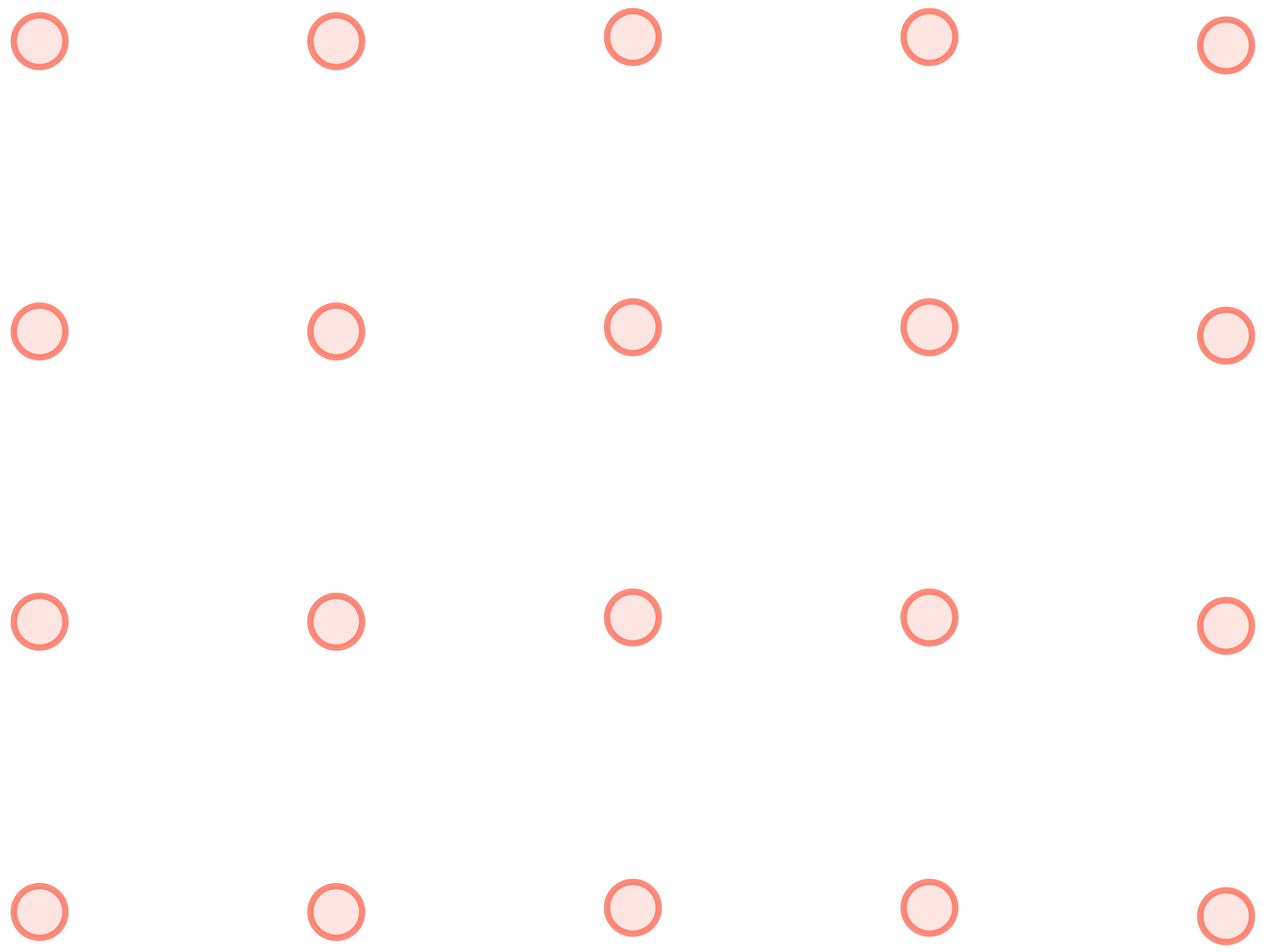
Integer Programming and IP Proof Systems

Noah Fleming
University of California, San Diego

Linear Programming

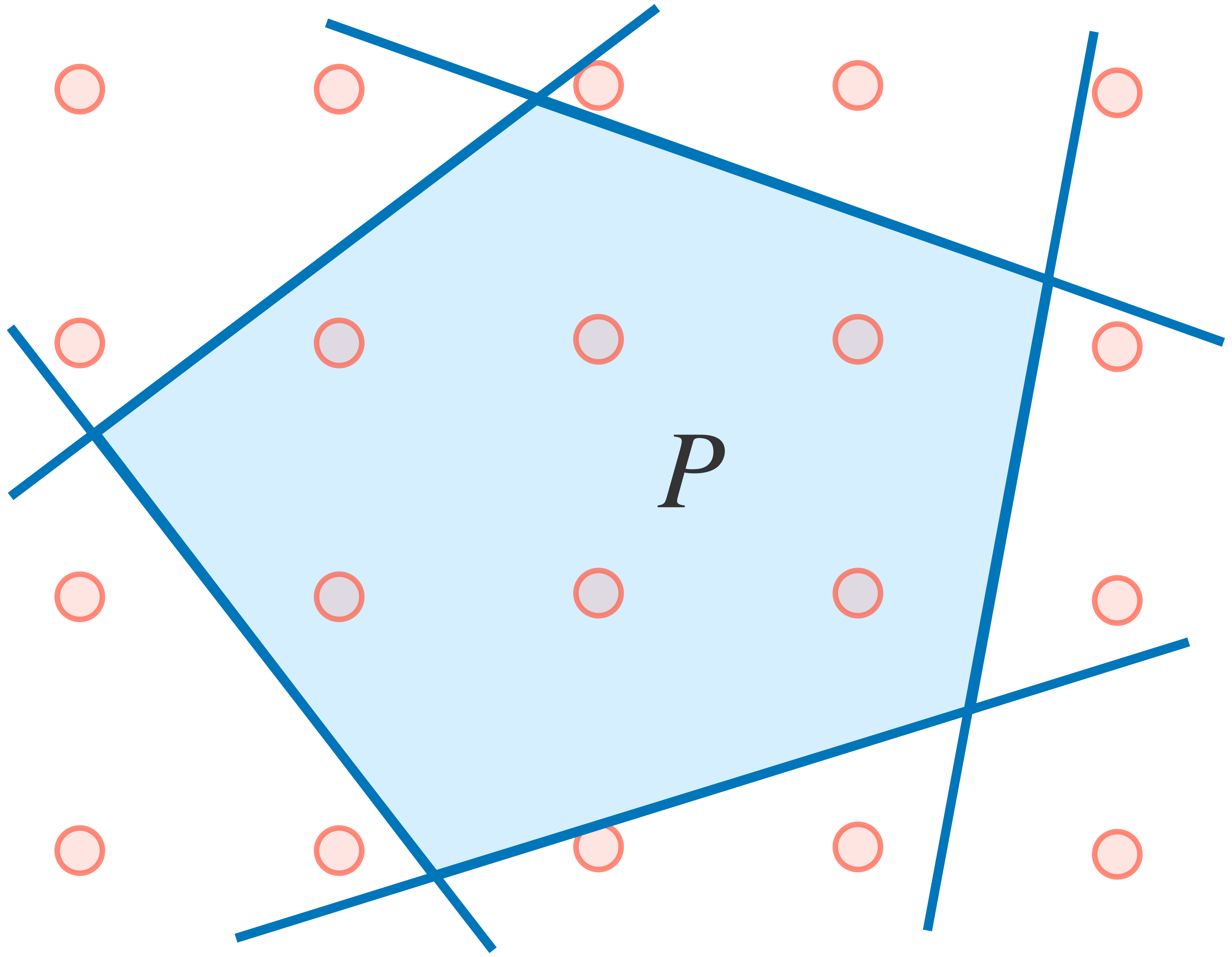
Input: set of linear inequalities $Ax \geq b$



Linear Programming

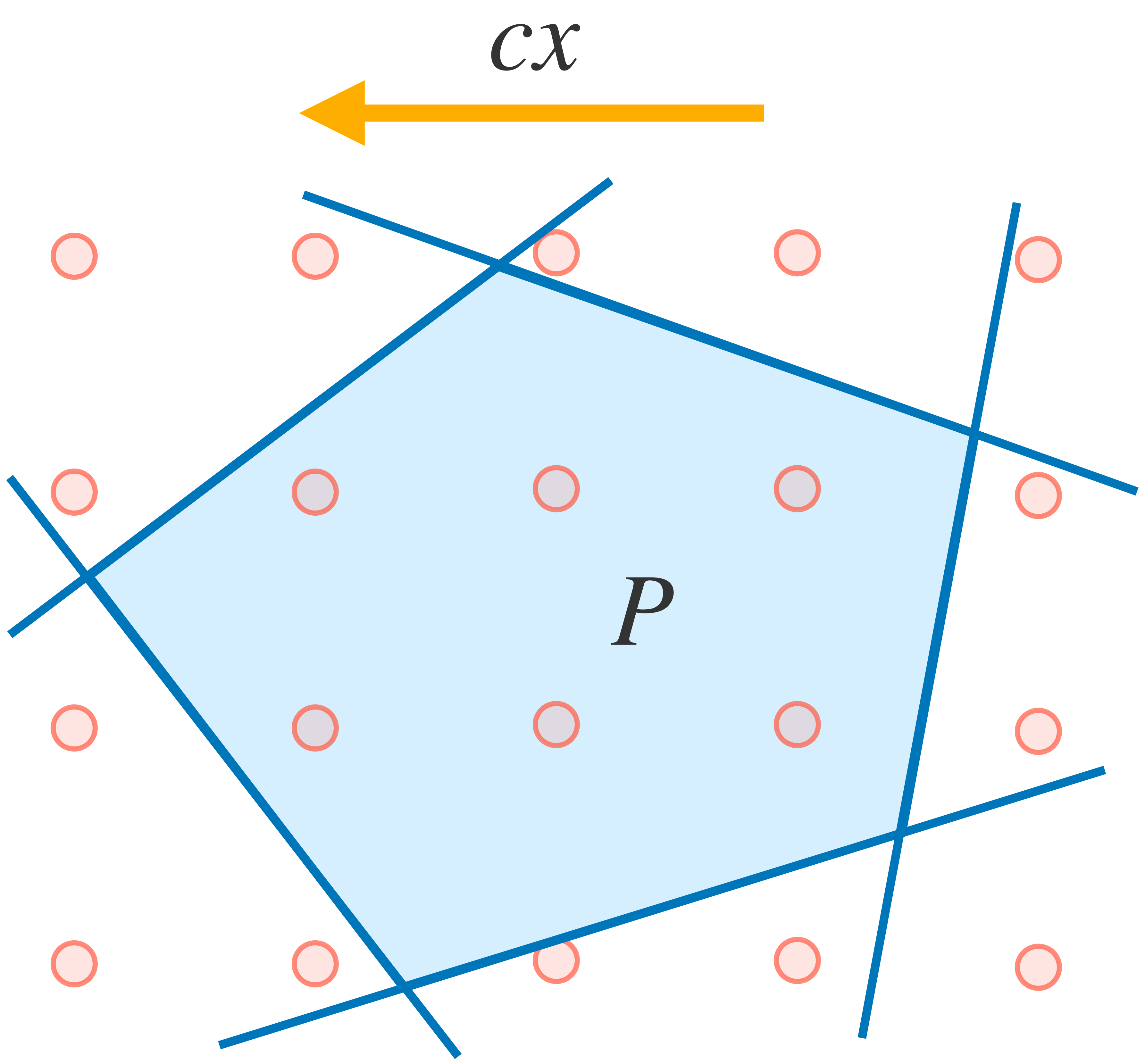
Input: set of linear inequalities $Ax \geq b$

Which defines a polytope $P = \{x : Ax \geq b\}$



Linear Programming

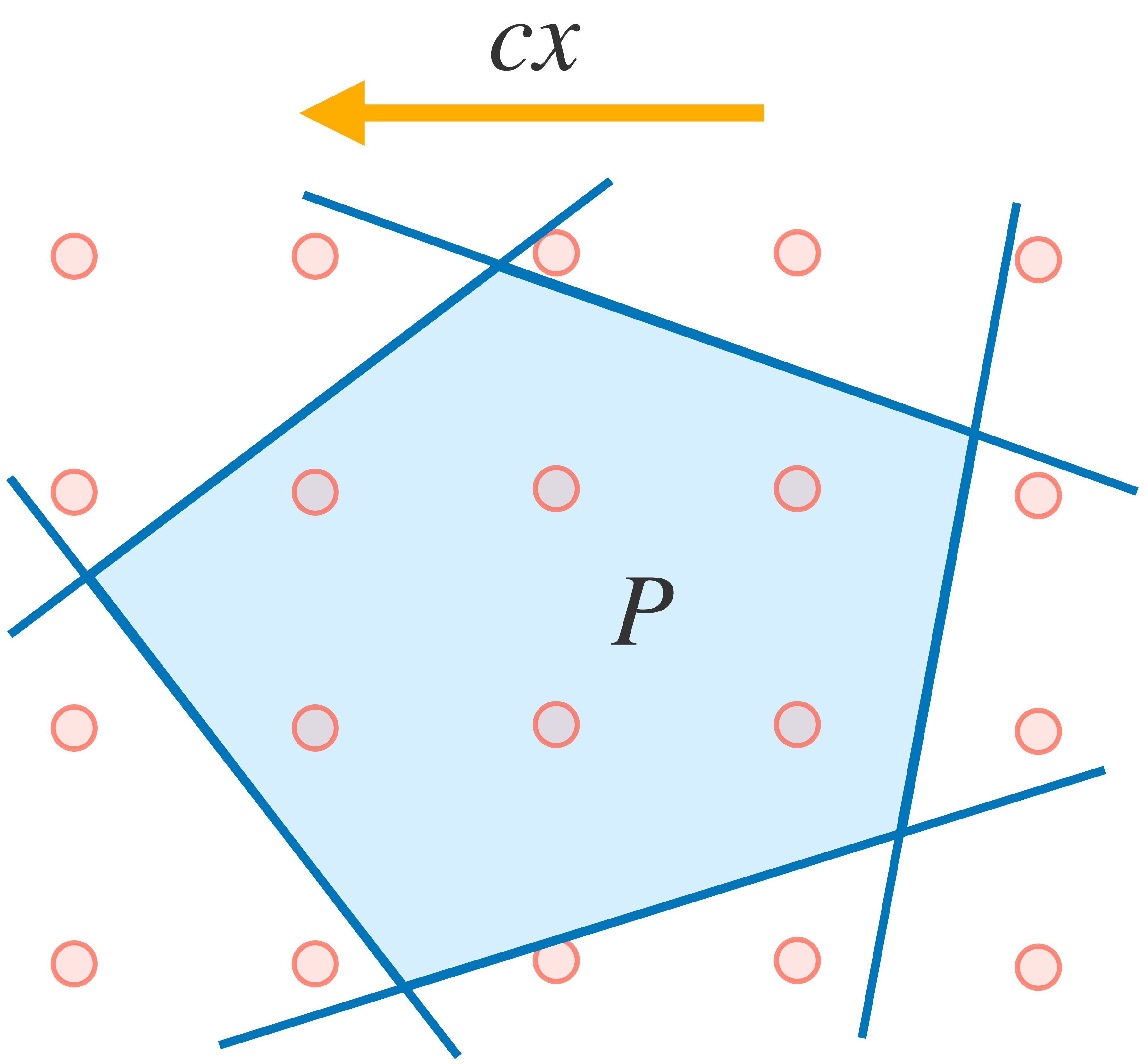
Input: set of linear inequalities $Ax \geq b$
linear objective function cx



Linear Programming

<u>Input:</u> set of linear inequalities $Ax \geq b$
linear objective function cx

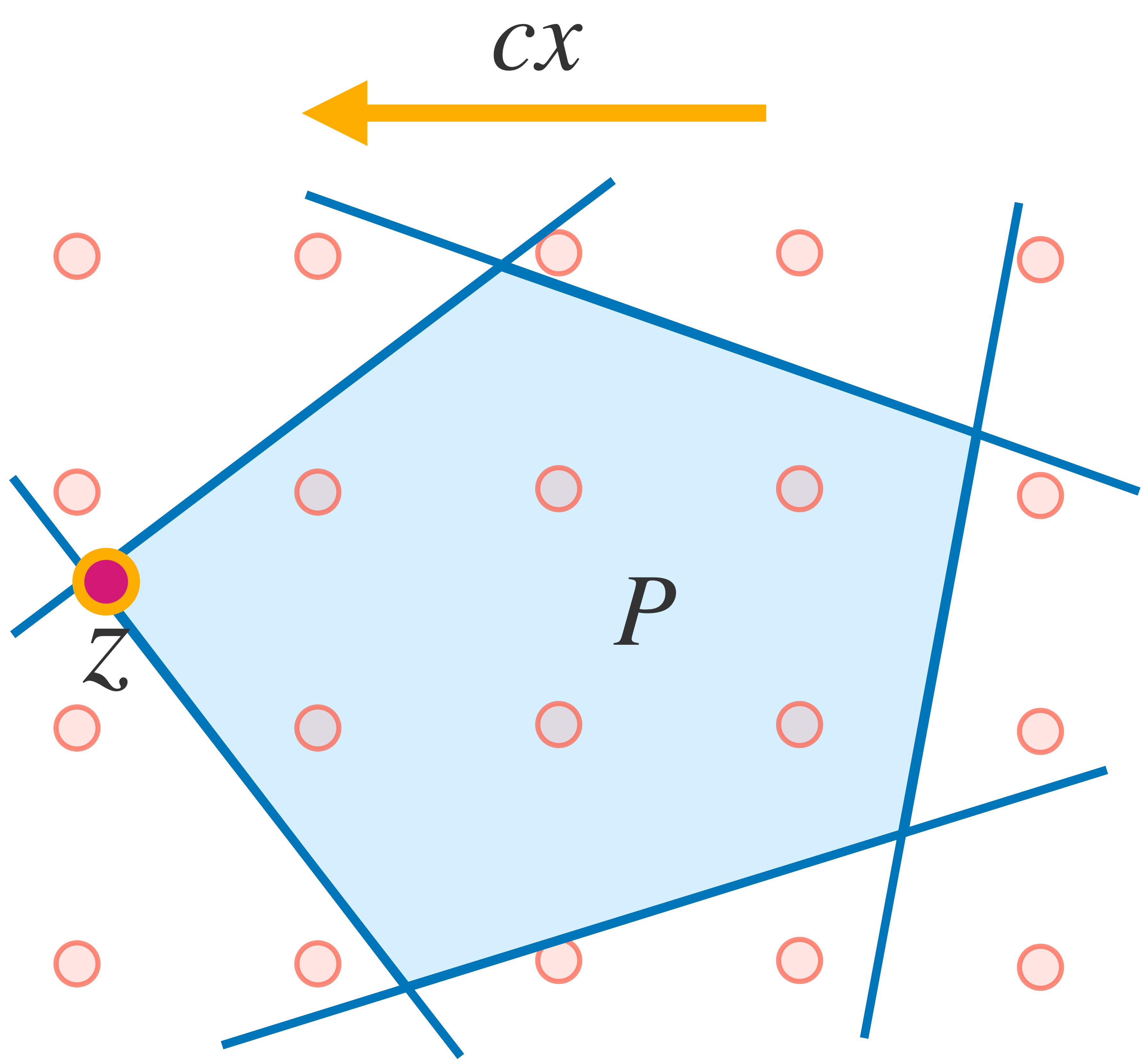
<u>Output:</u> Solution $z \in \mathbb{R}^n$
Satisfying $Az \geq b$
Maximizing cz



Linear Programming

Input: set of linear inequalities $Ax \geq b$
linear objective function cx

Output: Solution $z \in \mathbb{R}^n$
Satisfying $Az \geq b$
Maximizing cz

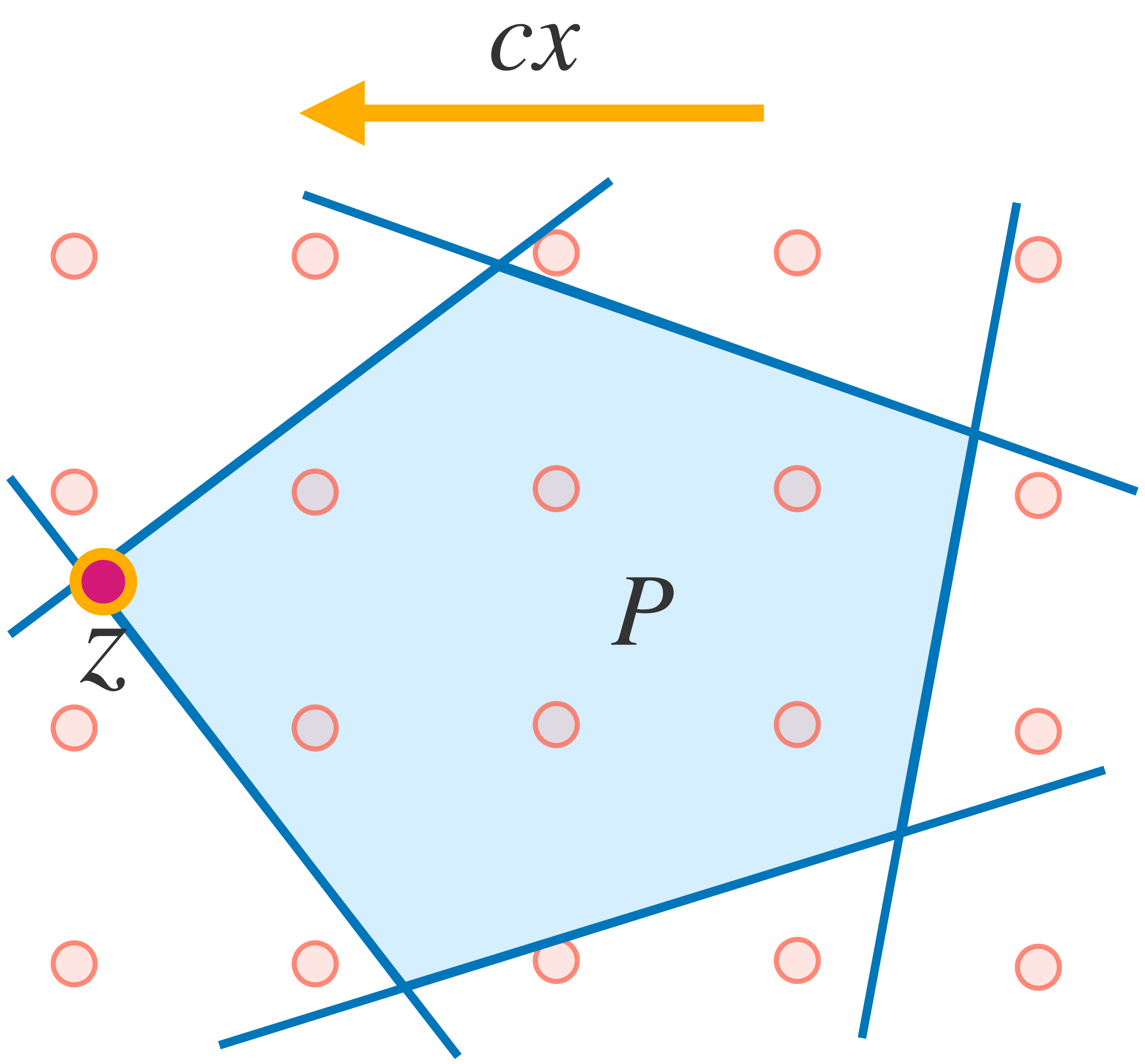


Linear Programming

Input: set of linear inequalities $Ax \geq b$
linear objective function cx

Output: Solution $z \in \mathbb{R}^n$
Satisfying $Az \geq b$
Maximizing cz

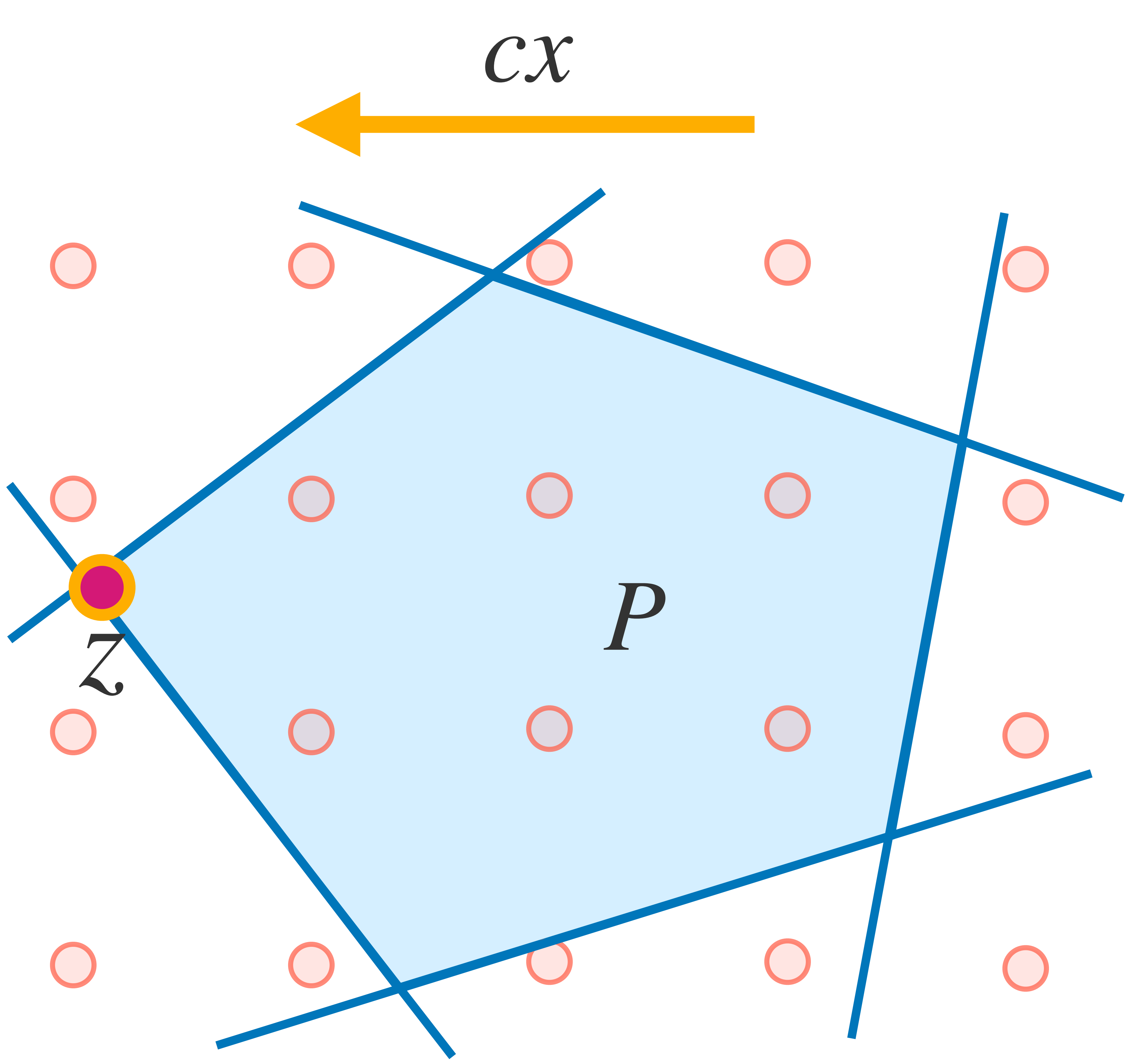
- Broadly applicable framework for optimization.
- Efficiently solvable!



Linear Programming

Input: set of linear inequalities $Ax \geq b$
linear objective function cx

Output: Solution $z \in \mathbb{R}^n$
Satisfying $Az \geq b$
Maximizing cz



- Broadly applicable framework for optimization.
- Efficiently solvable!

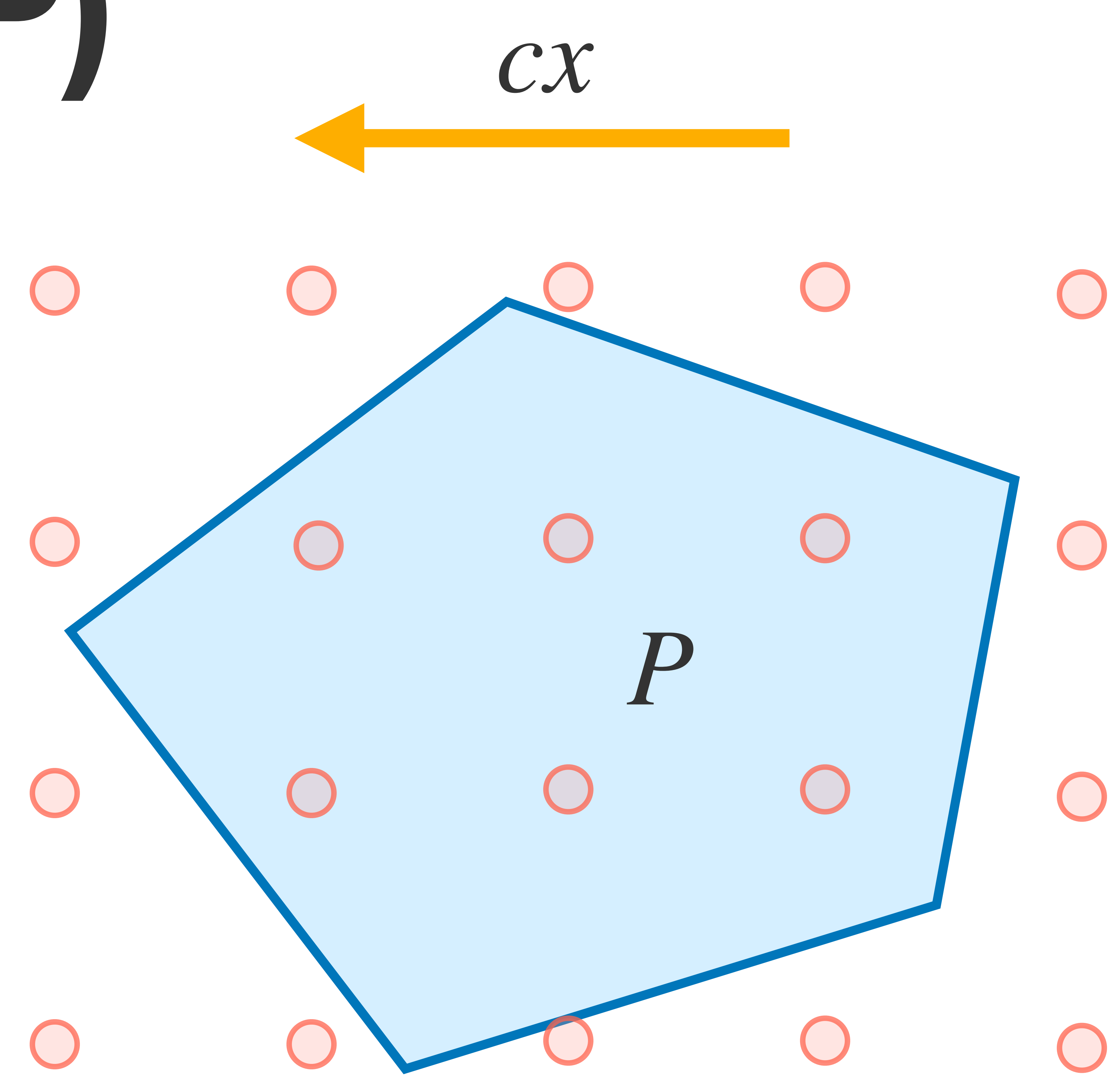
However, many **important problems** phrased most naturally as finding **integer solutions** to a linear program

- e.g. maxCut, maxSAT, maxClique, etc.

Integer Programming (IP)

<u>Input:</u> set of linear inequalities $Ax \geq b$
linear objective function cx

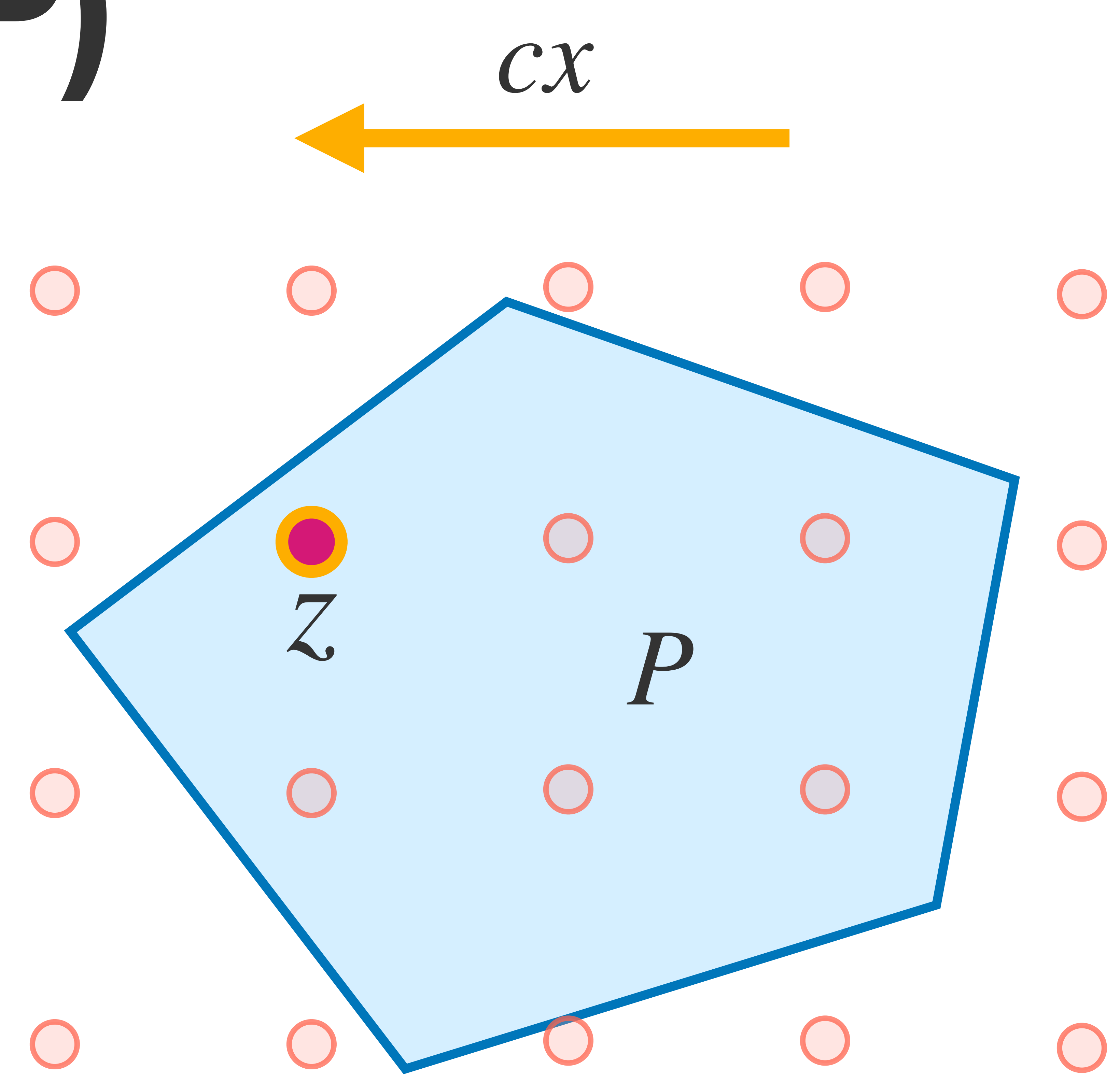
<u>Output:</u> Integer solution $z \in \mathbb{Z}^n$
Satisfying $Az \geq b$
Maximizing cz



Integer Programming (IP)

Input: set of linear inequalities $Ax \geq b$
linear objective function cx

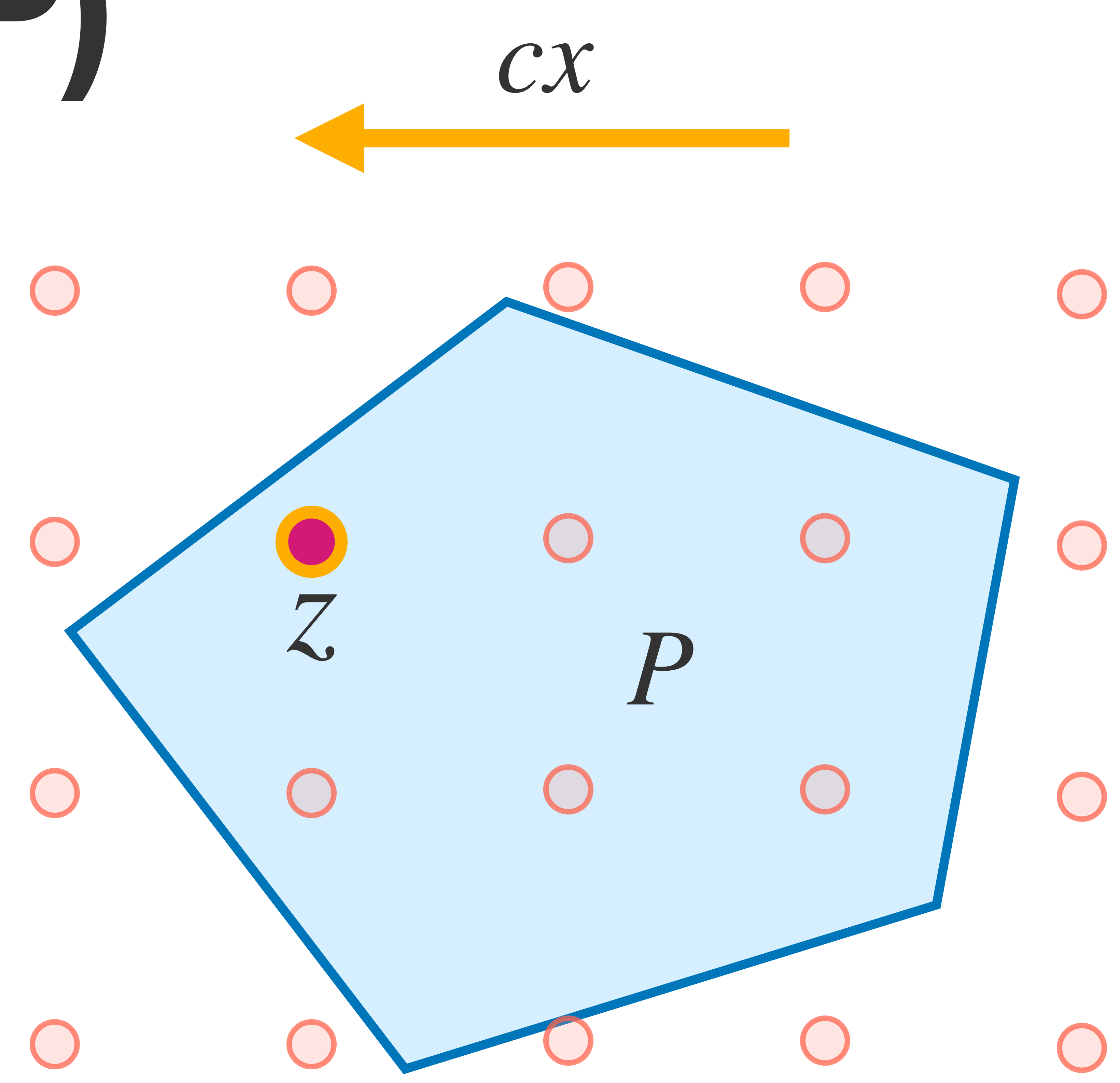
Output: Integer solution $z \in \mathbb{Z}^n$
Satisfying $Az \geq b$
Maximizing cz



Integer Programming (IP)

Input: set of linear inequalities $Ax \geq b$
linear objective function cx

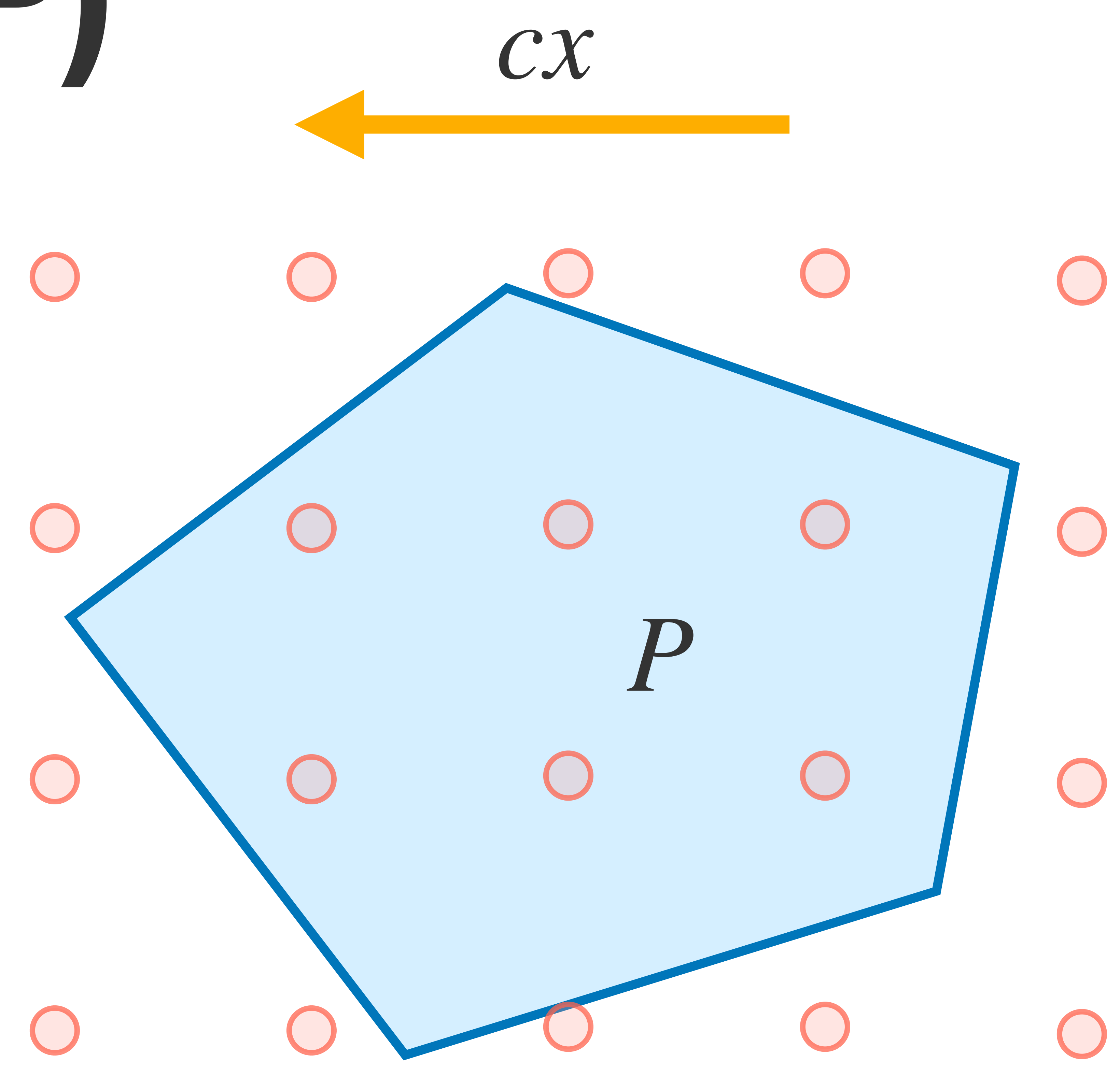
Output: Integer solution $z \in \mathbb{Z}^n$
Satisfying $Az \geq b$
Maximizing cz



Extremely general framework!
… but **NP**-complete.

