$
  - Is INTETER_FACTORISATION in P? For cryptography sake we hope not!

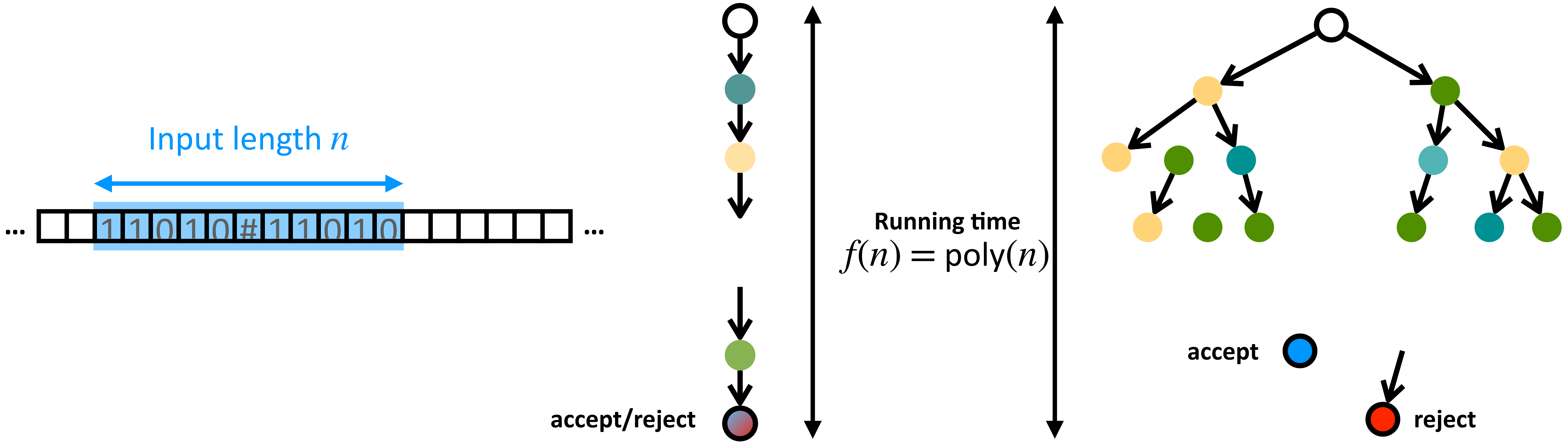
* Prime-testing is in **P** [M Agrawal, N Kayal, N Saxena, 2004]

# Outline

- More NP-Hardness proofs
  - 3SAT $\leq_p$ VERTEX-COVER
  - VERTEX-COVER $\leq_p$ INDEPENDENT SET
  - VERTEX-COVER $\leq_p$ FEEDBACK-VERTEX-SET
  - VERTEX-COVER $\leq_p$ Integer Linear Program
- Pseudo-polynomial time algorithms
- **NP** and **Co-NP**
- Turing undecidable languages

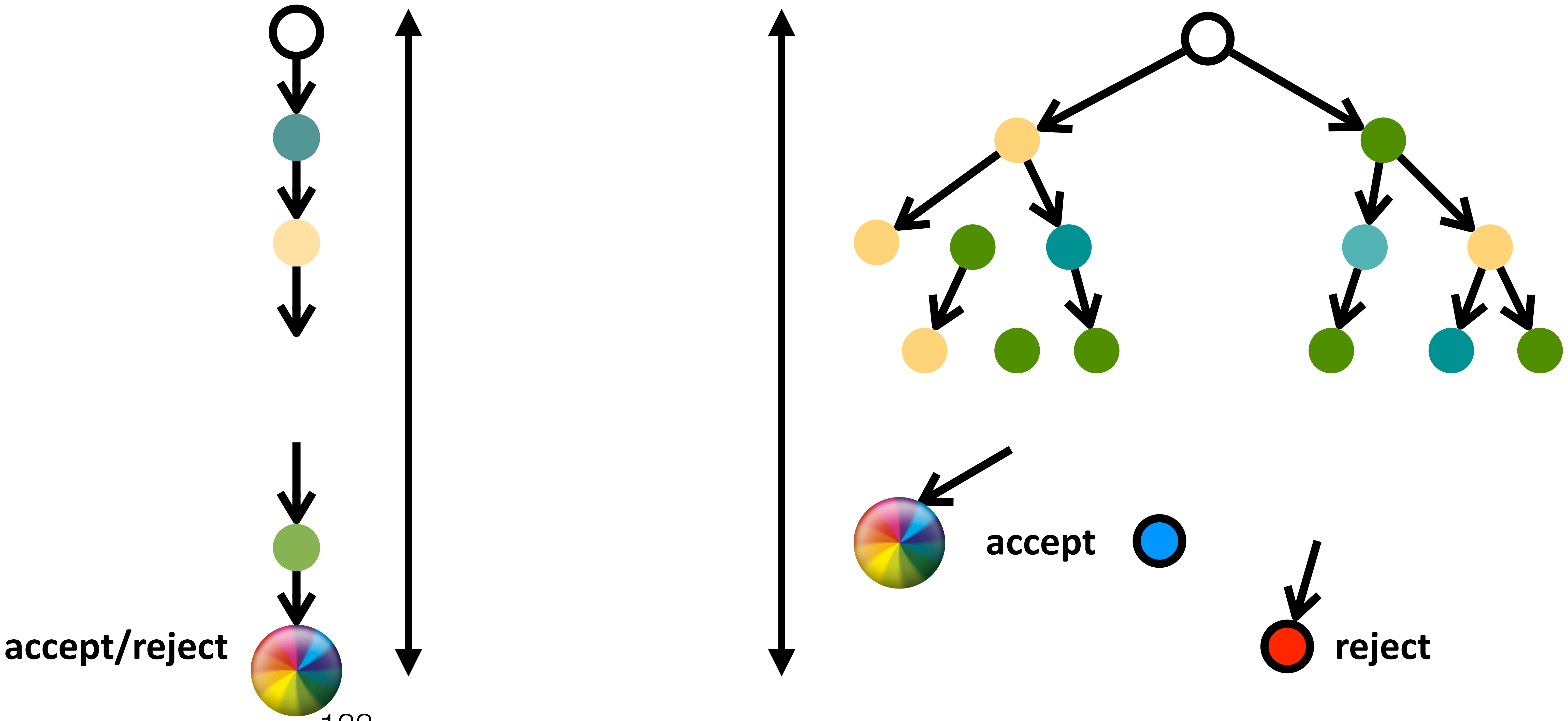
# Turing machine and Decidability

- The class **P** is the class of languages that are *accepted* or *rejected* in polynomial time by a deterministic Turing machine
- The class **NP** is the class of languages that can be verified in polynomial time by a deterministic Turing machine.