Integer Programming (IP)

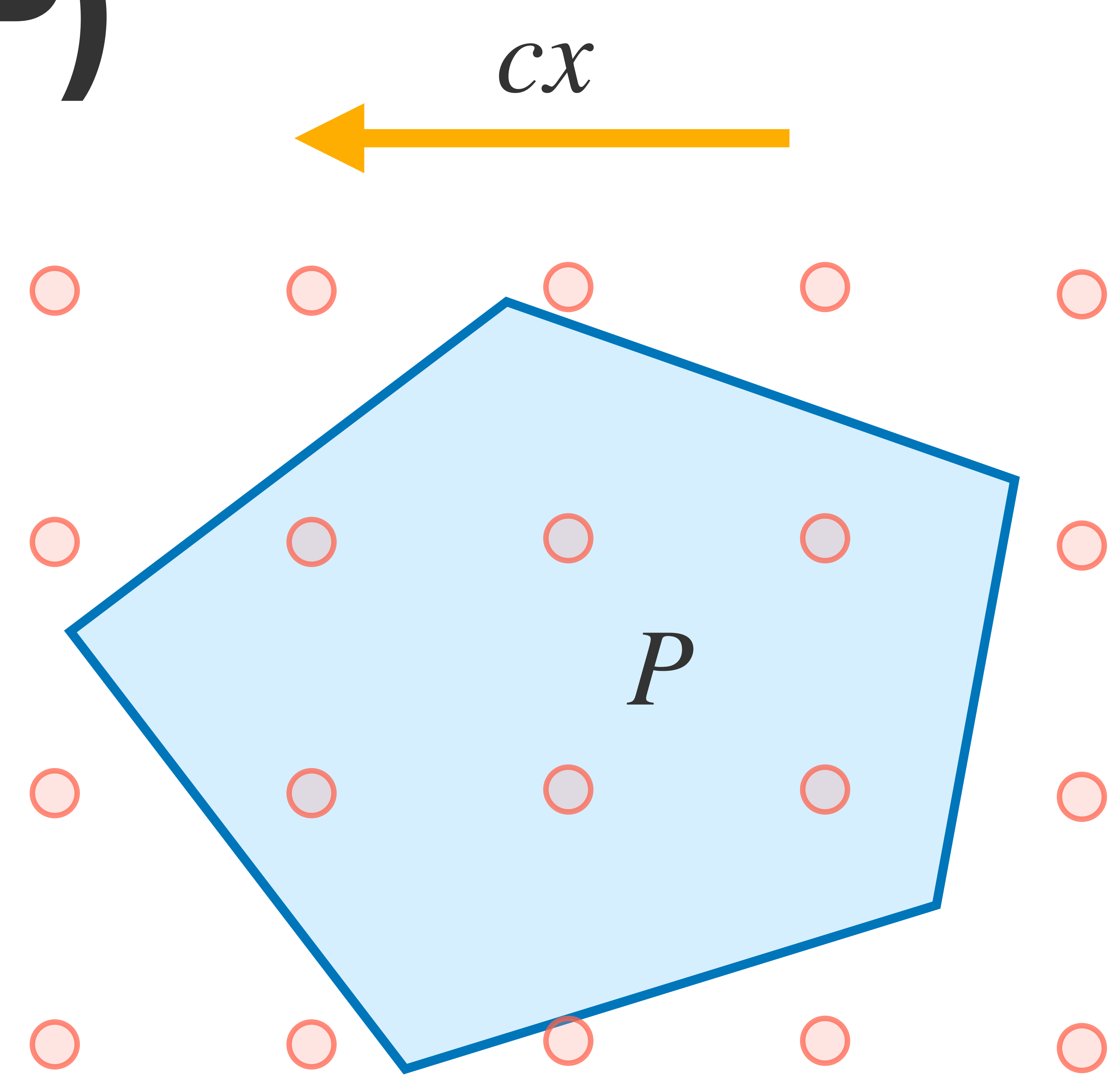
Nonetheless, Integer programs are solved **extremely** fast in **practice!**



Integer Programming (IP)

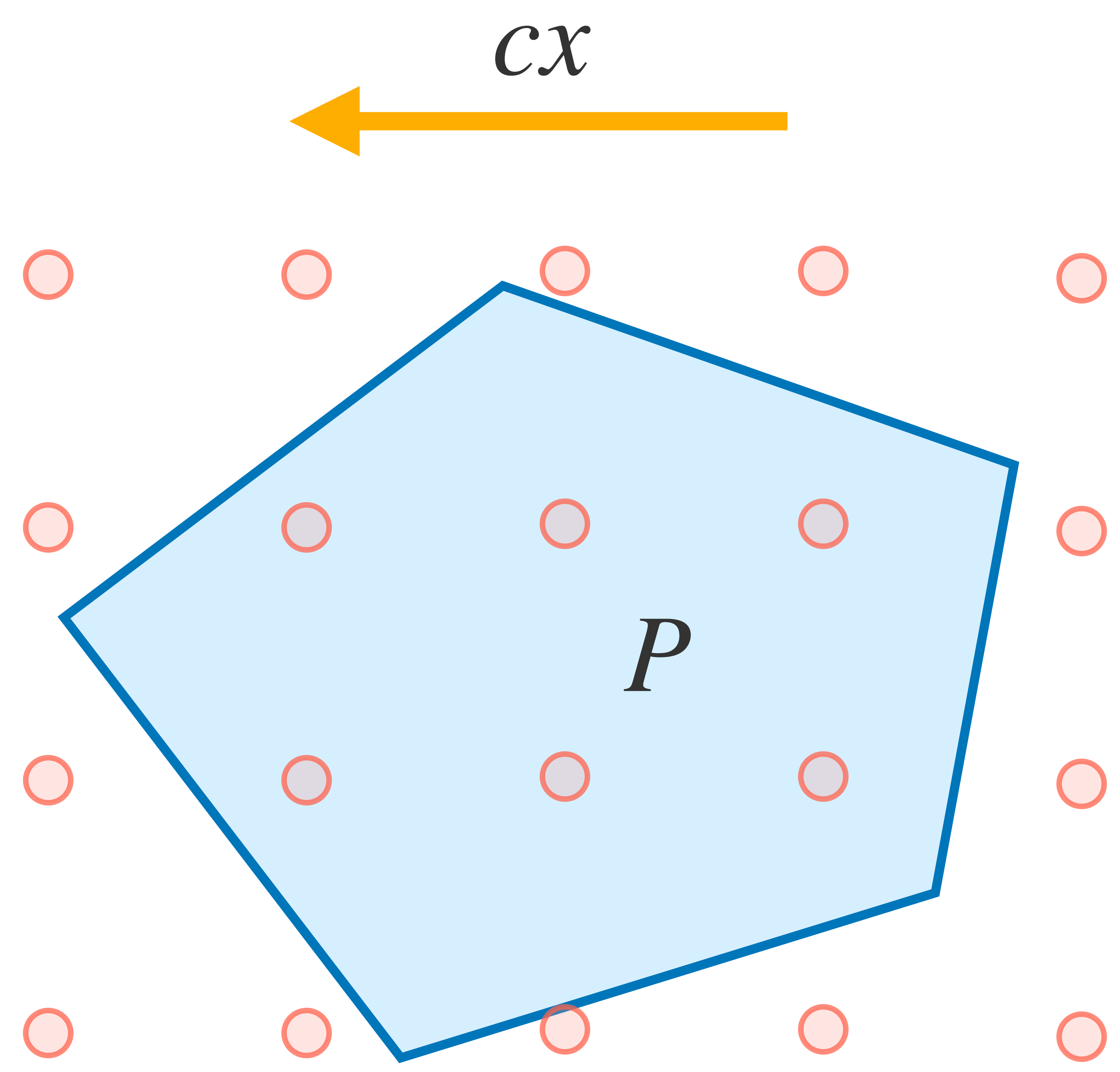
Nonetheless, Integer programs are solved **extremely** fast in **practice!**

How? Branch-and-Cut!



Branch-and-Cut

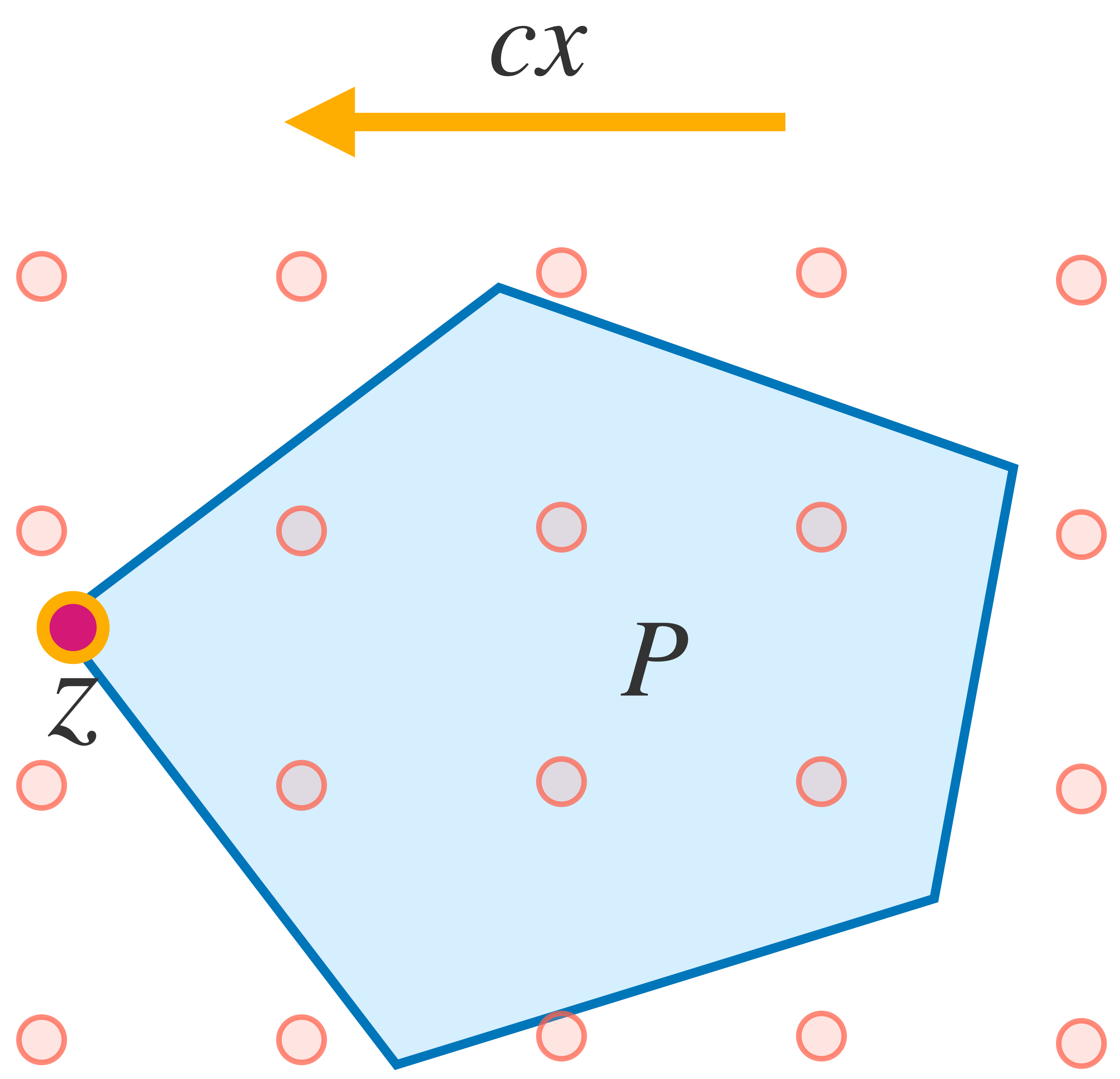
Idea: Try to use **linear** programming to solve **integer** programming!



Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Run a linear program

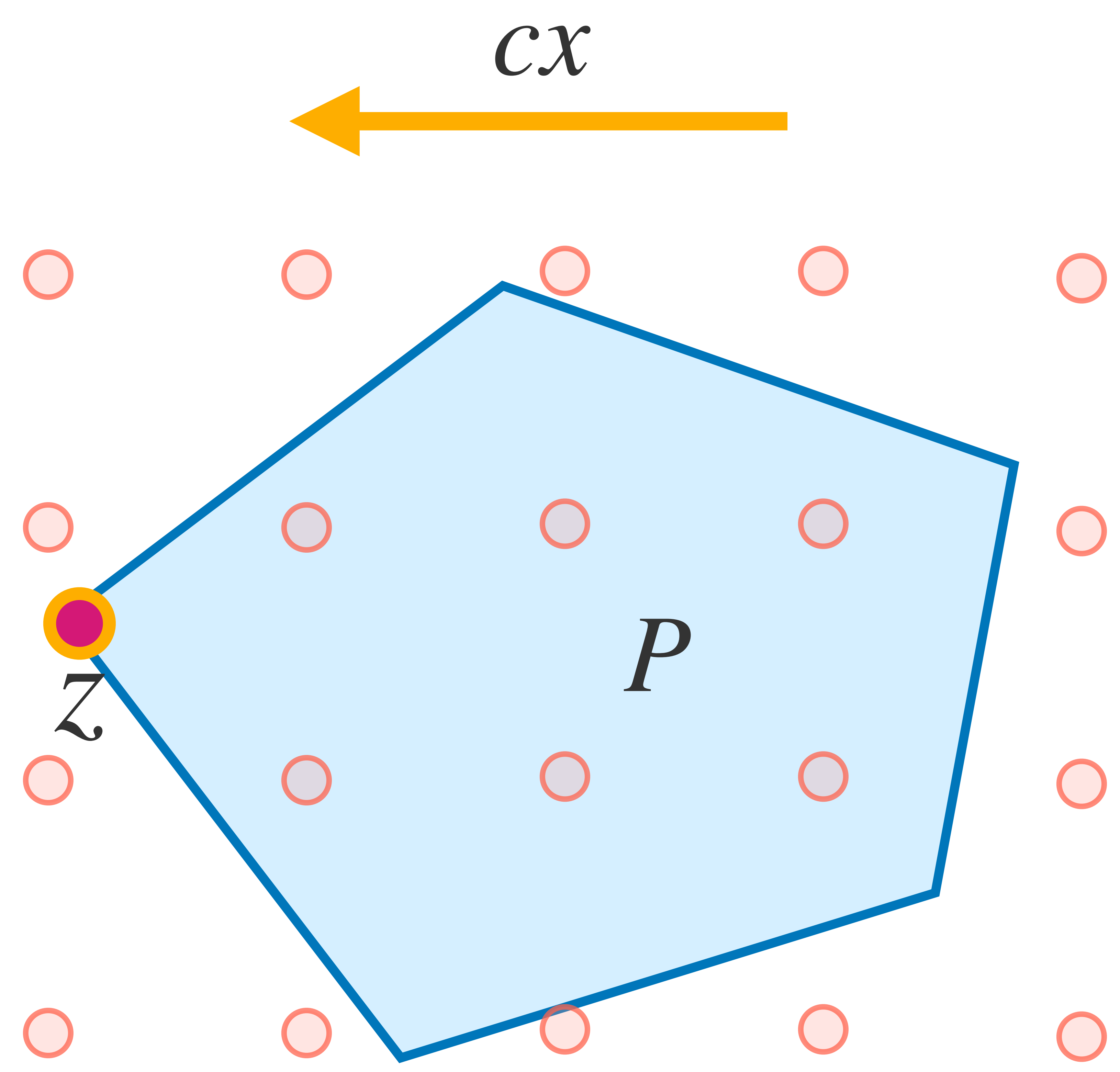


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Run a linear program

- If solution is integral, **done**!

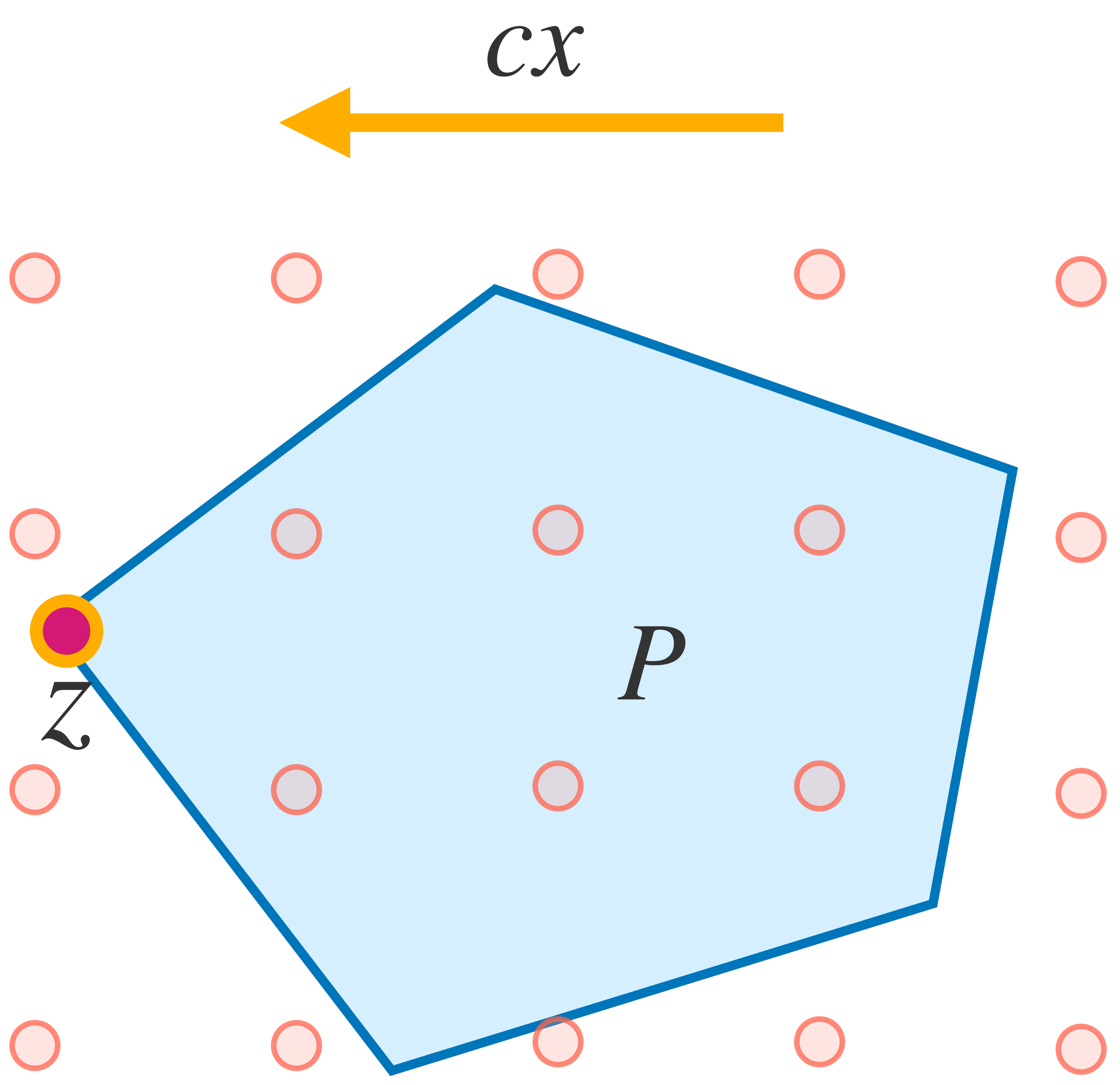


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Run a linear program

- If solution is integral, **done**!
- Otherwise, **refine** the polytope until an integer solution can be found by linear programming

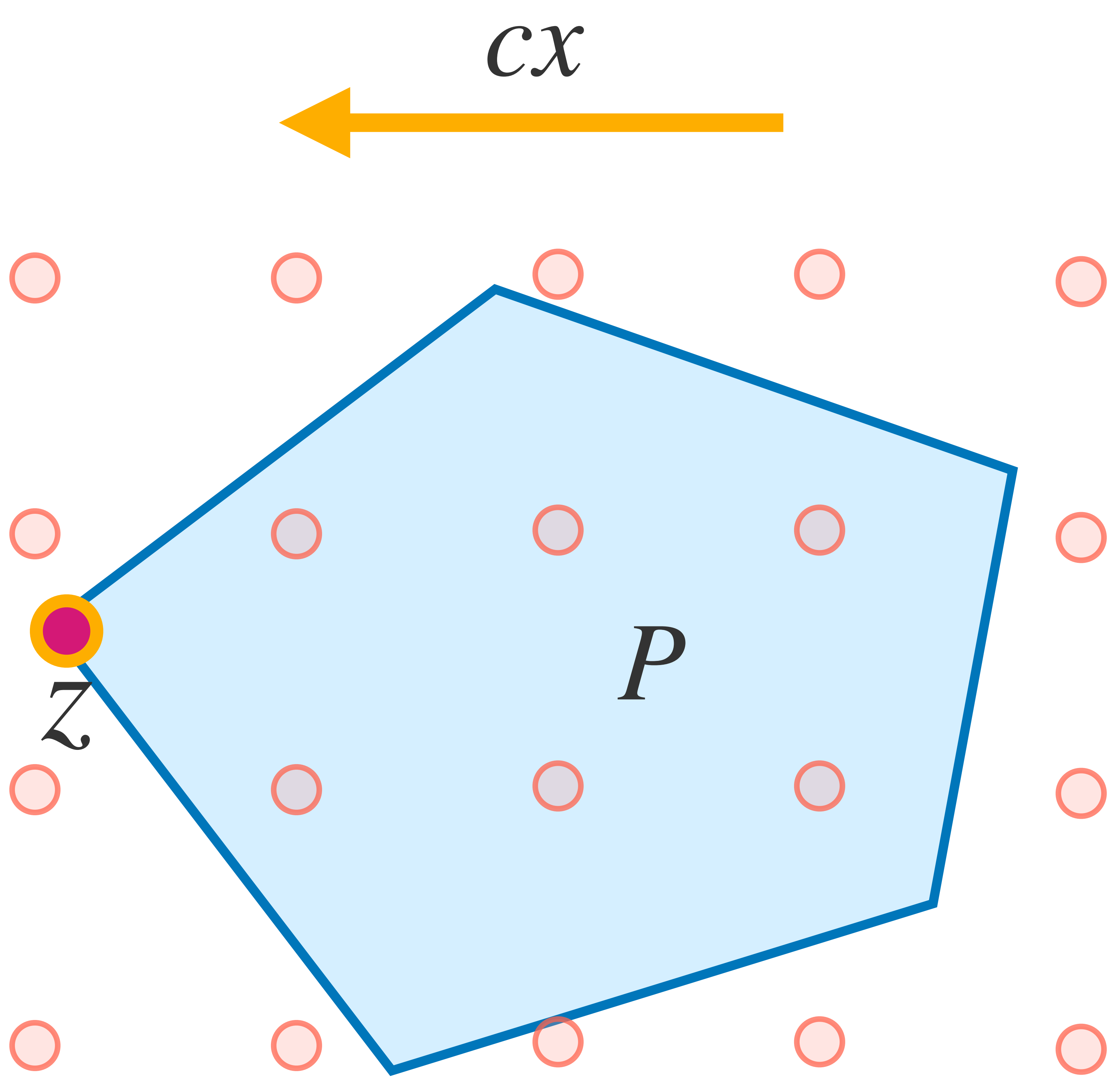


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Run a linear program

- If solution is integral, **done**!
- Otherwise, **refine** the polytope until an integer solution can be found by linear programming
  $\rightarrow$ **remove non-integer** solutions by adding **additional constraints** to P

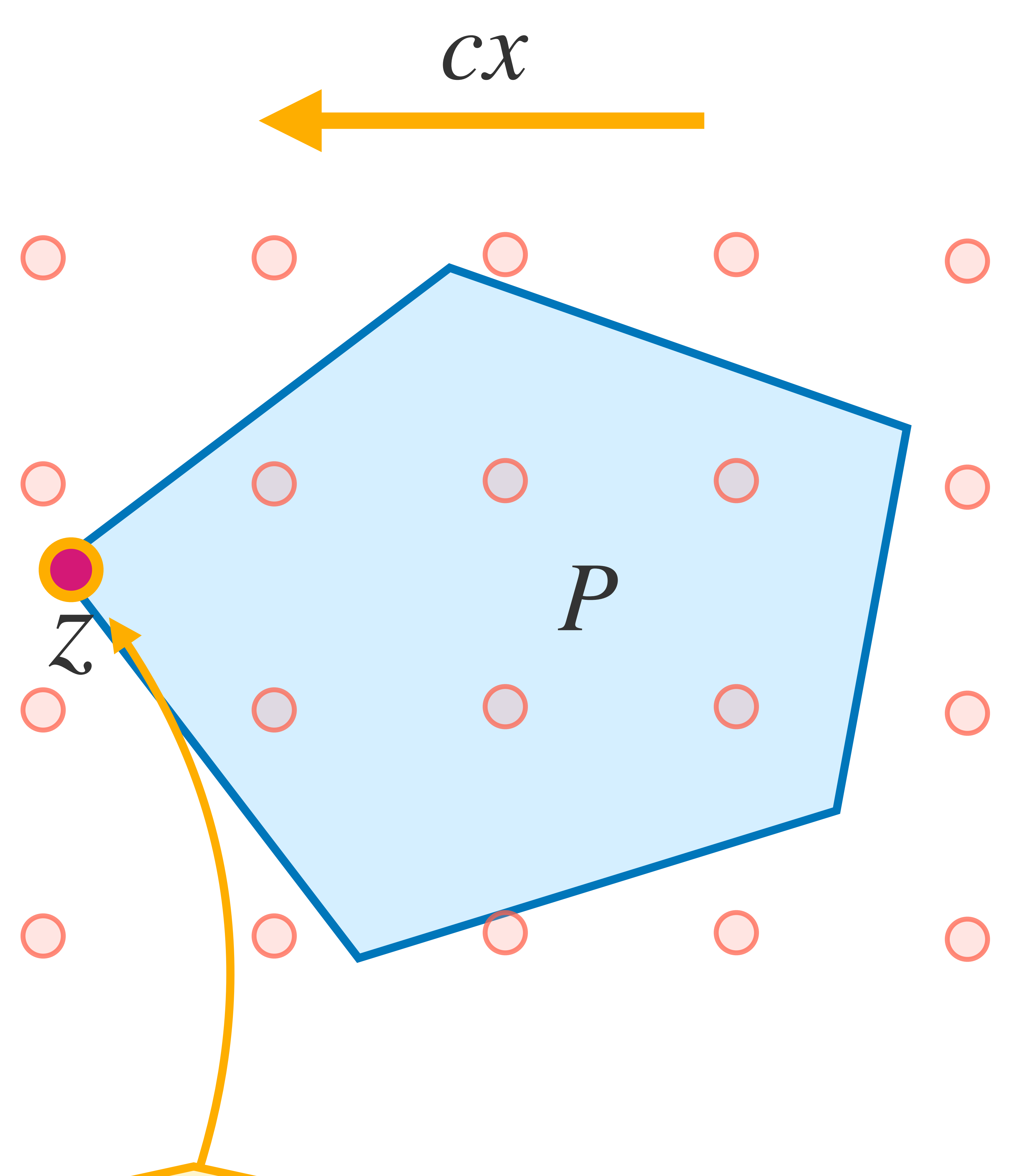


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Run a linear program

- If solution is integral, **done**!
- Otherwise, **refine** the polytope until an integer solution can be found by linear programming
  $\rightarrow$ **remove non-integer** solutions by adding **additional constraints** to P



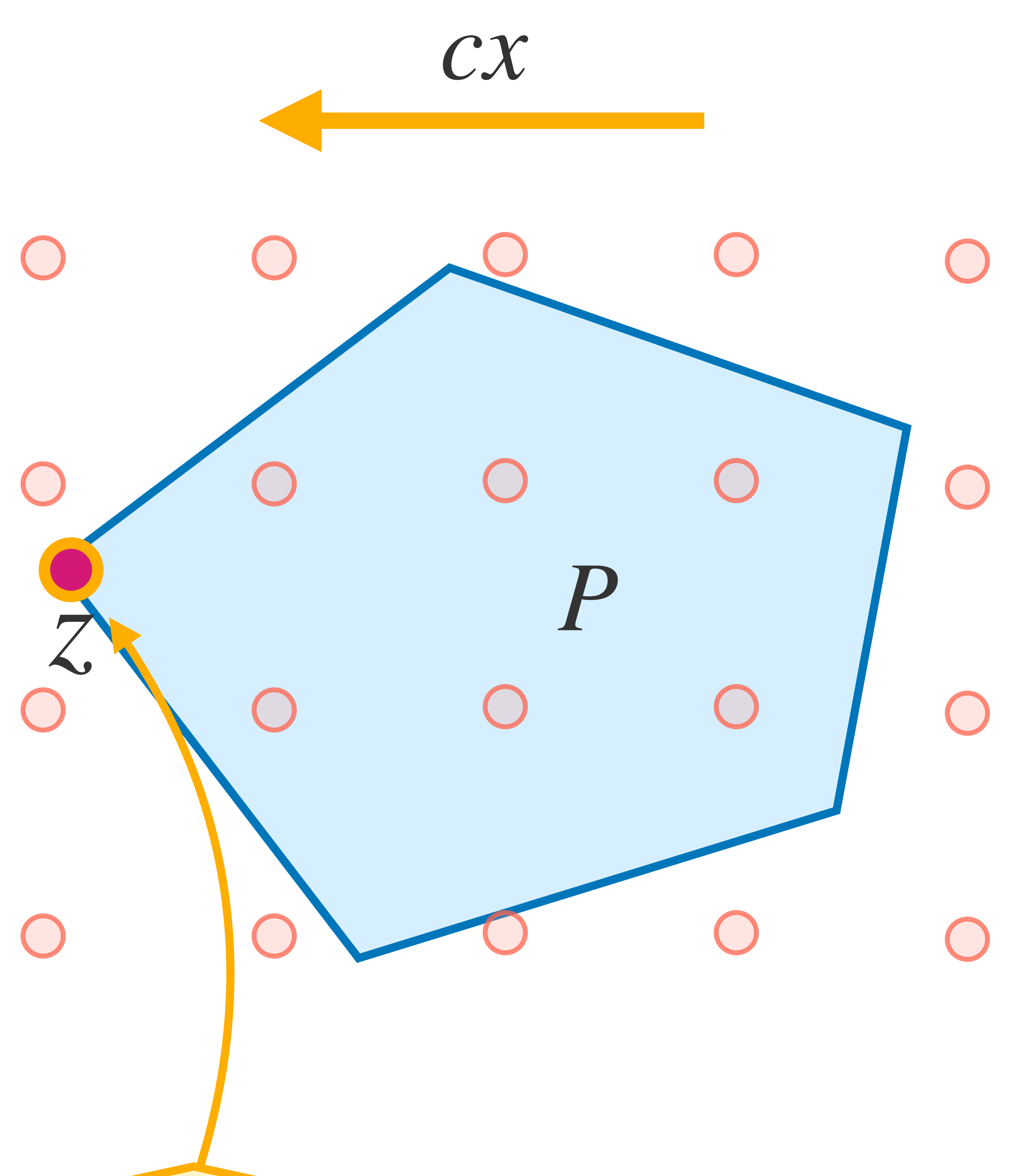
Remove this point so **better** solutions can be found!

Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Run a linear program

- If solution is integral, **done**!
- Otherwise, **refine** the polytope until an integer solution can be found by linear programming
  $\rightarrow$ **remove non-integer** solutions by adding **additional constraints** to P
- Recuse

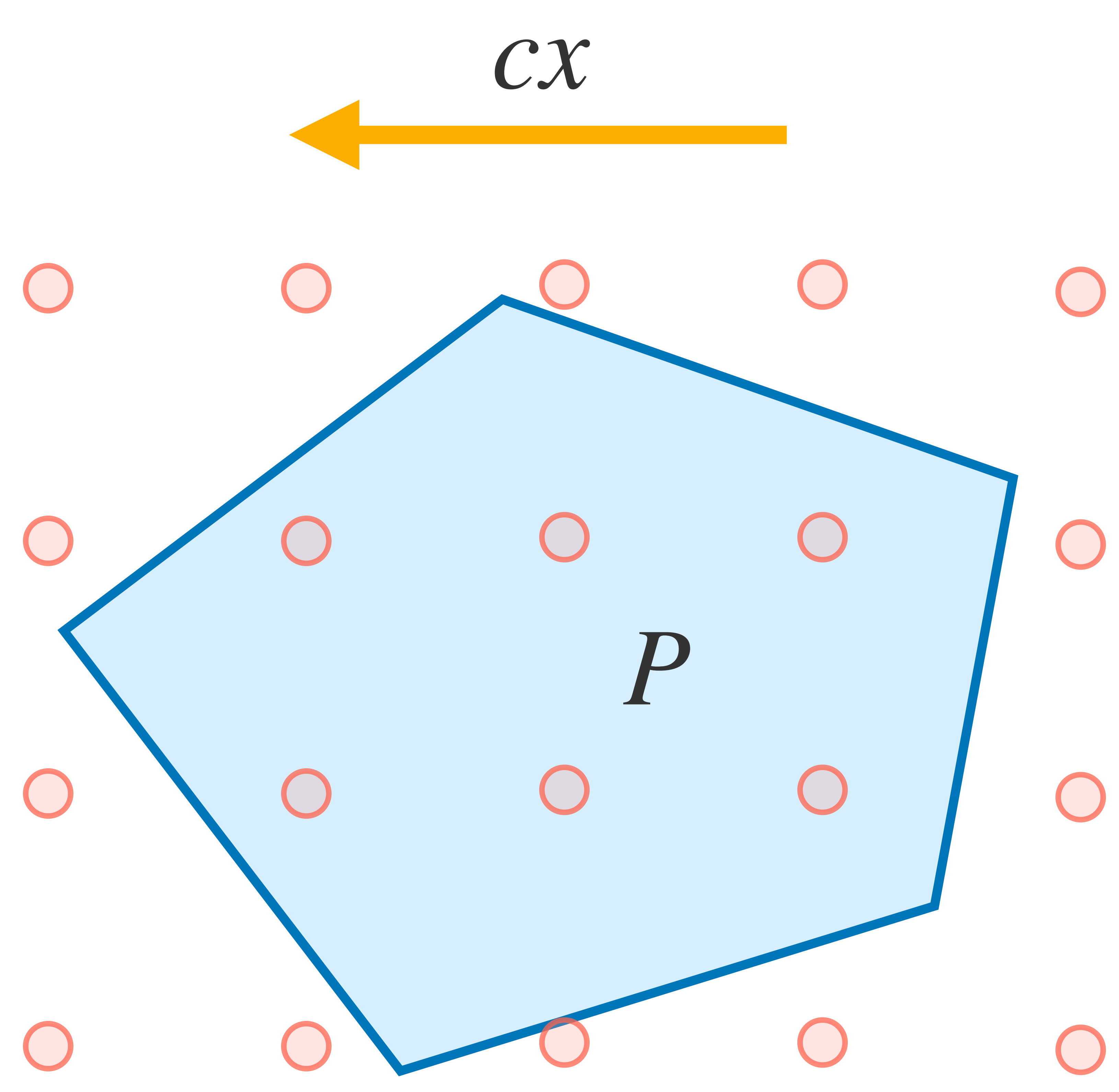


Remove this point so **better** solutions can be found!

Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes

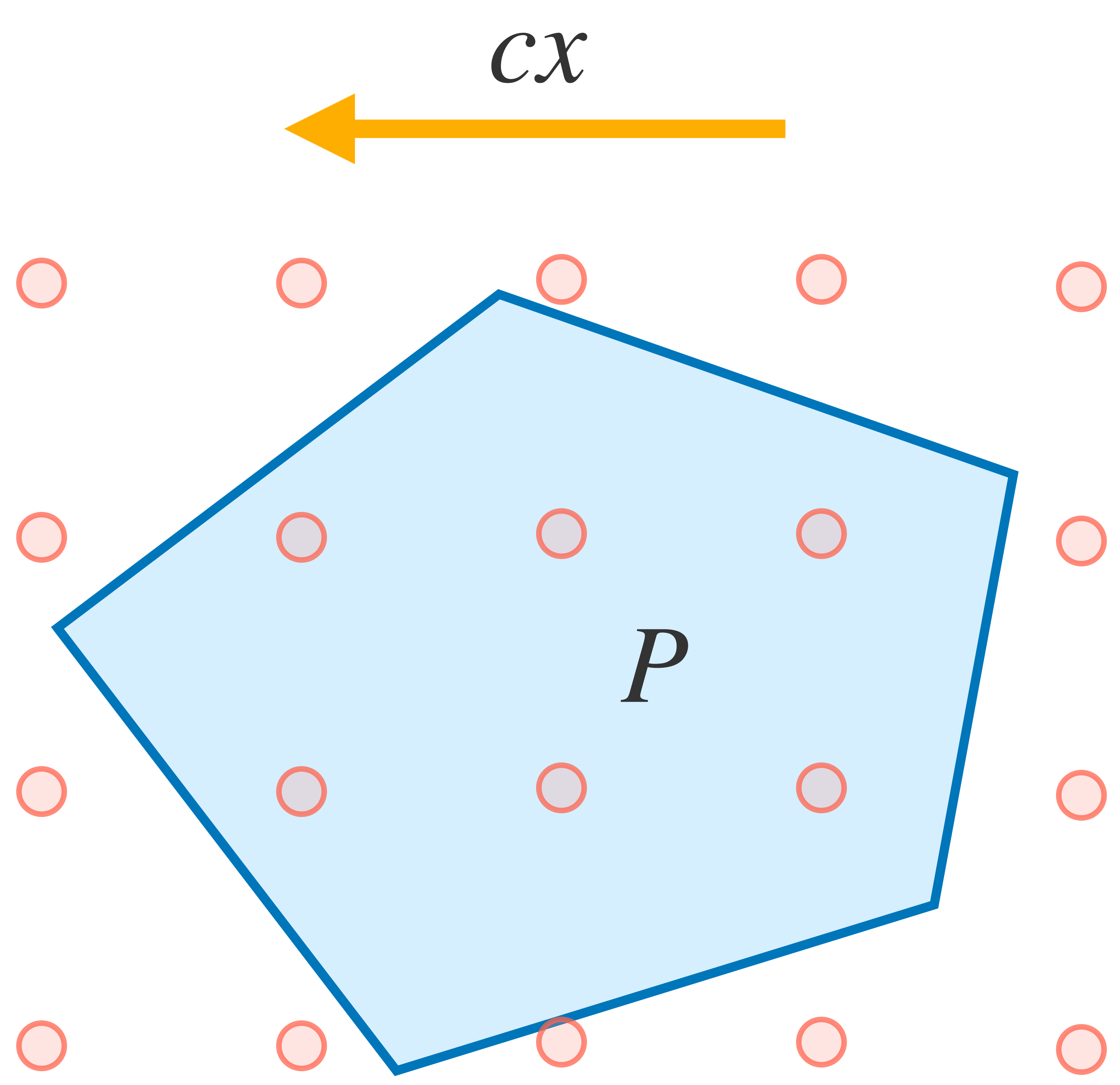


Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes

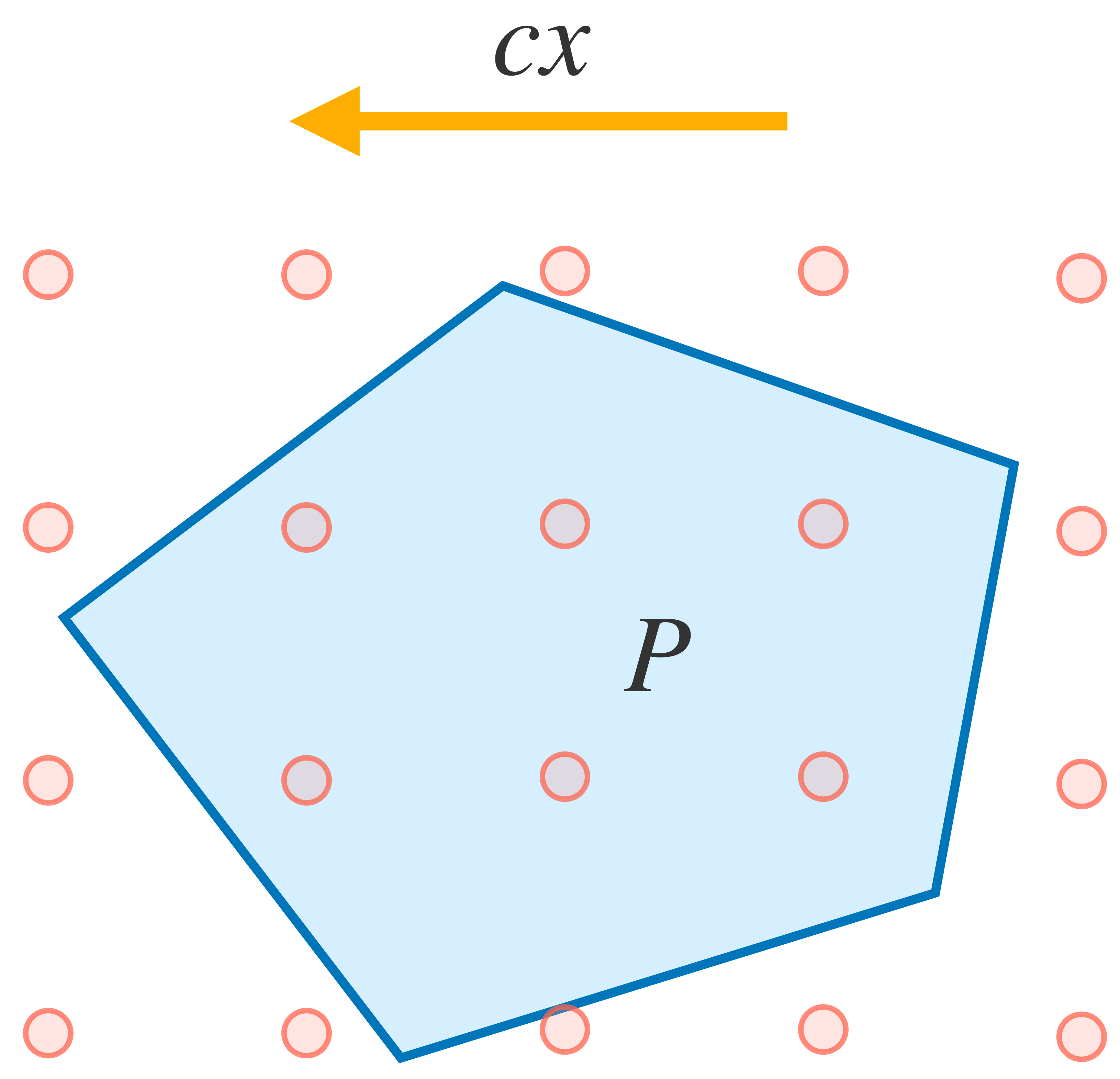
<u>**Branching**</u> — "Break P up into subpolytopes"



Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes



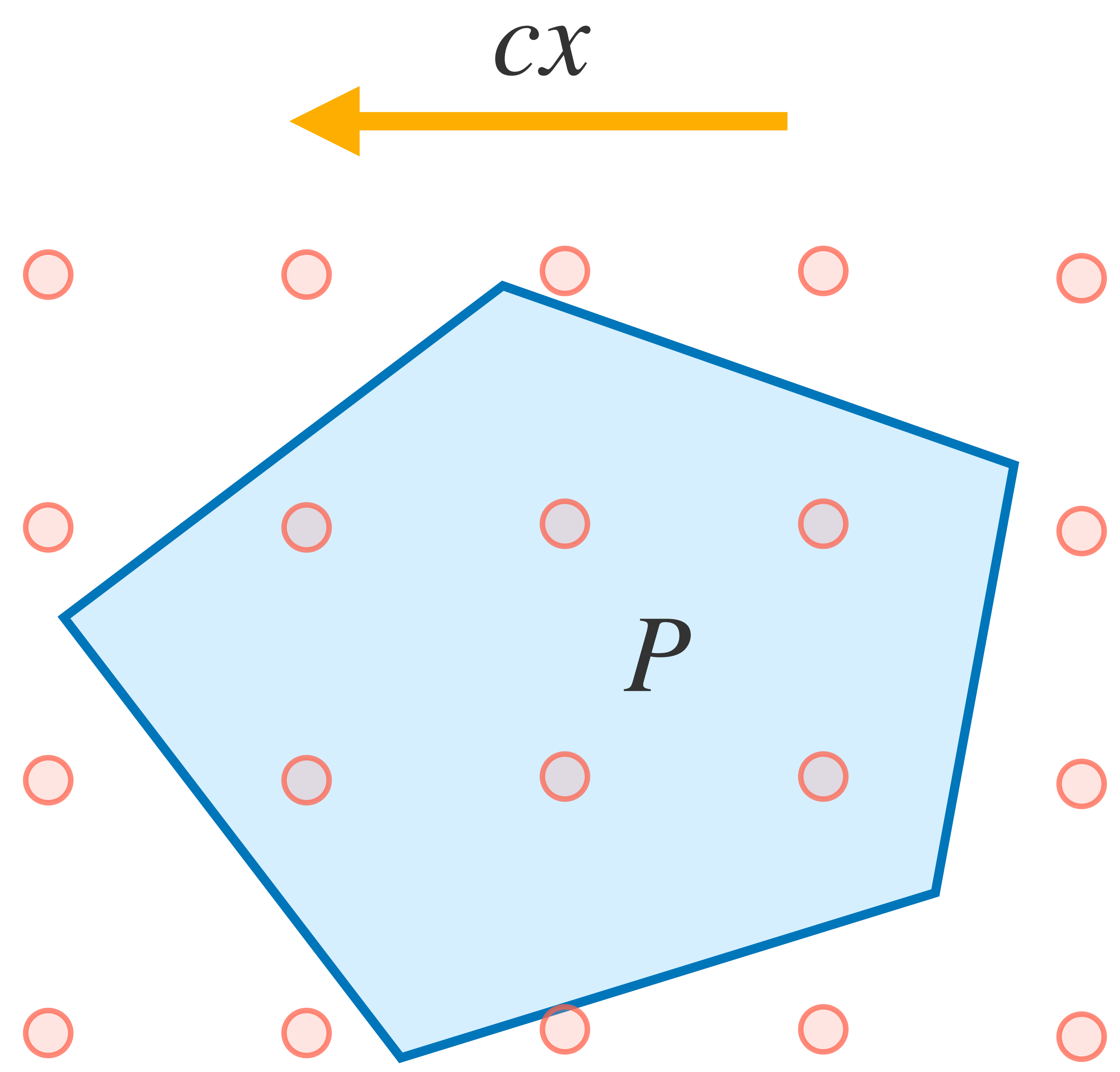
Branching —"Break P up into subpolytopes"

1. Heuristically choose an **integer**-linear inequality $ax \geq b$

Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes



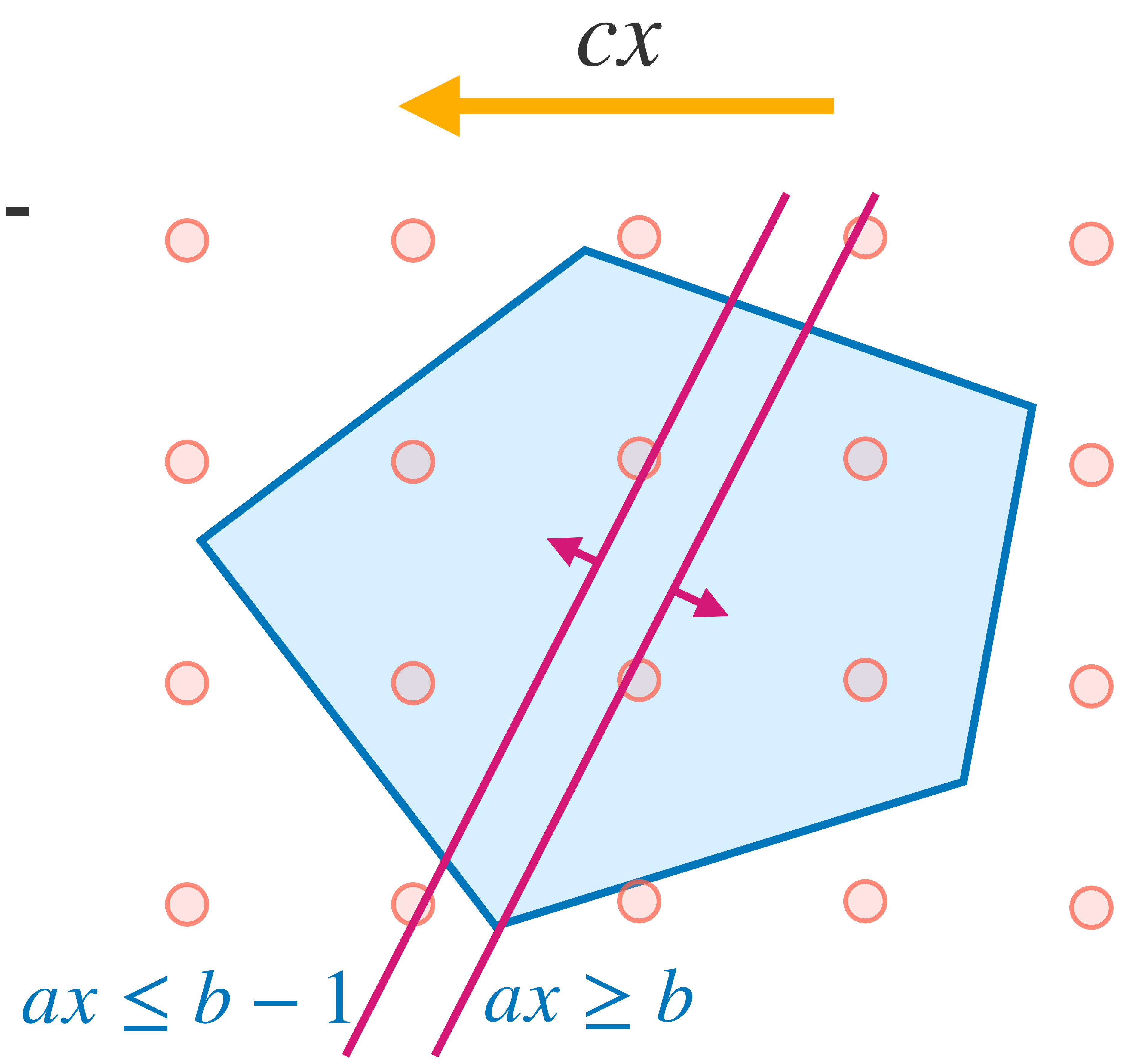
<u>**Branching**</u> —"Break P up into subpolytopes"

1. Heuristically choose an **integer**-linear inequality $ax \geq b$
2. Break P into $P \cap \{ax \geq b\}$ and $P \cap \{ax \leq b - 1\}$

Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes



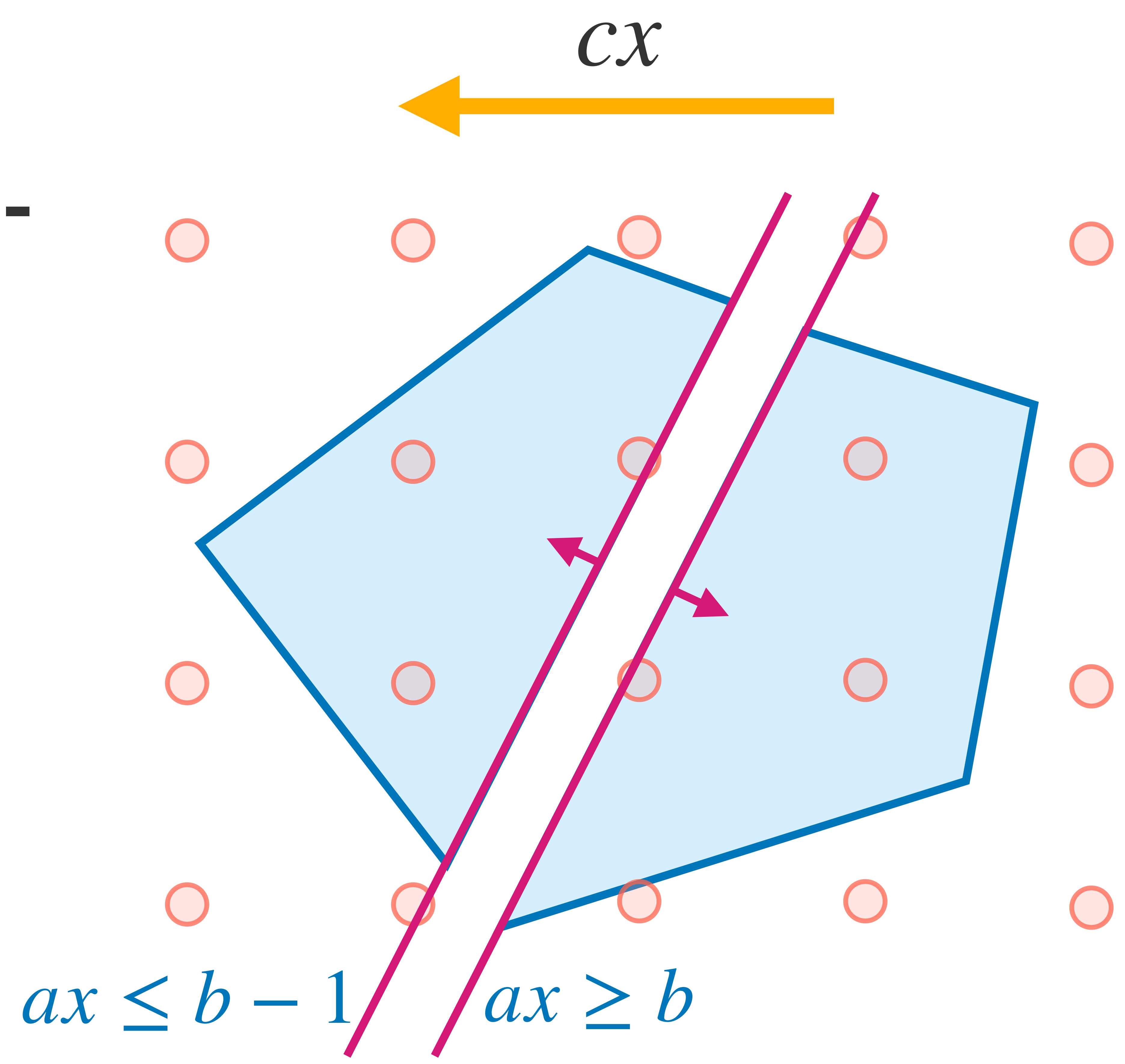
Branching —"Break P up into subpolytopes"

1. Heuristically choose an **integer**-linear inequality $ax \geq b$
2. Break P into $P \cap \{ax \geq b\}$ and $P \cap \{ax \leq b - 1\}$

Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes



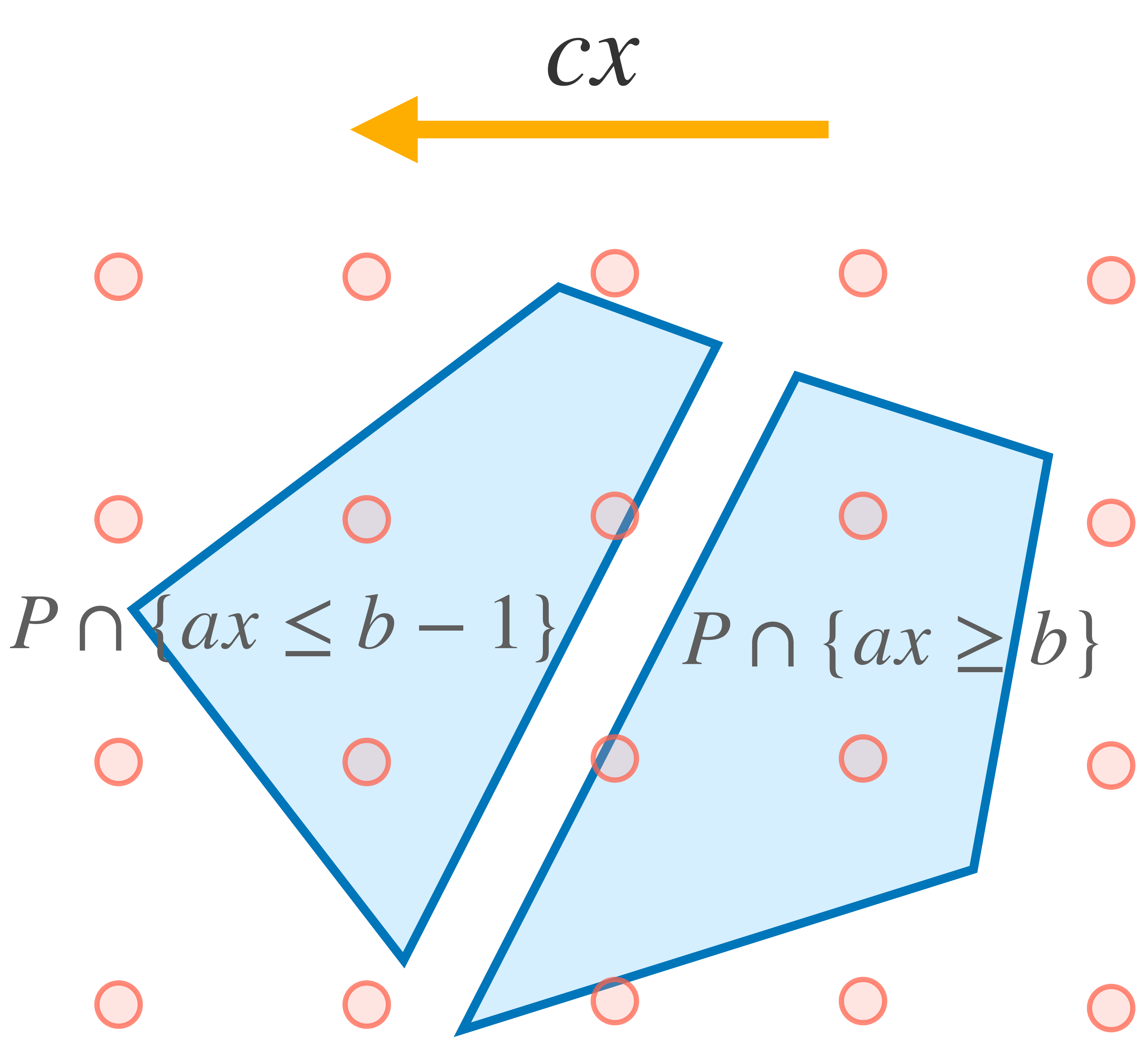
<u>**Branching**</u> — "Break P up into subpolytopes"

1. Heuristically choose an **integer**-linear inequality $ax \geq b$
2. Break P into $P \cap \{ax \geq b\}$ and $P \cap \{ax \leq b - 1\}$

Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes



Branching —"Break P up into subpolytopes"

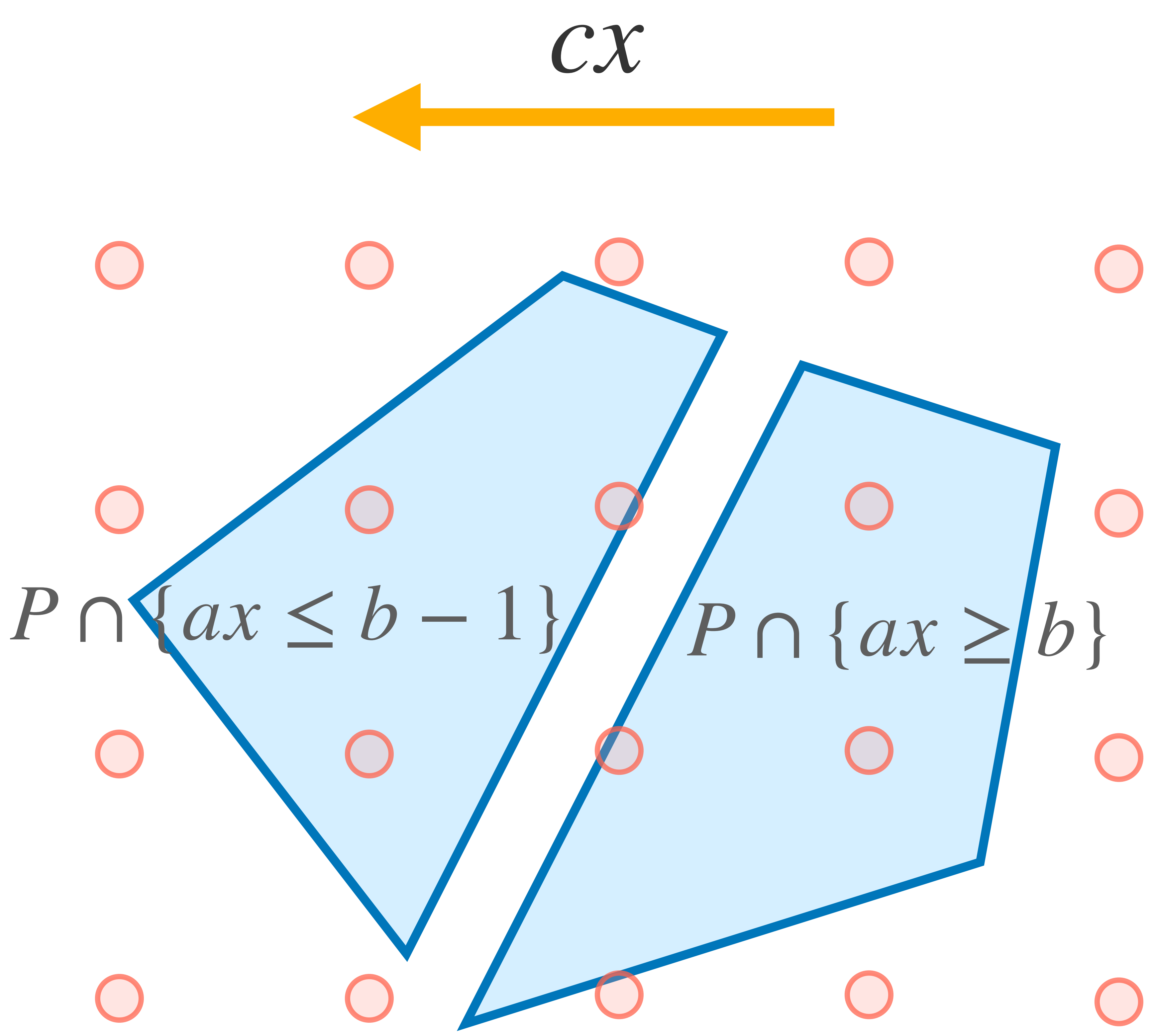
1. Heuristically choose an **integer**-linear inequality $ax \geq b$
2. Break P into $P \cap \{ax \geq b\}$ and $P \cap \{ax \leq b-1\}$

Preserves integer solutions! $x \in \mathbb{Z}^n$ satisfies $ax \leq b-1$ or $ax \geq b$

Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes



<u>**Branching**</u> —"Break P up into subpolytopes"

1. Heuristically choose an **integer**-linear inequality $ax \ge b$
2. **Recurse** on $P \cap \{ax \ge b\}$ and $P \cap \{ax \le b - 1\}$

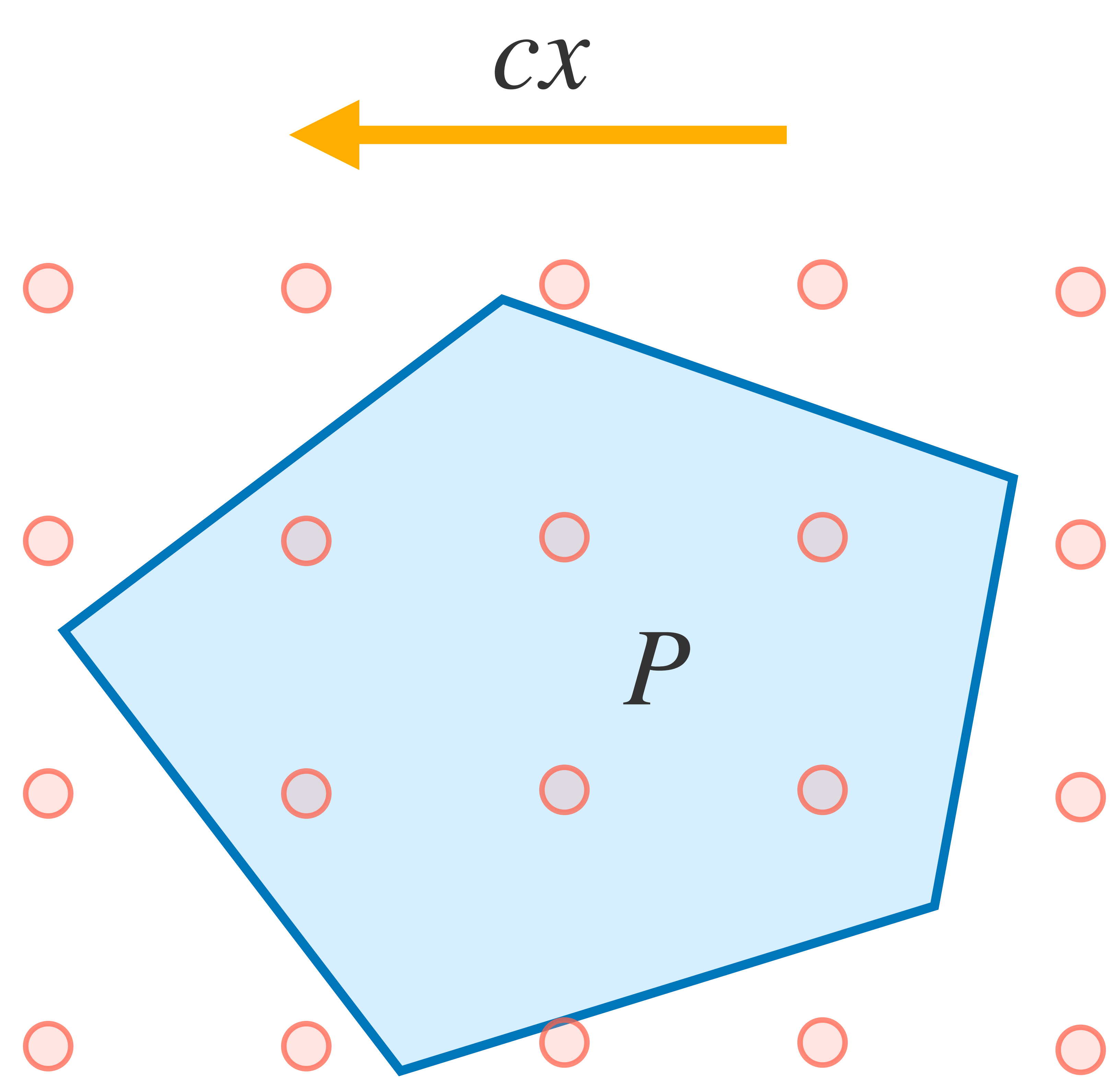
Preserves integer solutions! $x \in \mathbb{Z}^n$ satisfies $ax \le b - 1$ or $ax \ge b$

Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes

<u>**Cutting Planes**</u> — "Remove corners of P"



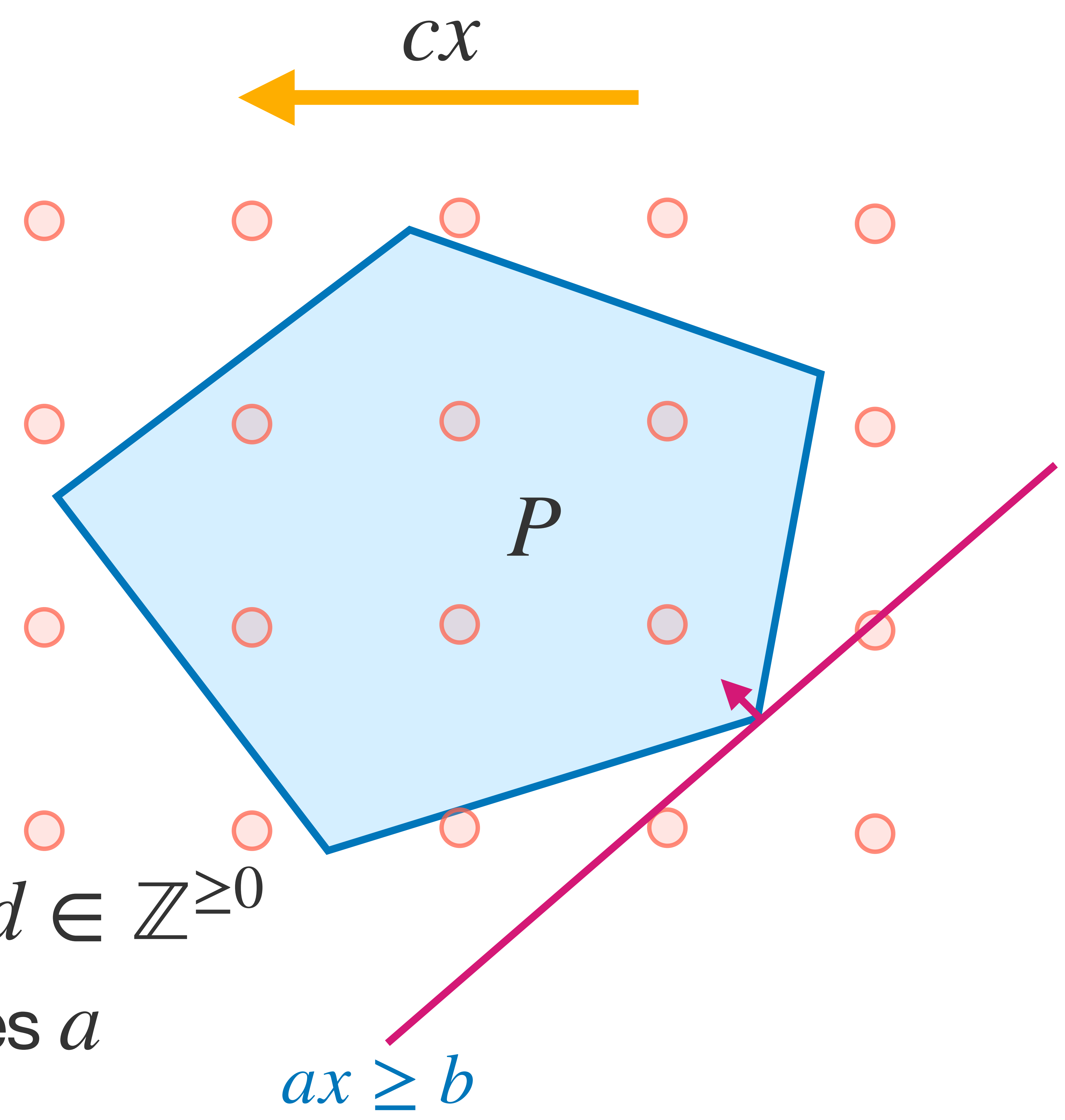
Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes

<u>Cutting Planes</u> — "Remove corners of P"

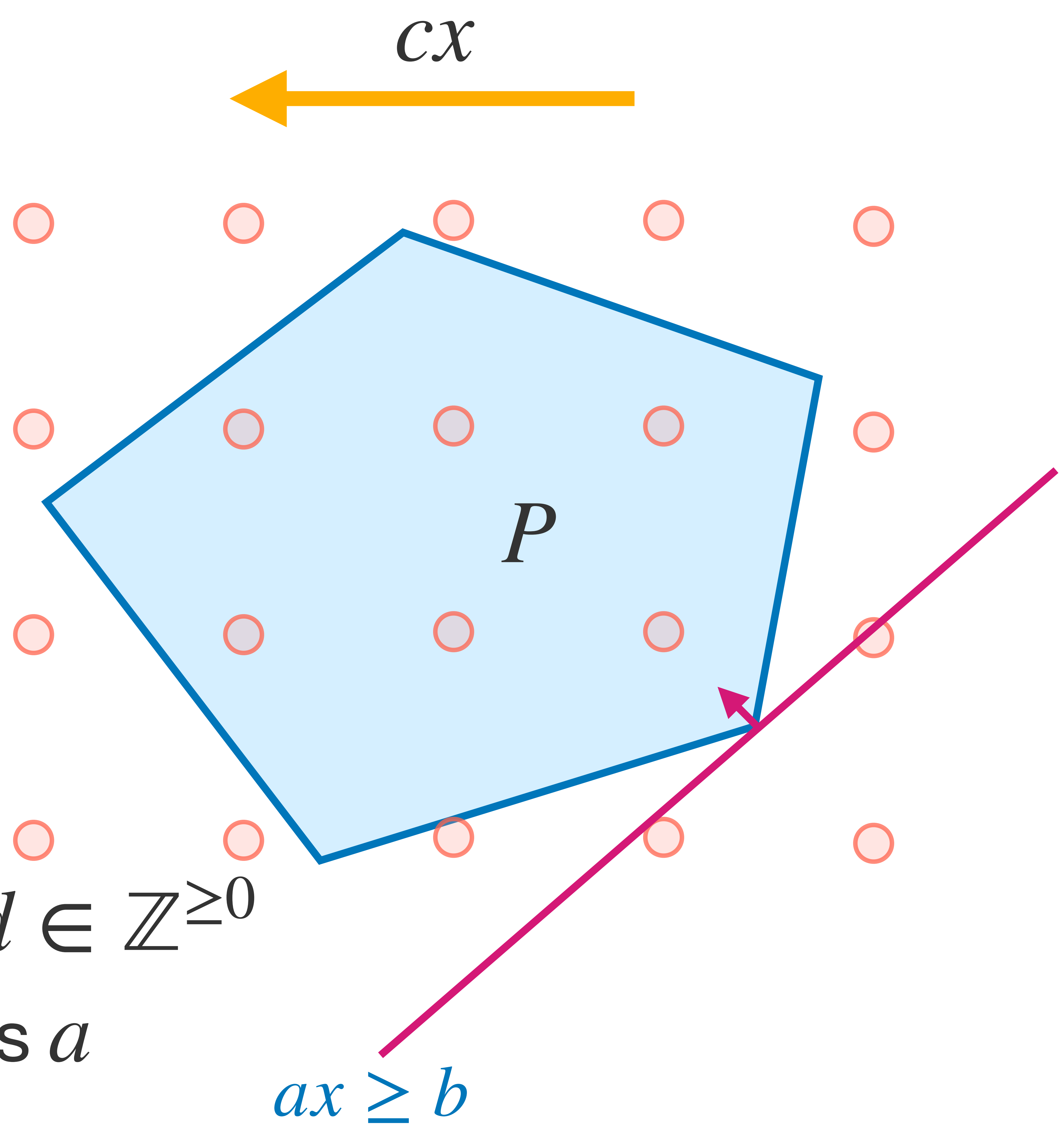
1. Choose an integer-linear inequality $ax \geq b$, and $d \in \mathbb{Z}^{\geq 0}$ s.t. every point in P satisfies $ax \geq b$ and d divides a



Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes



Cutting Planes — "Remove corners of P"

1. Choose an integer-linear inequality $ax \geq b$, and $d \in \mathbb{Z}^{\geq 0}$ s.t. every point in P satisfies $ax \geq b$ and d divides a
2. Add $(a/d)x \geq \lceil b/d \rceil$

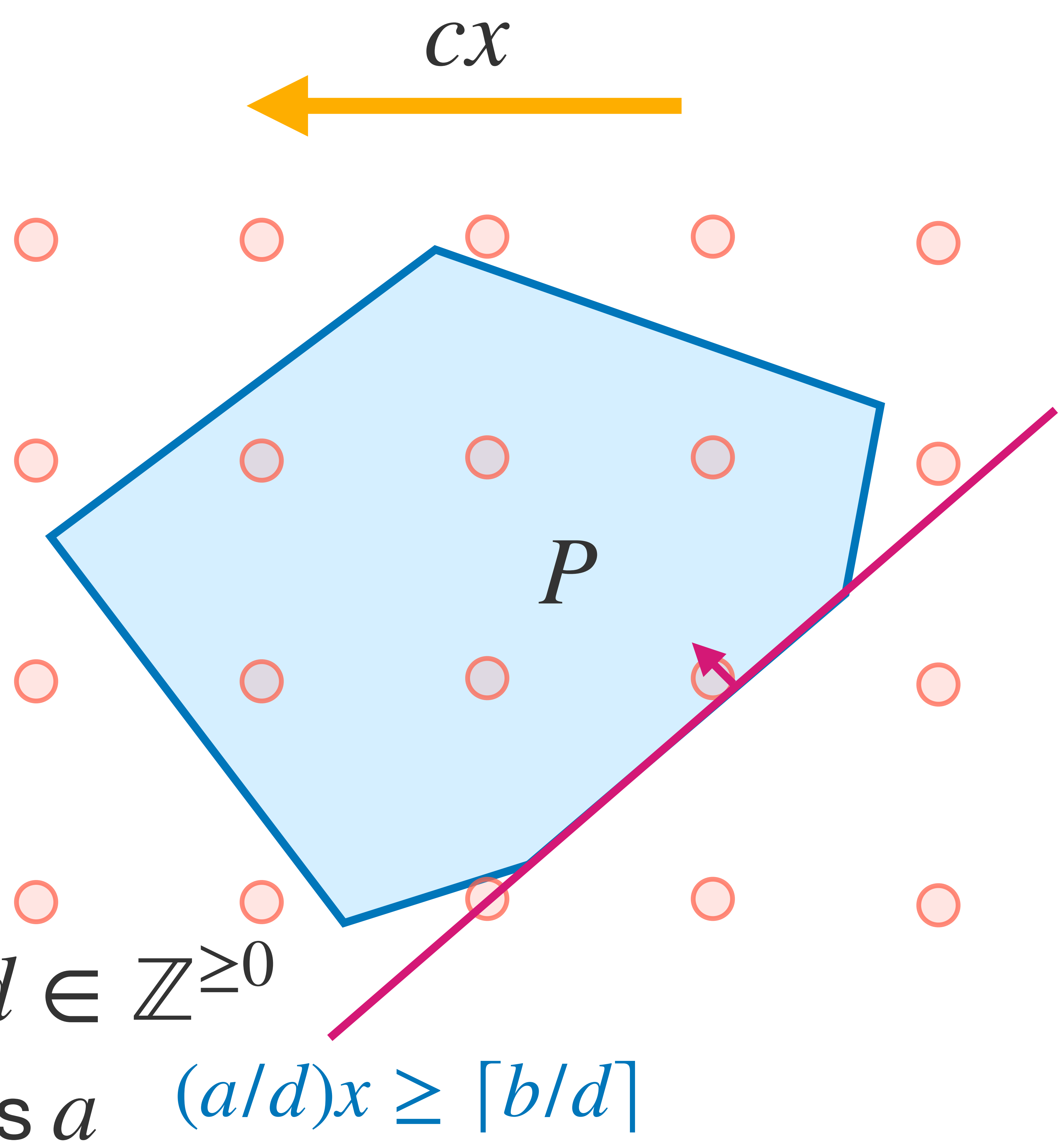
Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes

<u>Cutting Planes</u> — "Remove corners of P"

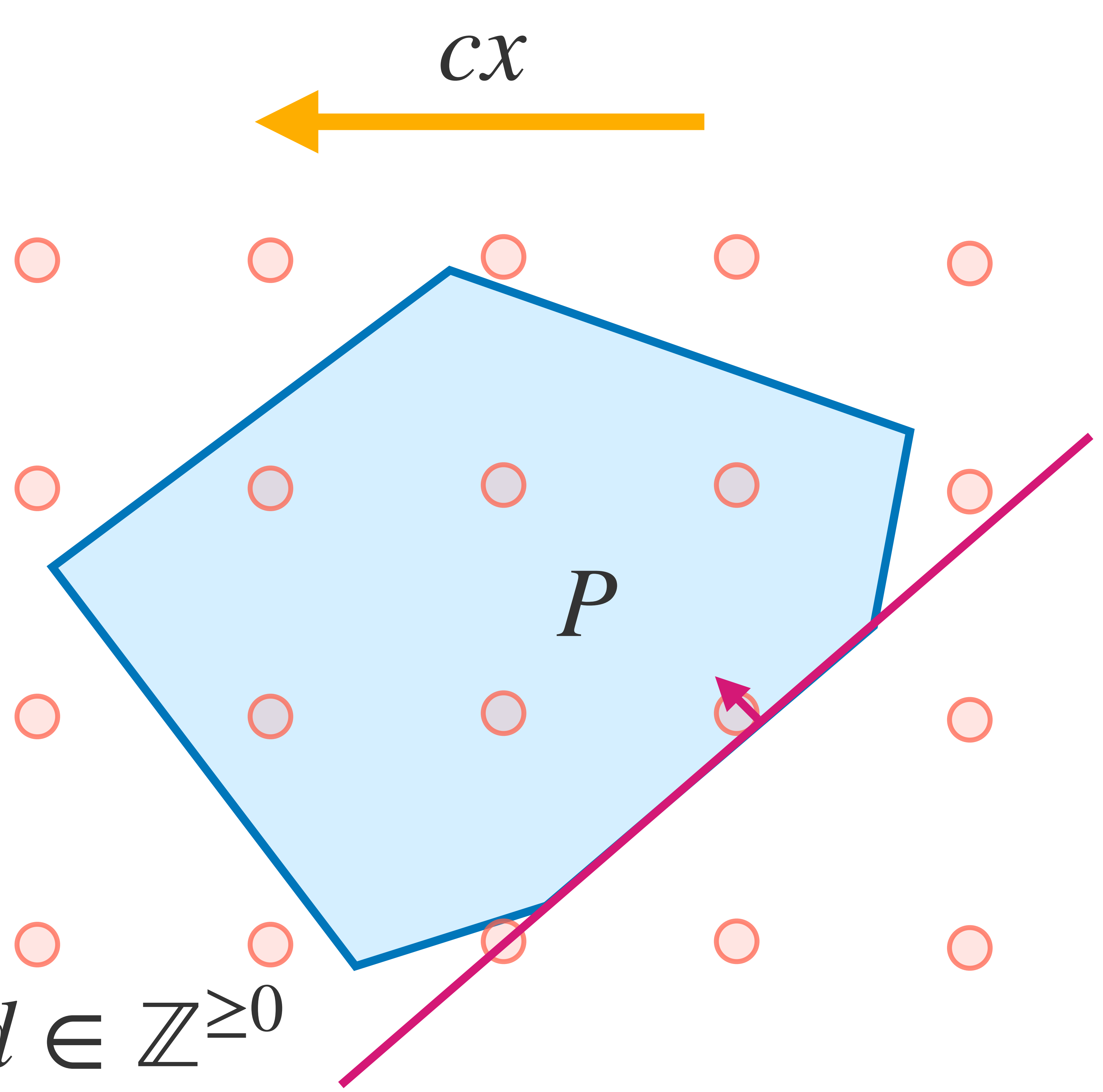
1. Choose an integer-linear inequality $ax \geq b$, and $d \in \mathbb{Z}^{\geq 0}$ s.t. every point in P satisfies $ax \geq b$ and d divides a
2. Add $(a/d)x \geq \lceil b/d \rceil$



Branch-and-Cut

Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes

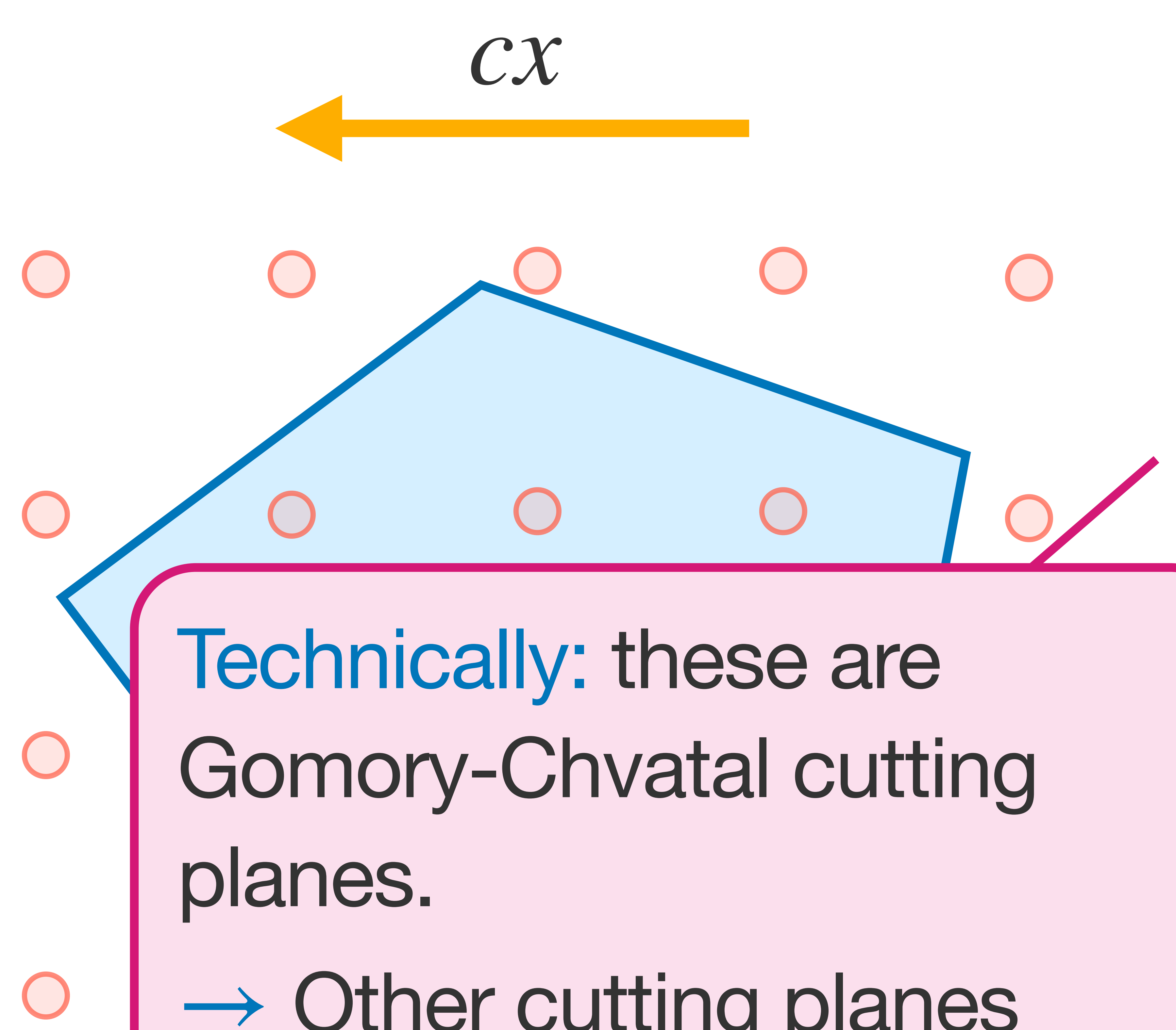


Cutting Planes — "Remove corners of P"

1. Choose an integer-linear inequality $ax \geq b$, and $d \in \mathbb{Z}^{\geq 0}$ s.t. every point in P satisfies $ax \geq b$ and d divides a
2. Add $(a/d)x \geq \lceil b/d \rceil$

Preserves **integer** points in P

Branch-and-Cut



Branch-and-Cut has two ways of **removing non-integer points** from P:

1. DPLL-style **branch**ing on linear inequalities
2. **Cut**ting planes

Cutting Planes — "Remove corners of P"

1. Choose an integer-linear inequality $ax \geq b$, and $d \in$
   s.t. every point in P satisfies $ax \geq b$ and d divides a
2. Add $(a/d)x \geq \lceil b/d \rceil$

Preserves **integer** points

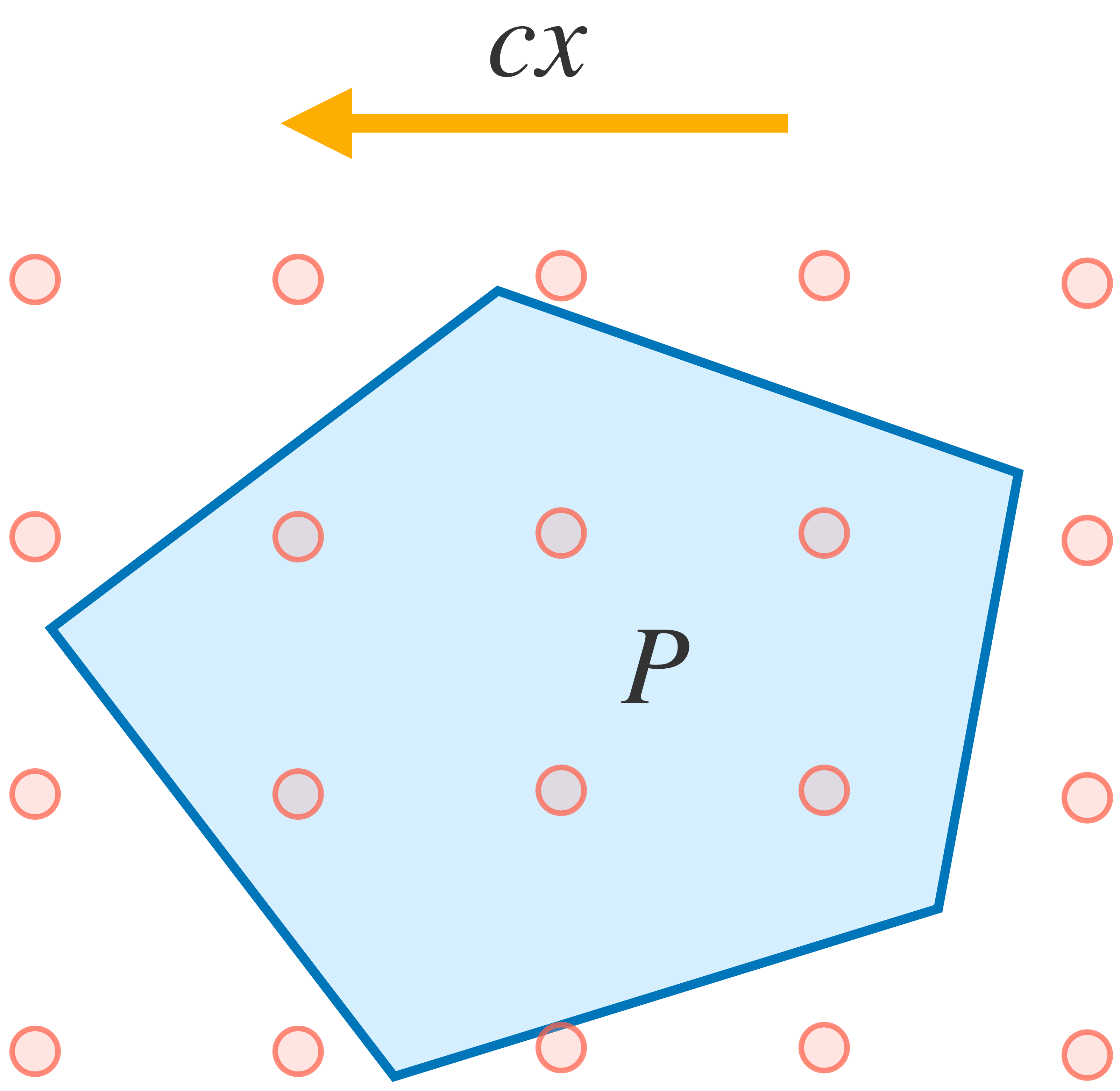
Technically: these are Gomory-Chvatal cutting planes.
→ Other cutting planes have been considered as well.
→ What we talk about today applies to them as well

Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Branch and Cut Template

1. Solve the linear program.
2. If solution z is non-integral, refine polytope by:
   i) Branching.
   ii) Cutting.
3. Repeat.