# Turing-Decidable Language

- Turing machine may not halt and enter a loop

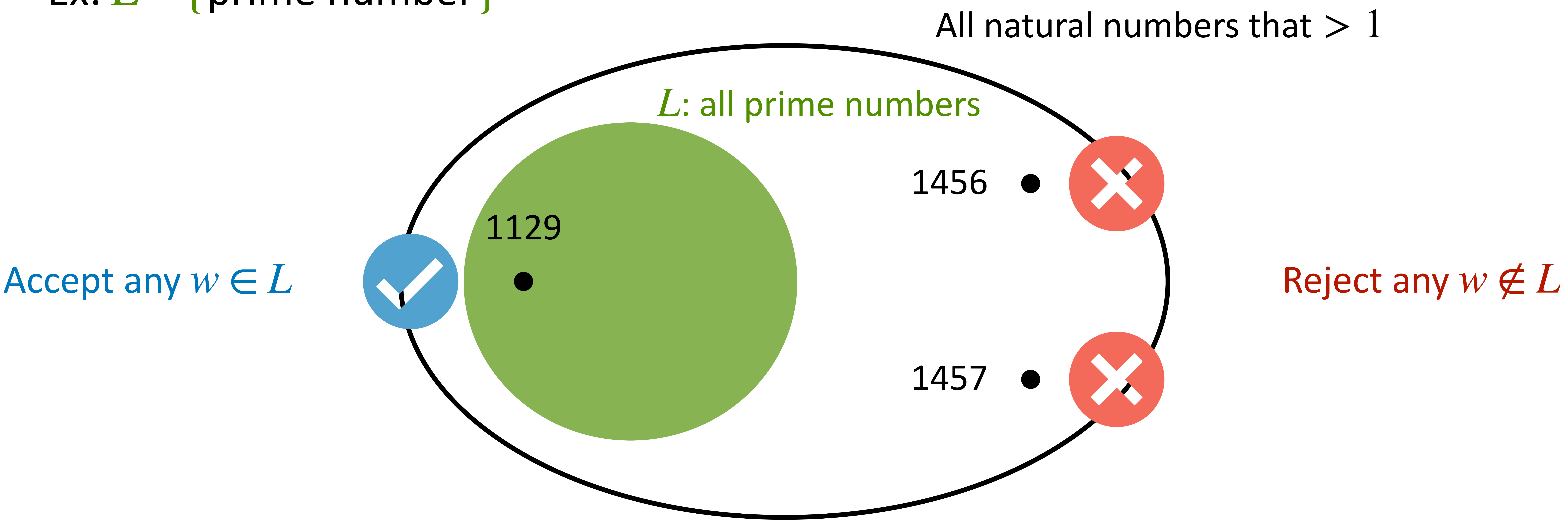


# Turing-Decidable Language

- A language $L$ is (Turing-)**decidable** if **some Turing machine *decides* it**
  - The Turing machine accepts all strings in $L$ and rejects all strings not in $L$