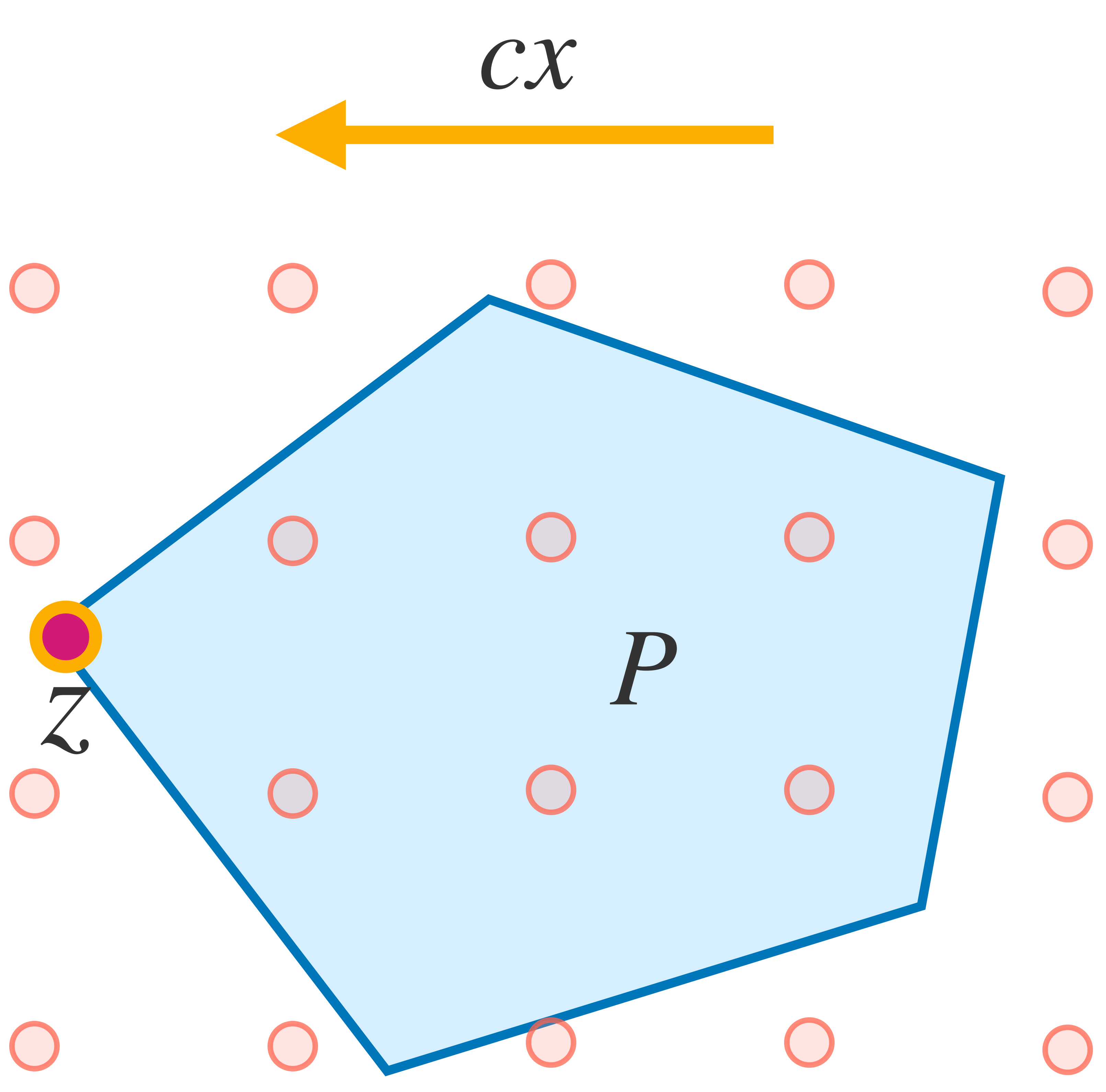


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Branch and Cut Template

1. Solve the linear program.
2. If solution z is non-integral, refine polytope by:
   i) Branching.
   ii) Cutting.
3. Repeat.

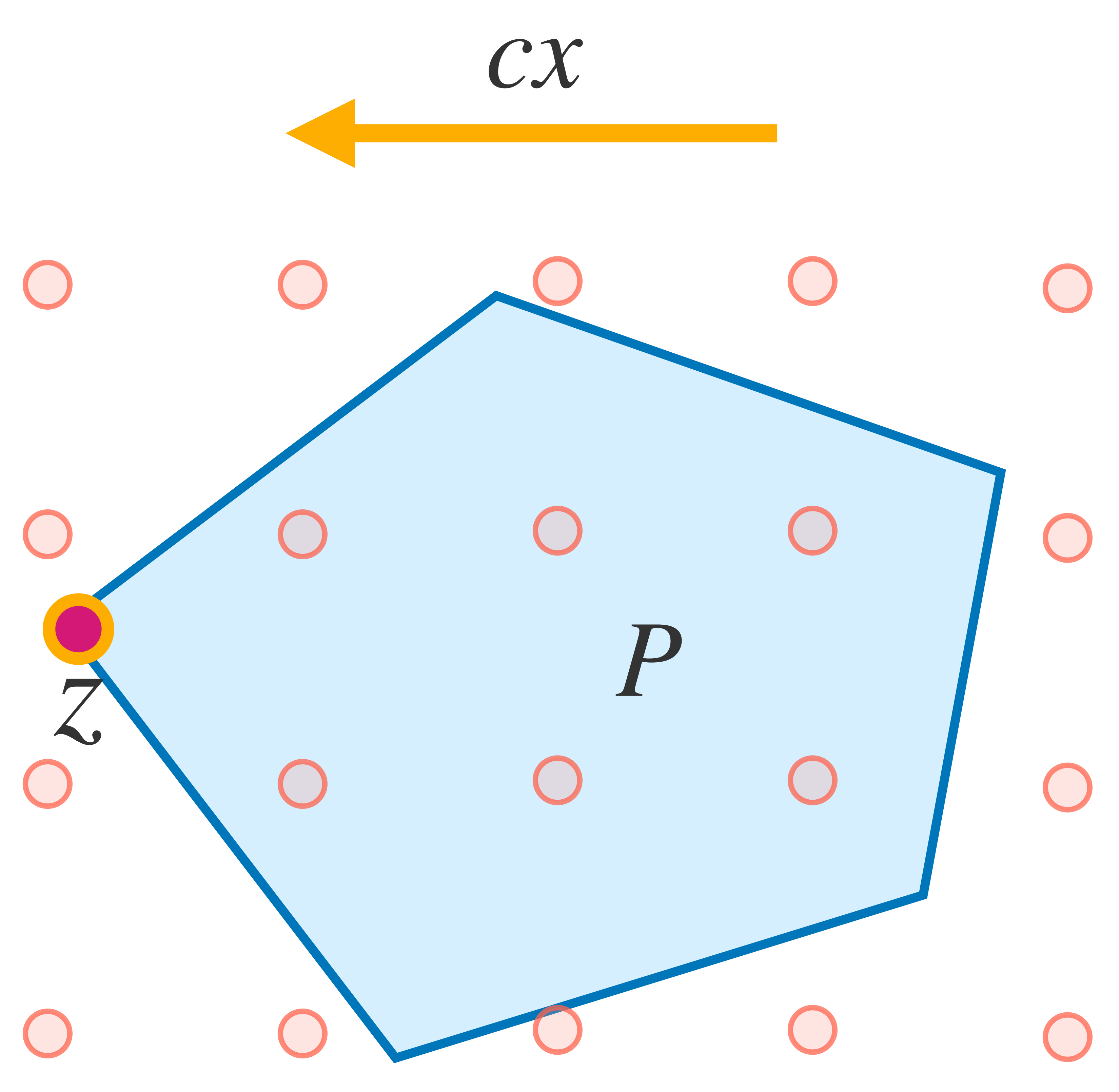


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Branch and Cut Template

1. Solve the linear program.
2. If solution z is non-integral, refine polytope by:
   i) Branching.
   ii) Cutting.
3. Repeat.

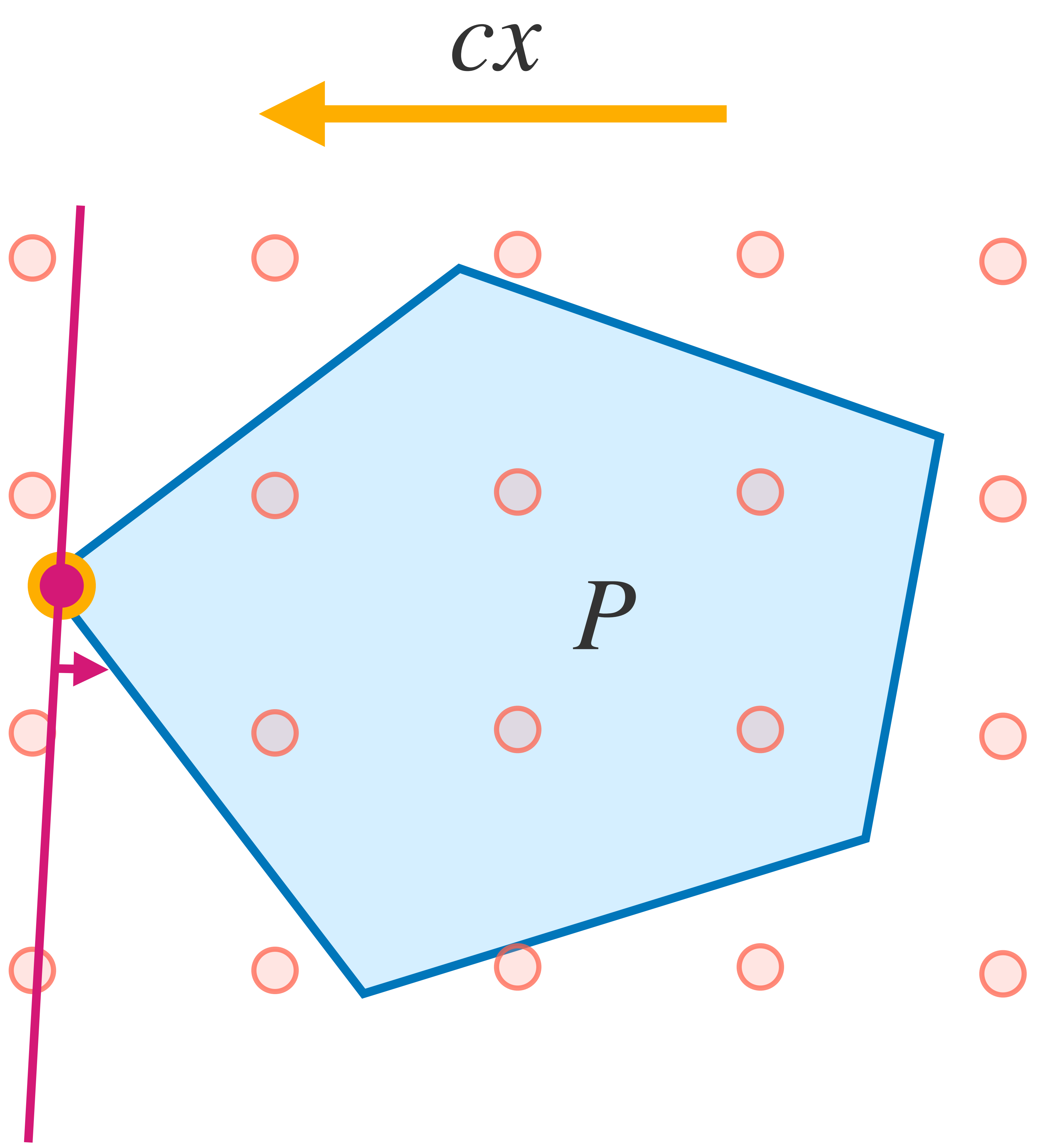


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Branch and Cut Template

1. Solve the linear program.
2. If solution z is non-integral, refine polytope by:
   i) Branching.
   ii) Cutting.
3. Repeat.

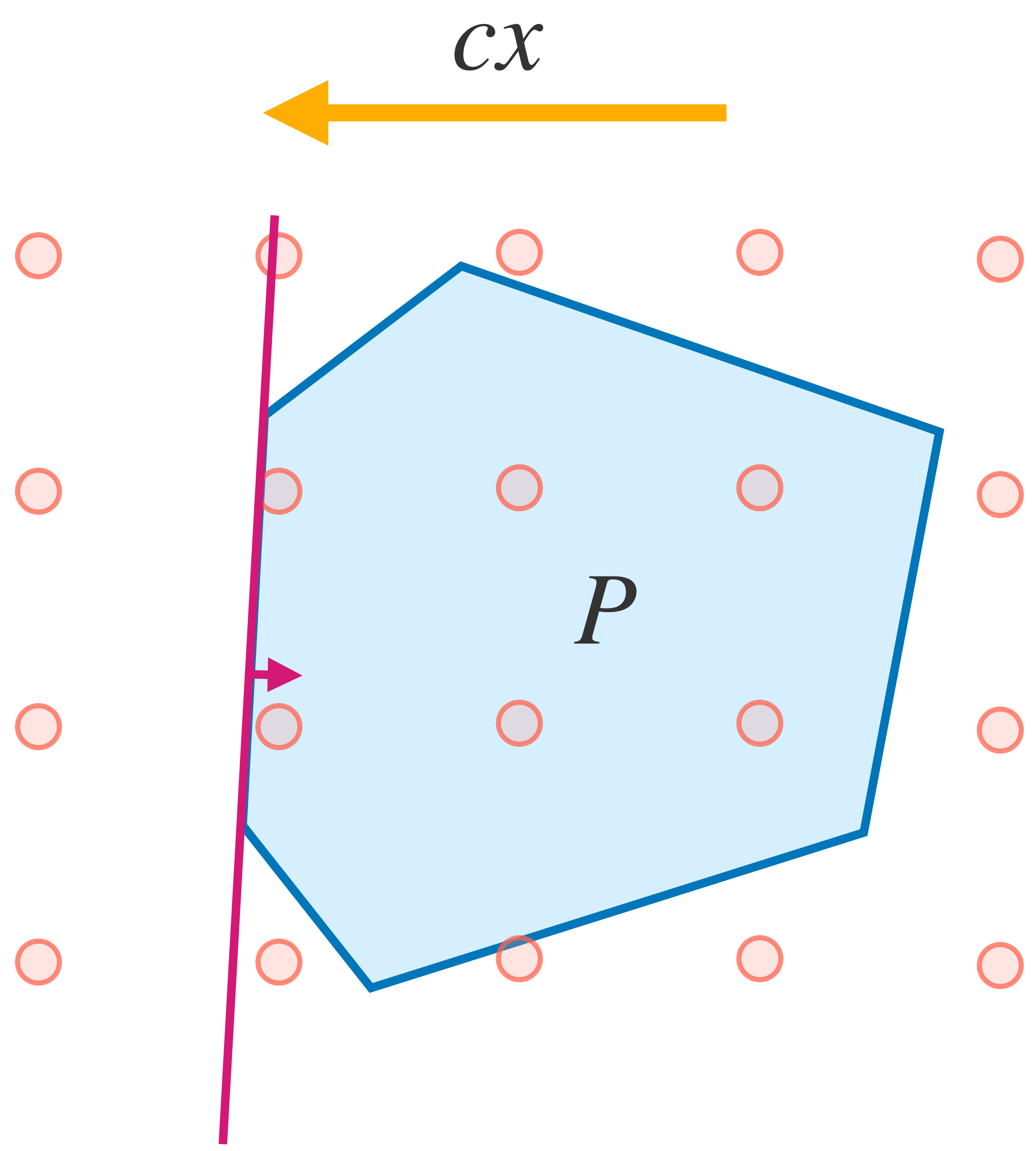


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Branch and Cut Template

1. Solve the linear program.
2. If solution z is non-integral, refine polytope by:
   i) Branching.
   ii) Cutting.
3. Repeat.

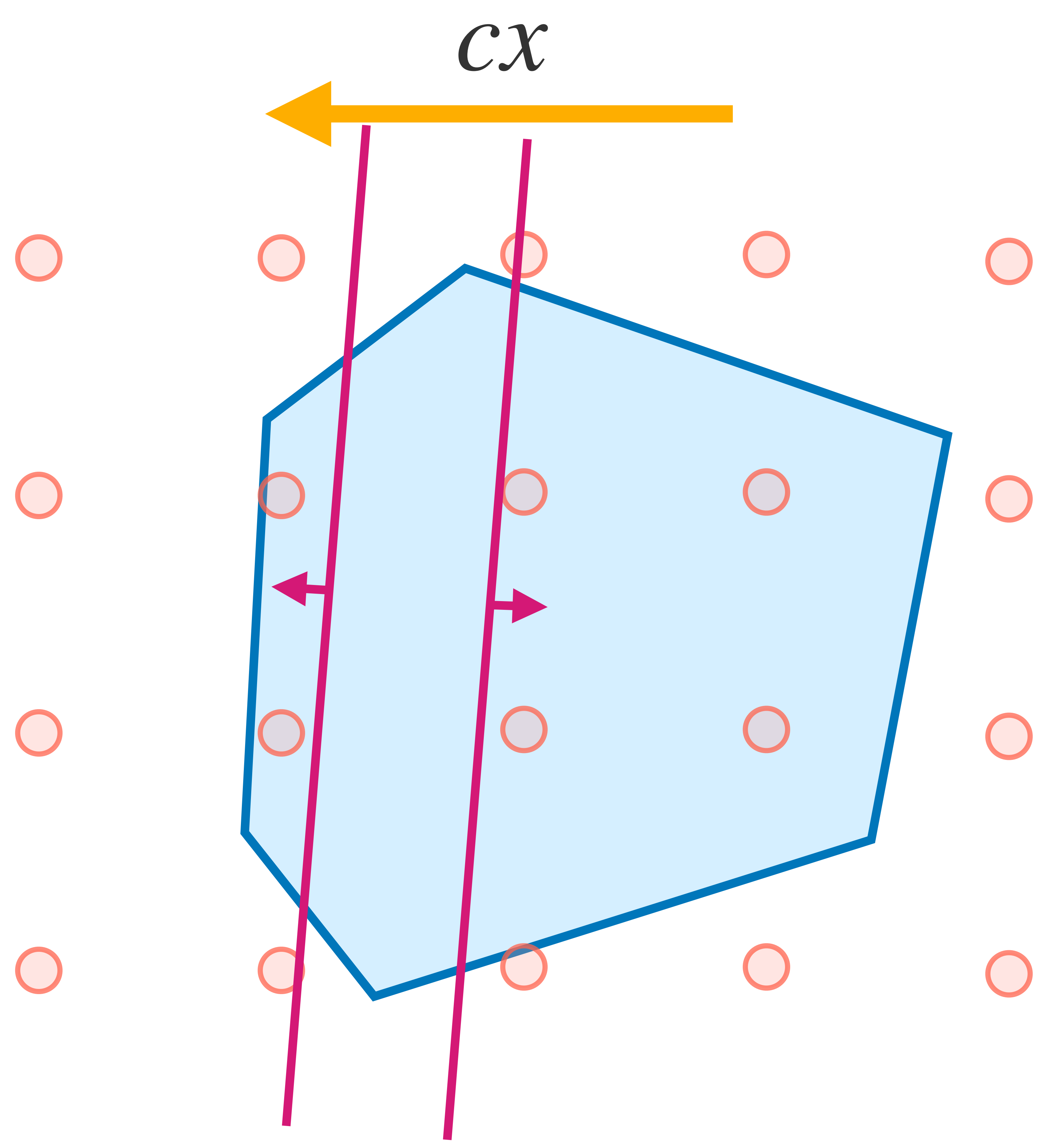


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Branch and Cut Template

1. Solve the linear program.
2. If solution z is non-integral, refine polytope by:
   i) Branching.
   ii) Cutting.
3. Repeat.

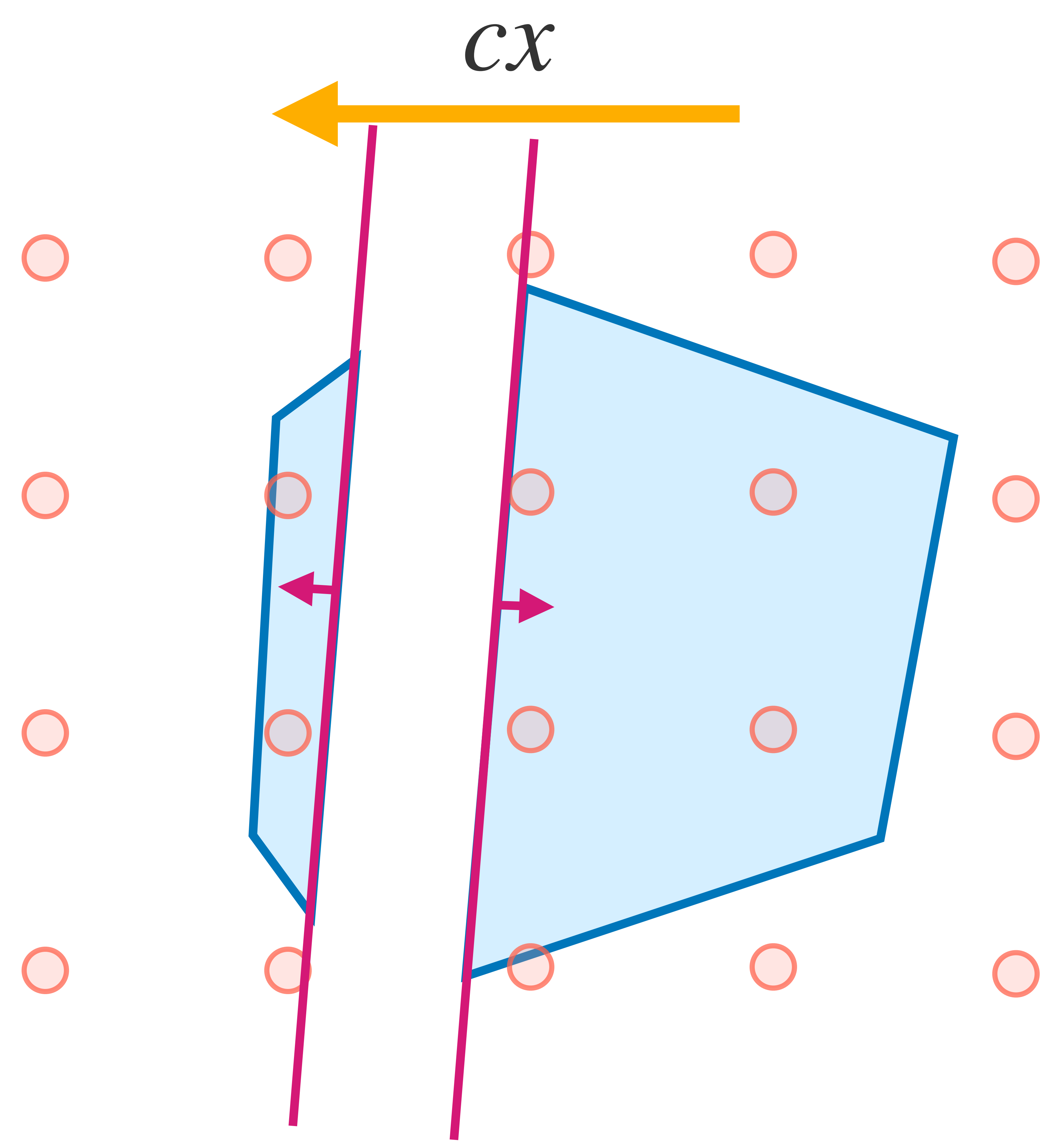


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Branch and Cut Template

1. Solve the linear program.
2. If solution z is non-integral, refine polytope by:
   i) Branching.
   ii) Cutting.
3. Repeat.

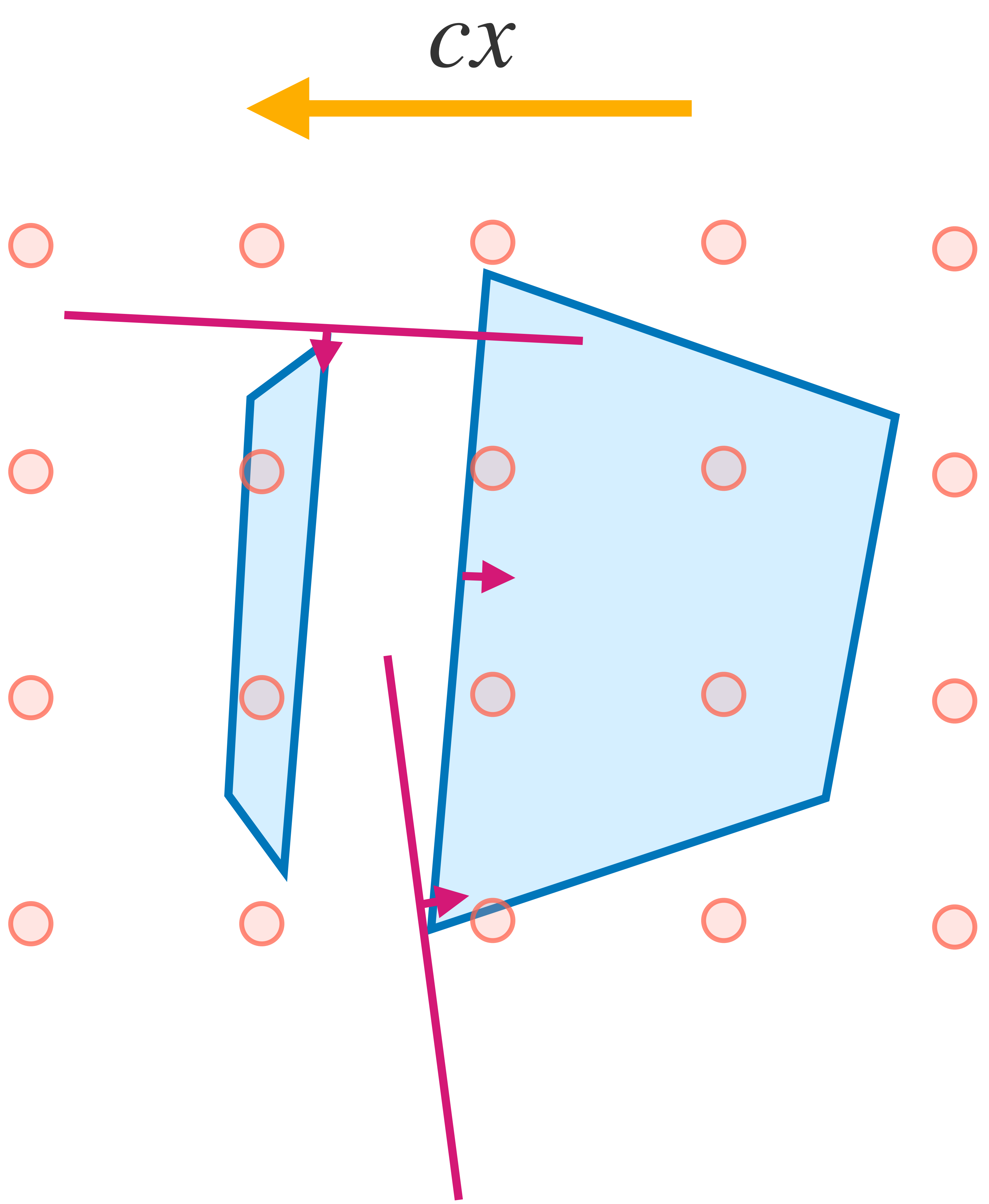


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

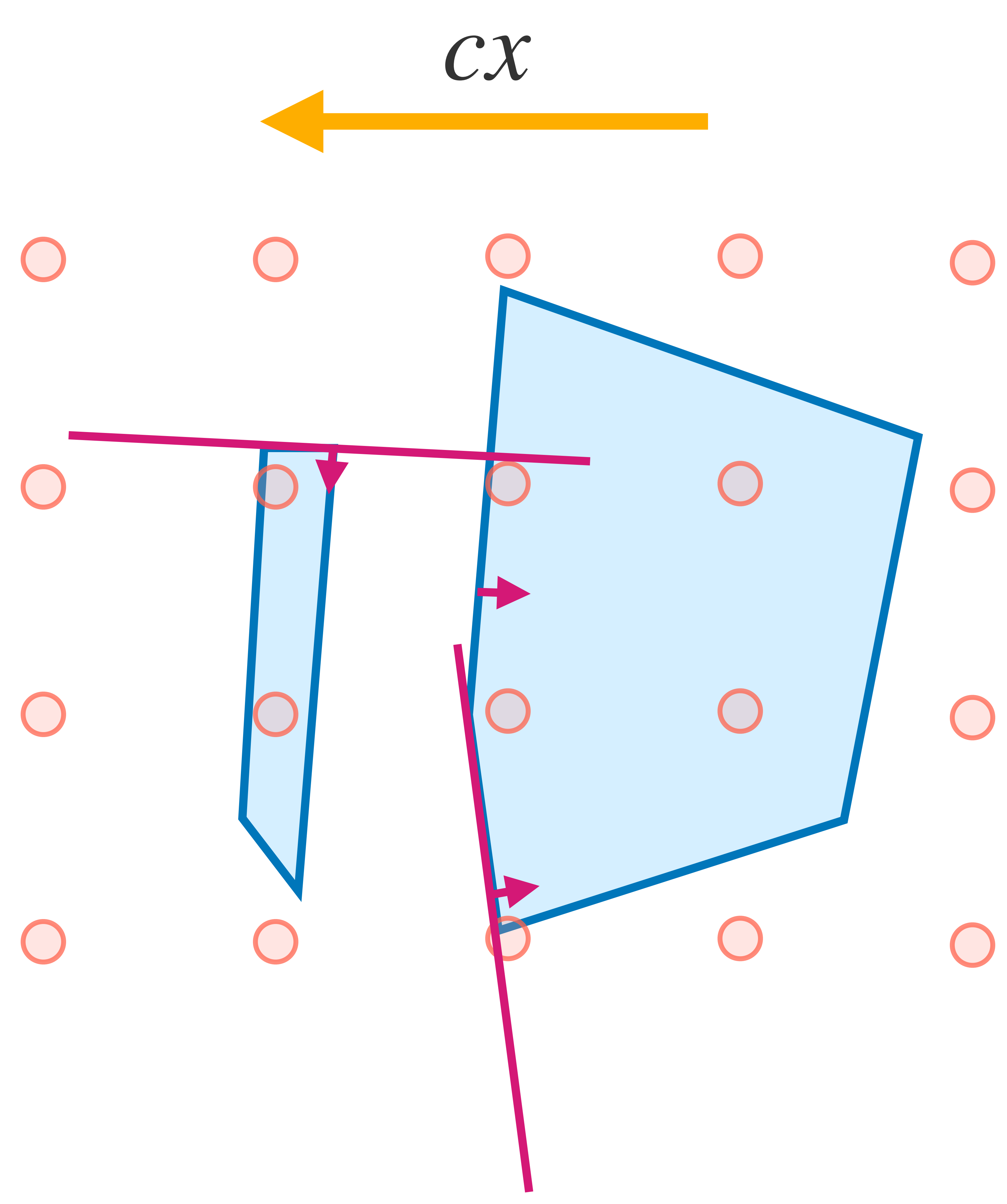
Branch and Cut Template

1. Solve the linear program.
2. If solution z is non-integral, refine polytope by:
   i) Branching.
   ii) Cutting.
3. Repeat.



Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!



Branch and Cut Template

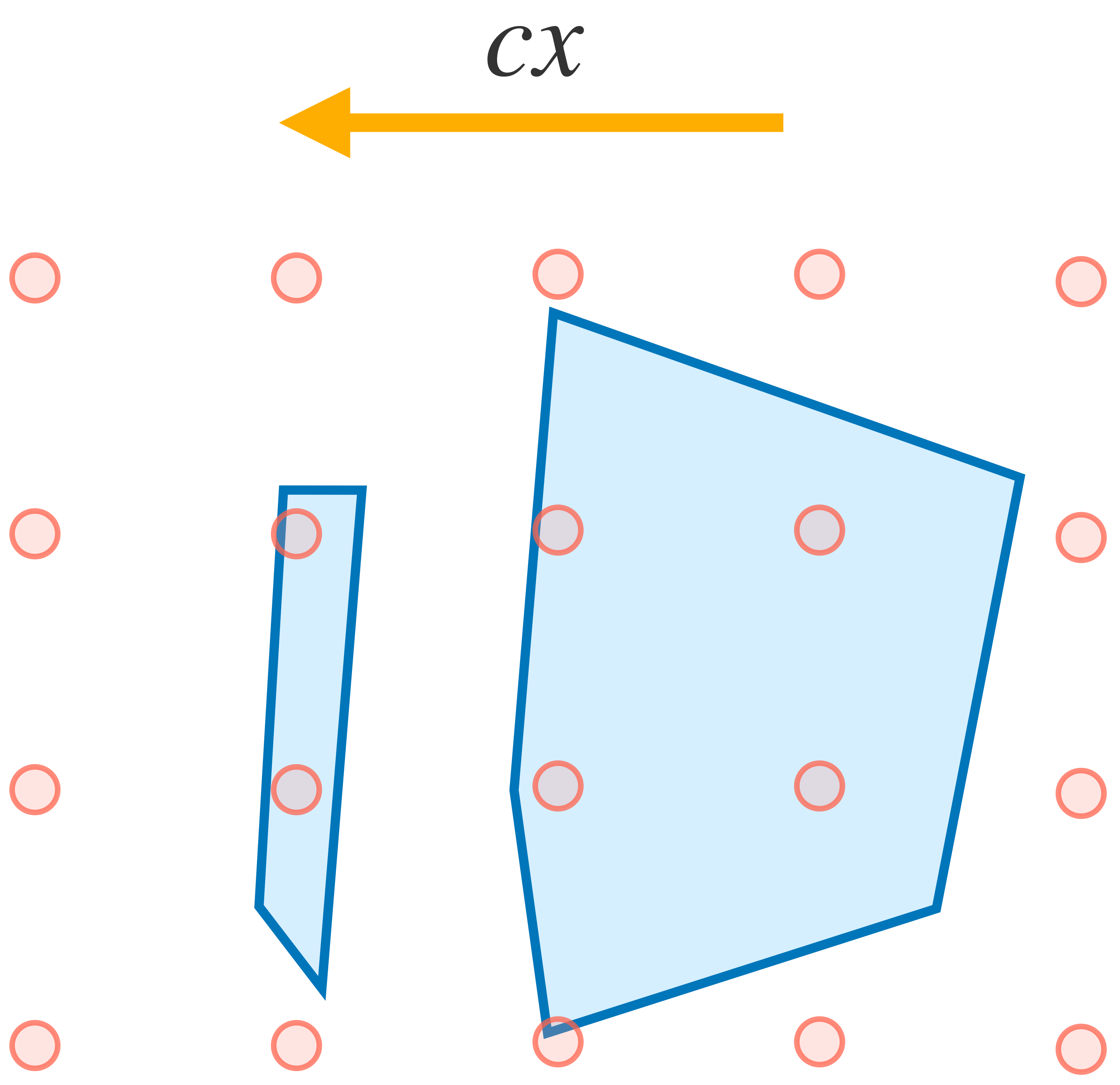
1. Solve the linear program.
2. If solution z is non-integral, refine polytope by:
   i) Branching.
   ii) Cutting.
3. Repeat.

Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Branch and Cut Template

1. Solve the linear program.
2. If solution z is non-integral, refine polytope by:
   i) Branching.
   ii) Cutting.
3. Repeat.

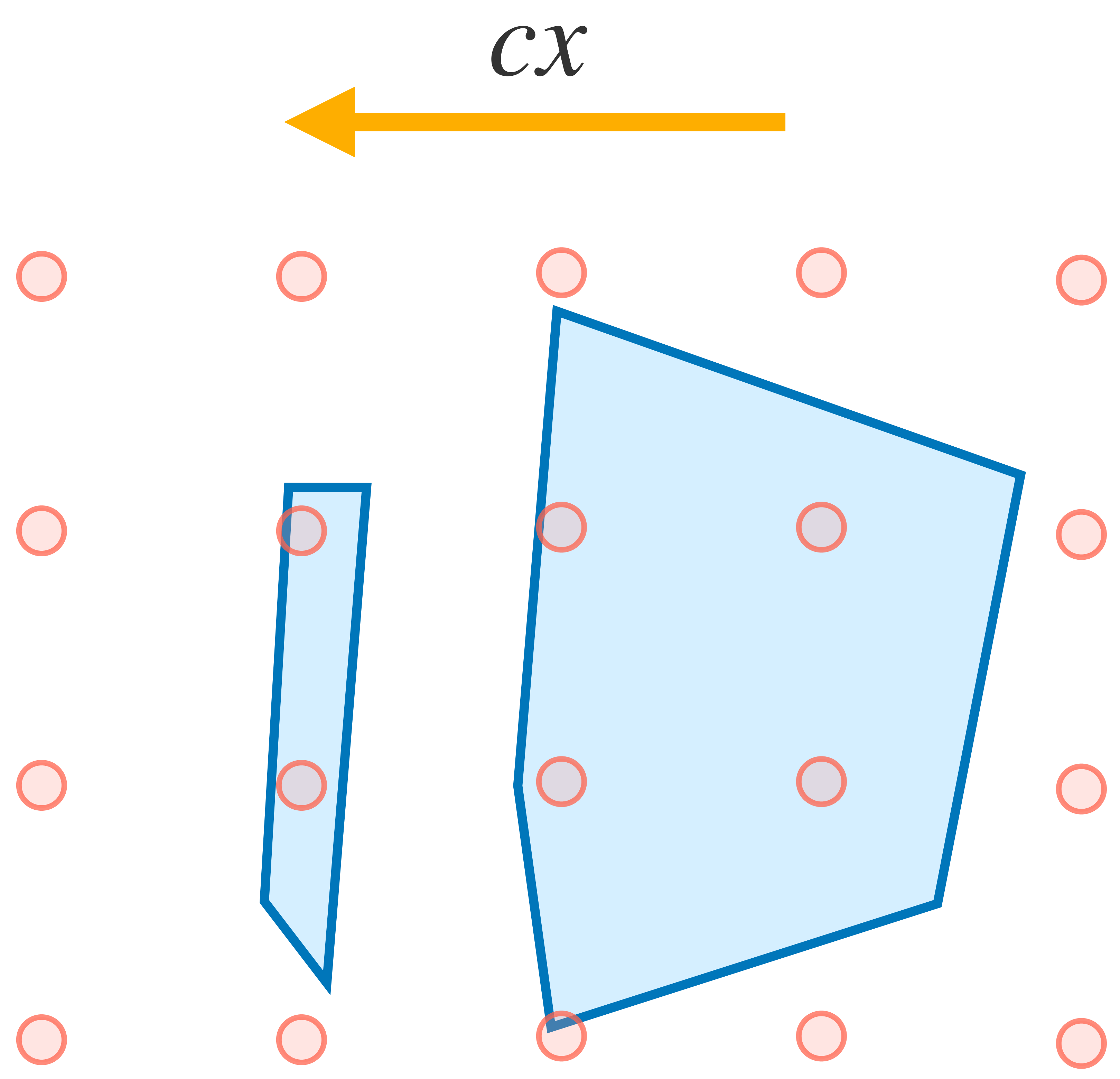


Branch-and-Cut

Idea: Try to use **linear** programming to solve **integer** programming!

Branch and Cut Template

1. Solve the linear program.
2. If solution z is non-integral, refine polytope by:
   i) Branching.
   ii) Cutting.
3. Repeat.



If the polytope is refined only by cutting, then this is known as a **cutting planes algorithm**

Formalizing Modern IP Solvers

[Chvatal73] Introduced the **Cutting Planes** proof system to formalize cutting planes algorithms.

Formalizing Modern IP Solvers

[Chvatal73] Introduced the **Cutting Planes** proof system to formalize cutting planes algorithms.

- Only captures the **cutting** part of branch-and-cut, not **branching**.

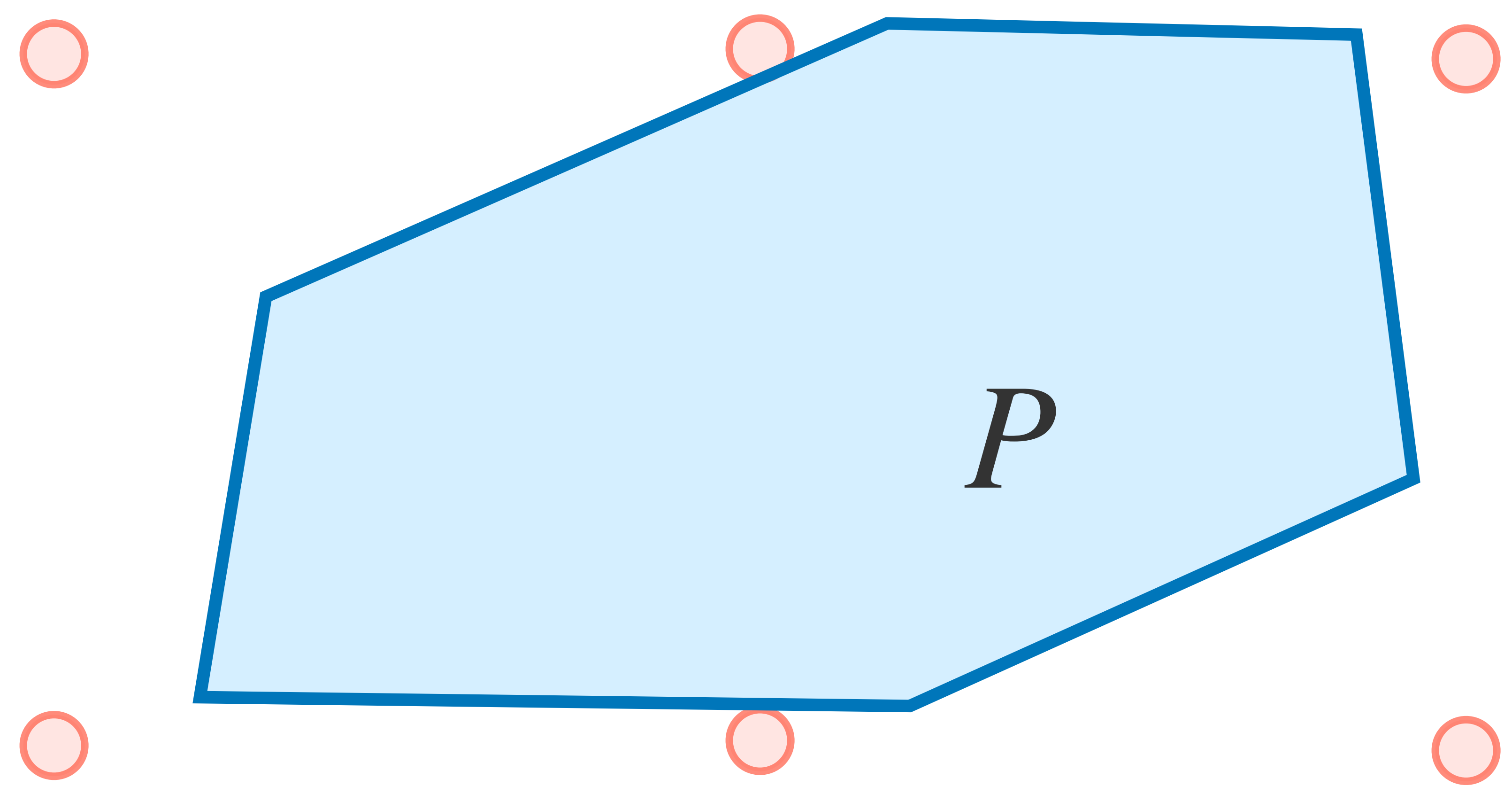
Formalizing Modern IP Solvers

[Chvatal73] Introduced the **Cutting Planes** proof system to formalize cutting planes algorithms.

- Only captures the **cutting** part of branch-and-cut, not **branching**.
- Even so, it is an important and heavily studied proof system!

Cutting Planes Proofs

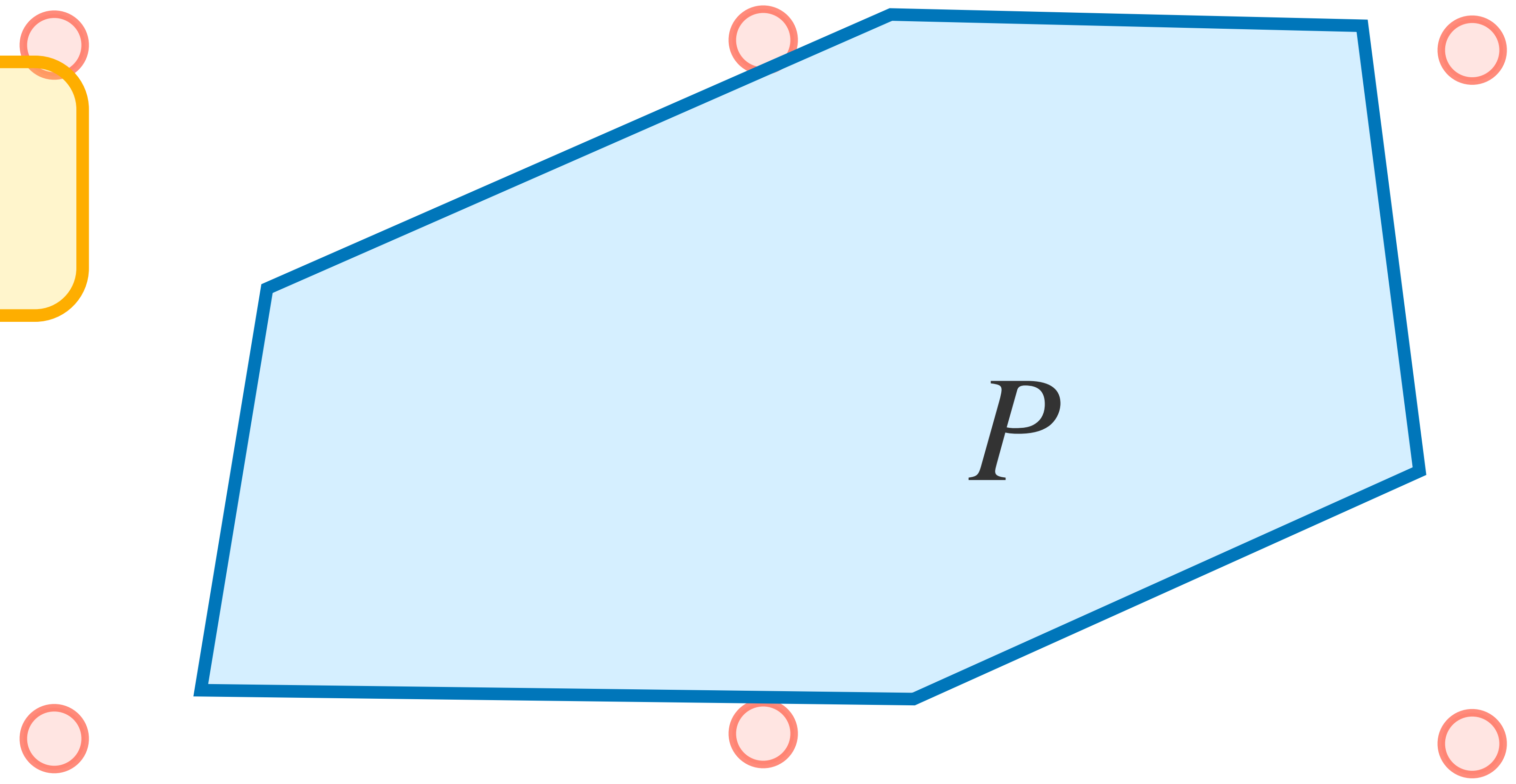
Suppose $Ax \geq b$ has **no integer solutions**



Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

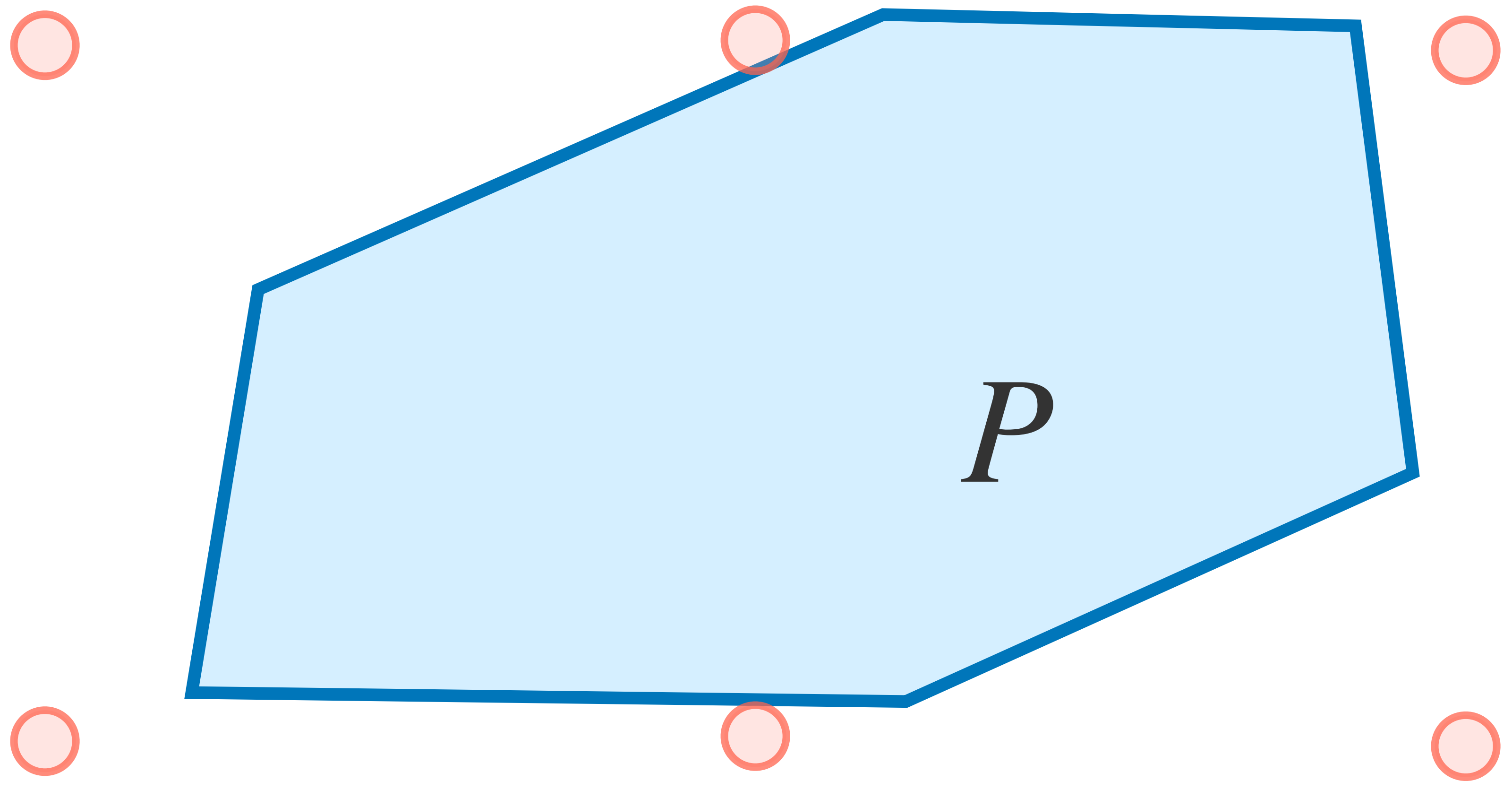
Analogous to running DPLL on unsatisfiable CNF



Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

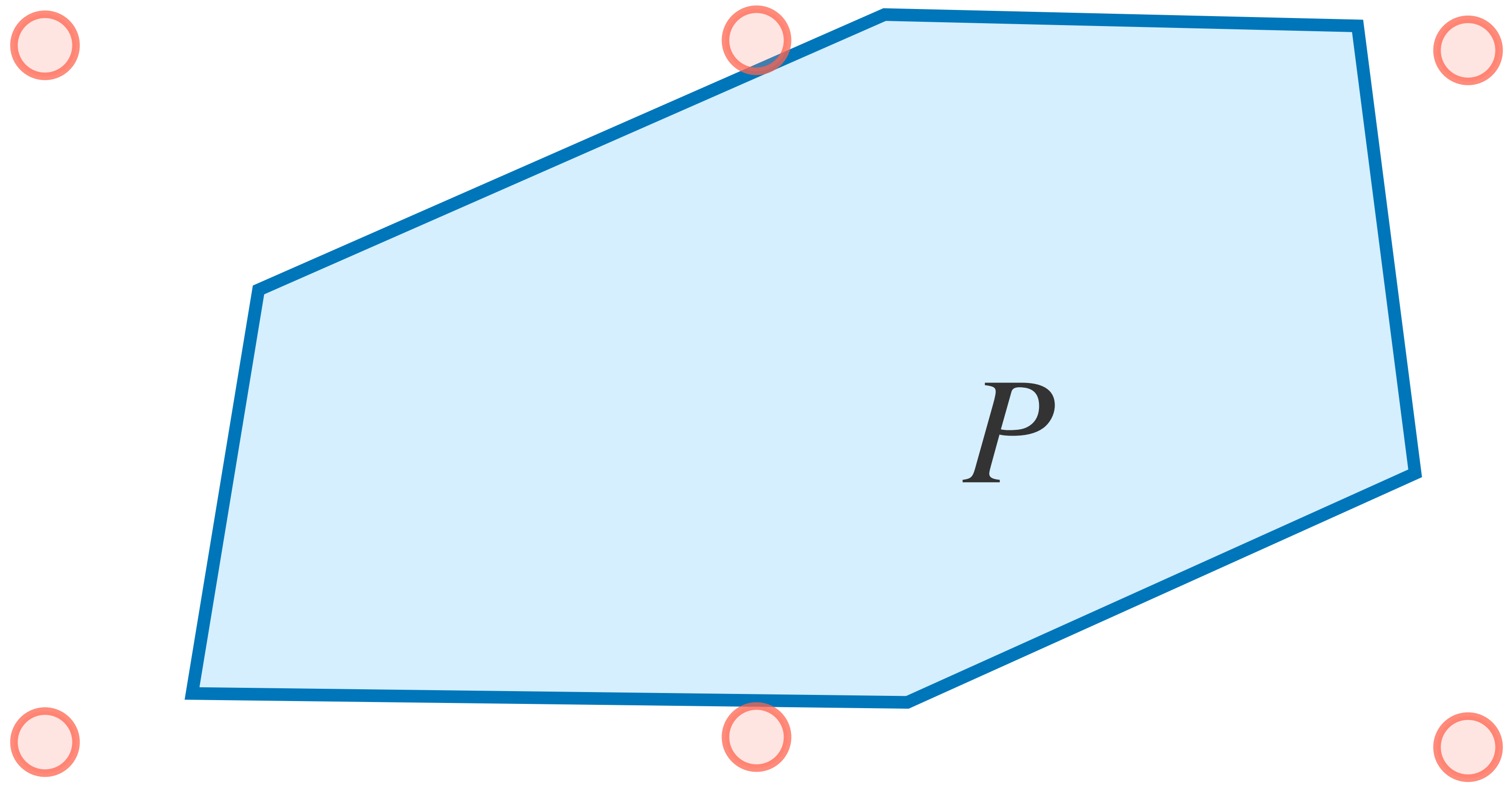
$\rightarrow$ **Prove** this fact using cutting planes!



Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!



<u>Rules</u>

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b, \; cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

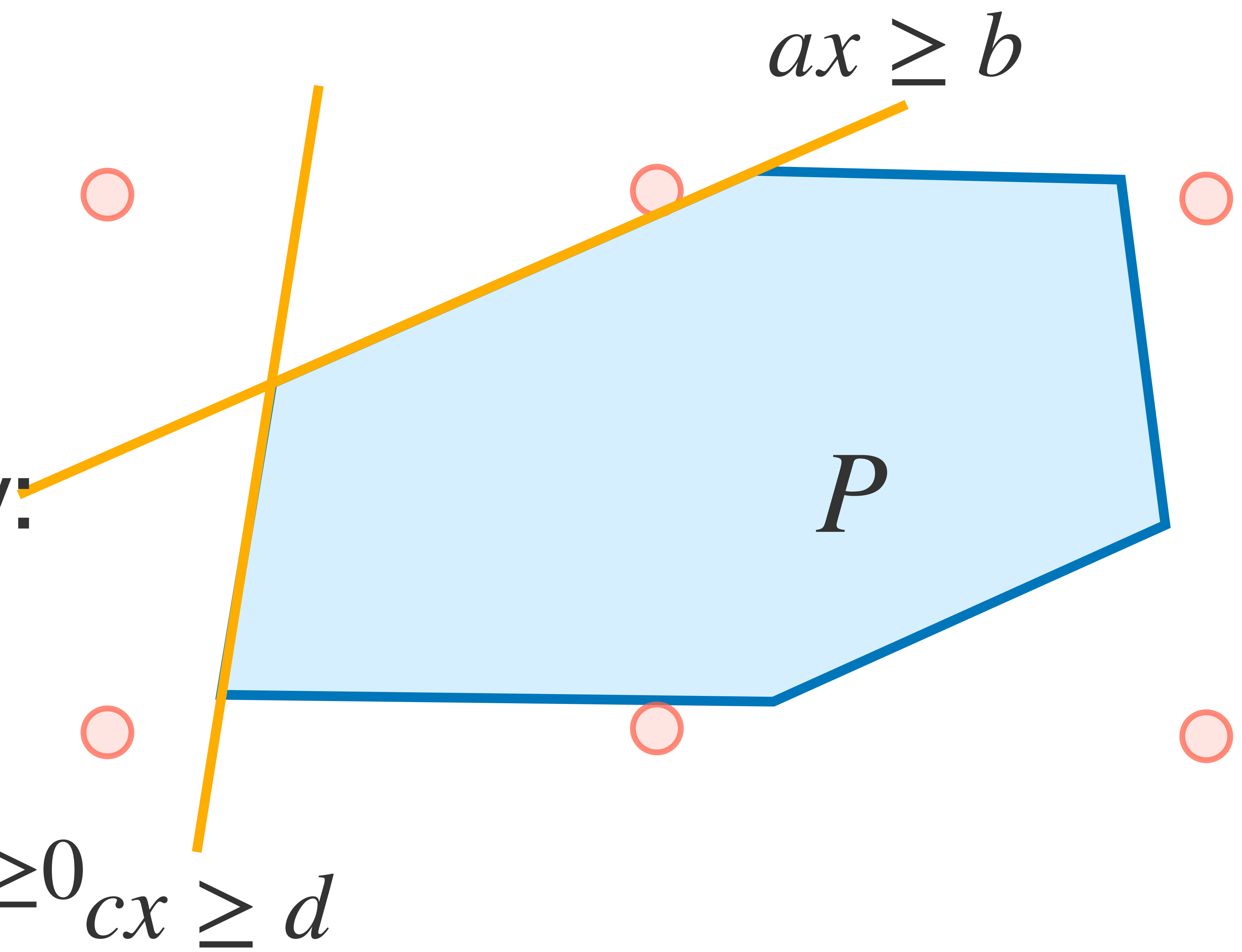
$\rightarrow$ **Prove** this fact using cutting planes!

<u>Rules</u>

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$



Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

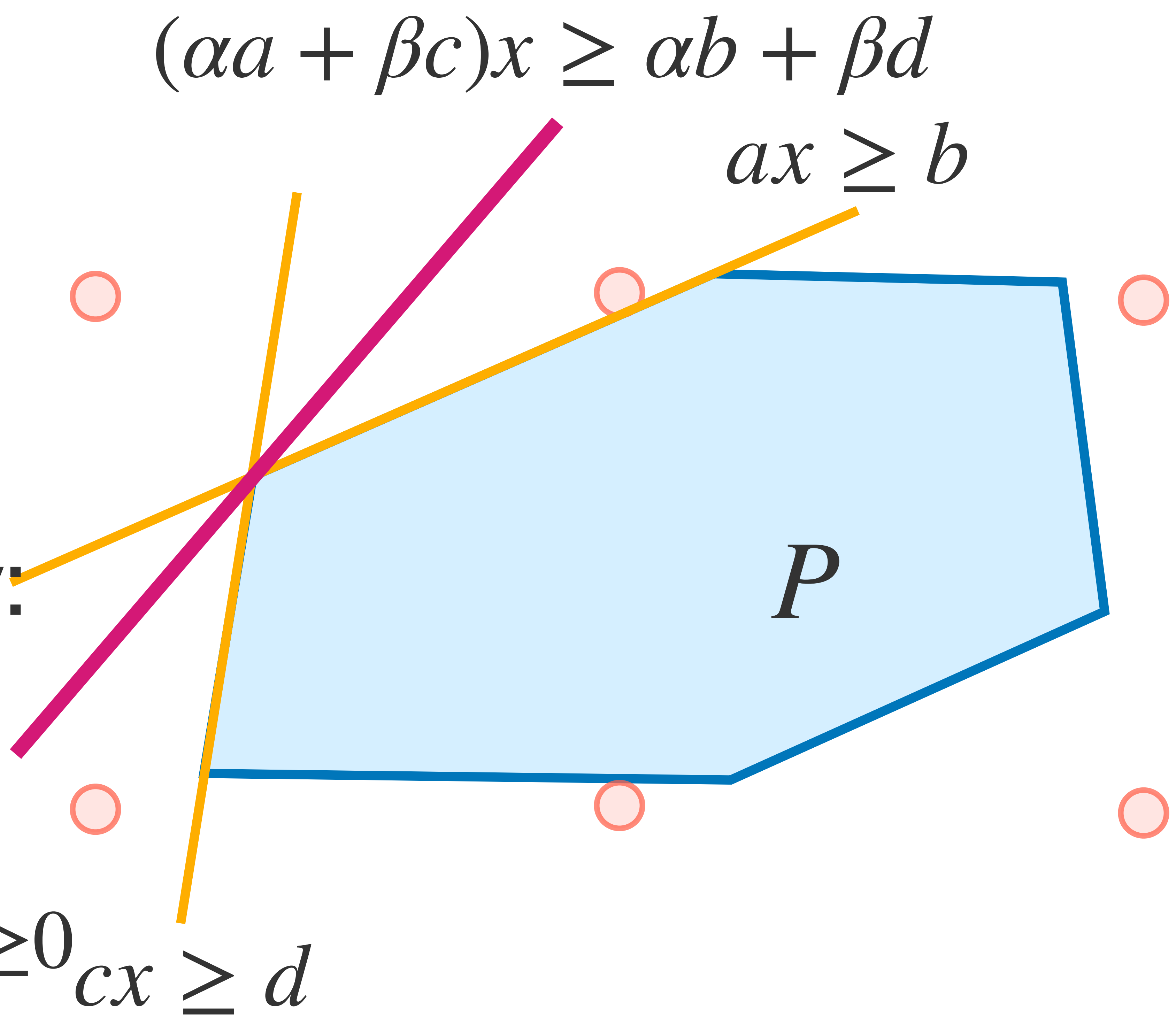
$\rightarrow$ **Prove** this fact using cutting planes!

Rules

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$



Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!

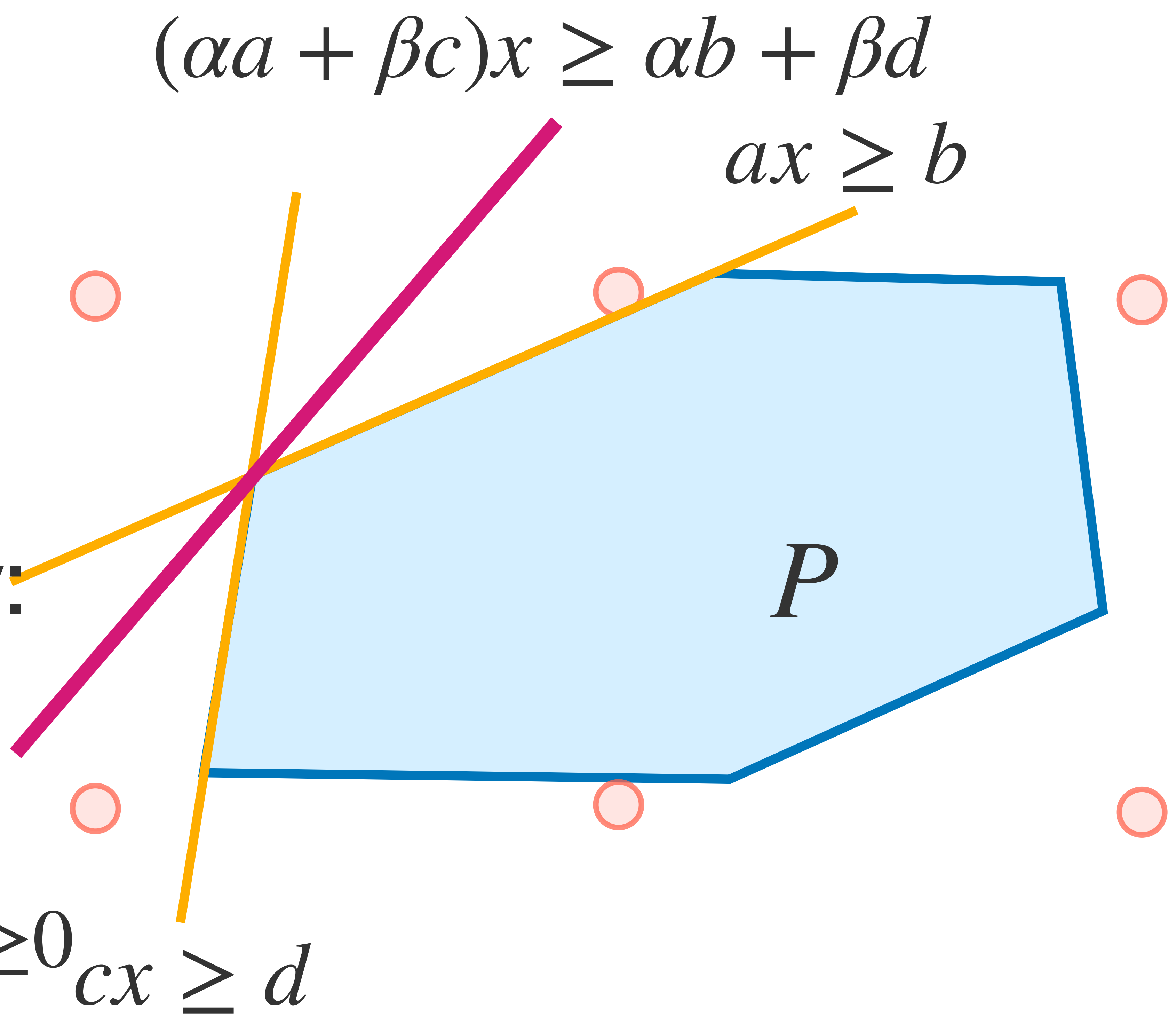
Rules

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

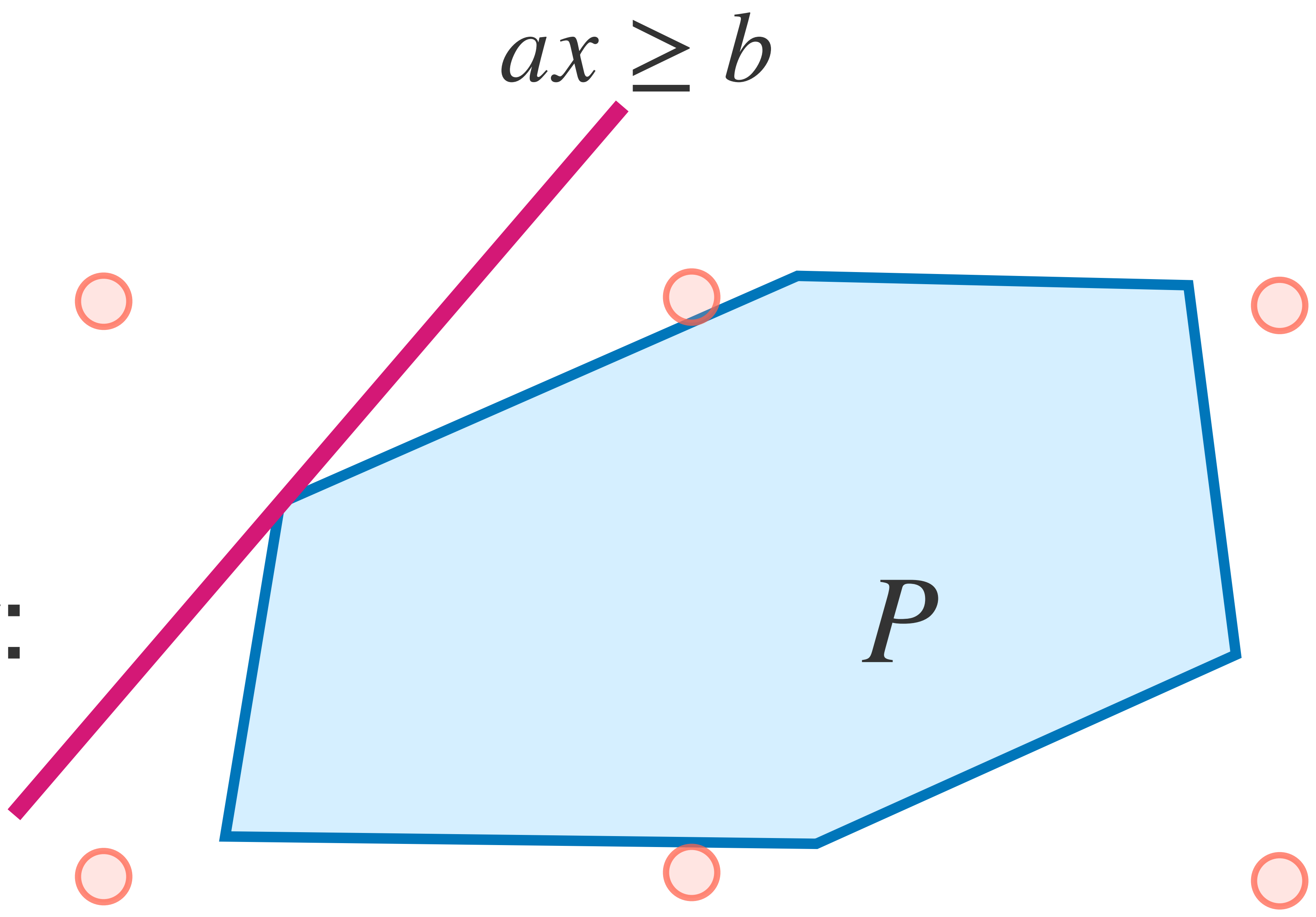
Preserves **all** points in P



Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!



<u>Rules</u>

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

- **Cut:**

$$\frac{ax \geq b}{(a/d)x \geq \lceil b/d \rceil}, \text{ if } d \in \mathbb{Z}^{\geq 0} \text{ divides } a$$

Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!

Rules

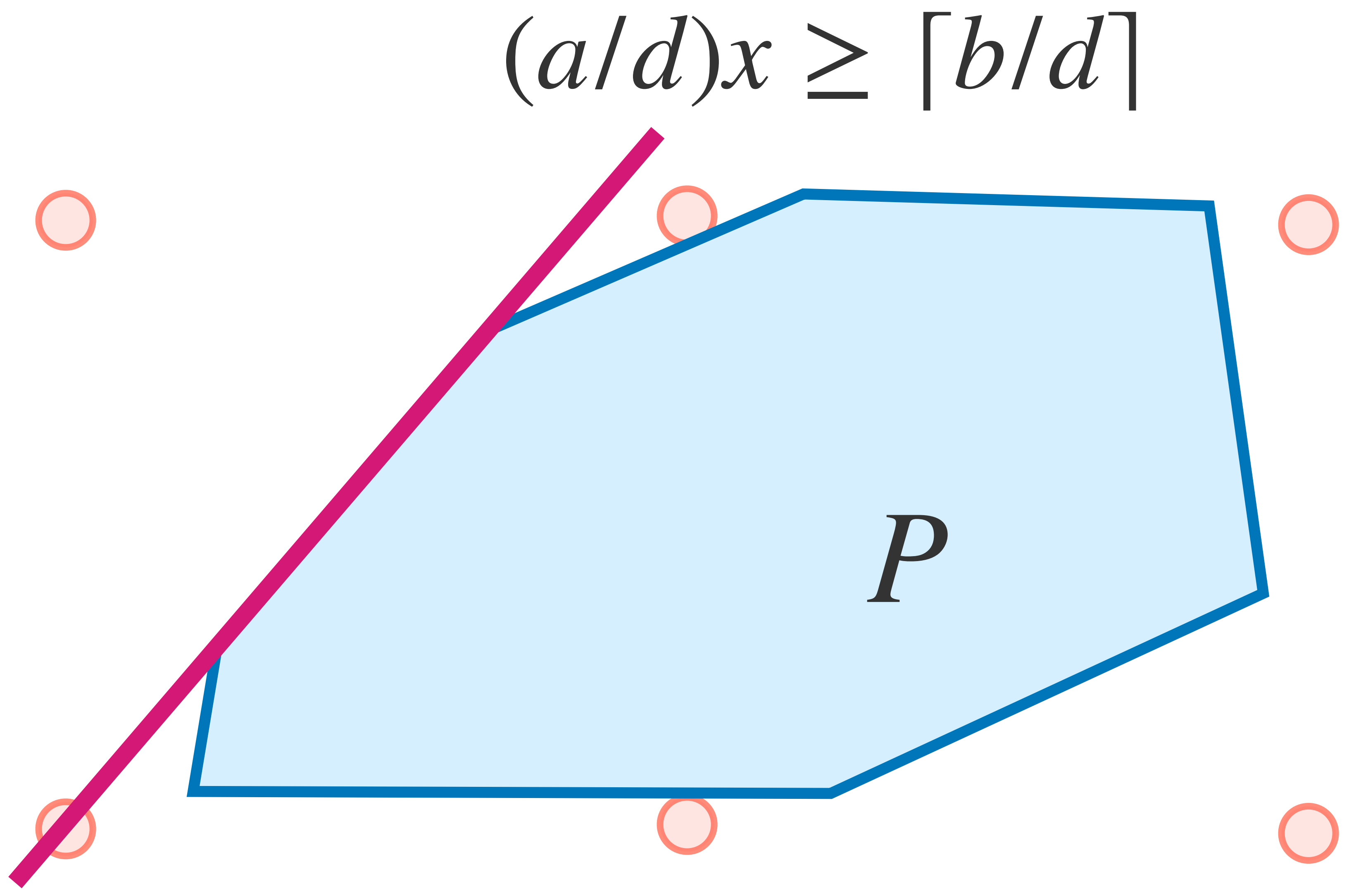
Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

- **Cut:**

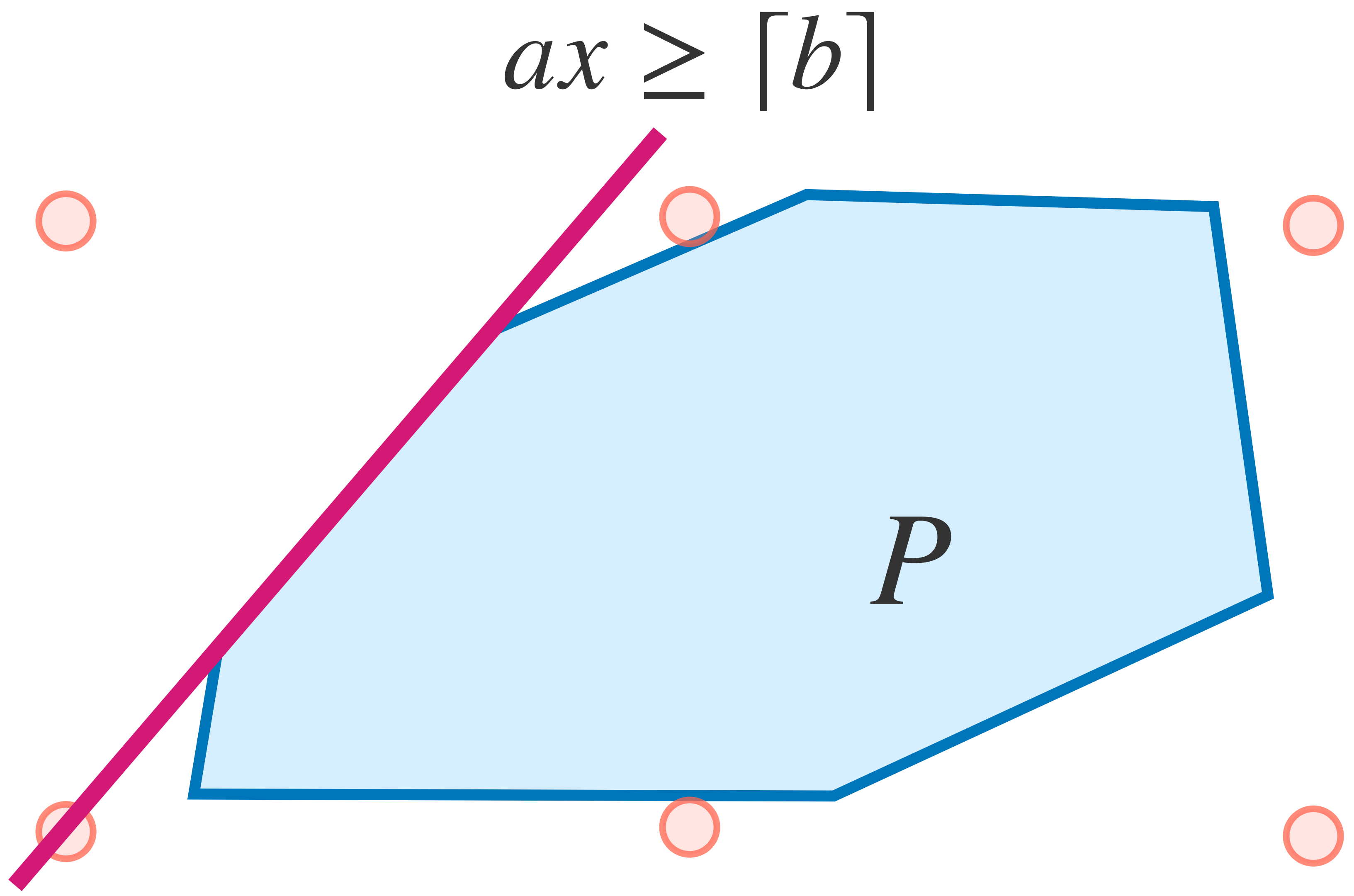
$$\frac{ax \geq b}{(a/d)x \geq \lceil b/d \rceil}, \text{ if } d \in \mathbb{Z}^{\geq 0} \text{ divides } a$$



Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!



Rules

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

- **Cut:**

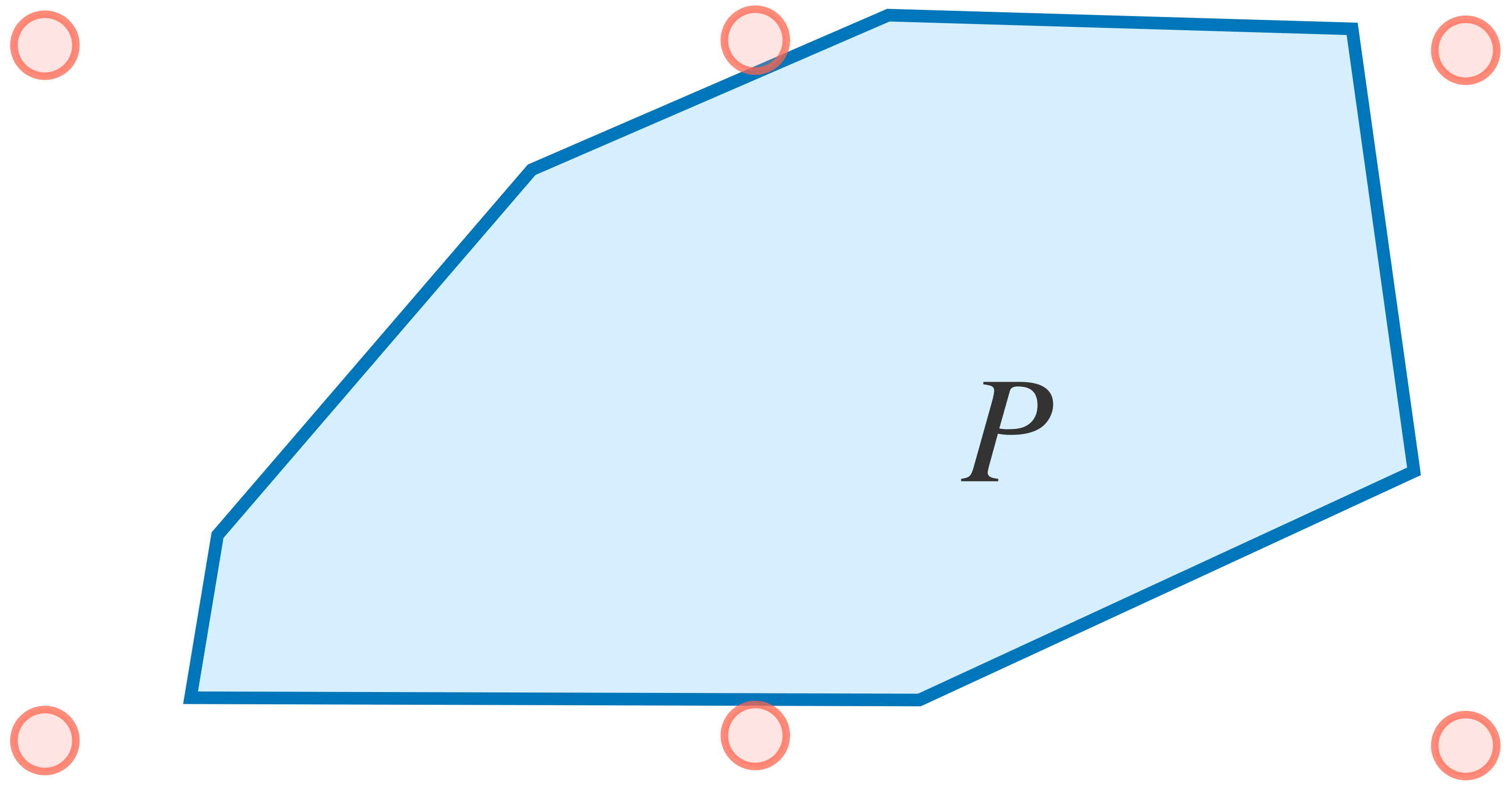
$$\frac{ax \geq b}{(a/d)x \geq \lceil b/d \rceil},$$ if $d \in \mathbb{Z}^{\geq 0}$ divides a

Preserves **integer** points in P

Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!



Rules

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

- **Cut:**

$$\frac{ax \geq b}{(a/d)x \geq \lceil b/d \rceil}, \text{ if } d \in \mathbb{Z}^{\geq 0} \text{ divides } a$$

Cutting Planes Proof

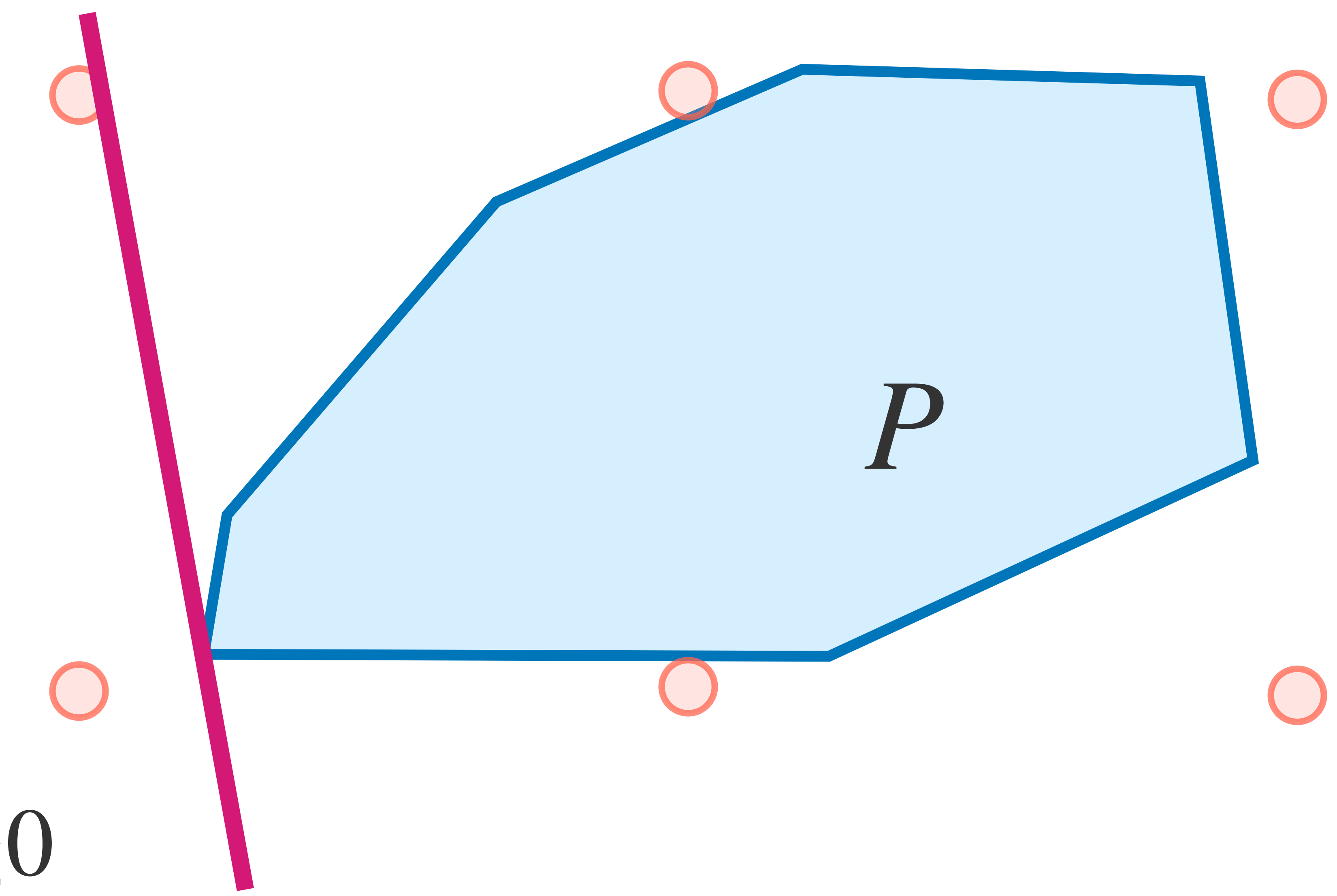
Derivation of $0 \geq 1$ from $Ax \geq b$

- equivalently, the **empty polytope**

Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!



Rules

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b, \; cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

- **Cut:**

$$\frac{ax \geq b}{(a/d)x \geq \lceil b/d \rceil}, \text{ if } d \in \mathbb{Z}^{\geq 0} \text{ divides } a$$

Cutting Planes Proof

Derivation of $0 \geq 1$ from $Ax \geq b$

- equivalently, the **empty polytope**

Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!

Rules

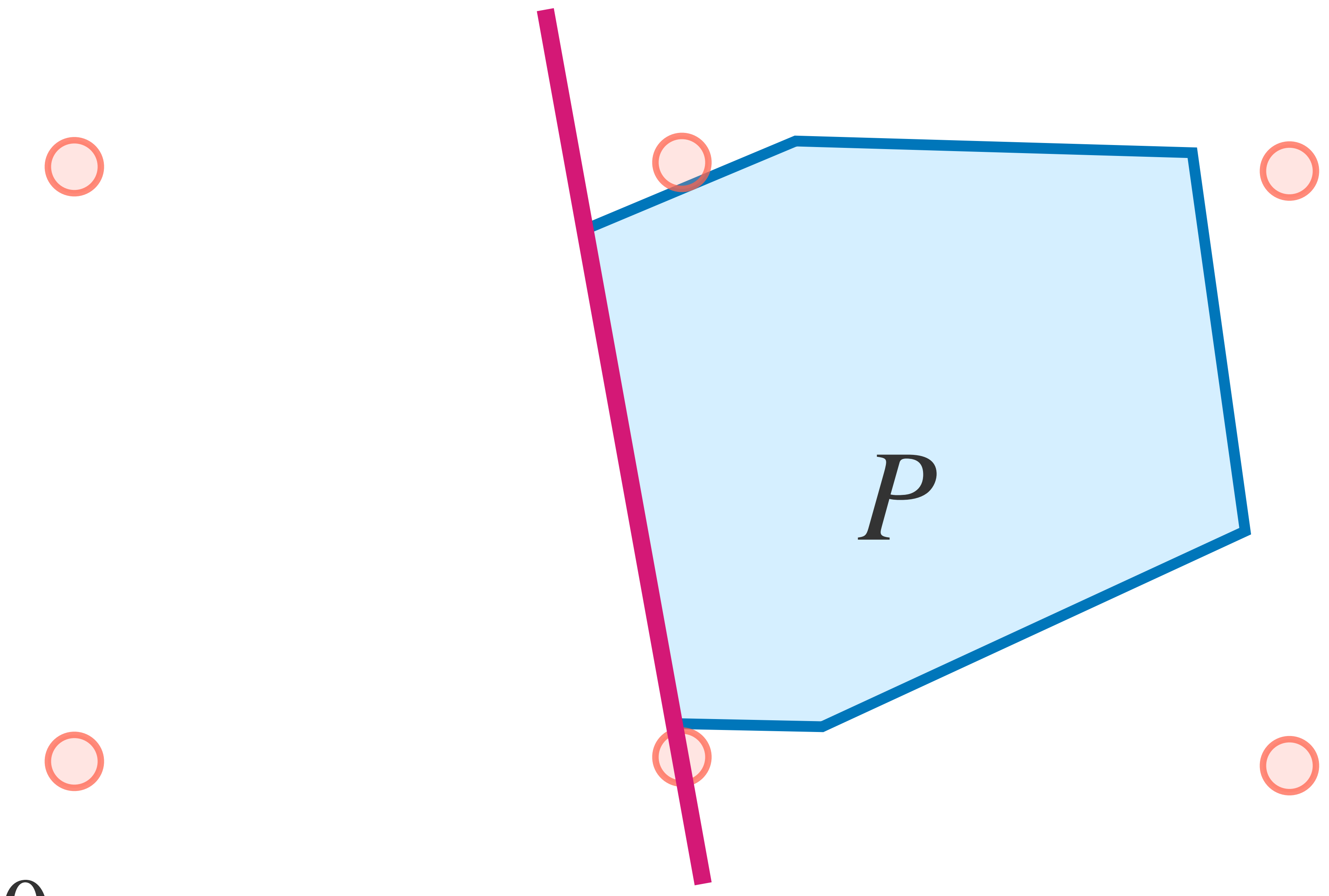
Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

- **Cut:**

$$\frac{ax \geq b}{(a/d)x \geq \lceil b/d \rceil}, \text{ if } d \in \mathbb{Z}^{\geq 0} \text{ divides } a$$



Cutting Planes Proof

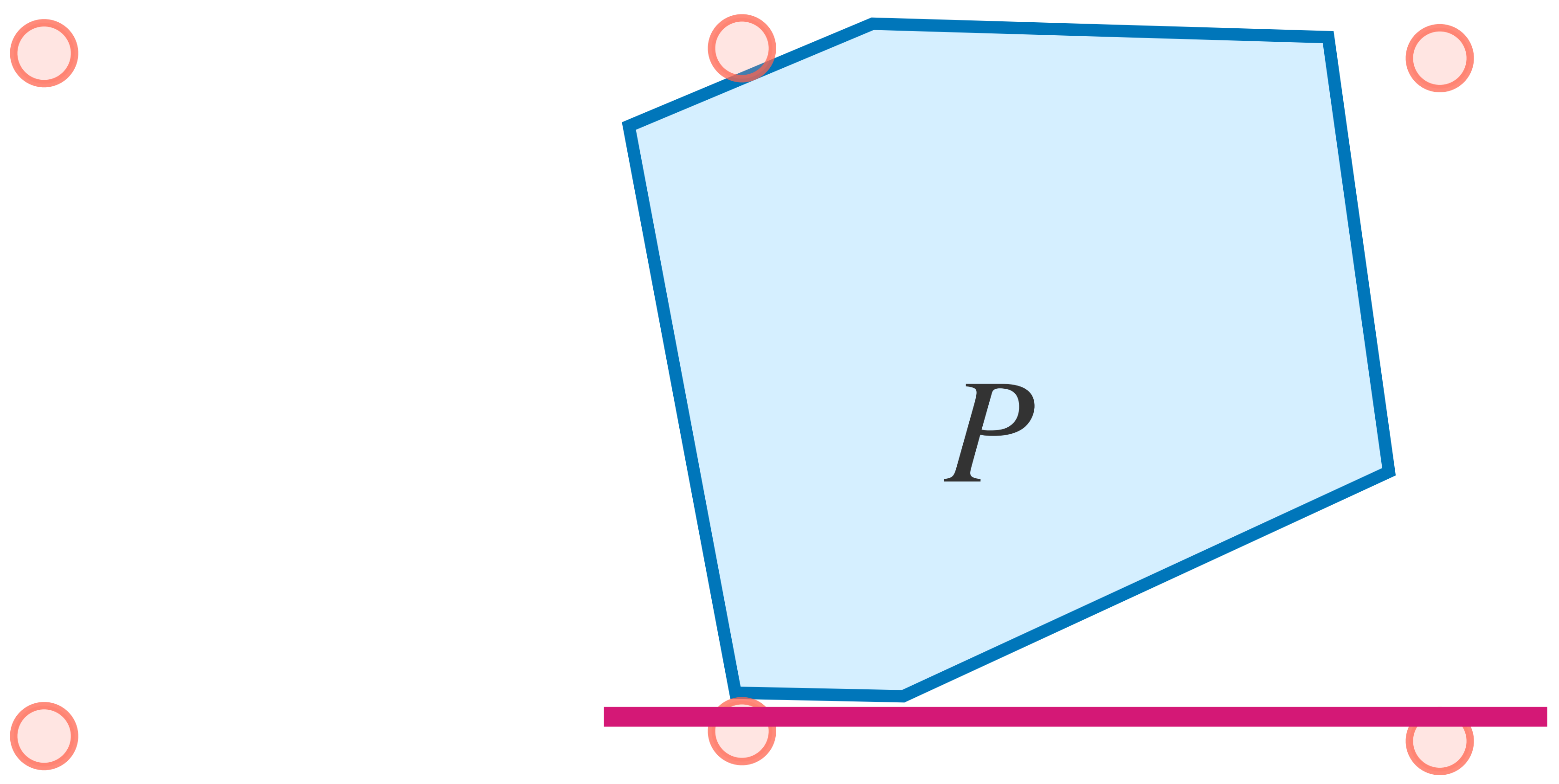
Derivation of $0 \geq 1$ from $Ax \geq b$

- equivalently, the **empty polytope**

Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!