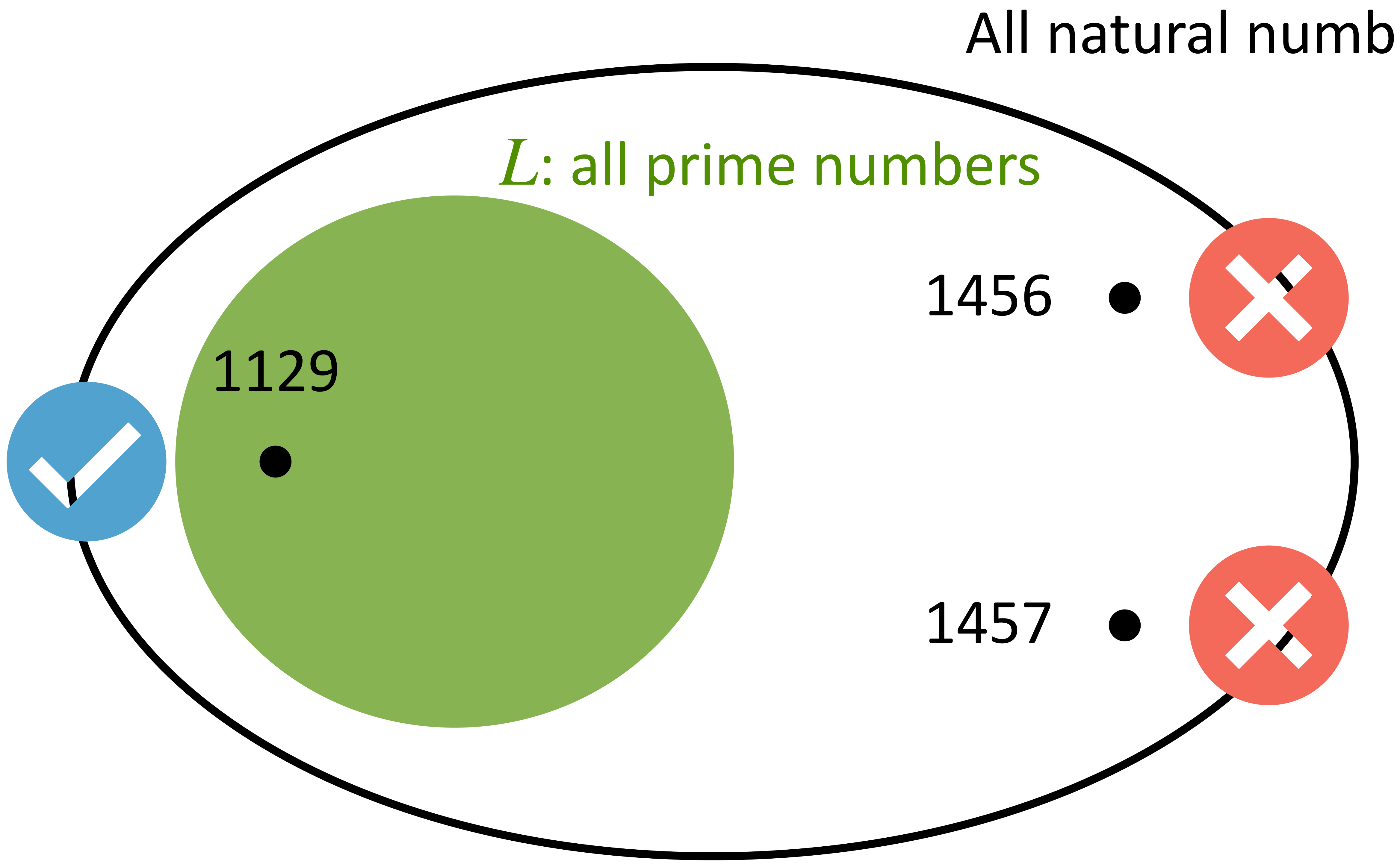
# Turing-Decidable Language

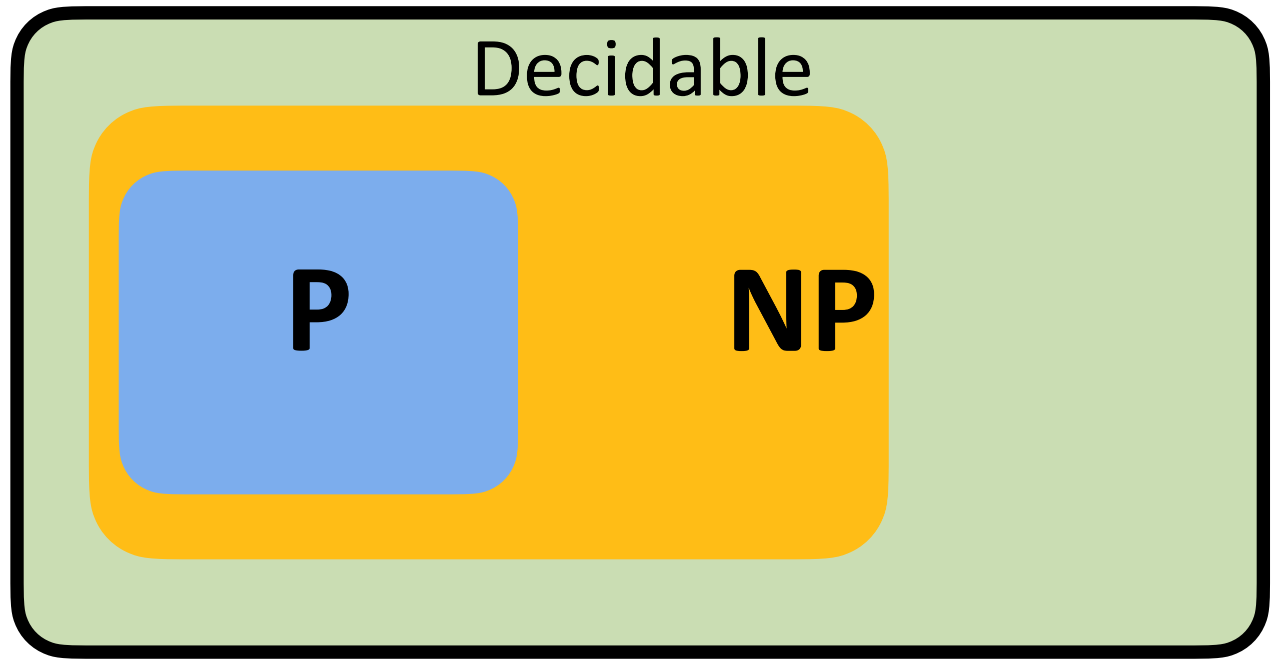
- A language $L$ is (Turing-)**decidable** if **some Turing machine *decides* it**
  - The Turing machine accepts all strings in $L$ and rejects all strings not in $L$
- Ex: $L = \{\text{prime number}\}$



# Turing-Decidable Language

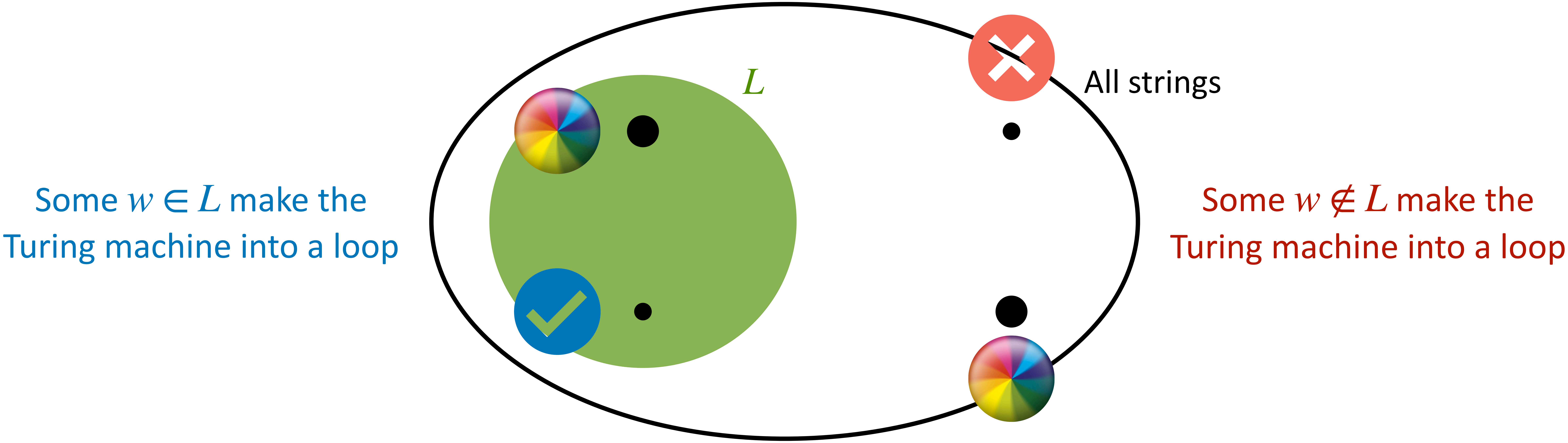
- A language $L$ is (Turing-)**decidable** if **some Turing machine *decides* it**
  - The Turing machine accepts all strings in $L$ and rejects all strings not in $L$
- All the problems in NP are decidable



# Undecidable Language

- A language $L$ is **undecidable** if
  for all Turing machine $M$,
  there exists $w \in L$ such that $M$ does not accept $w$ or there exists $w \notin L$ such that $M$ does not reject $w$



# Undecidable Languages

- $\mathsf{A}_{\mathsf{TM}} = \{\langle M, w\rangle \mid M$ is a Turing machine and $M$ accepts input string $w\}$
- Halting problem:
  $\mathsf{HALT}_{\mathsf{TM}} = \{\langle M, w\rangle \mid M$ is a Turing machine and $M$ accepts or rejects input string $w\}$

# Undecidable Languages

- $\mathsf{A}_{\mathsf{TM}} = \{\langle M, w\rangle \mid M$ is a Turing machine and $M$ accepts input string $w\}$
- Halting problem:
  $\mathsf{HALT}_{\mathsf{TM}} = \{\langle M, w\rangle \mid M$ is a Turing machine and $M$ accepts or rejects input string $w\}$

# $A_{TM}$ is undecidable

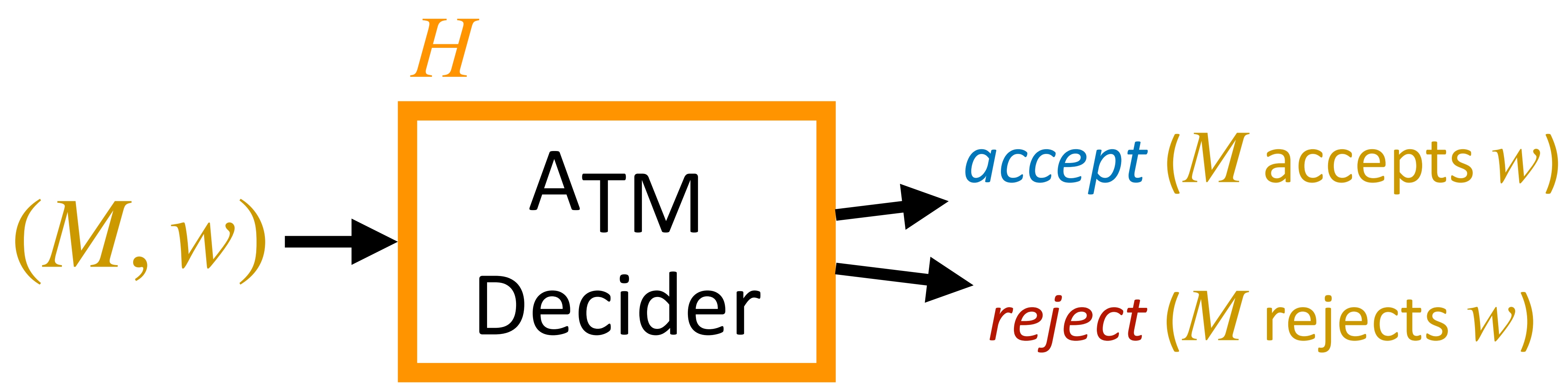
- $A_{TM} = \{\langle M, w \rangle \mid M$ is a Turing machine and $M$ accepts input string $w\}$