Rules

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

- **Cut:**

$$\frac{ax \geq b}{(a/d)x \geq \lceil b/d \rceil}, \text{ if } d \in \mathbb{Z}^{\geq 0} \text{ divides } a$$

Cutting Planes Proof

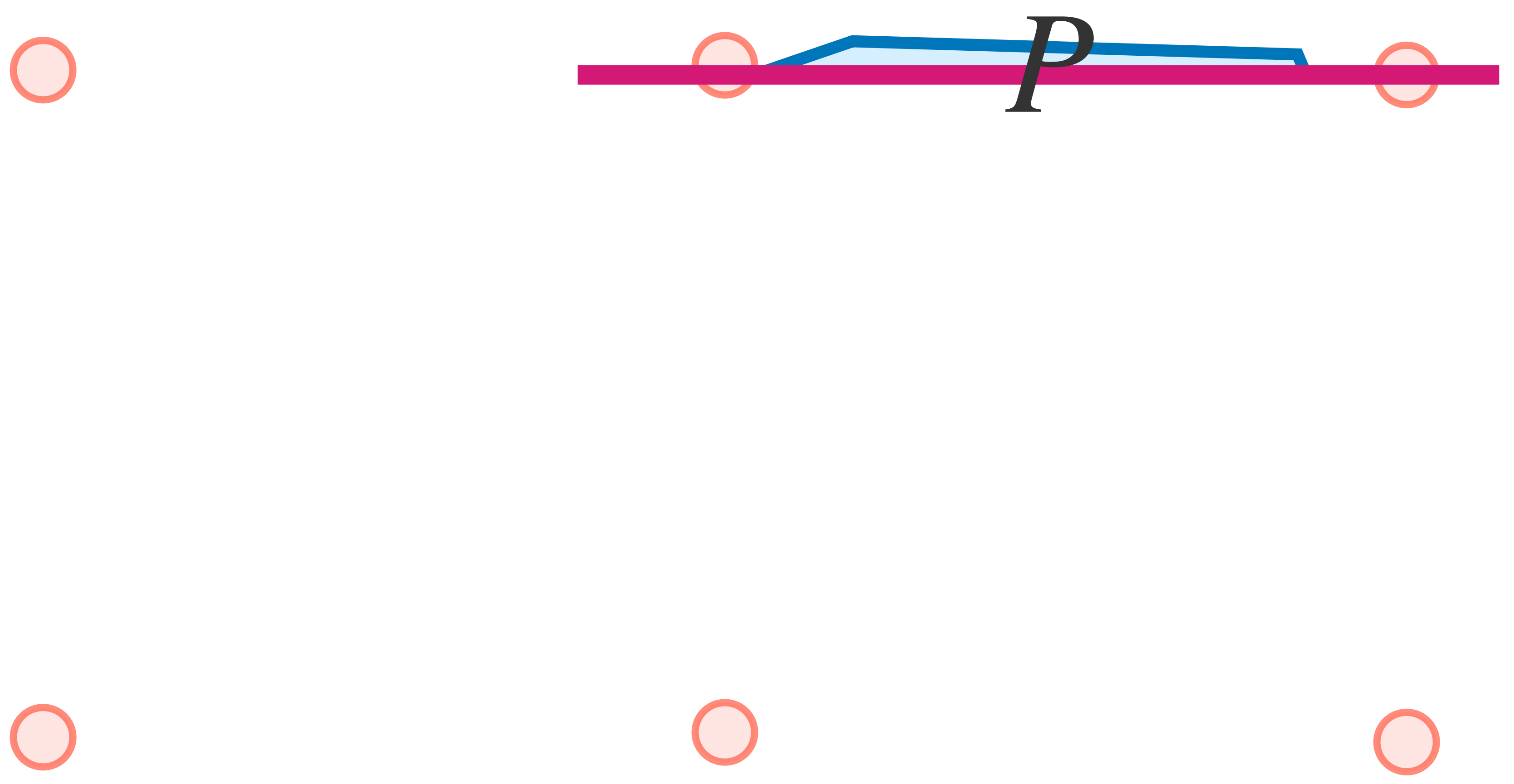
Derivation of $0 \geq 1$ from $Ax \geq b$

- equivalently, the **empty polytope**

Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!



Rules

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

- **Cut:**

$$\frac{ax \geq b}{(a/d)x \geq \lceil b/d \rceil}, \text{ if } d \in \mathbb{Z}^{\geq 0} \text{ divides } a$$

Cutting Planes Proof

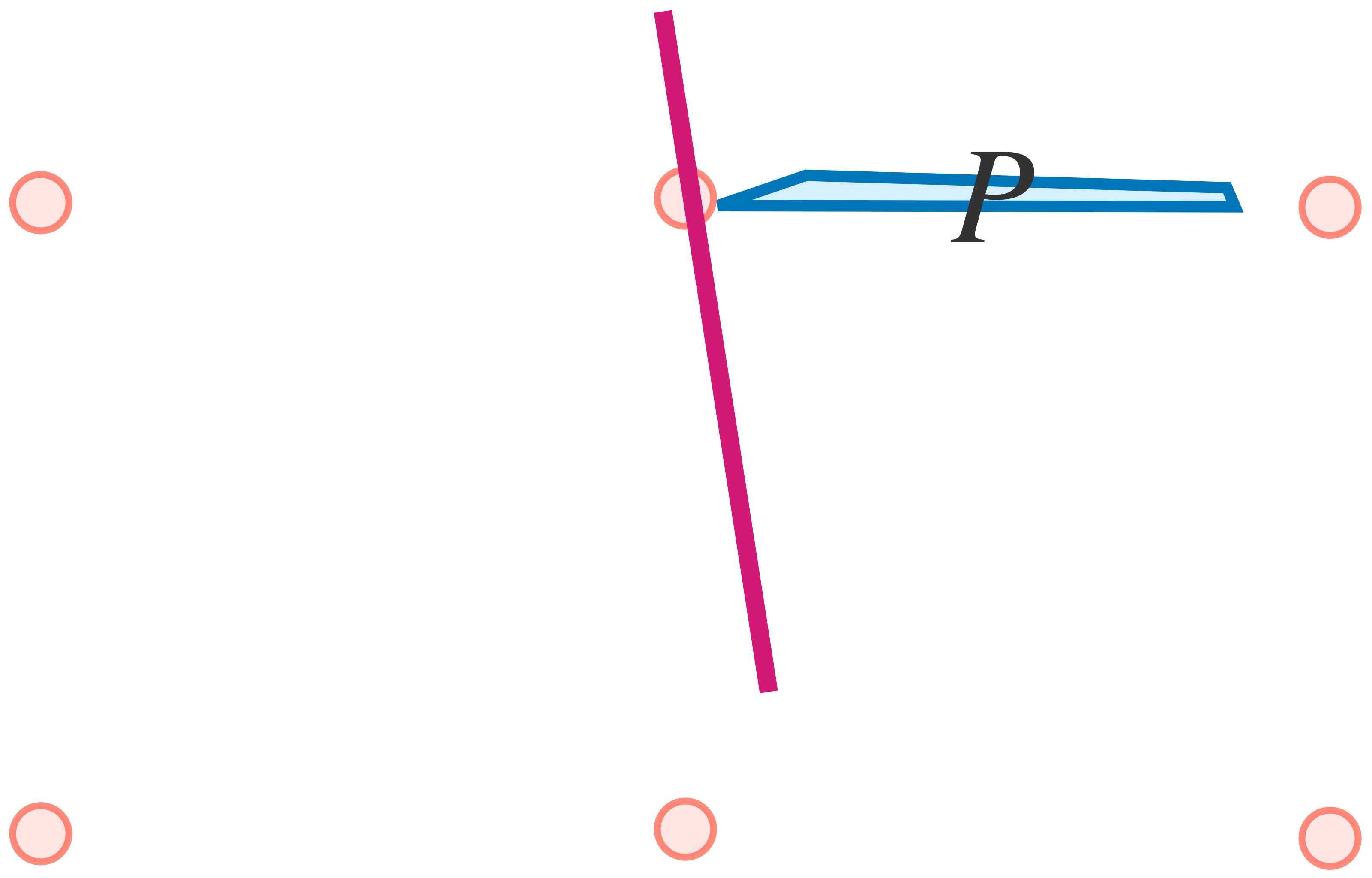
Derivation of $0 \geq 1$ from $Ax \geq b$

- equivalently, the **empty polytope**

Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!



Rules

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

- **Cut:**

$$\frac{ax \geq b}{(a/d)x \geq \lceil b/d \rceil}, \text{ if } d \in \mathbb{Z}^{\geq 0} \text{ divides } a$$

Cutting Planes Proof

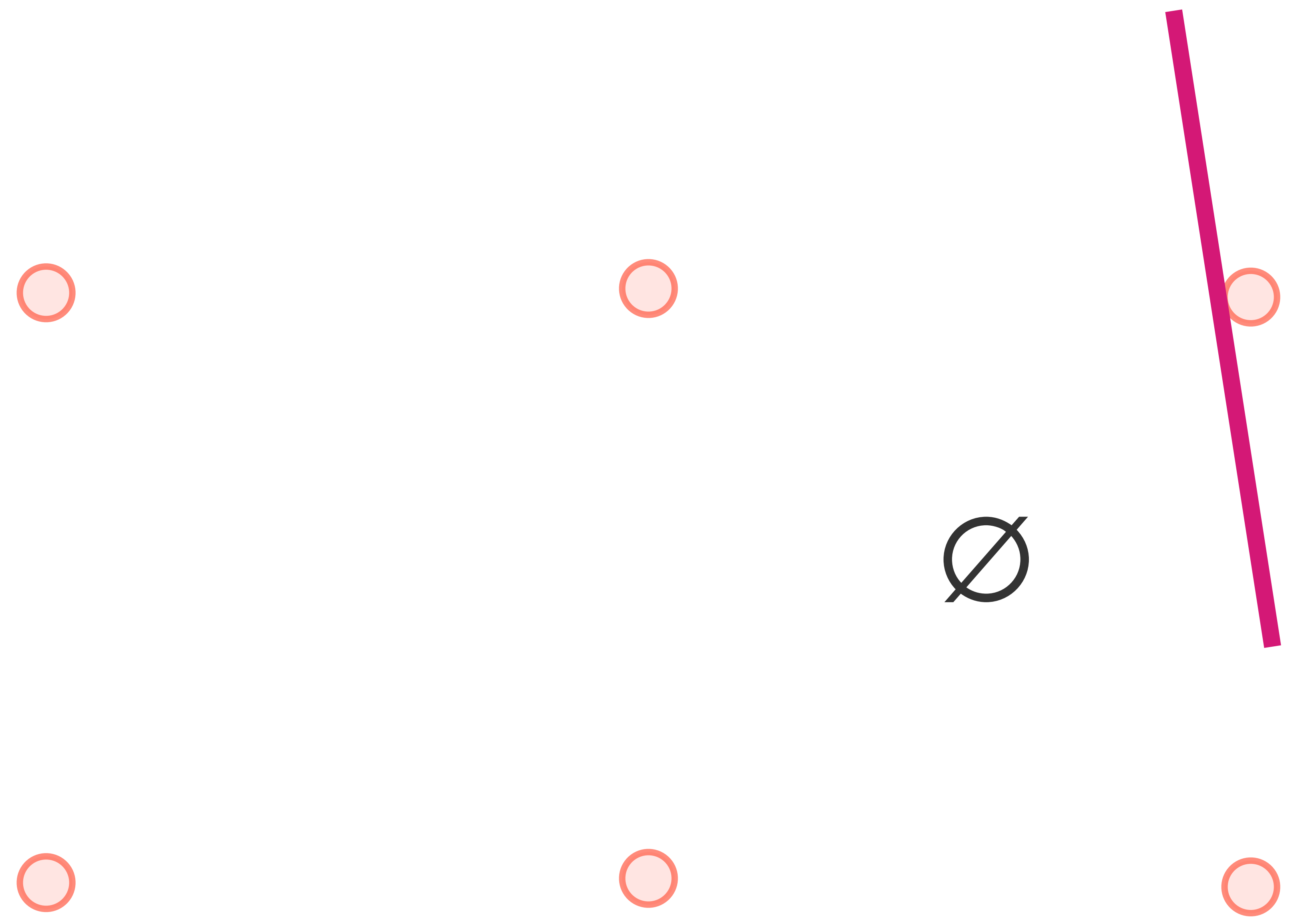
Derivation of $0 \geq 1$ from $Ax \geq b$

- equivalently, the **empty polytope**

Cutting Planes Proofs

Suppose $Ax \geq b$ has **no integer solutions**

$\rightarrow$ **Prove** this fact using cutting planes!



Rules

Derive new integer-inequalities from old ones by:

- **Non-negative linear Combination:**

$$\frac{ax \geq b,\ cx \geq d}{(\alpha a + \beta c)x \geq \alpha b + \beta d}, \quad \alpha, \beta \in \mathbb{Z}^{\geq 0}$$

- **Cut:**

$$\frac{ax \geq b}{(a/d)x \geq \lceil b/d \rceil}, \text{ if } d \in \mathbb{Z}^{\geq 0} \text{ divides } a$$

Cutting Planes Proof

Derivation of $0 \geq 1$ from $Ax \geq b$

- equivalently, the **empty polytope**

Proving CNF Formulas

In order to talk about CP as a proof system, we need to encode CNF formulas as a system of linear inequalities — easy because integer programming is NP-complete!

Proving CNF Formulas

In order to talk about CP as a proof system, we need to encode CNF formulas as a system of linear inequalities — easy because integer programming is NP-complete!

1. Convert clauses into inequalities:

$$x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \quad \rightarrow \quad x_1 + (1 - x_2) + (1 - x_3) + x_4 \geq 1$$

Proving CNF Formulas

In order to talk about CP as a proof system, we need to encode CNF formulas as a system of linear inequalities — easy because integer programming is NP-complete!

1. Convert clauses into inequalities:

$$x_1 \lor \neg x_2 \lor \neg x_3 \lor x_4 \quad \rightarrow \quad x_1 + (1 - x_2) + (1 - x_3) + x_4 \geq 1$$

2. Add boolean constraints:

$$x_i \geq 0 \quad \text{and} \quad x_i \leq 1$$

Cutting Planes Proofs

Lower bounds on Cutting Planes proofs $\rightarrow$ lower bounds on the runtime of cutting planes algorithms

Cutting Planes Proofs

Lower bounds on Cutting Planes proofs $\rightarrow$ lower bounds on the runtime of cutting planes algorithms

- [Pudlak97] (also [BPR97] under restriction): Refuting that a graph has both a k-clique and a $(k-1)$-coloring requires exponential size Cutting Planes proofs

Cutting Planes Proofs

Lower bounds on Cutting Planes proofs $\rightarrow$ lower bounds on the runtime of cutting planes algorithms

- [Pudlak97] (also [BPR97] under restriction): Refuting that a graph has both a k-clique and a $(k-1)$-coloring requires exponential size Cutting Planes proofs
- [FPPR17, HP17]: Uniformly random CNF formulas require exponential size Cutting Planes refutations

Cutting Planes Proofs

Lower bounds on Cutting Planes proofs $\rightarrow$ lower bounds on the runtime of cutting planes algorithms

- [Pudlak97] (also [BPR97] under restriction): Refuting that a graph has both a k-clique and a $(k-1)$-coloring requires exponential size Cutting Planes proofs
- [FPPR17, HP17]: Uniformly random CNF formulas require exponential size Cutting Planes refutations

However… Cutting Planes does not capture modern algorithms for IP (branch-and-cut)

The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut**.

The Stabbing Planes Proof System

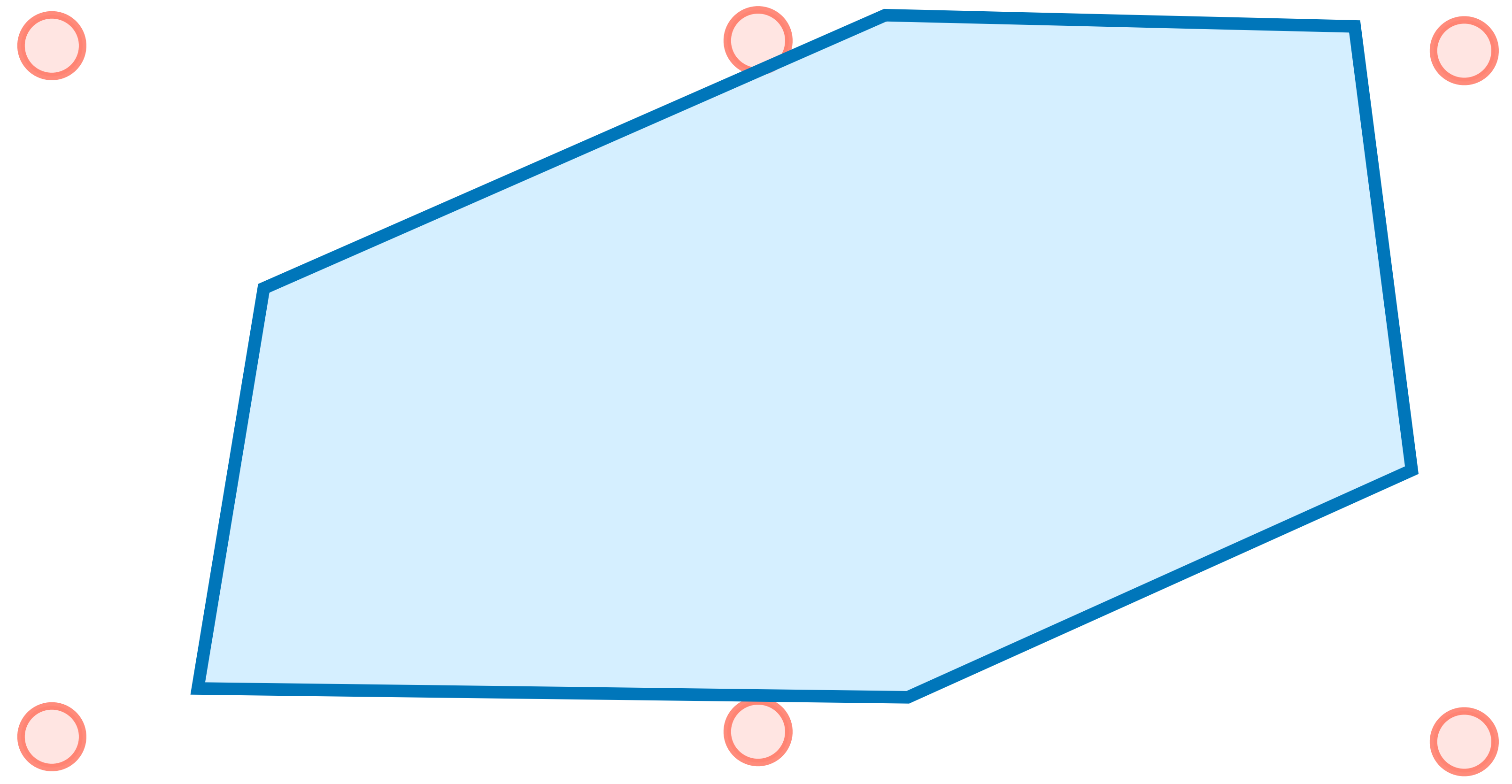
[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut**.

- DPLL querying integer linear inequalities!
- No Cutting Planes rule needed!

The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut**.

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \emptyset$.

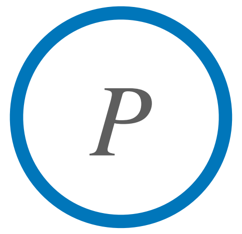


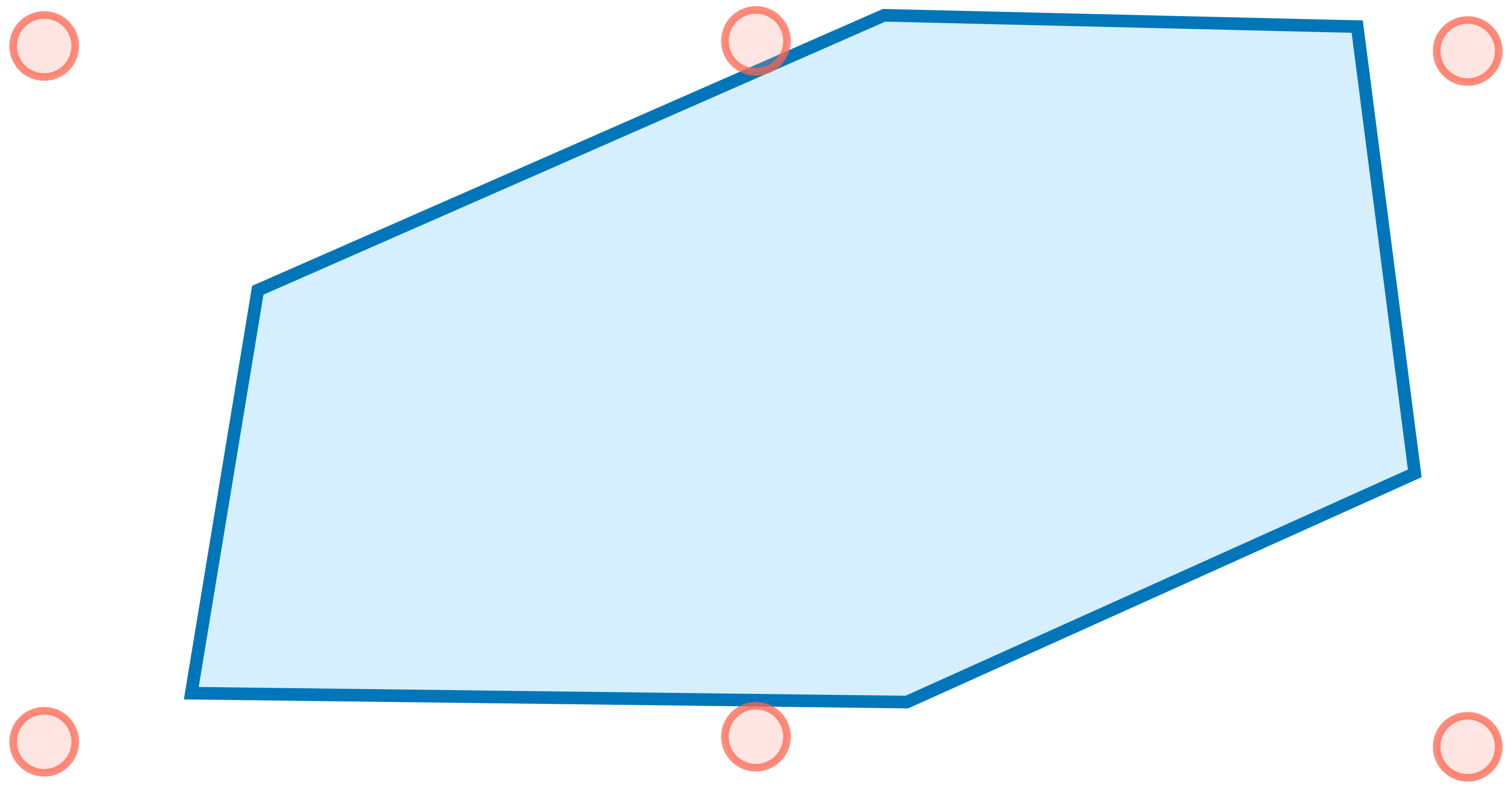
The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut**.

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \emptyset$.

<u>Rule</u>: query an arbitrary **integer linear inequality**



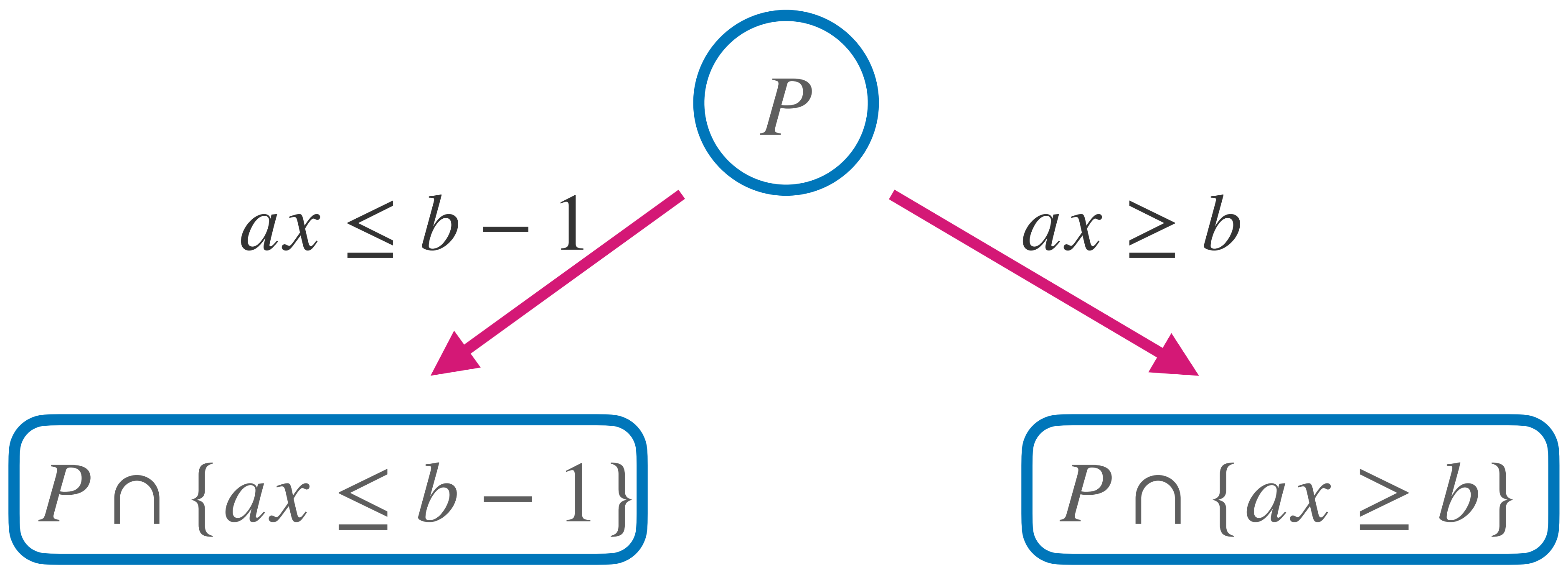


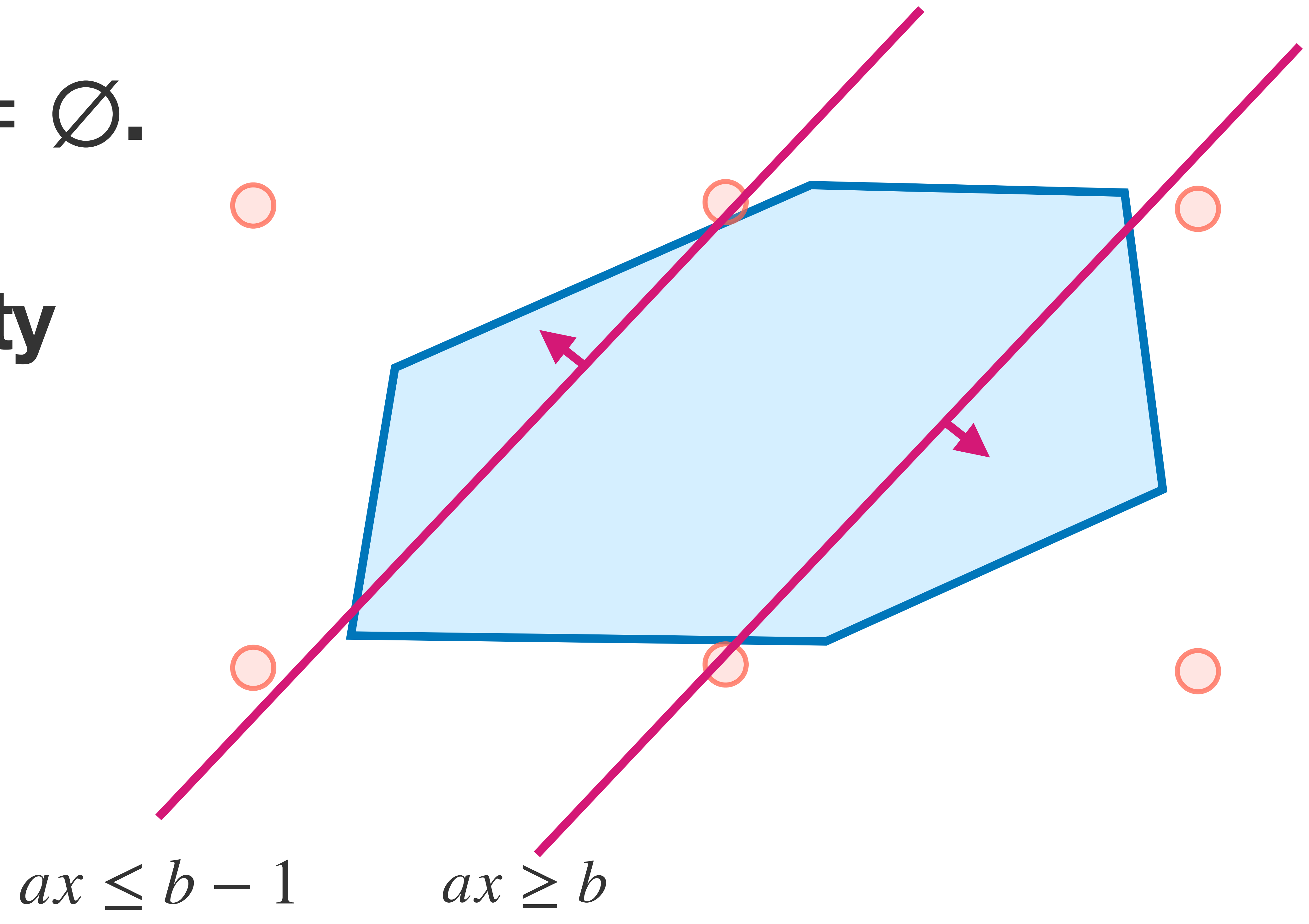
The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut.**

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \emptyset$.

<u>Rule</u>: query an arbitrary **integer linear inequality**



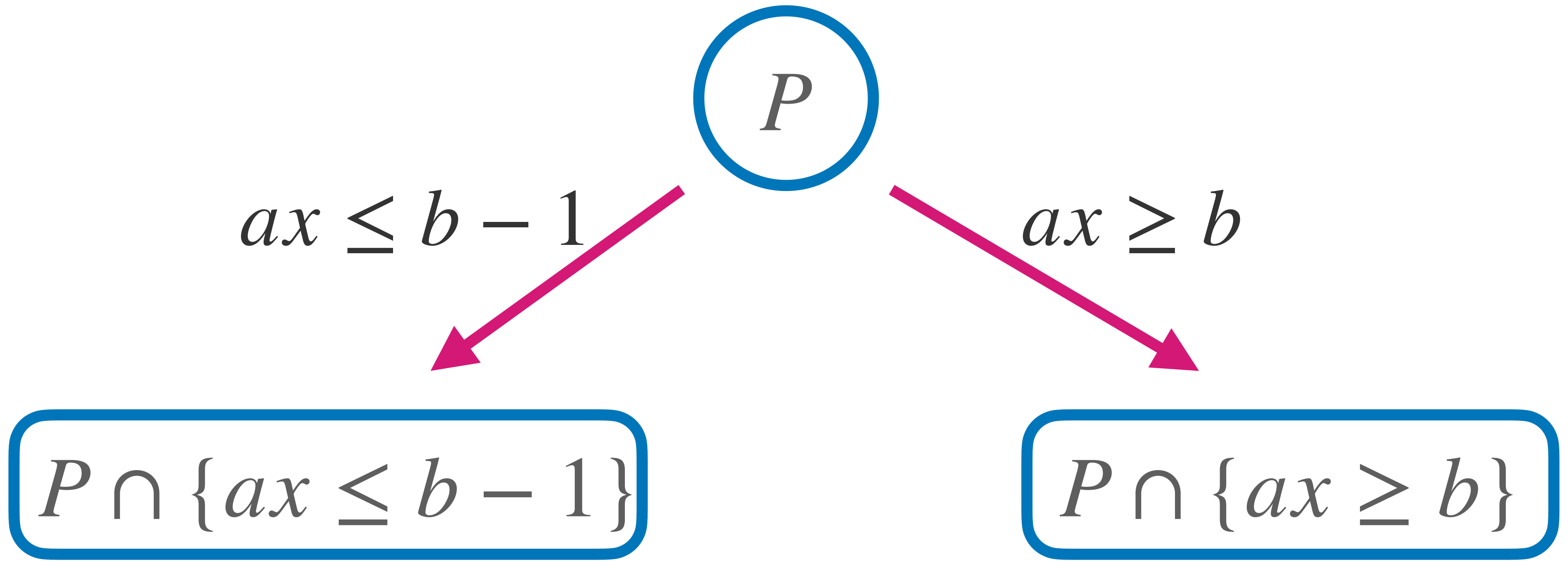


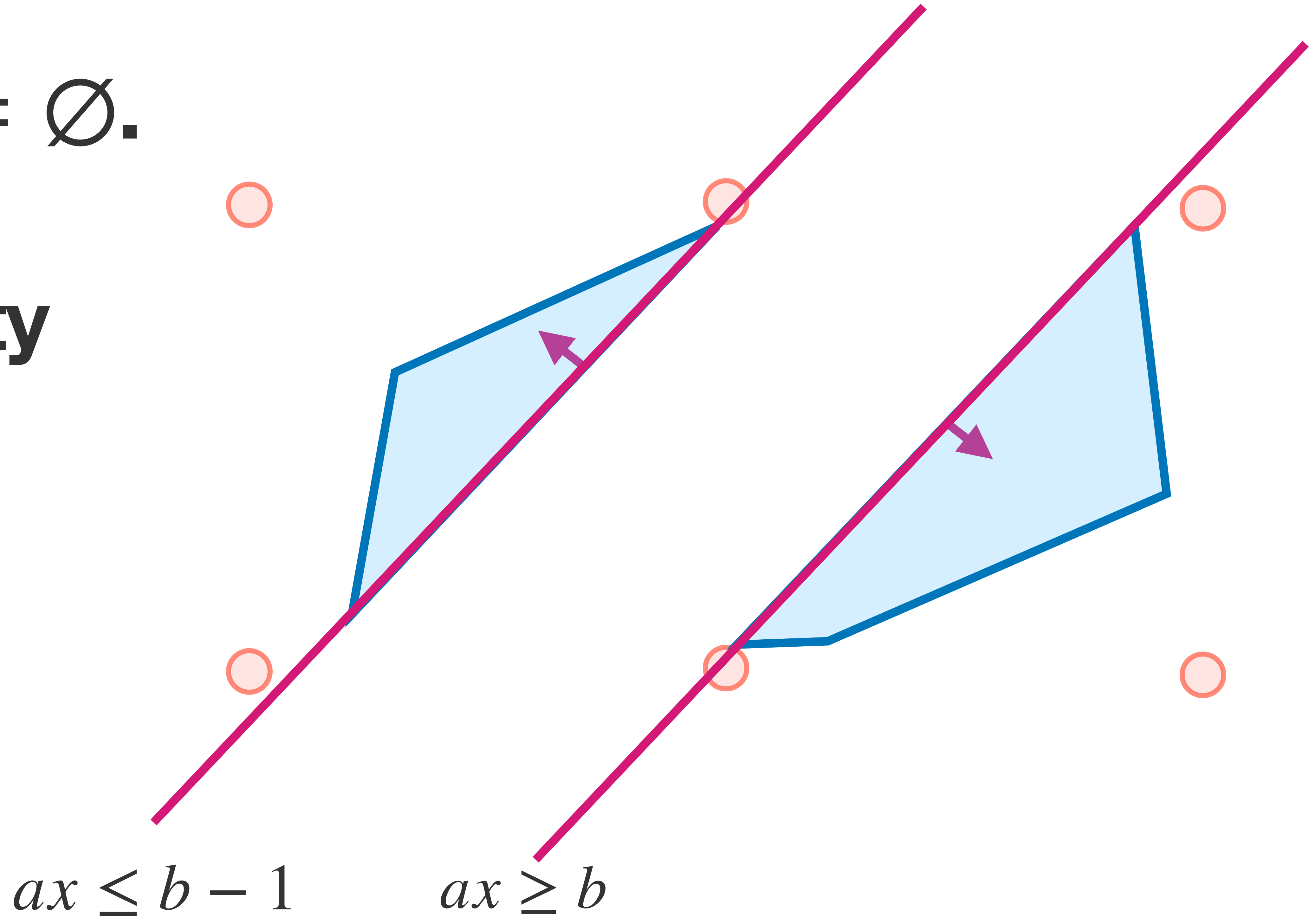
The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut.**

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \emptyset$.

<u>**Rule**</u>**:** query an arbitrary **integer linear inequality**



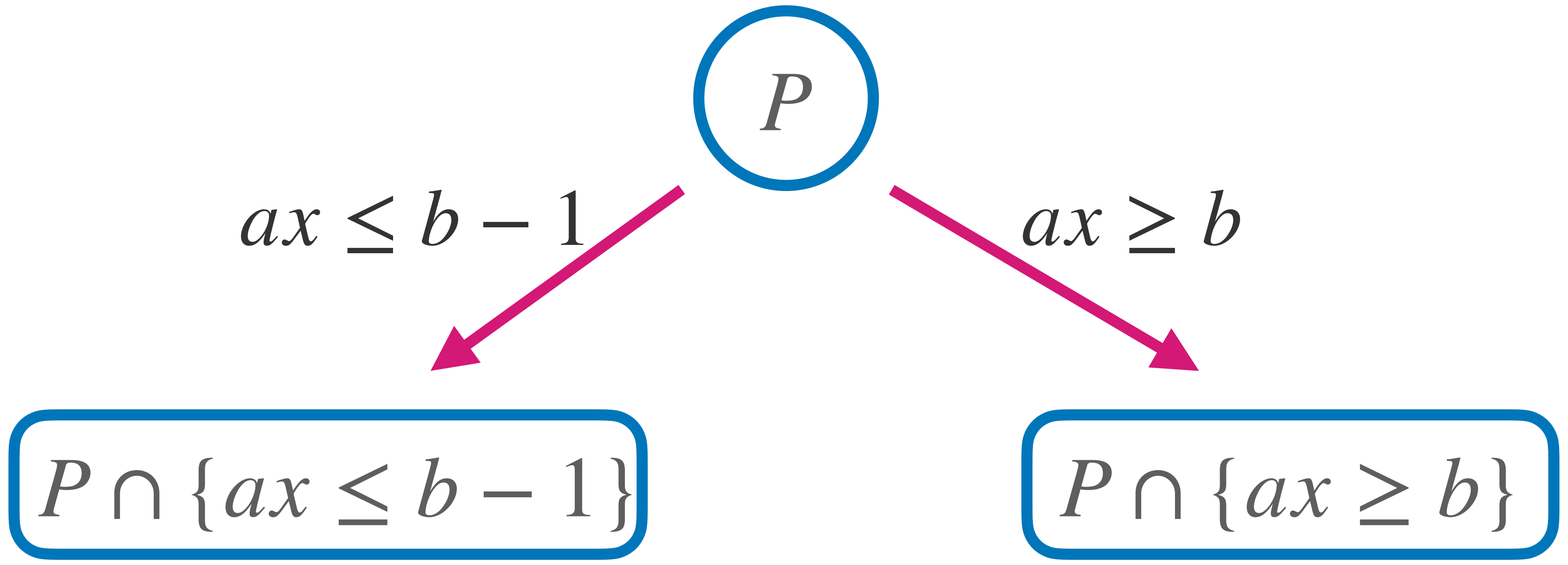


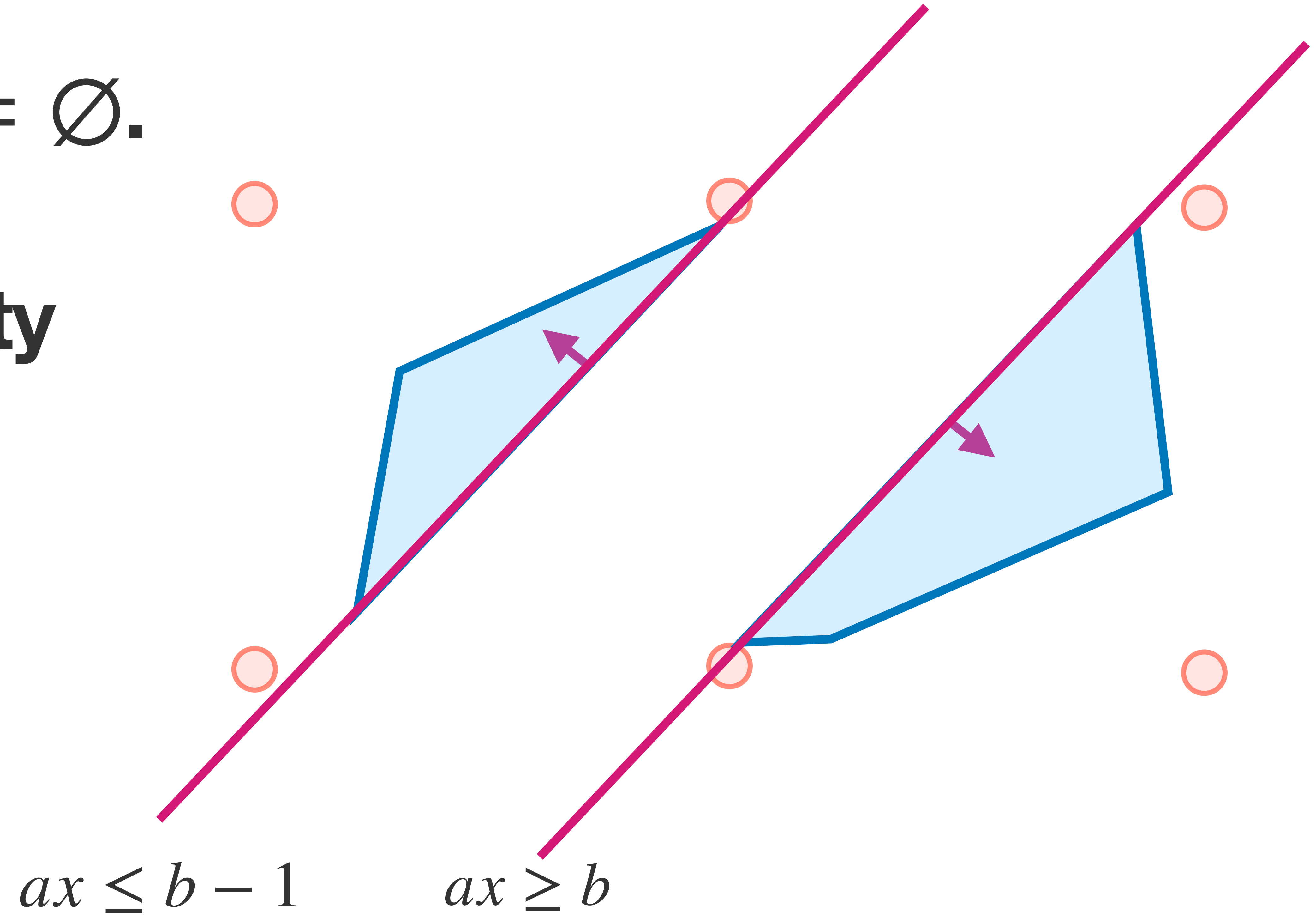
The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut.**

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \emptyset$.

<u>Rule</u>: query an arbitrary **integer linear inequality**





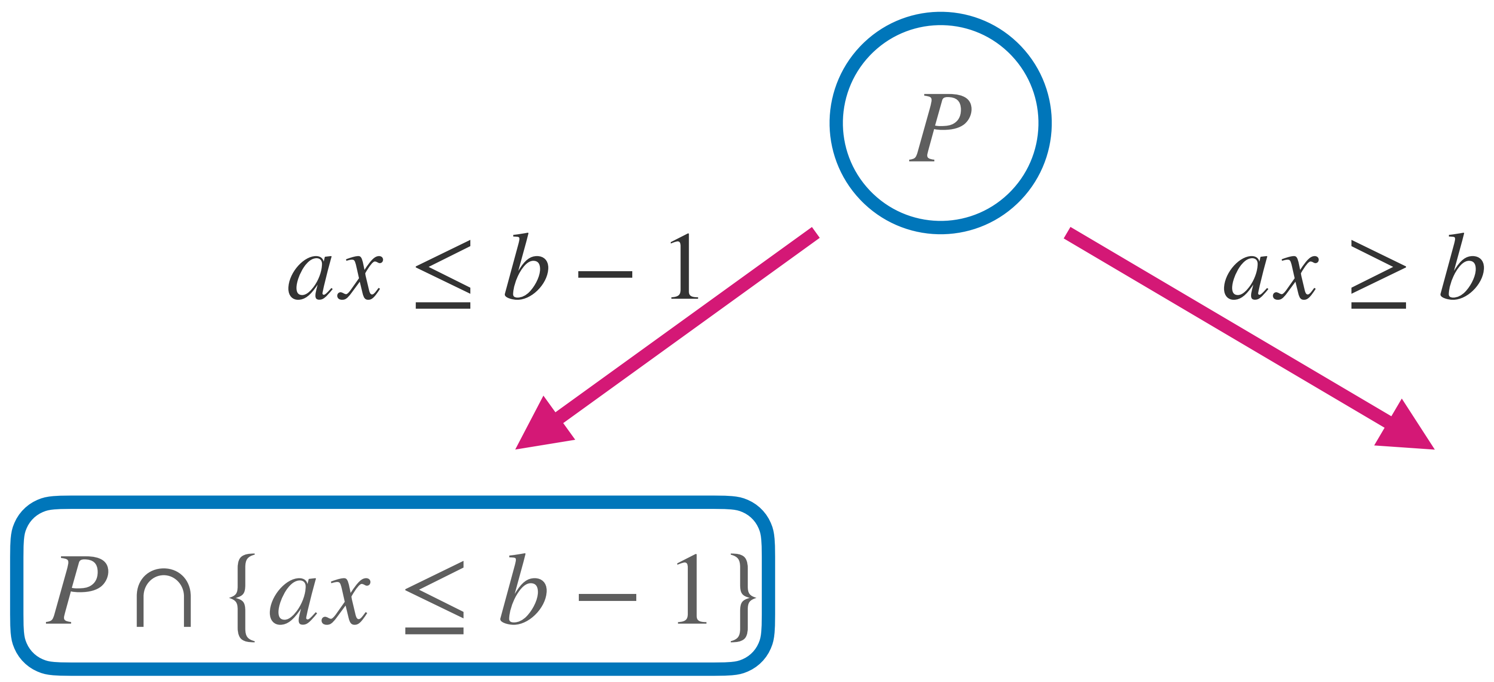
Because a, b are integral, **preserves** integer points!

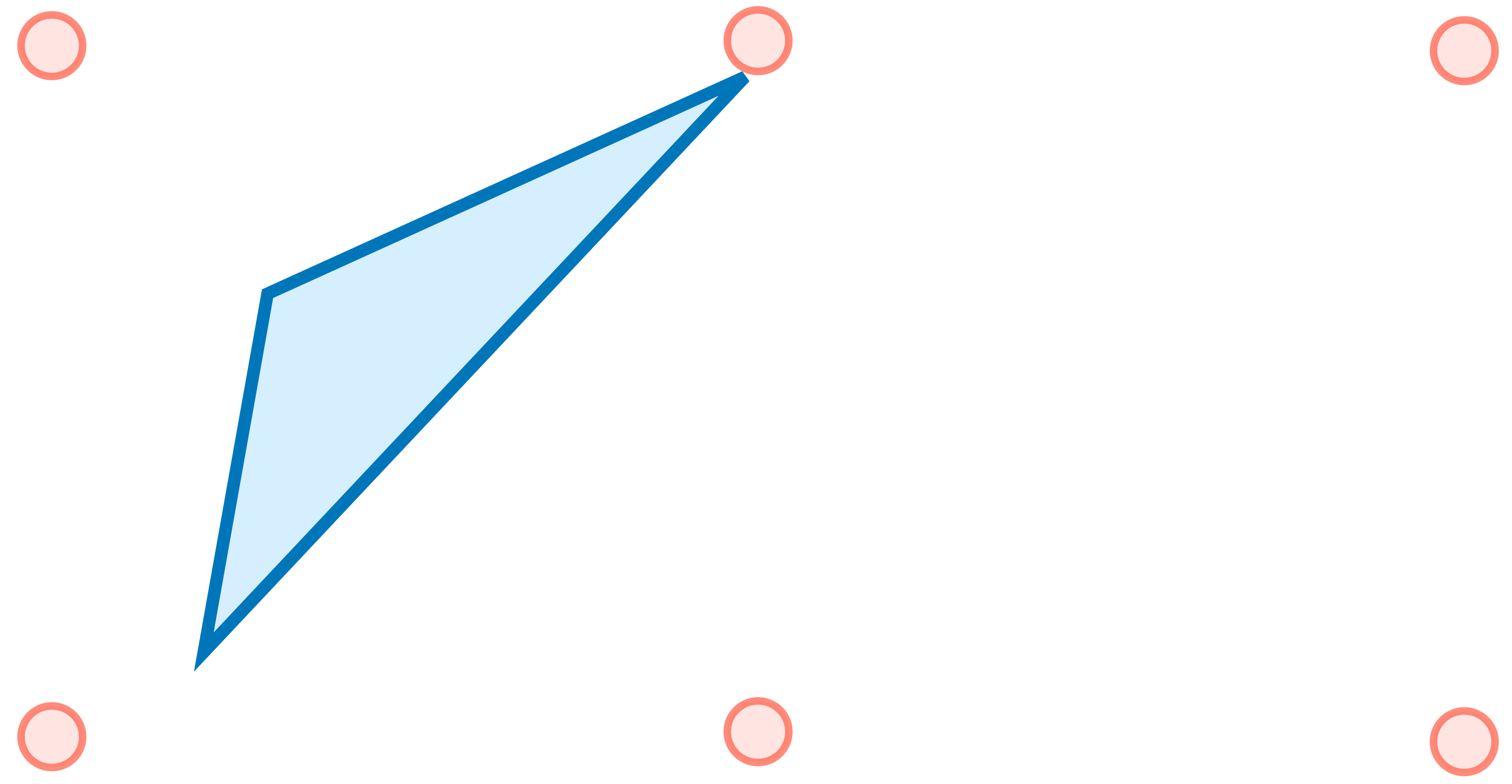
The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut**.

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \varnothing$.

<u>**Rule:**</u> query an arbitrary **integer linear inequality**

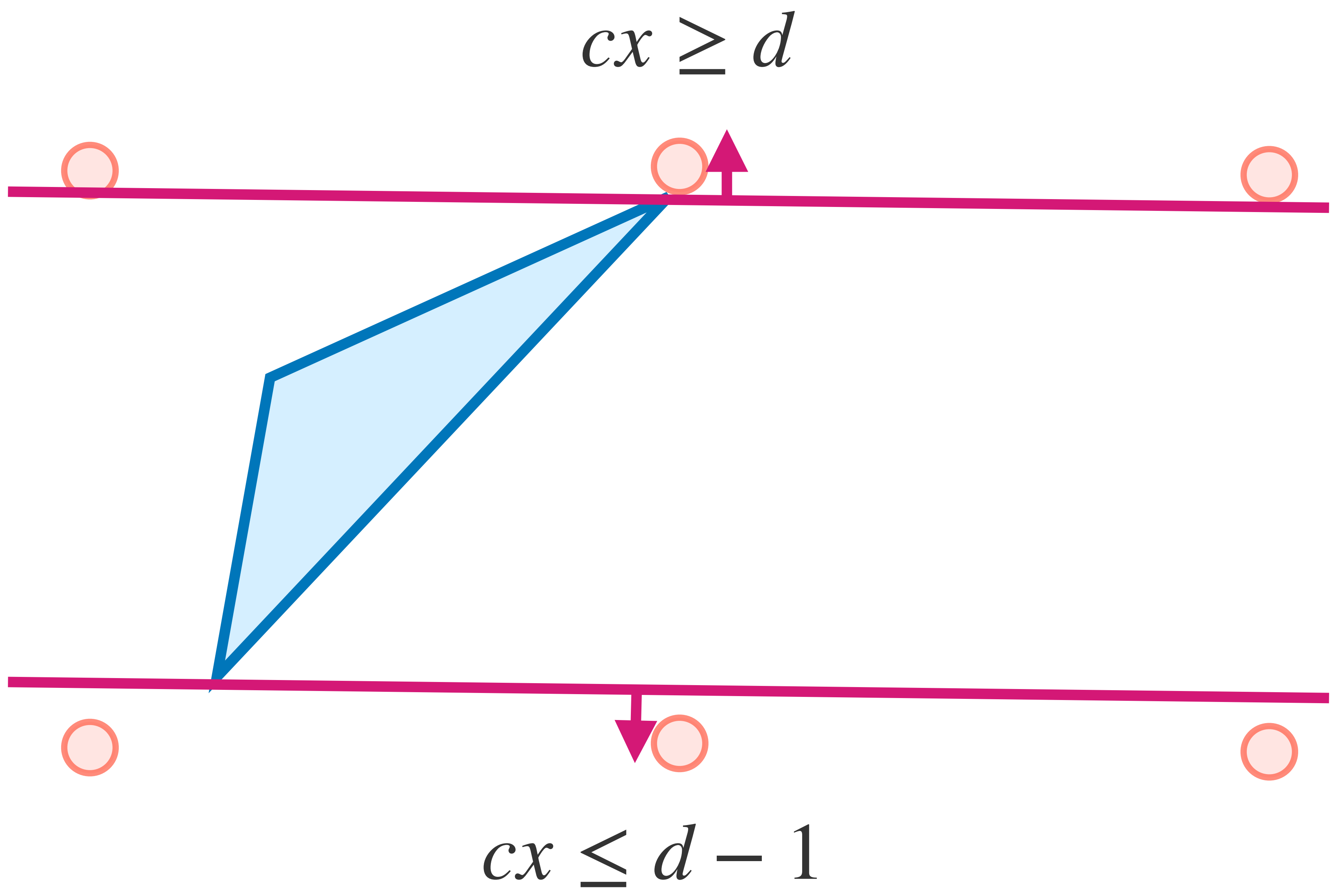




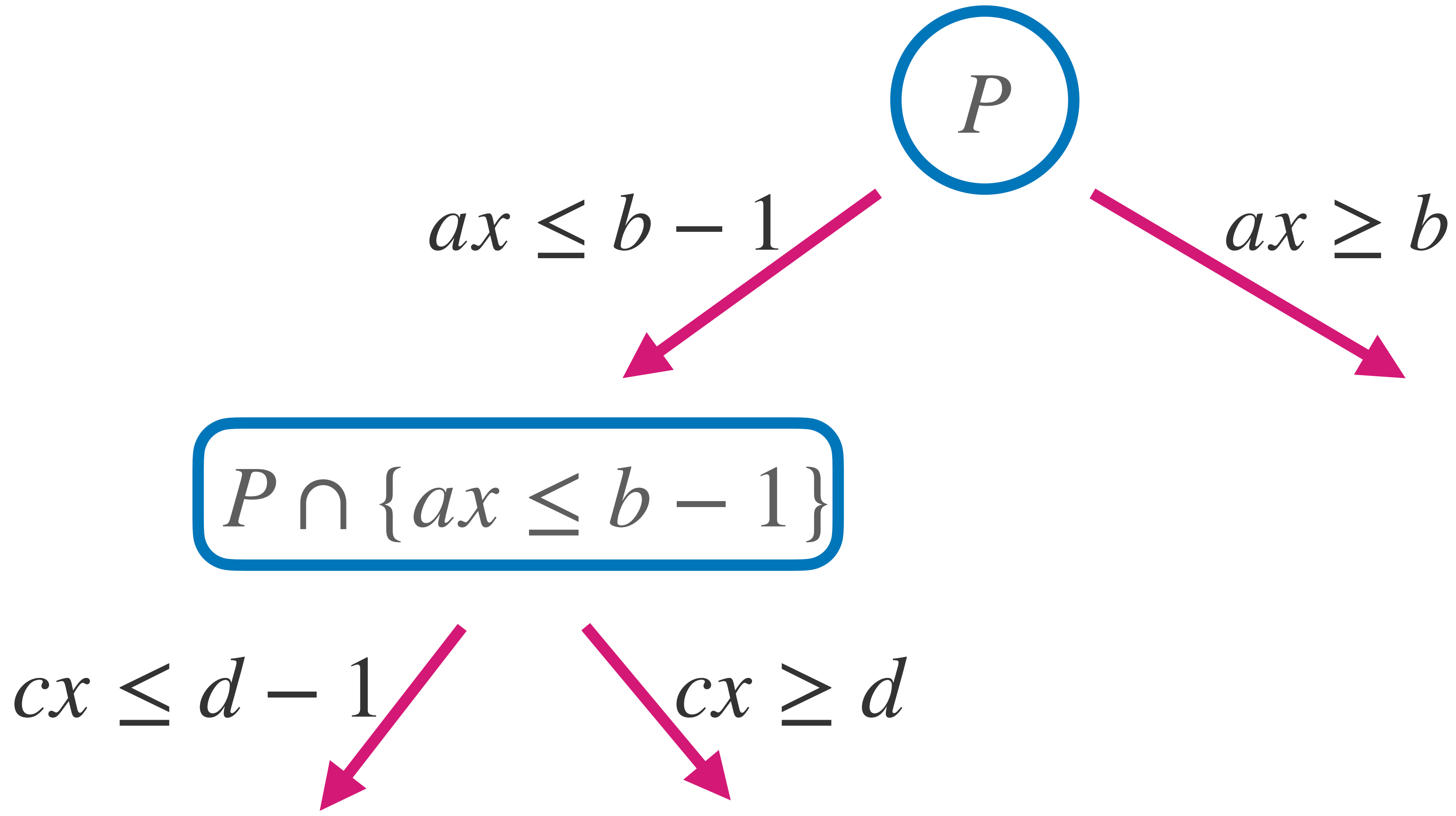
The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut.**

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \emptyset$.



Rule: query an arbitrary **integer linear inequality**

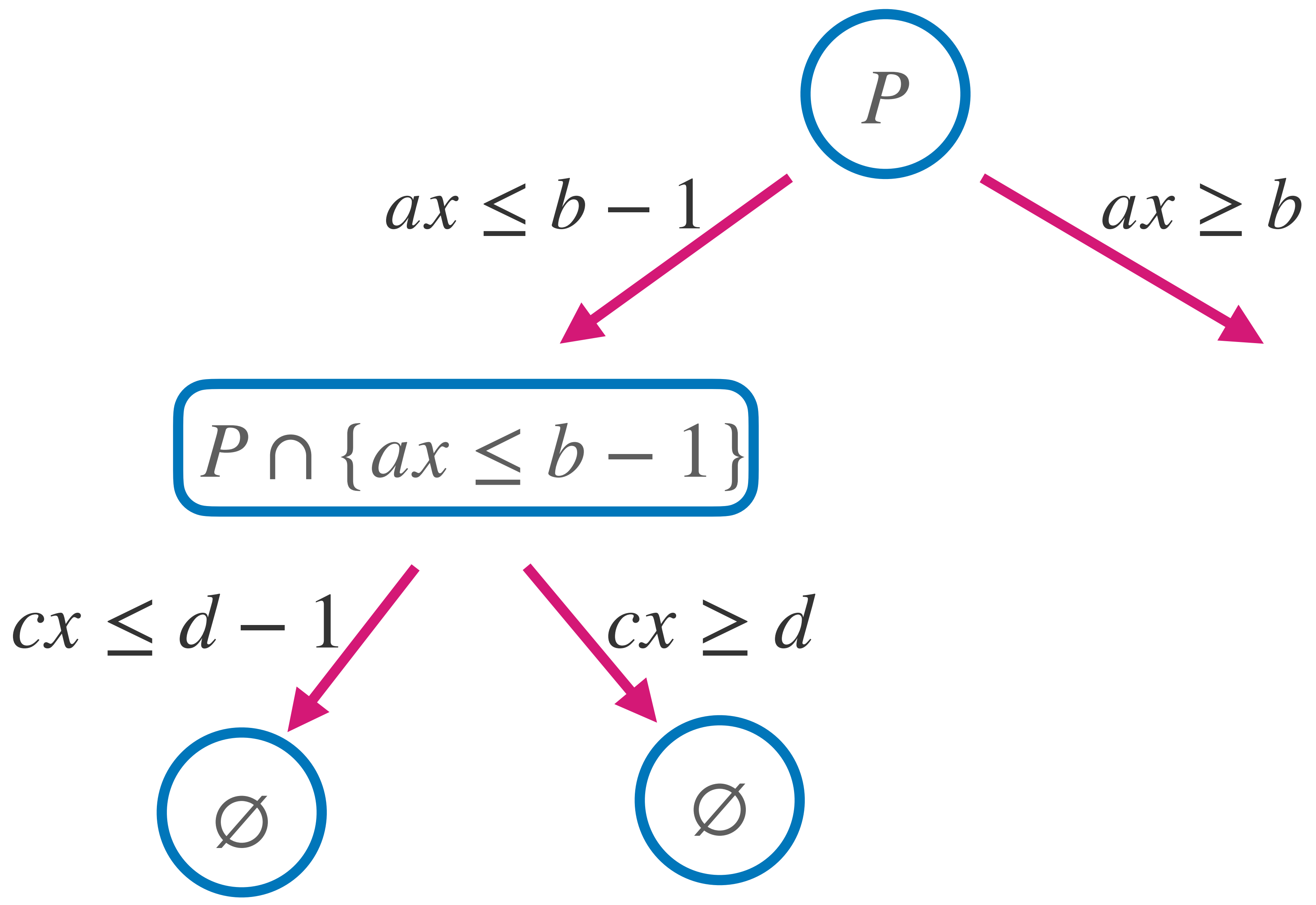


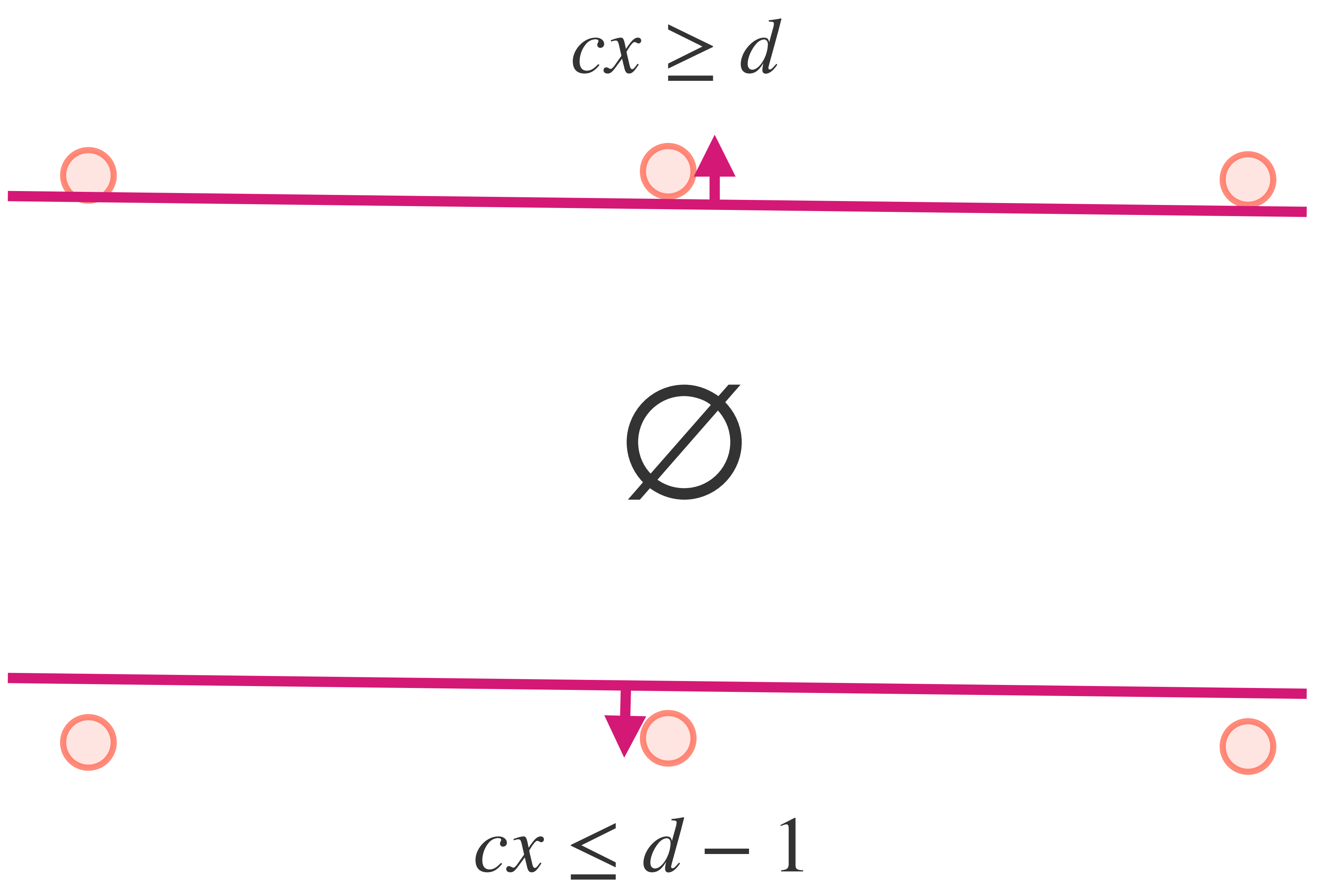
The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut**.

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \varnothing$.

Rule: query an arbitrary **integer linear inequality**

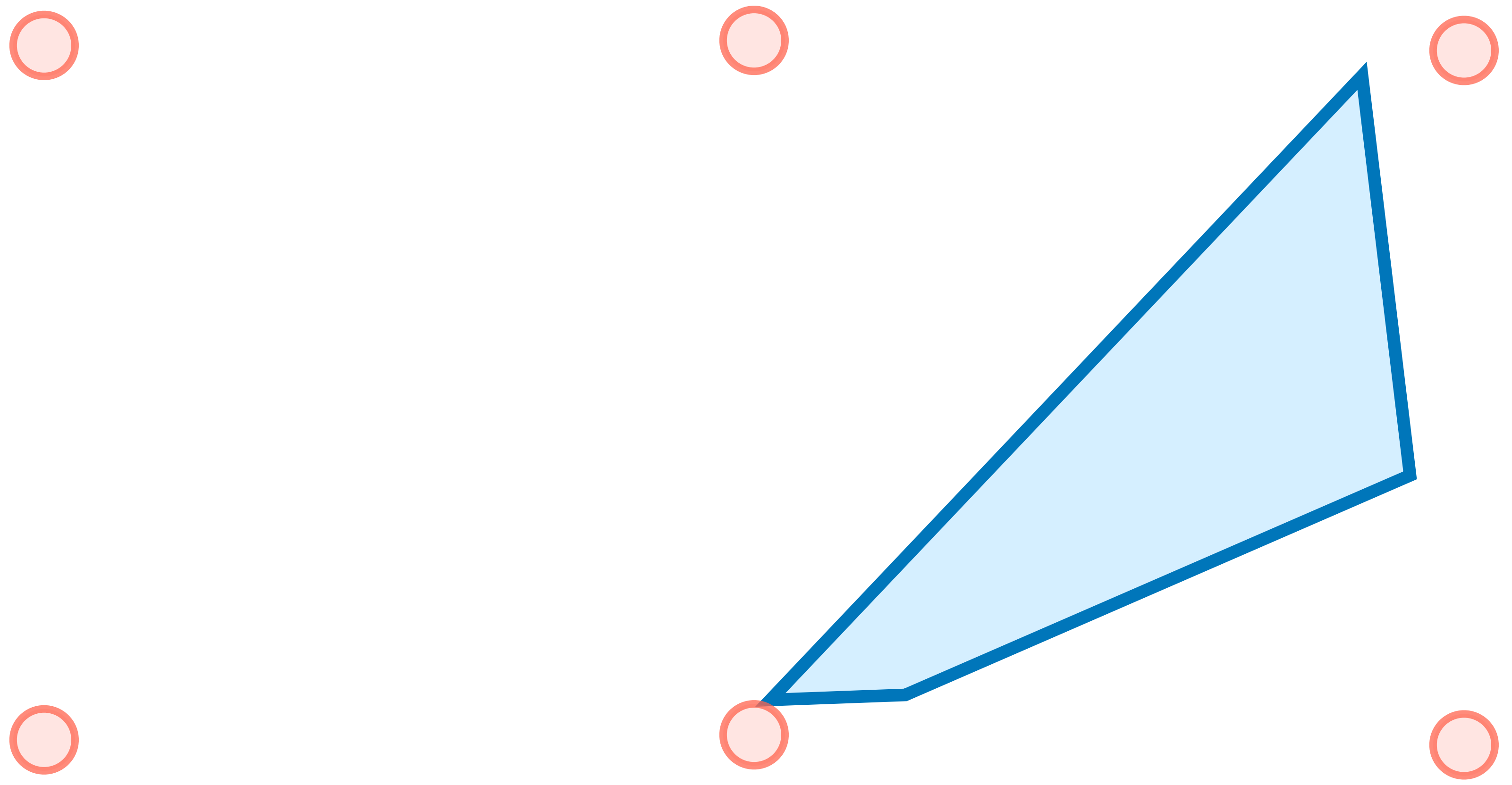




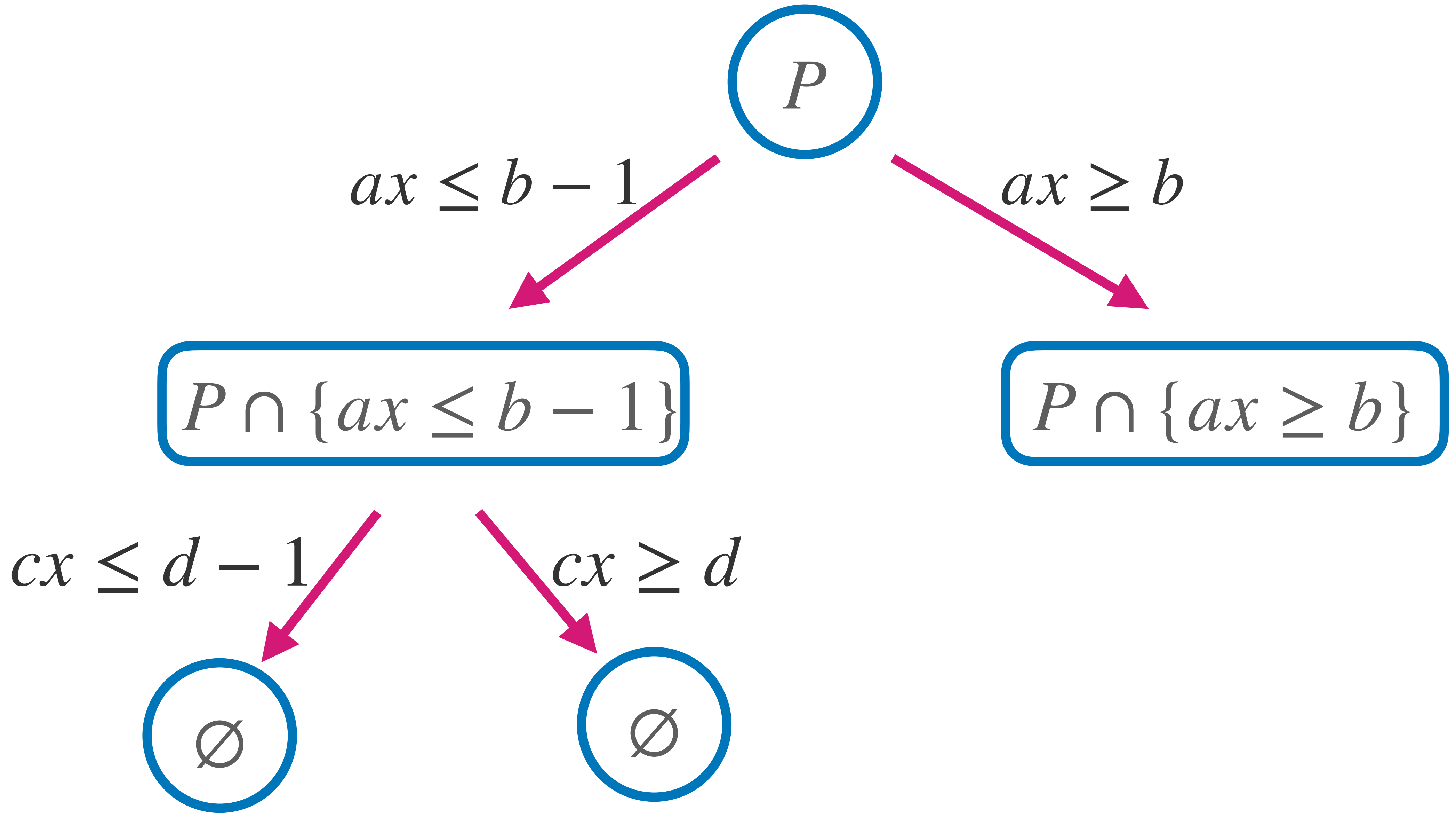
The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut.**

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \varnothing$.



<u>Rule</u>: query an arbitrary **integer linear inequality**

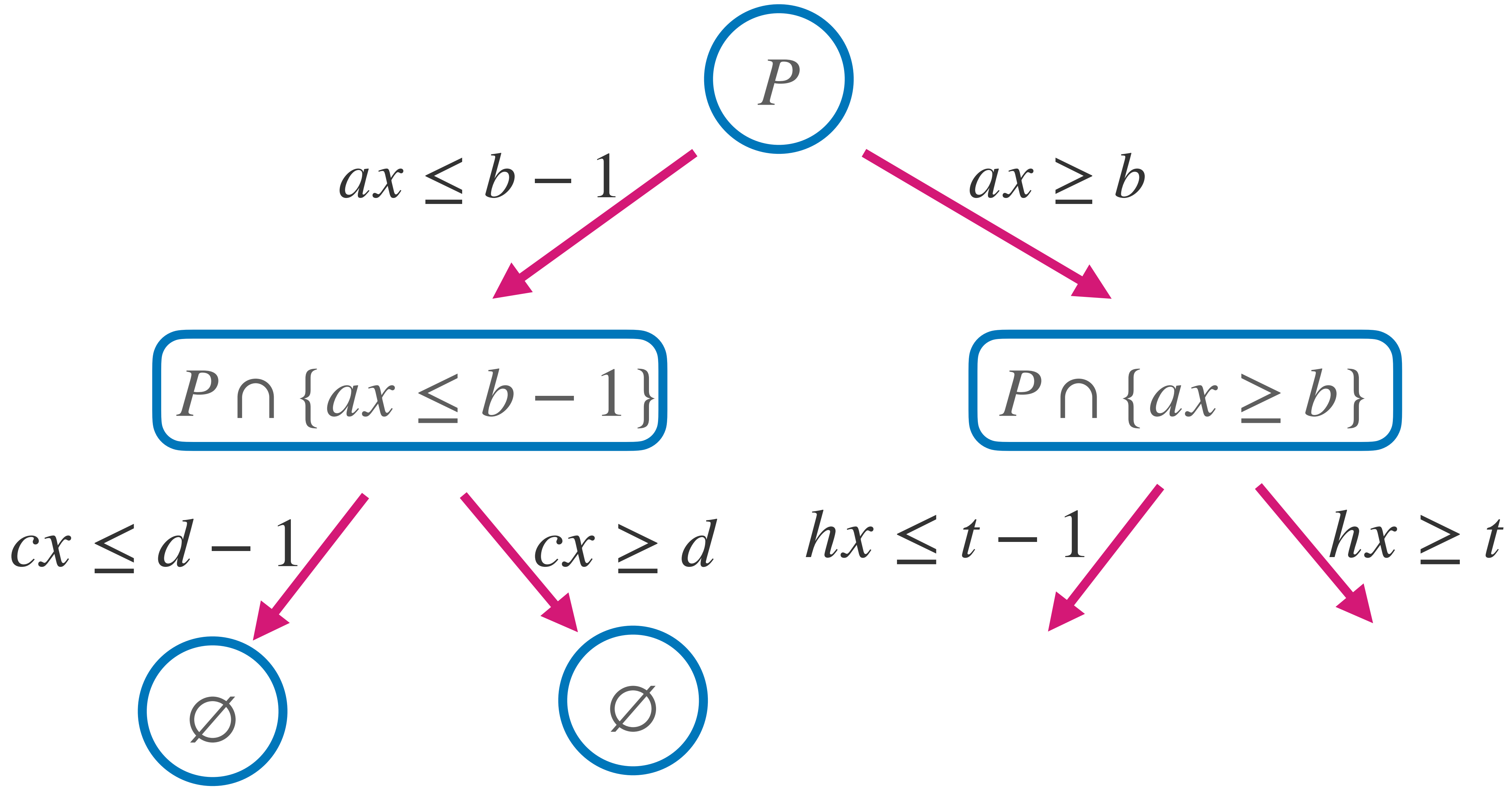


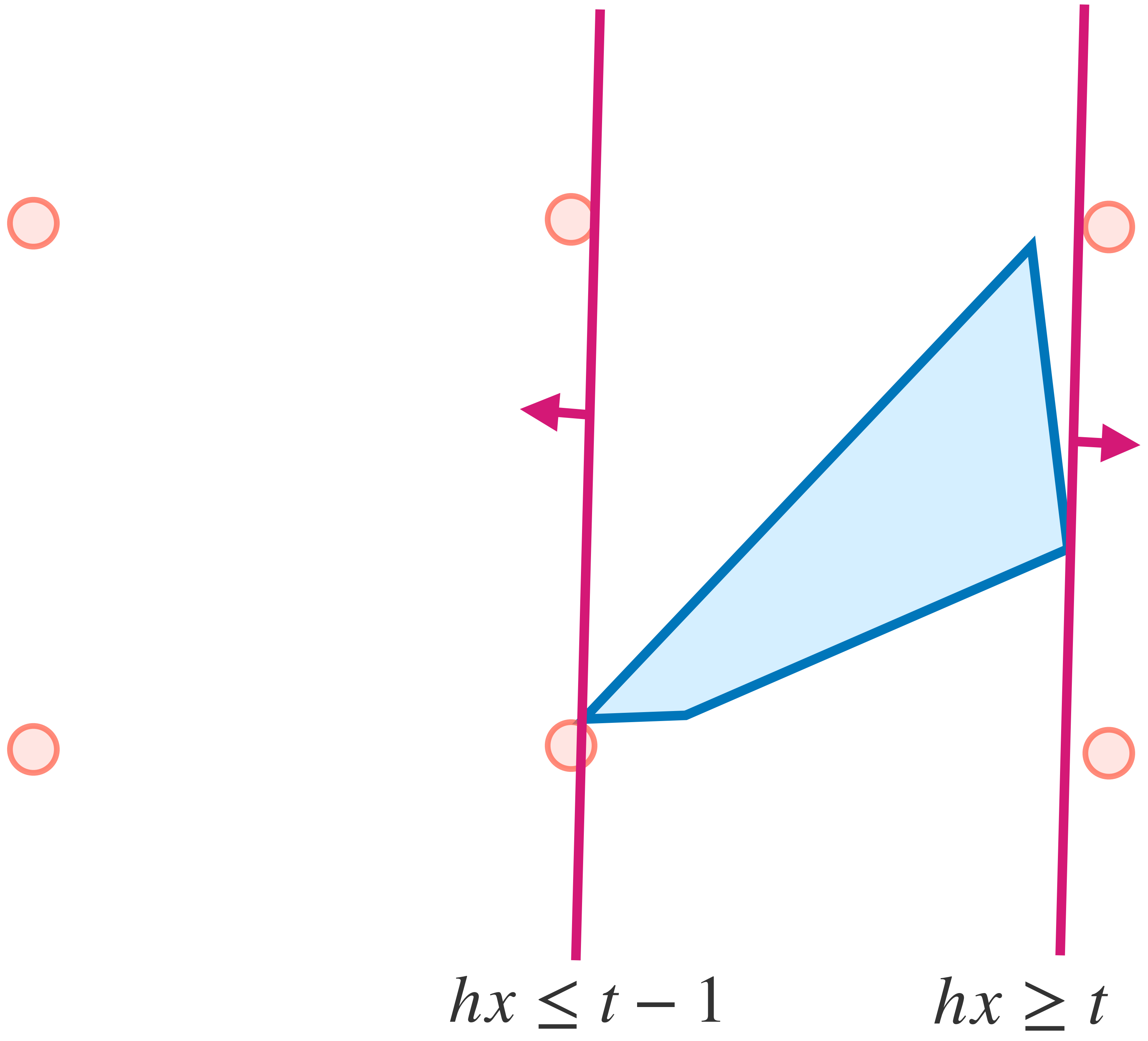
The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut.**

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \varnothing$.

<u>Rule</u>: query an arbitrary **integer linear inequality**



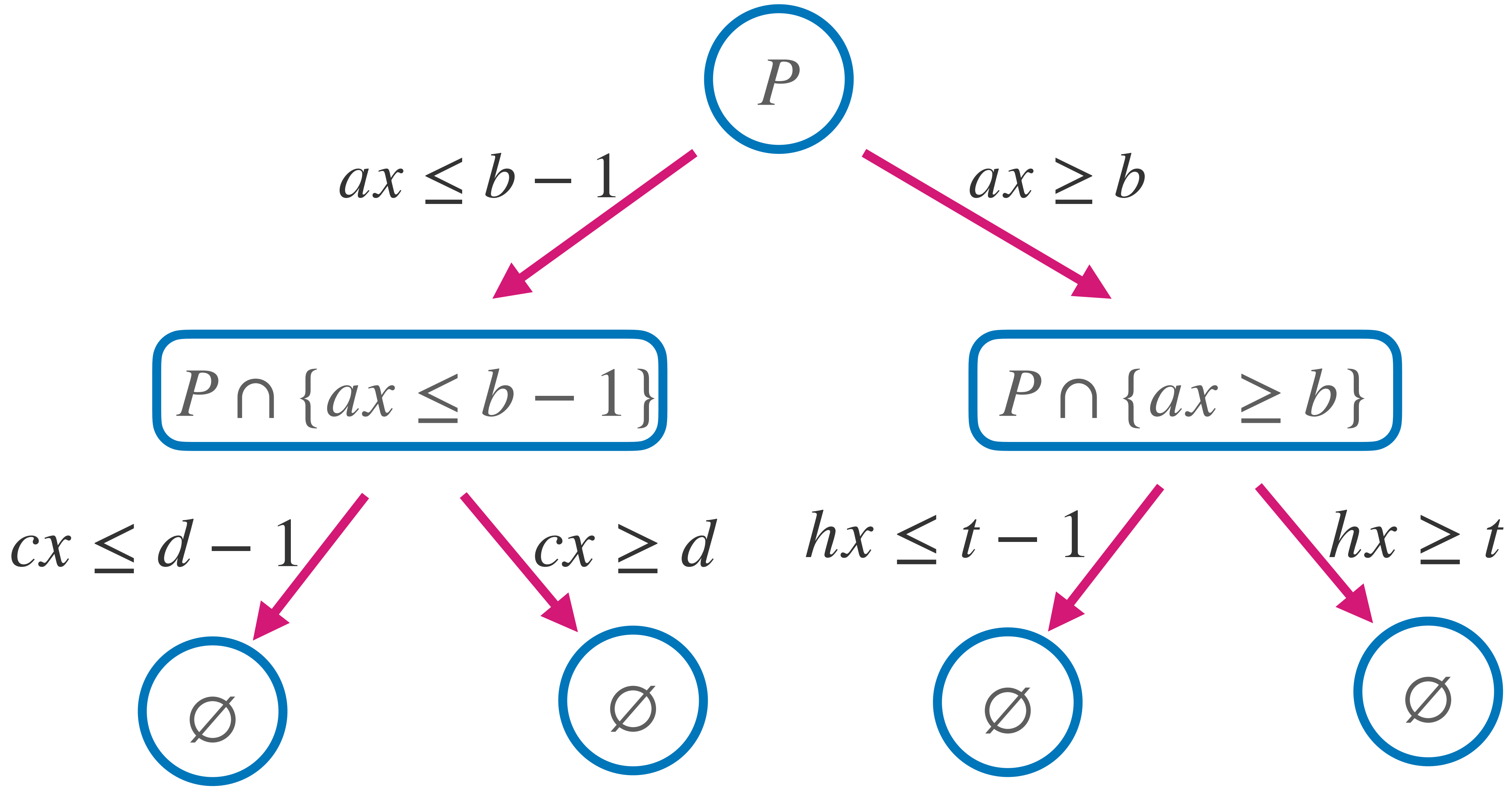


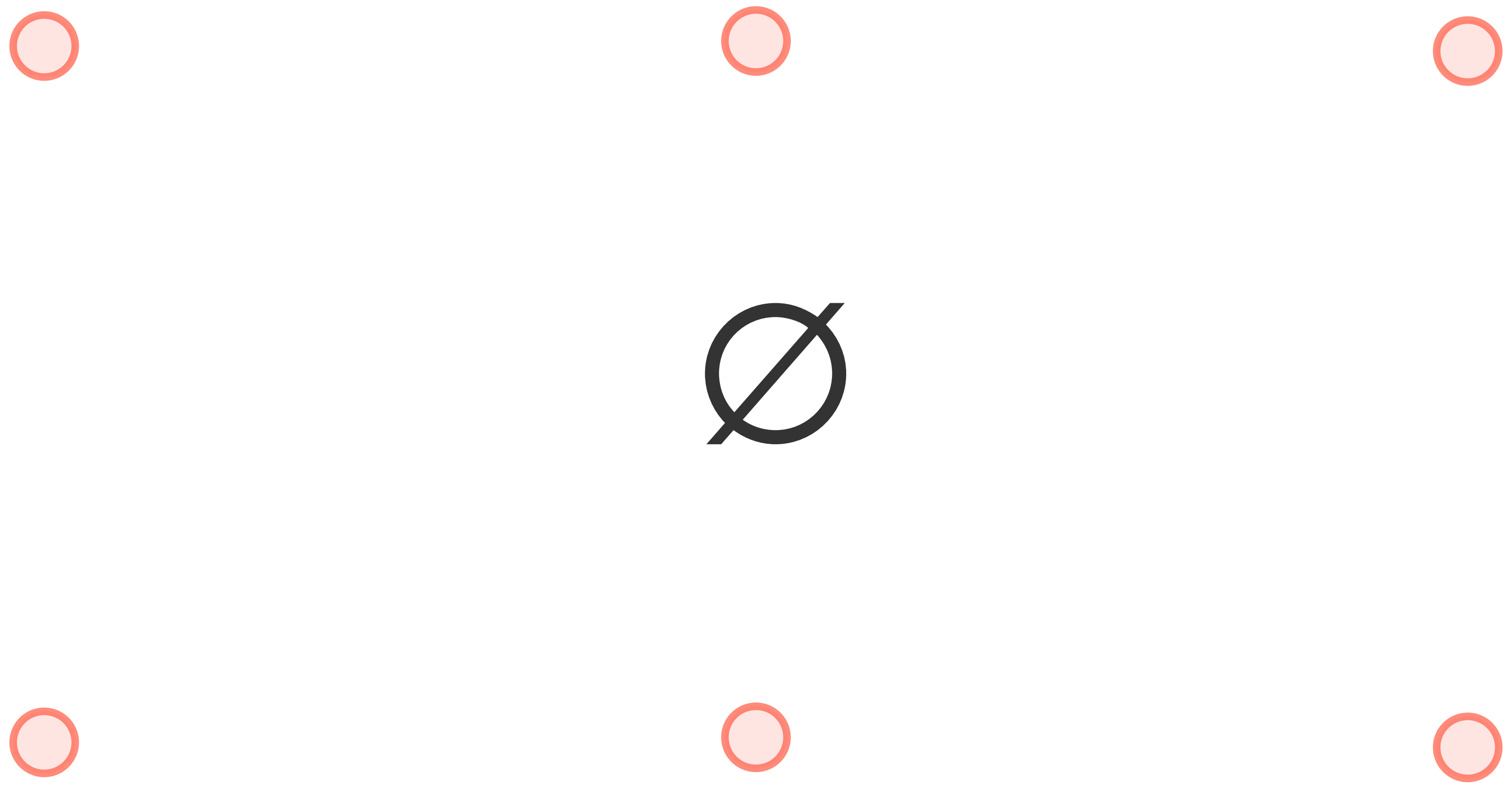
The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut**.

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \varnothing$.