# $A_{\mathrm{TM}}$ is undecidable

- $A_{\mathrm{TM}} = \{\langle M, w\rangle \mid M$ is a Turing machine and $M$ accepts input string $w\}$

<Pf> Assume on the contrary that $A_{\mathrm{TM}}$ is decidable

$\Rightarrow$ there is a Turing machine $H$ that can decide $A_{\mathrm{TM}}$

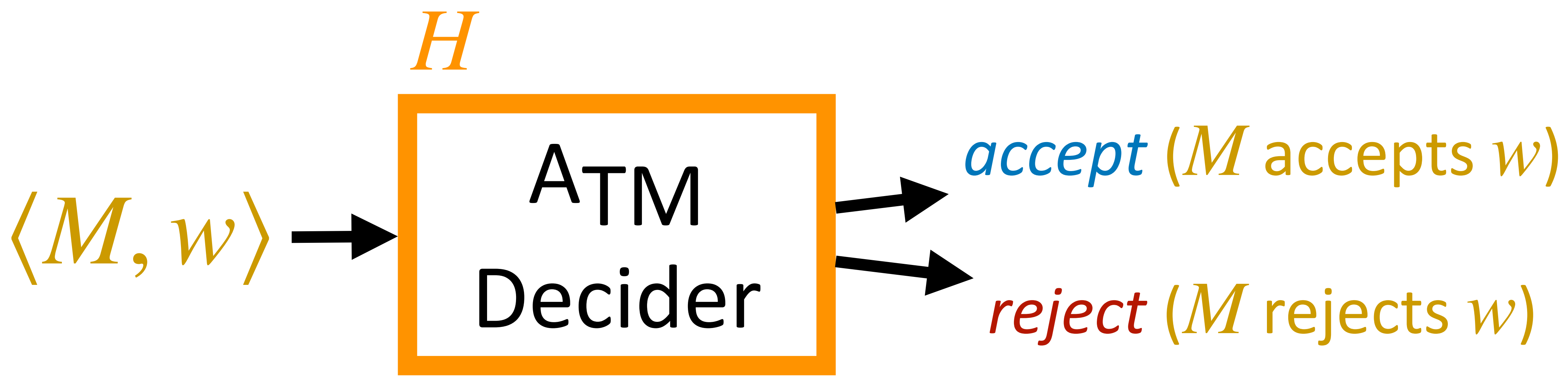


# $A_{TM}$ is undecidable

- $A_{TM} = \{\langle M, w\rangle \mid M$ is a Turing machine and $M$ accepts input string $w\}$

<Pf> Assume on the contrary that $A_{TM}$ is decidable

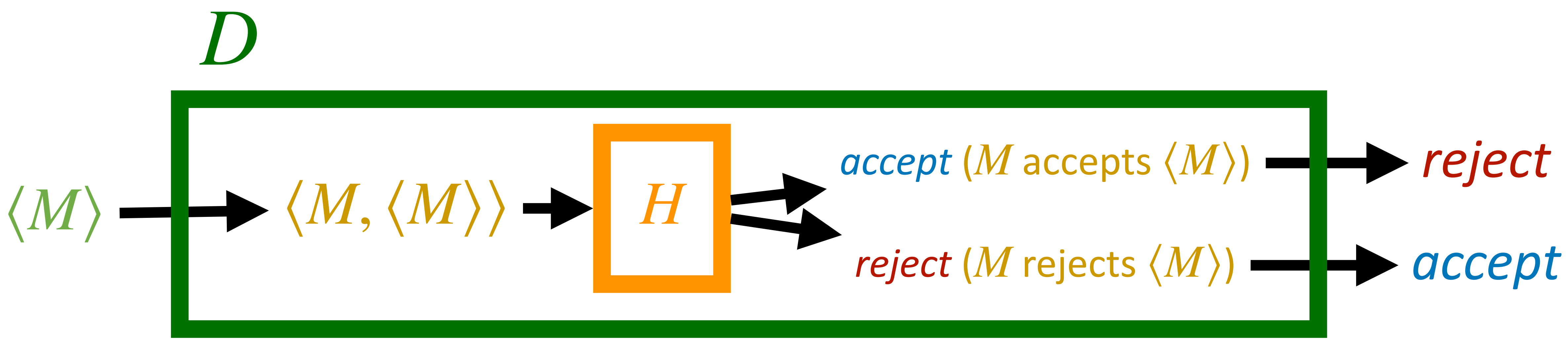
$\Rightarrow$ there is a Turing machine $H$ that can decide $A_{TM}$




Design a Turing machine $D$:

On input $\langle M\rangle$, where $M$ is a Turing machine

1. Run $H$ on input $\langle M, \langle M\rangle\rangle$
2. If $H$ accepts, *reject* and if $H$ rejects, *accept*

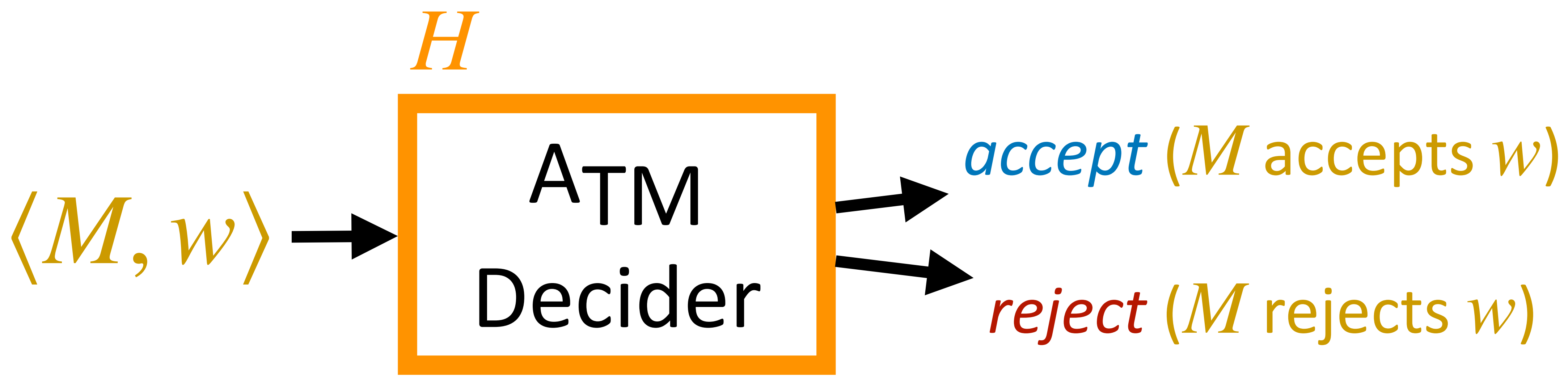


# $\mathrm{A_{TM}}$ is undecidable

- $\mathrm{A_{TM}} = \{\langle M, w\rangle \mid M$ is a Turing machine and $M$ accepts input string $w\}$

<Pf> Assume on the contrary that $\mathrm{A_{TM}}$ is decidable

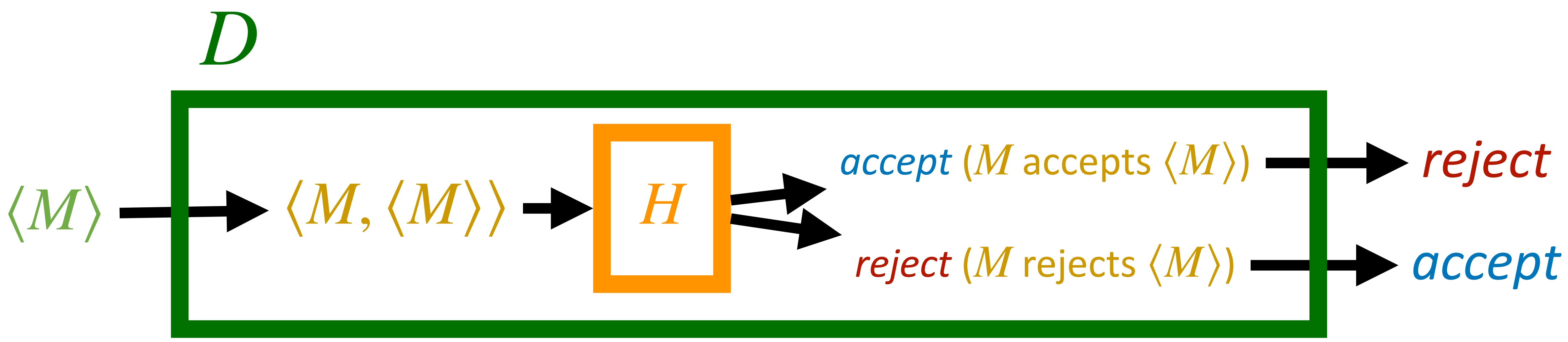
$\Rightarrow$ there is a Turing machine $H$ that can decide $\mathrm{A_{TM}}$



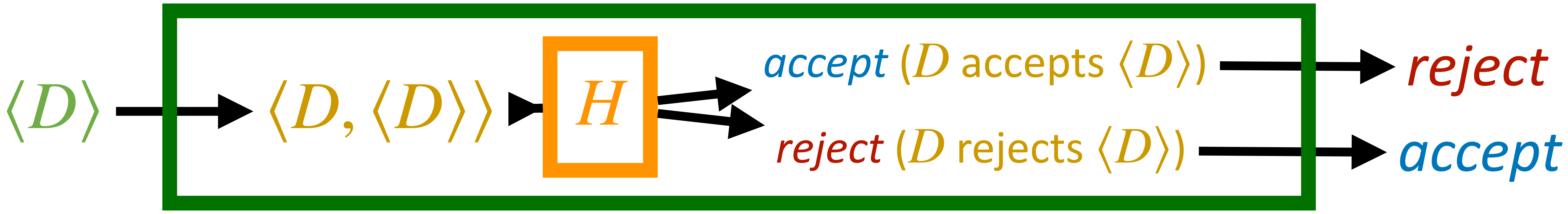

Design a Turing machine $D$:

On input $\langle M\rangle$, where $M$ is a Turing machine

1. Run $H$ on input $\langle M, \langle M\rangle\rangle$
2. If $H$ accepts, *reject* and if $H$ rejects, *accept*




Run $D$ on $\langle D\rangle$:

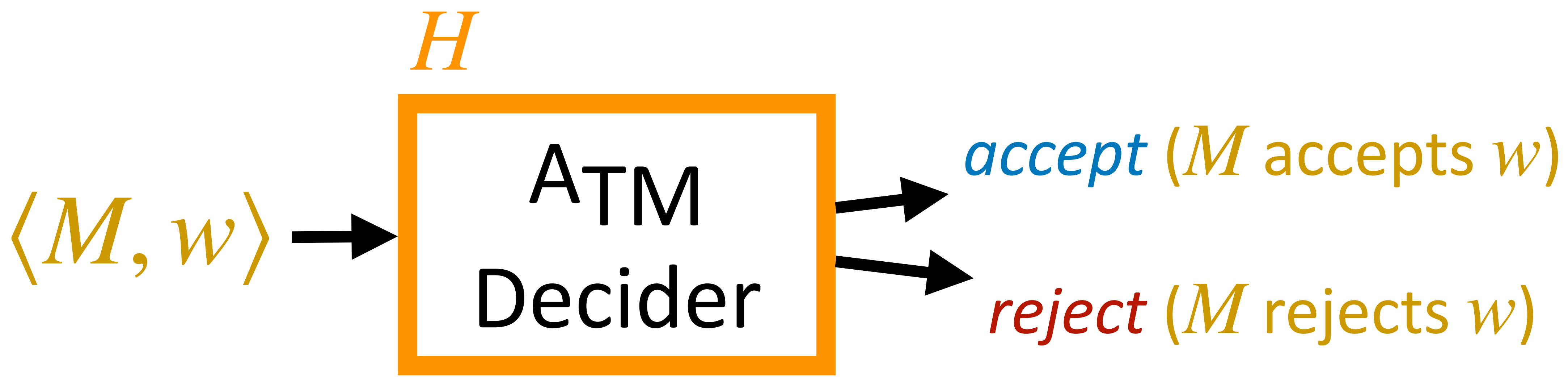


# $A_{TM}$ is undecidable

- $A_{TM} = \{\langle M, w\rangle \mid M$ is a Turing machine and $M$ accepts input string $w\}$

<Pf> Assume on the contrary that $A_{TM}$ is decidable

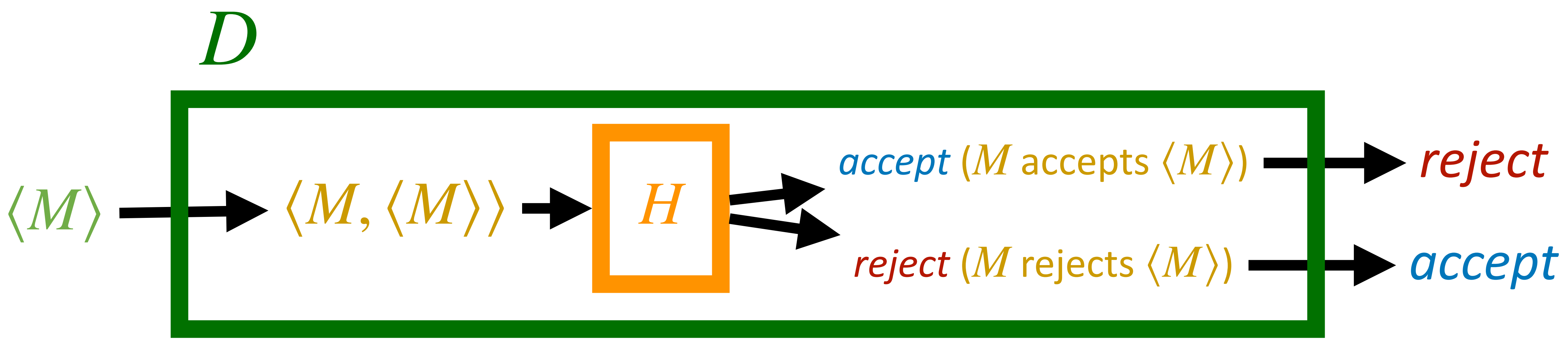
$\Rightarrow$ there is a Turing machine $H$ that can decide $A_{TM}$



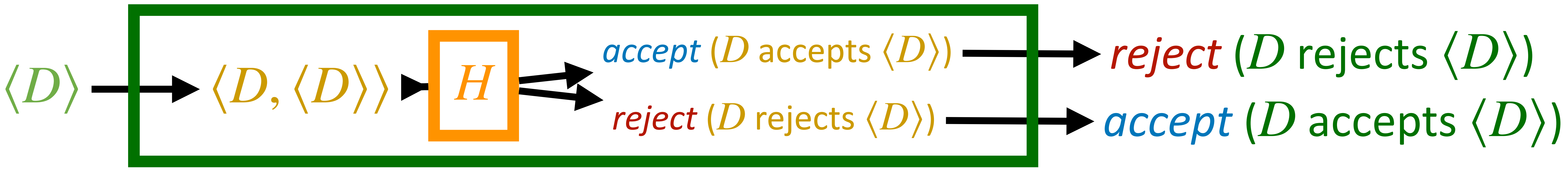

Design a Turing machine $D$:

On input $\langle M\rangle$, where $M$ is a Turing machine

1. Run $H$ on input $\langle M, \langle M\rangle\rangle$
2. If $H$ accepts, *reject* and if $H$ rejects, *accept*




Run $D$ on $\langle D\rangle$:

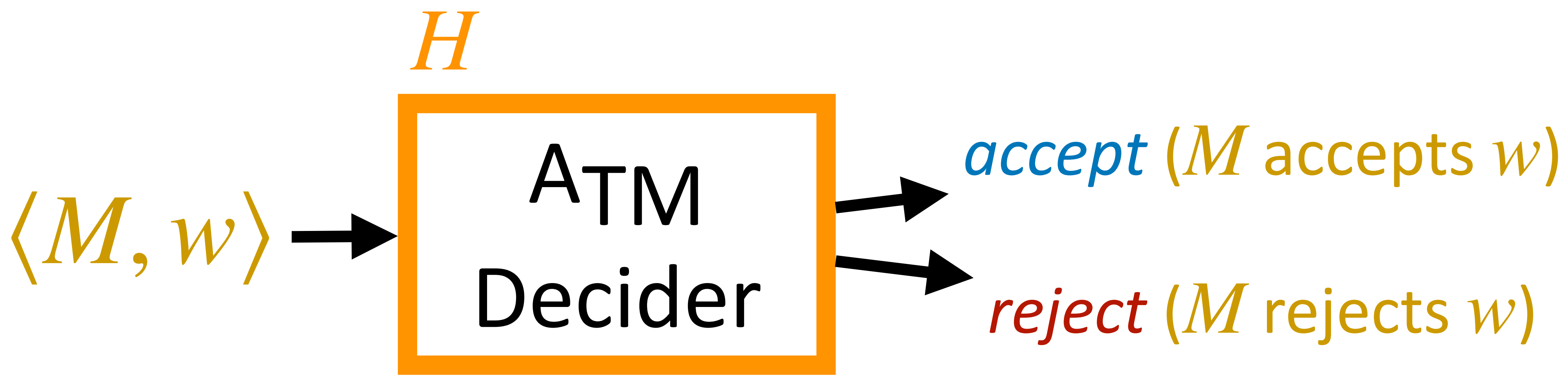


# $A_{TM}$ is undecidable

- $A_{TM} = \{\langle M, w\rangle \mid M$ is a Turing machine and $M$ accepts input string $w\}$

<Pf> Assume on the contrary that $A_{TM}$ is decidable

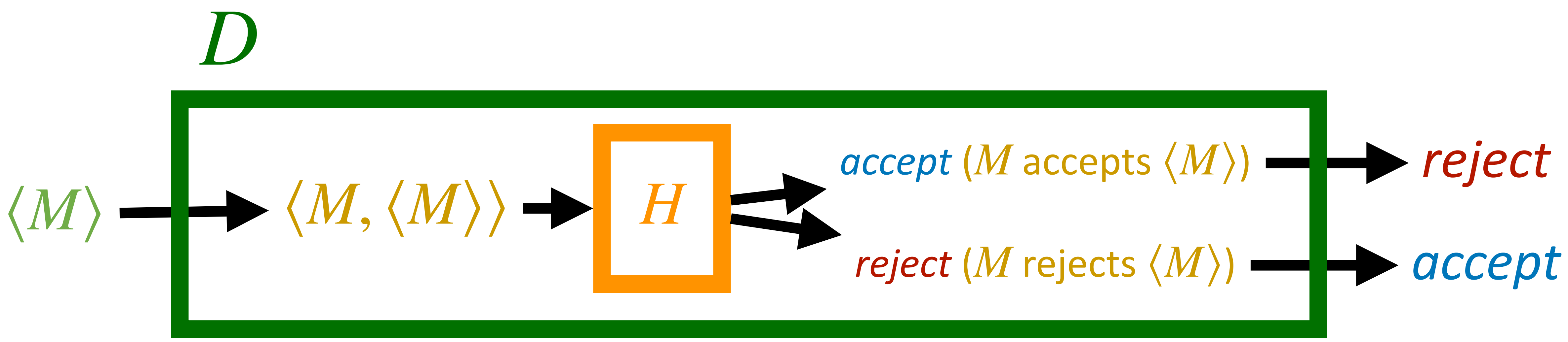
$\Rightarrow$ there is a Turing machine $H$ that can decide $A_{TM}$



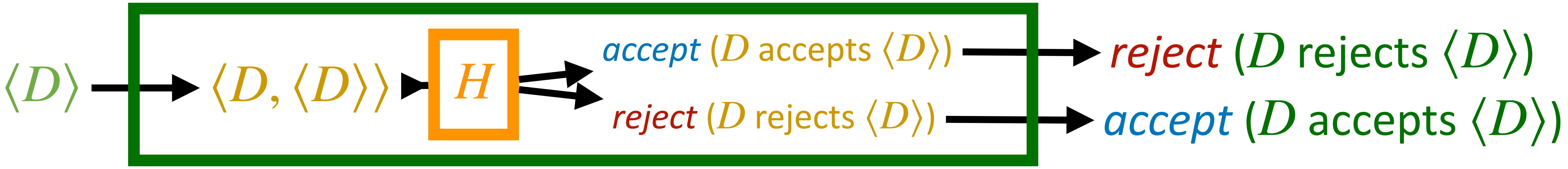

Design a Turing machine $D$:

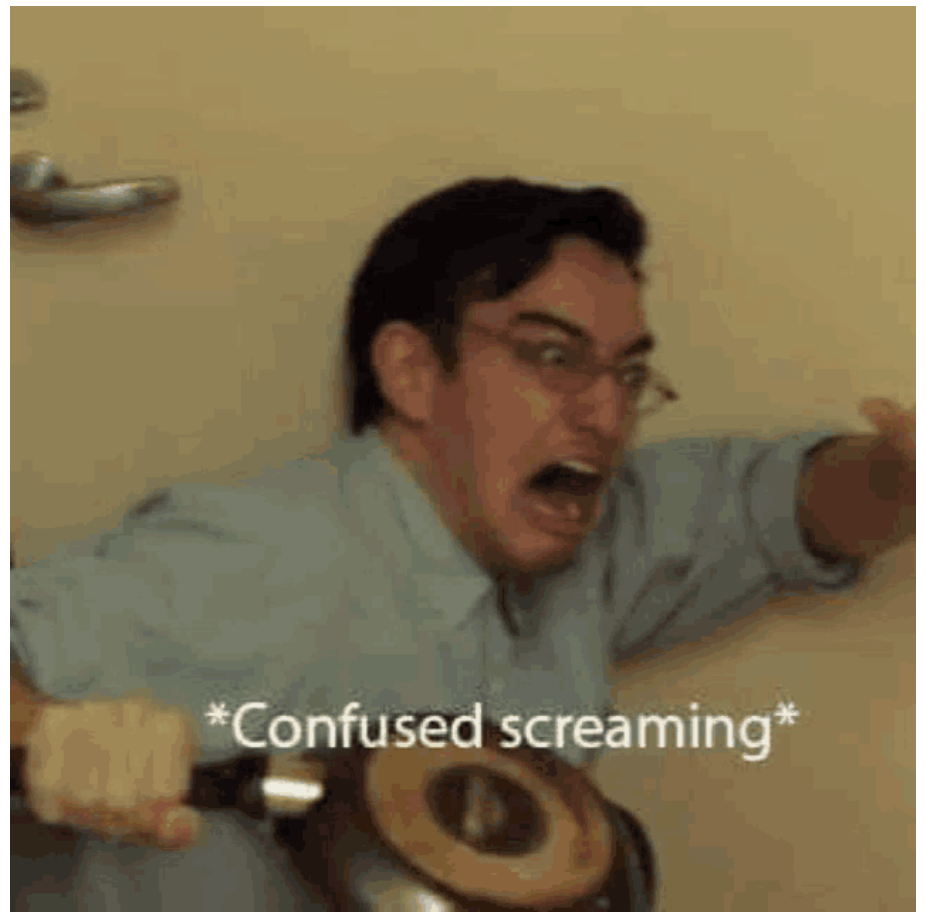
On input $\langle M\rangle$, where $M$ is a Turing machine

1. Run $H$ on input $\langle M, \langle M\rangle\rangle$
2. If $H$ accepts, *reject* and if $H$ rejects, *accept*




Run $D$ on $\langle D\rangle$:



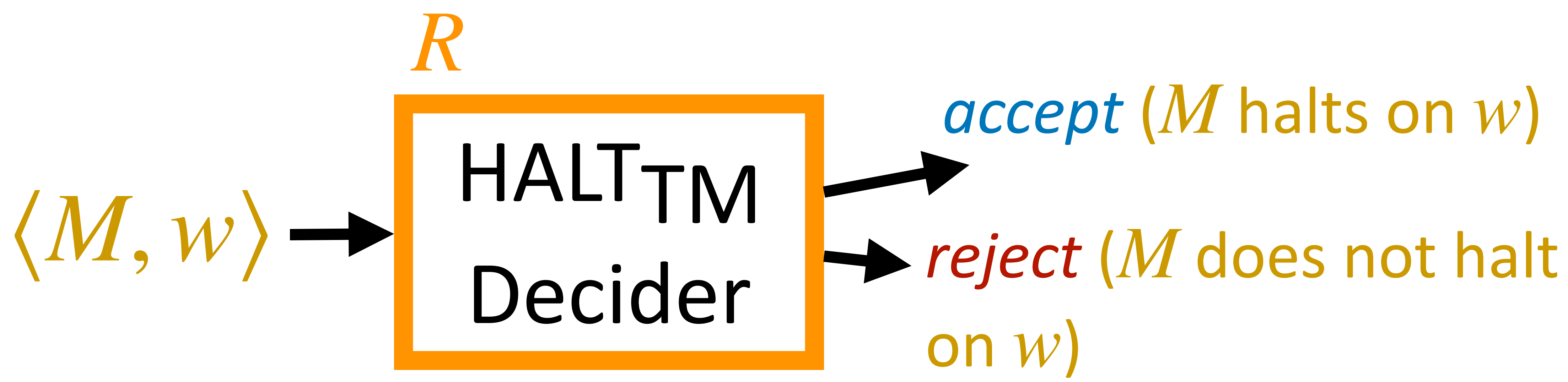
# Undecidable Languages

- $A_{TM} = \{\langle M, w \rangle \mid M$ is a Turing machine and $M$ accepts input string $w\}$
- Halting problem:
  $HALT_{TM} = \{\langle M, w \rangle \mid M$ is a Turing machine and $M$ accepts or rejects input string $w\}$

# $HALT_{TM}$ is undecidable

- $HALT_{TM} = \{\langle M, w\rangle \mid M$ is a Turing machine and $M$ halts on input string $w\}$

<Pf> Assume on the contrary that $HALT_{TM}$ is decidable $\Rightarrow$ there is a Turing machine $R$ that decides $HALT_{TM}$

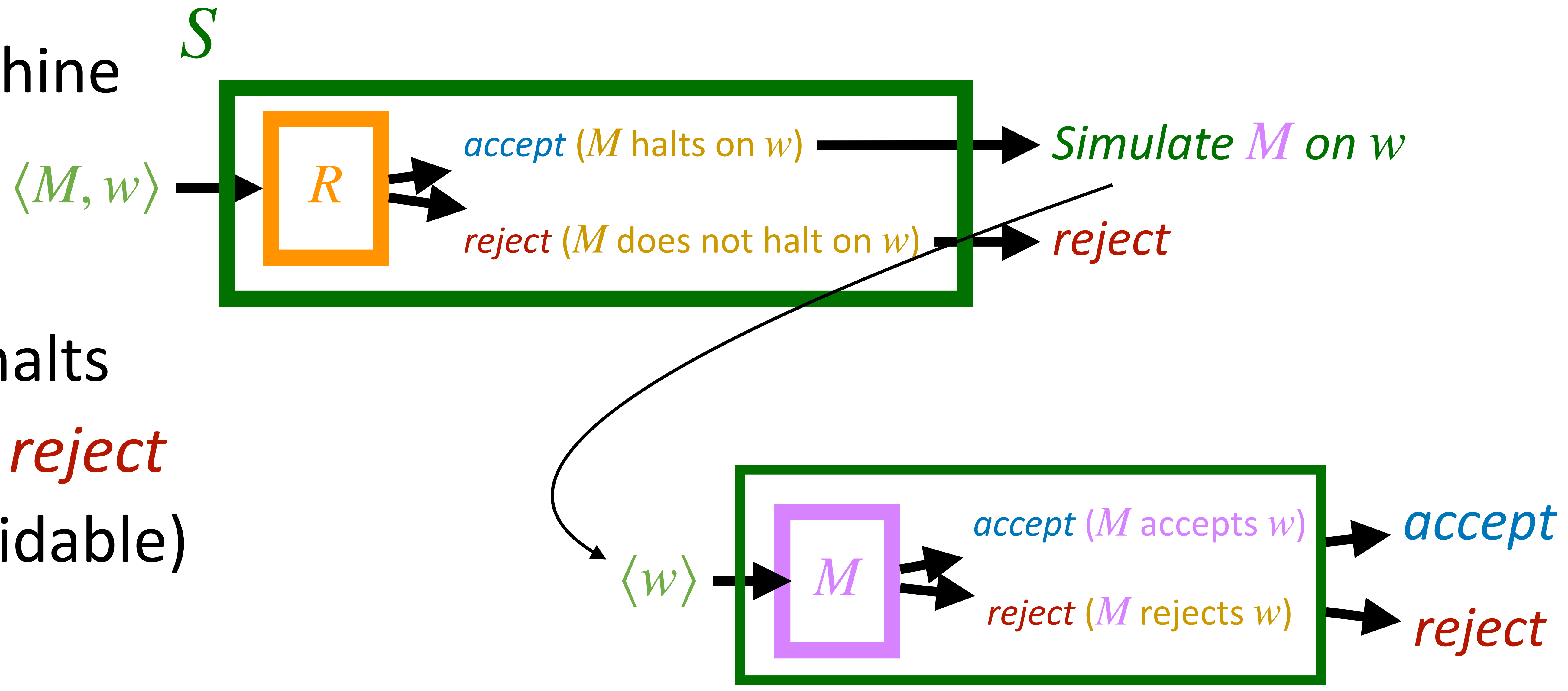



Design a Turing machine $S$ that decides $A_{TM}$:

On input $\langle M, w\rangle$, where $M$ is a Turing machine

1. Run $R$ on input $\langle M, w\rangle$
2. If $R$ rejects, *reject*
3. If $R$ accepts, *simulate* $M$ on $w$ until it halts
4. If $M$ accepts $w$, *accept*. If $M$ rejects $w$, *reject*

$\rightarrow \leftarrow$ ($A_{TM}$ is undecidable)



# Halting problem is undecidable

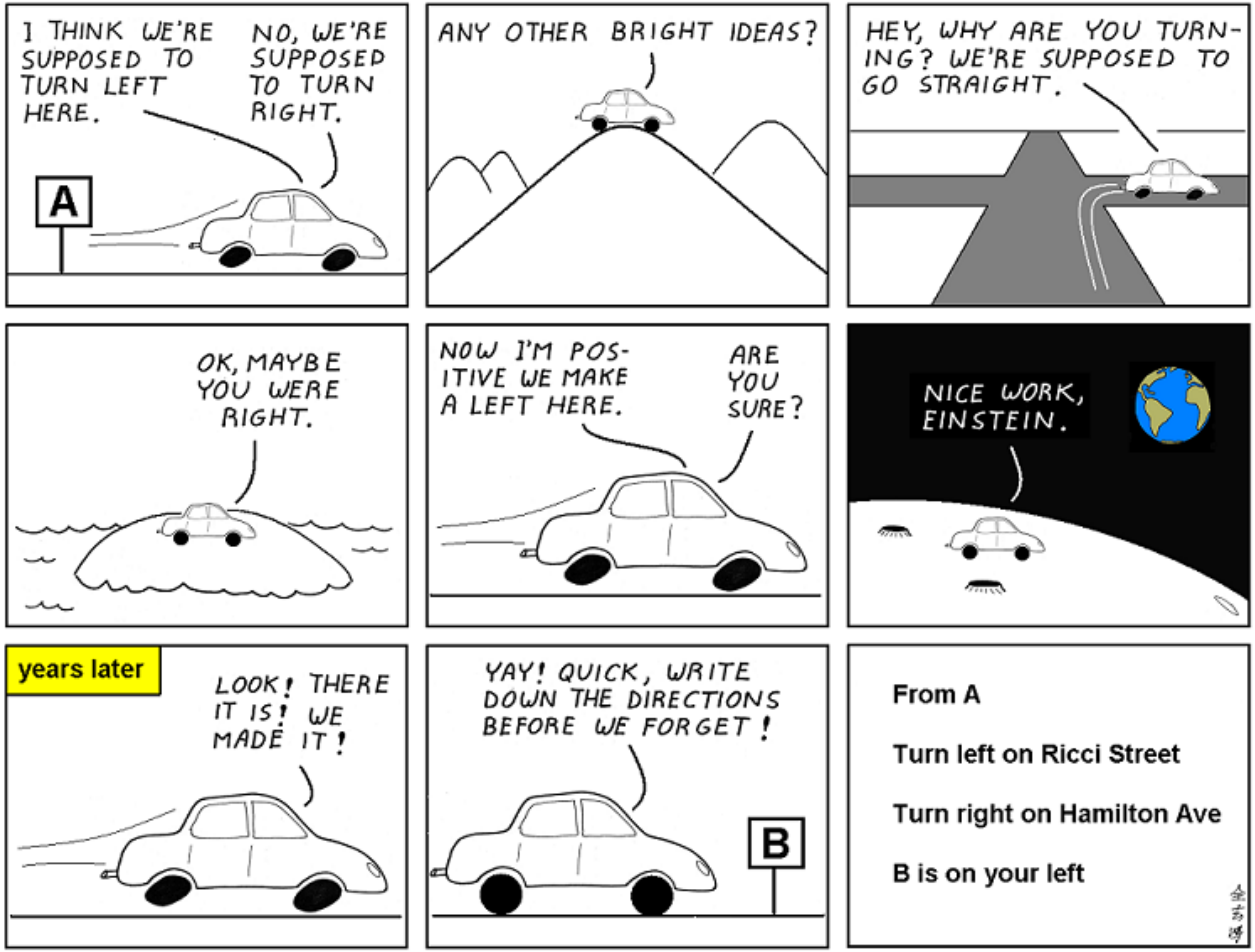
- Proof by reduction!
  - If $A \leq B$ and $B$ is decidable, then $A$ is decidable
  - The reduction $\leq$ doesn't need to be polynomial time

# Undecidable Languages

- $A_{TM} = \{\langle M, w \rangle \mid M$ is a Turing machine and $M$ accepts input string $w\}$
- Halting problem:
  $HALT_{TM} = \{\langle M, w \rangle \mid M$ is a Turing machine and $M$ accepts or rejects input string $w\}$
- Hilbert's 10th problem:
  $H = \{\langle p \rangle \mid p$ is a polynomial with an integral root$\}$
- Post correspondence problem (PCP):
  Given a collection $D$ of dominos, each containing two strings, one on each side. A match is a list of these dominos (repetition permitted) such that the string on the top is the same as the string on the bottom.

# It’s **obvious**

— by Abstruse Goose




This is how most mathematical proofs are written.