<u>Rule</u>: query an arbitrary **integer linear inequality**



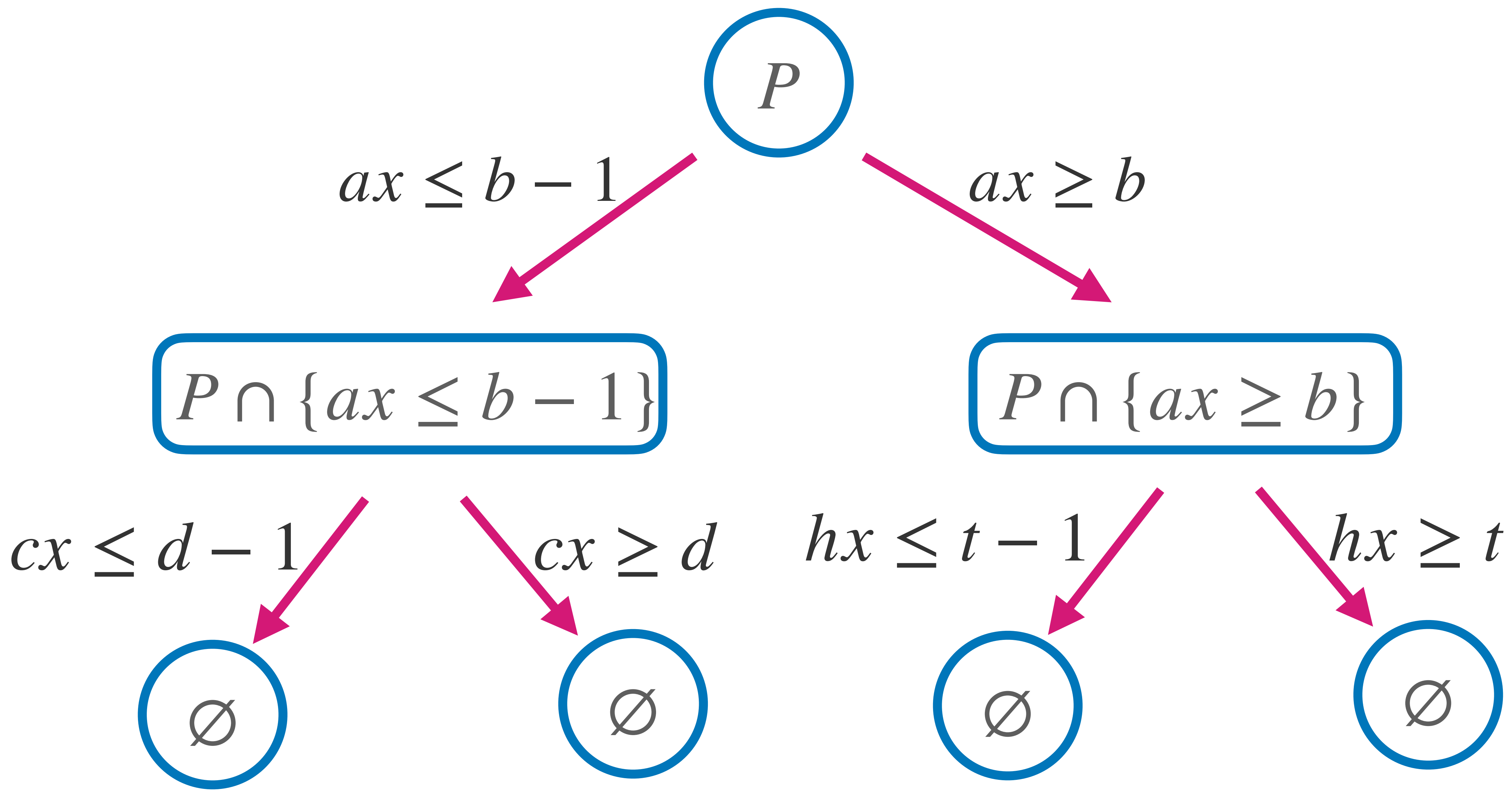


The Stabbing Planes Proof System

[BFI+18] Introduced **Stabbing Planes** to formalize **branch-and-cut**.

Let $P = \{x : Ax \geq b\}$ be such that $P \cap \mathbb{Z}^n = \emptyset$.

<u>Rule</u>: query an arbitrary **integer linear inequality**



Proof that $P \cap \mathbb{Z}^n = \emptyset$!

Stabbing Planes

Claim

The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

Stabbing Planes

Claim

The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Stabbing Planes

Claim

The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

$\rightarrow$ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Stabbing Planes

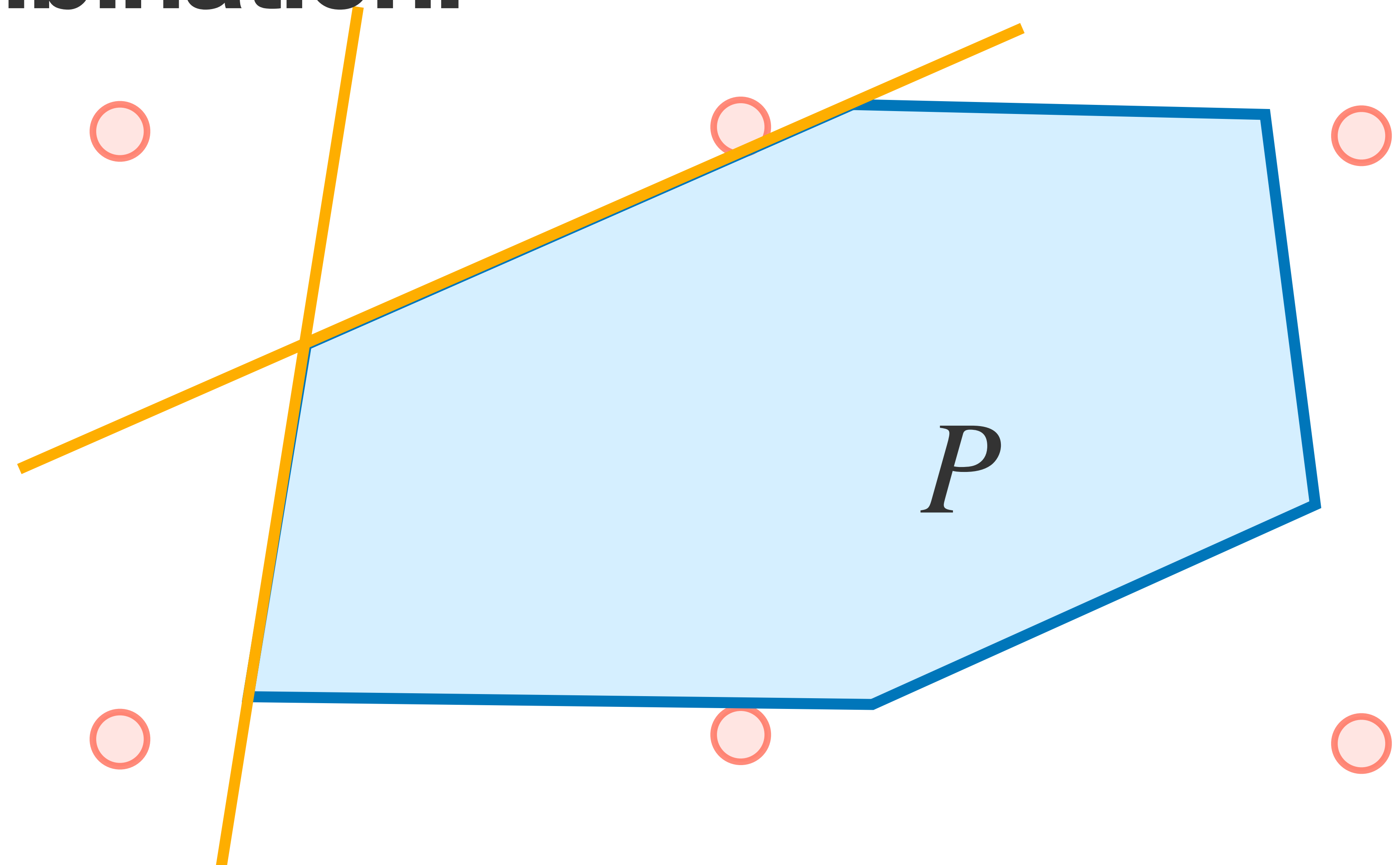
Claim

The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Non-negative linear combination:



Stabbing Planes

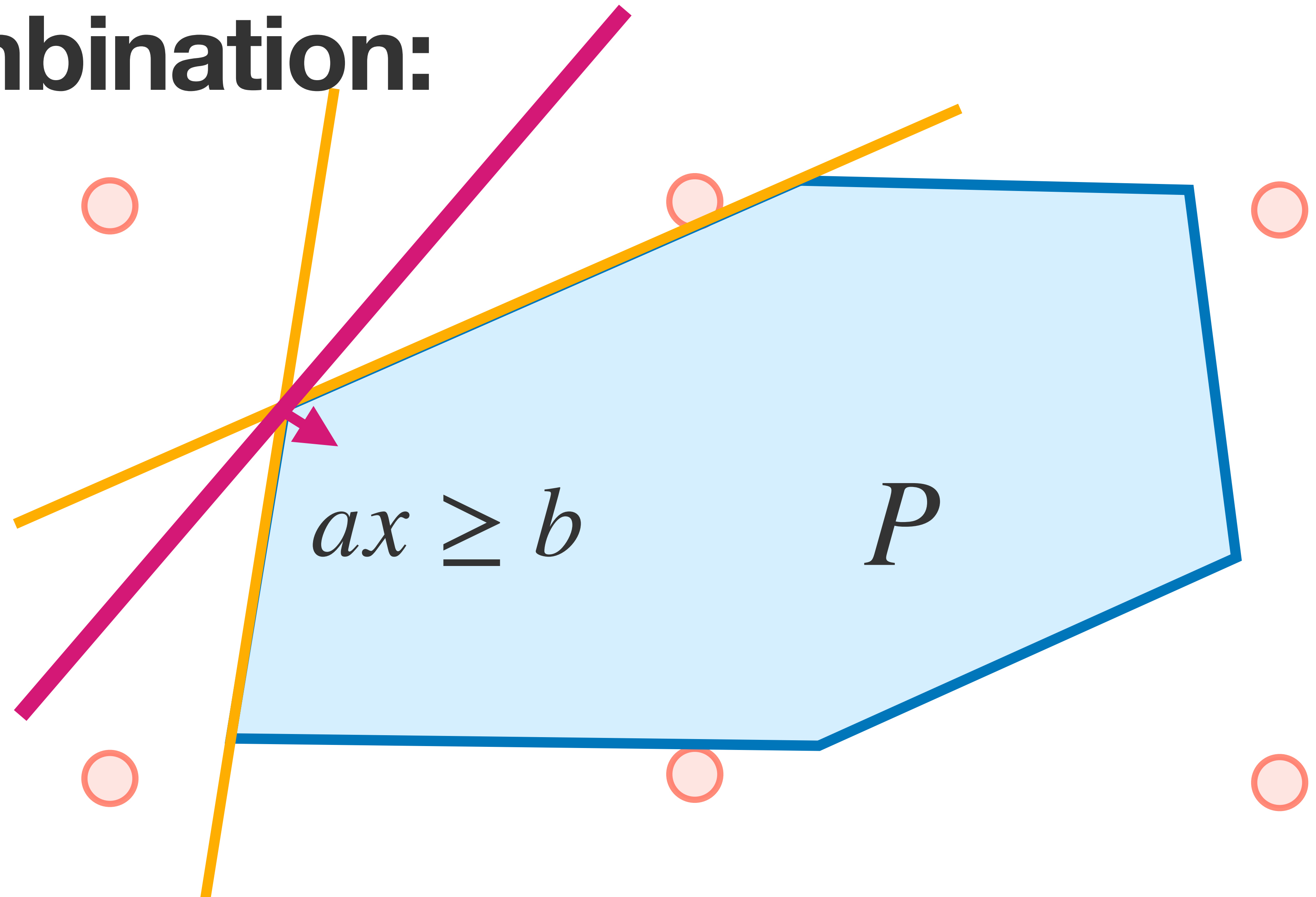
Claim

The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Non-negative linear combination:



Stabbing Planes

Claim

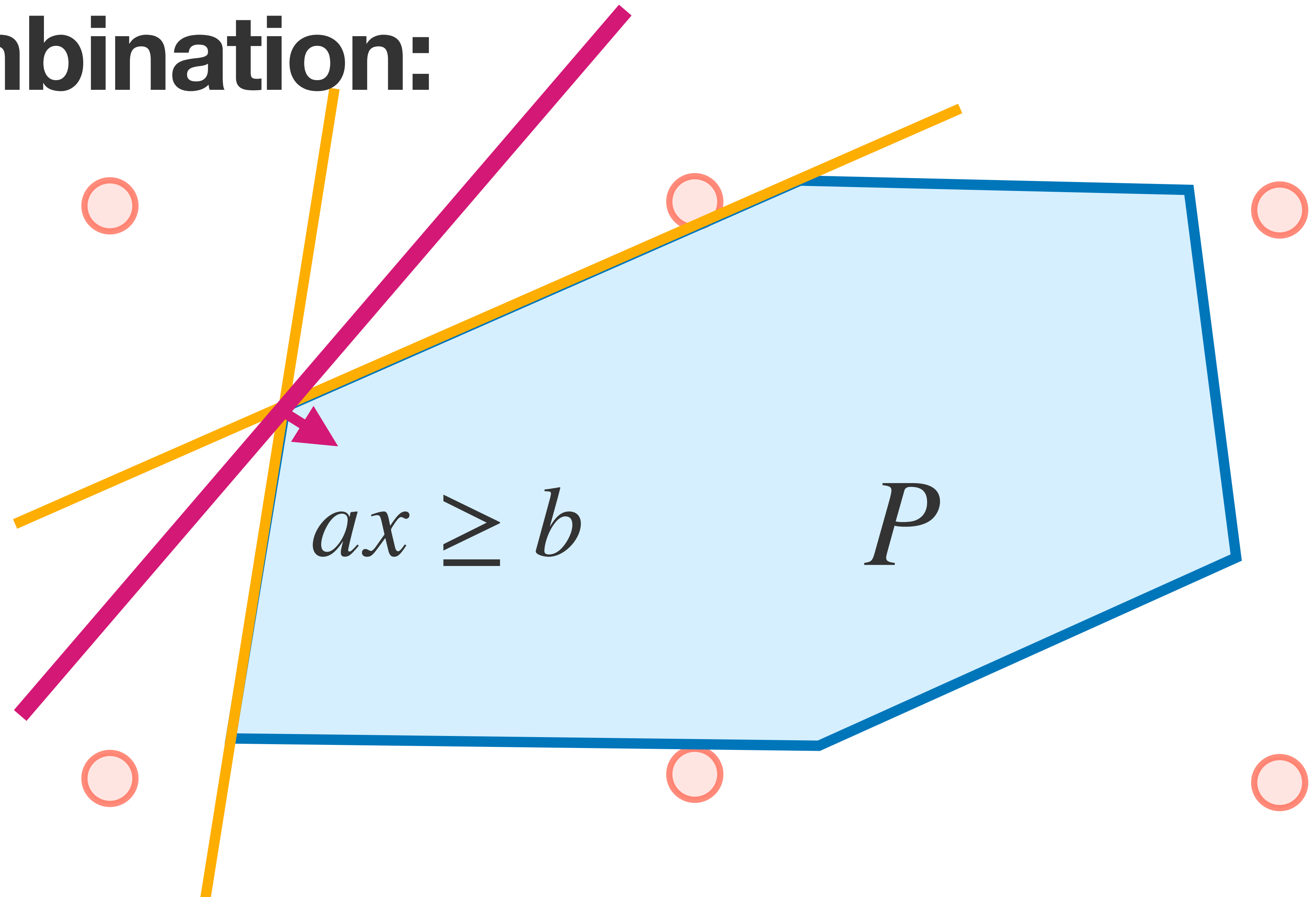
The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Non-negative linear combination:

In SP query:



Stabbing Planes

Claim

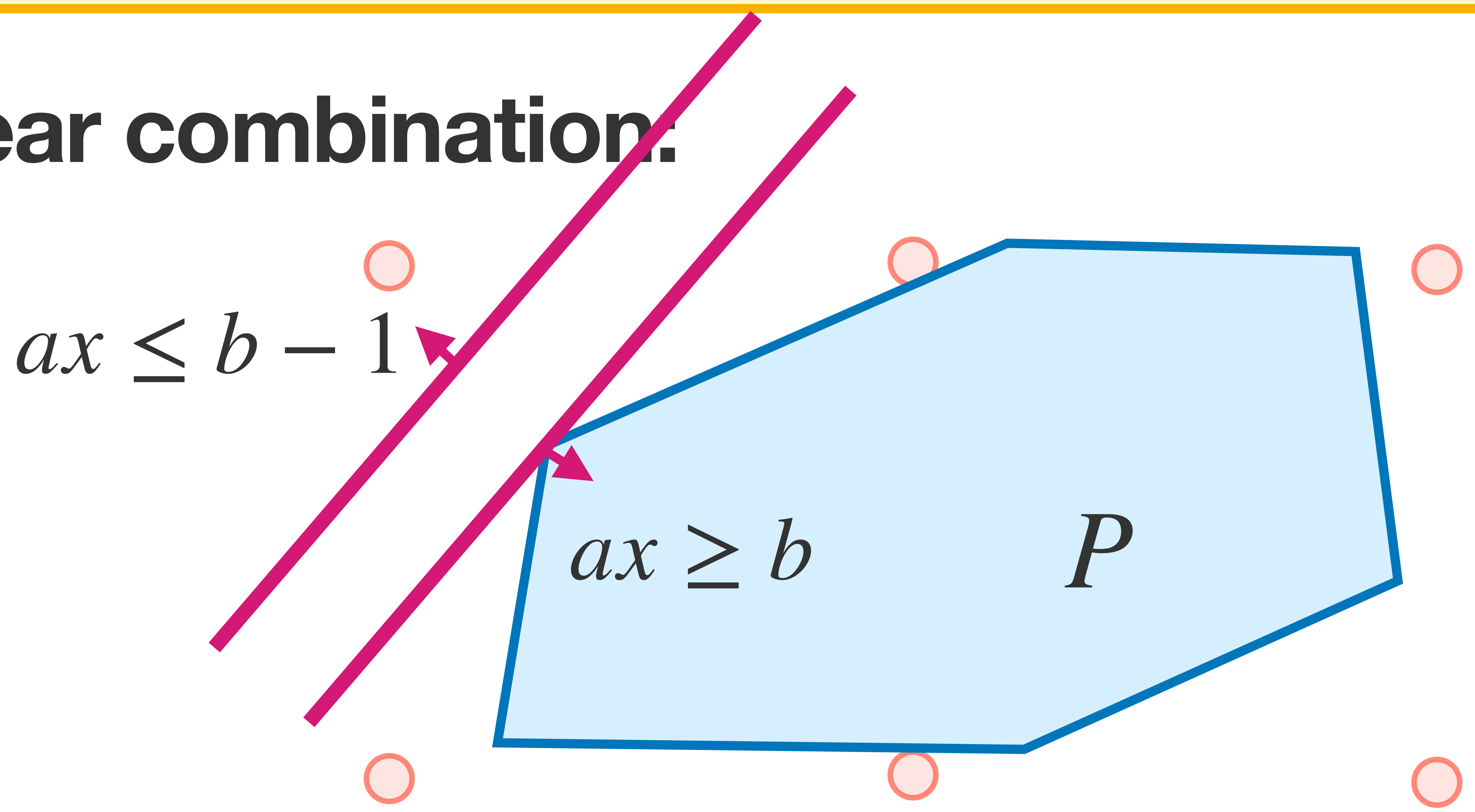
The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Non-negative linear combination:

In SP query:



Stabbing Planes

Claim

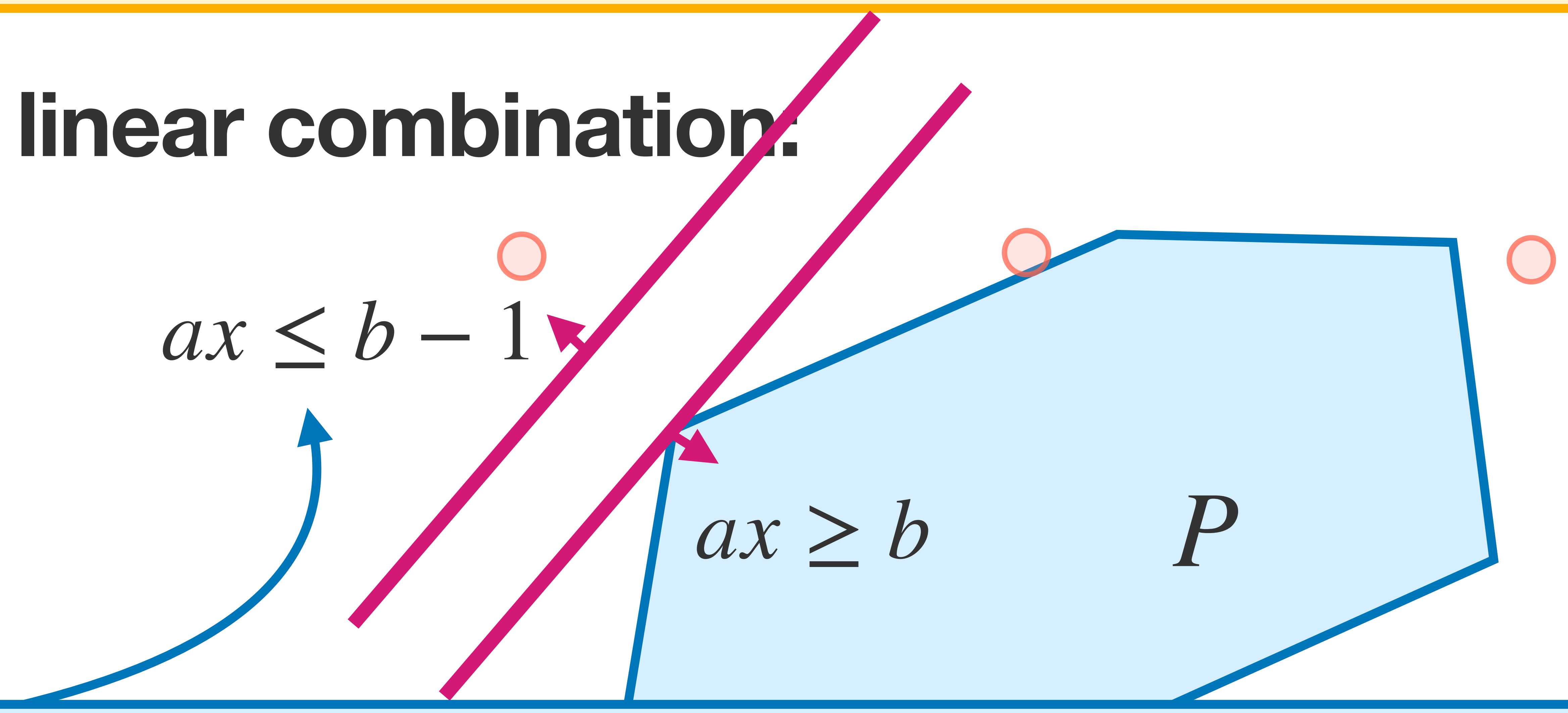
The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Non-negative linear combination:

In SP query:



$P \cap \{ax \leq b - 1\}$ is empty!

Stabbing Planes

Claim

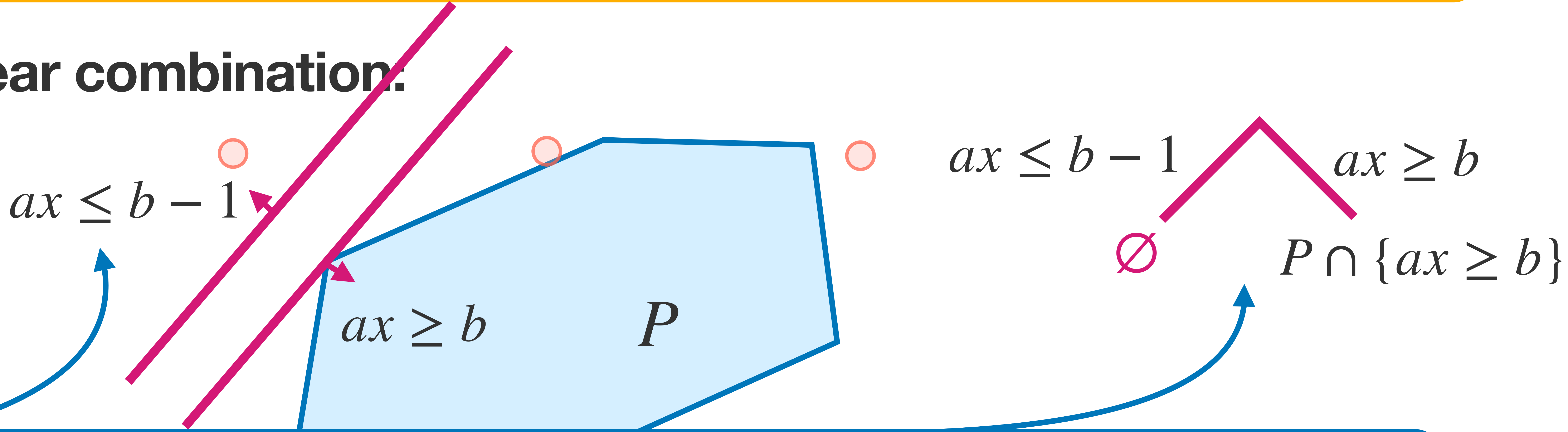
The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Non-negative linear combination:

In SP query:



$P \cap \{ax \leq b - 1\}$ is empty!

Stabbing Planes

Claim

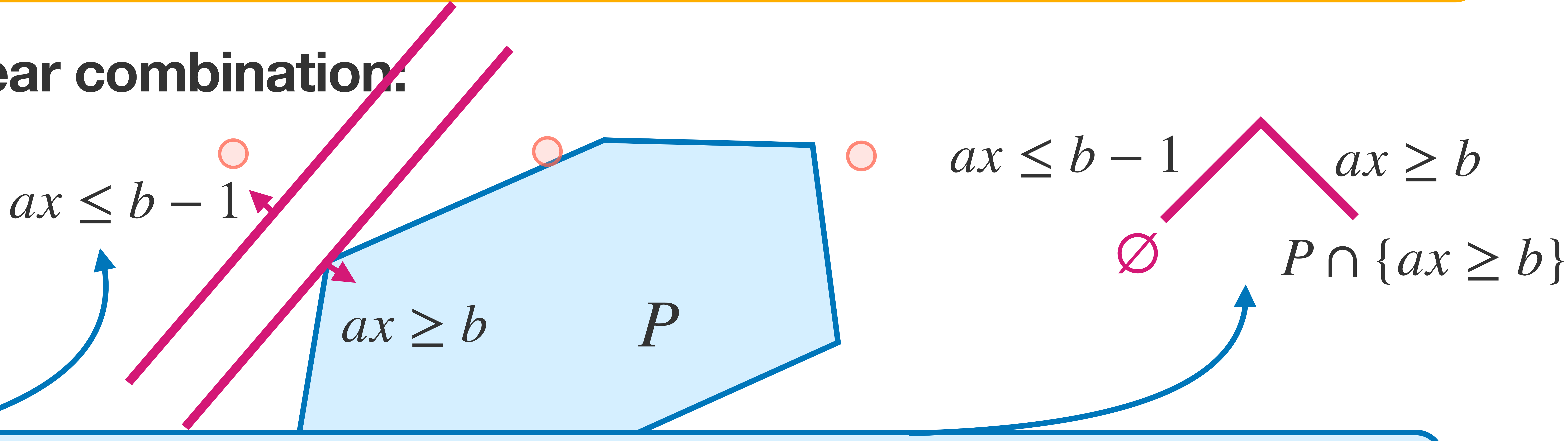
The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

$\rightarrow$ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Non-negative linear combination:

In SP query:



$P \cap \{ax \leq b - 1\}$ is empty! So this only increases the size by 1!

Stabbing Planes

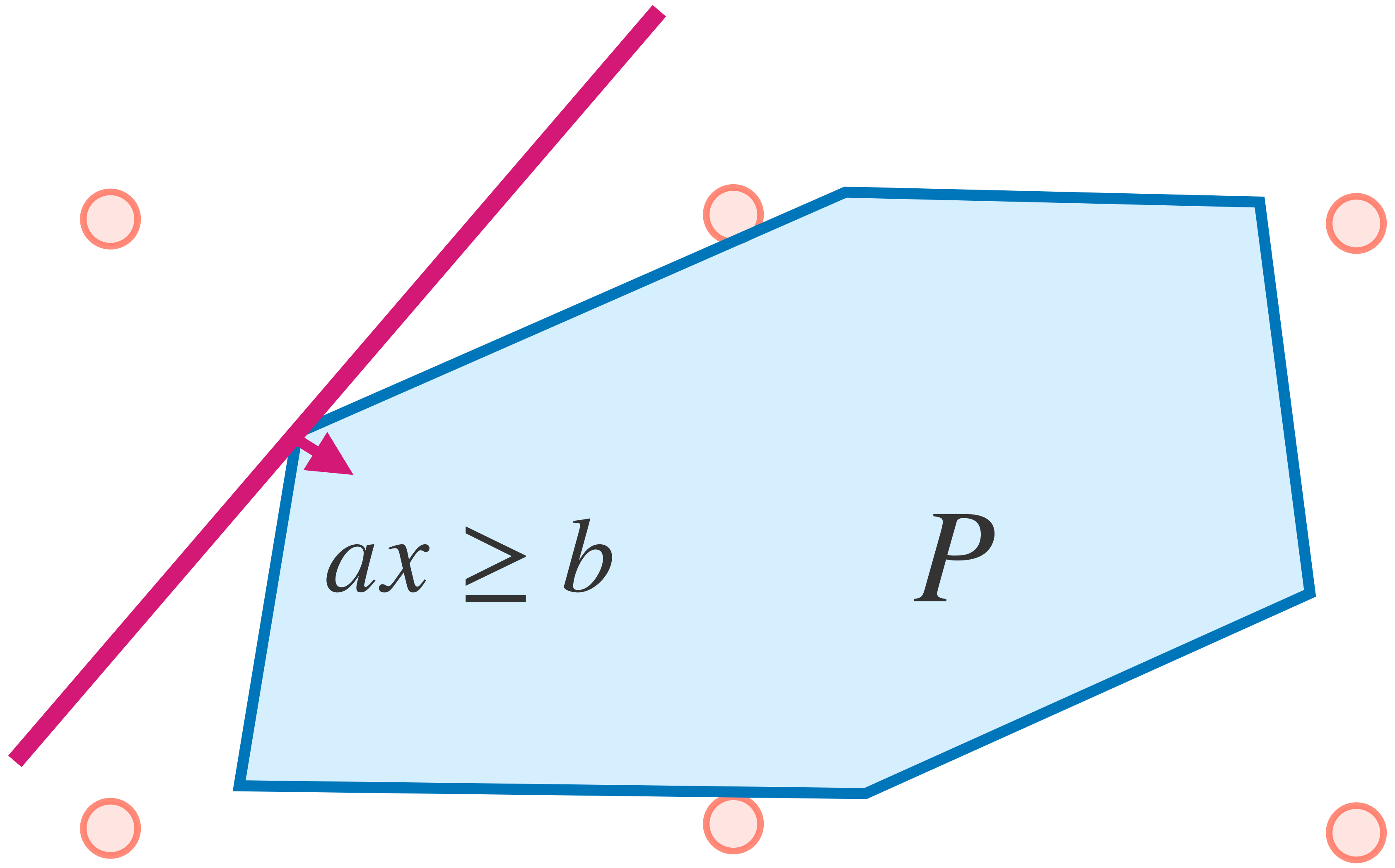
Claim

The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Cut:



Stabbing Planes

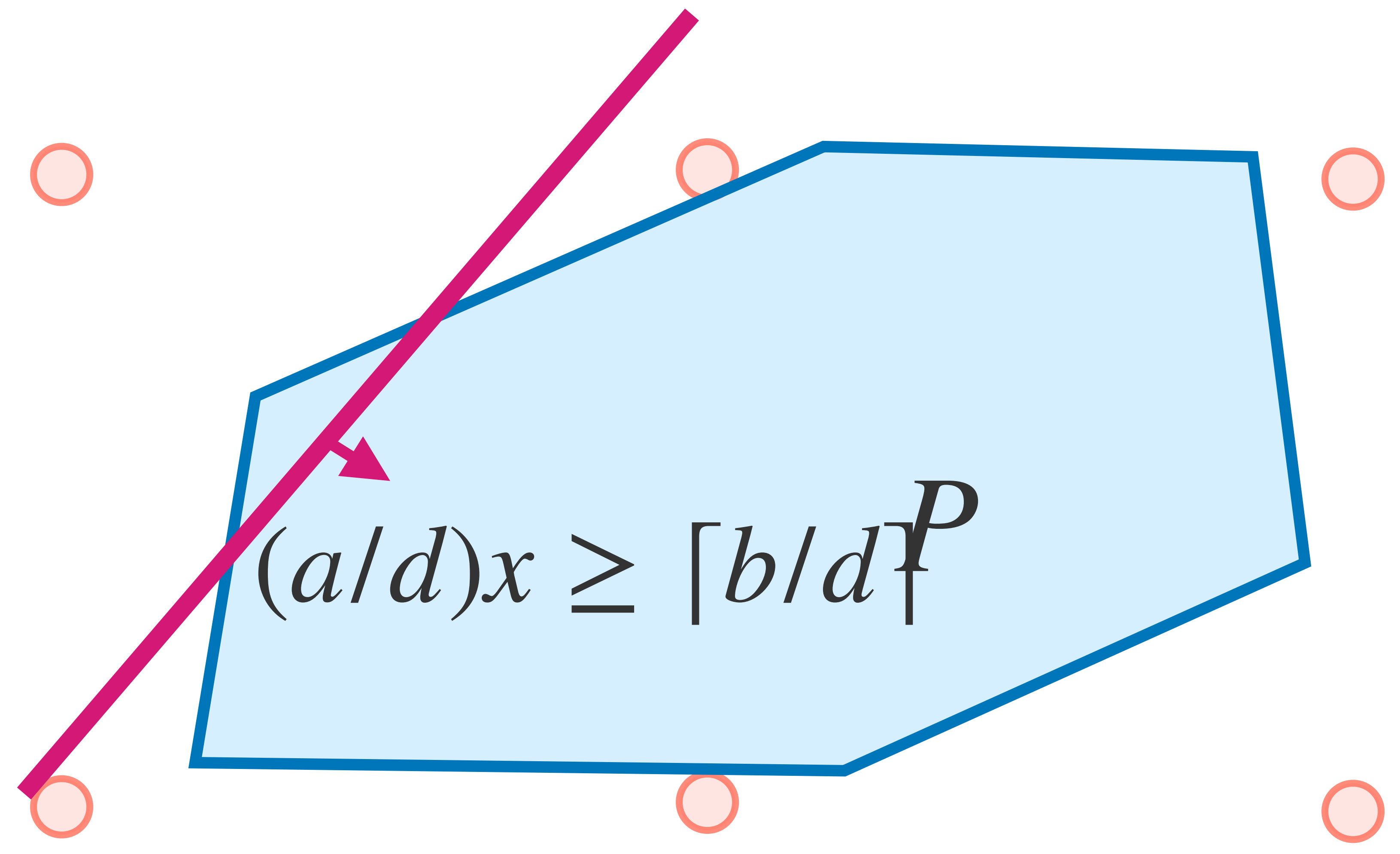
Claim

The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Cut:



Stabbing Planes

Claim

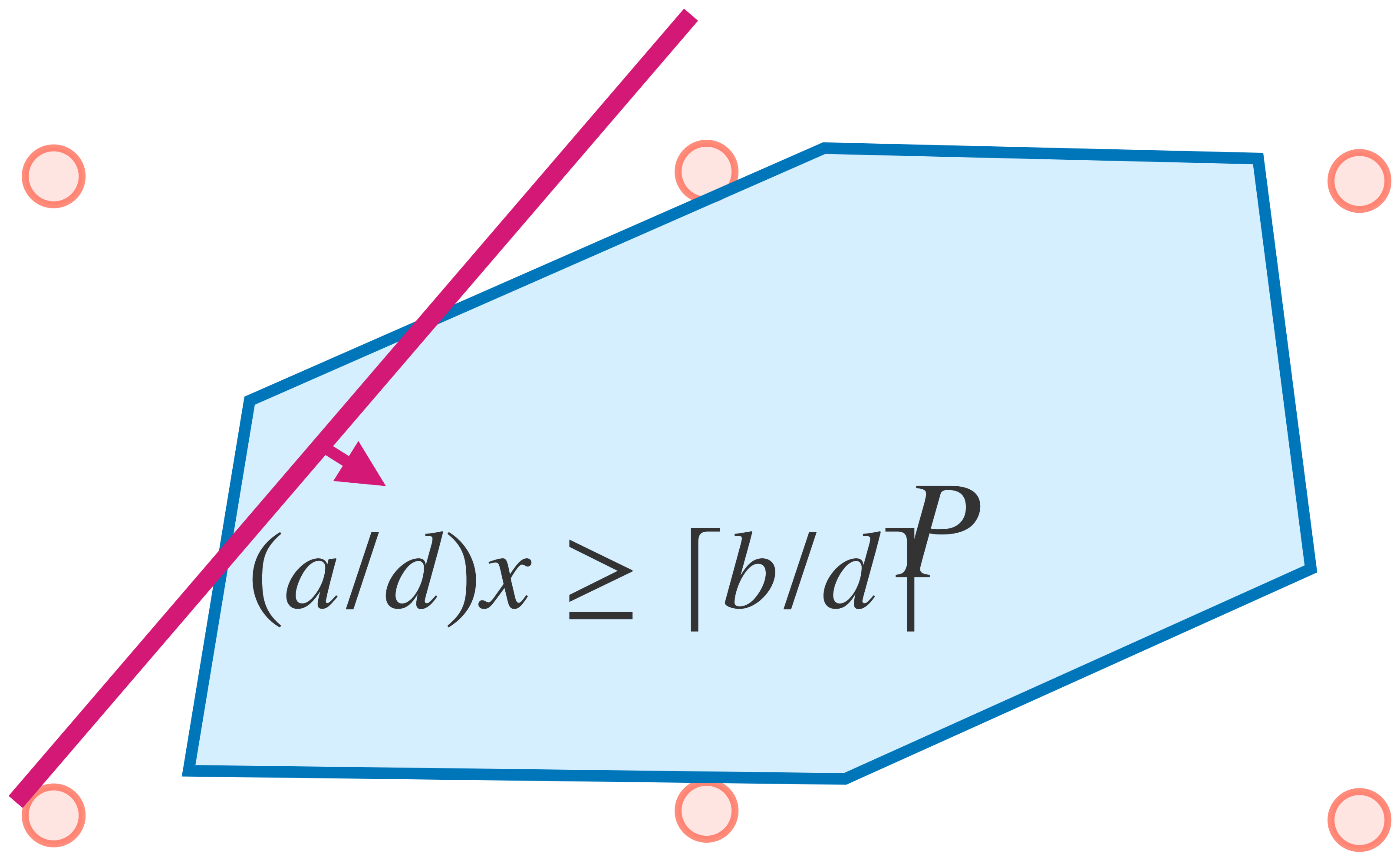
The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Cut:

In SP query:



Stabbing Planes

Claim

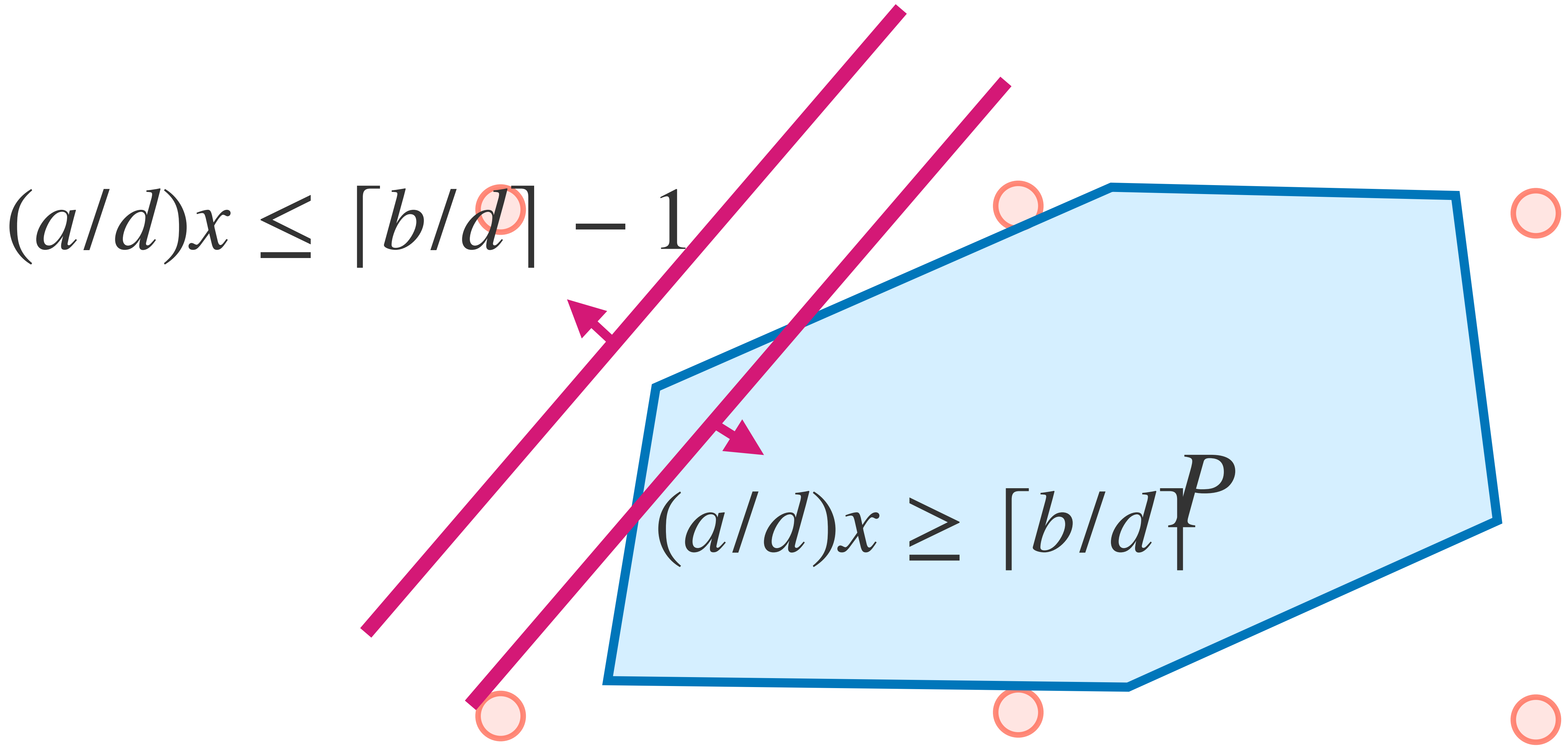
The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Cut:

In SP query:



Stabbing Planes

Claim

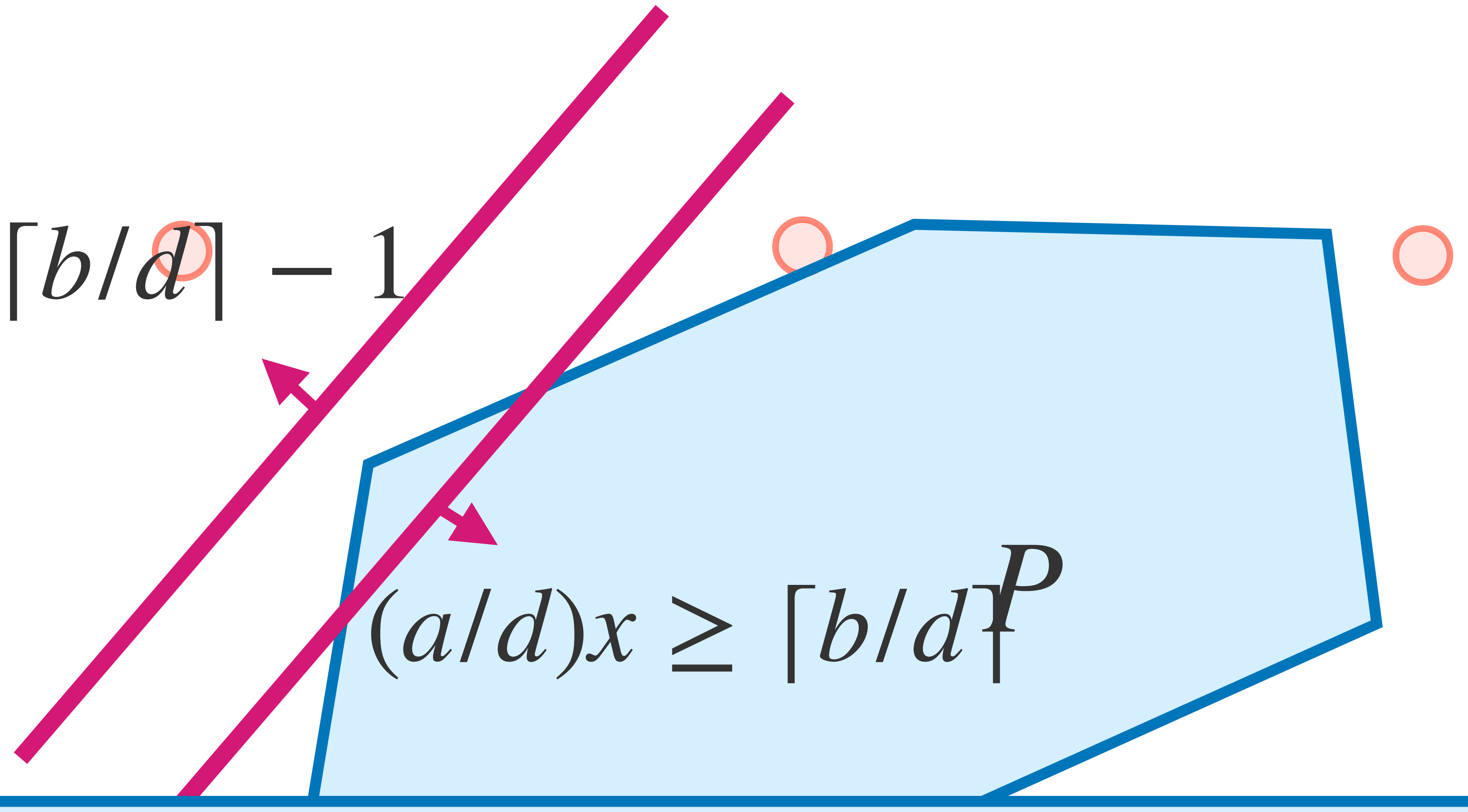
The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Cut:

In SP query: $(a/d)x \leq \lceil b/d \rceil - 1$



$P \cap \{(a/d)x \leq \lceil b/d \rceil - 1\}$ is empty! So this only increases the size by 1!

Stabbing Planes

Claim

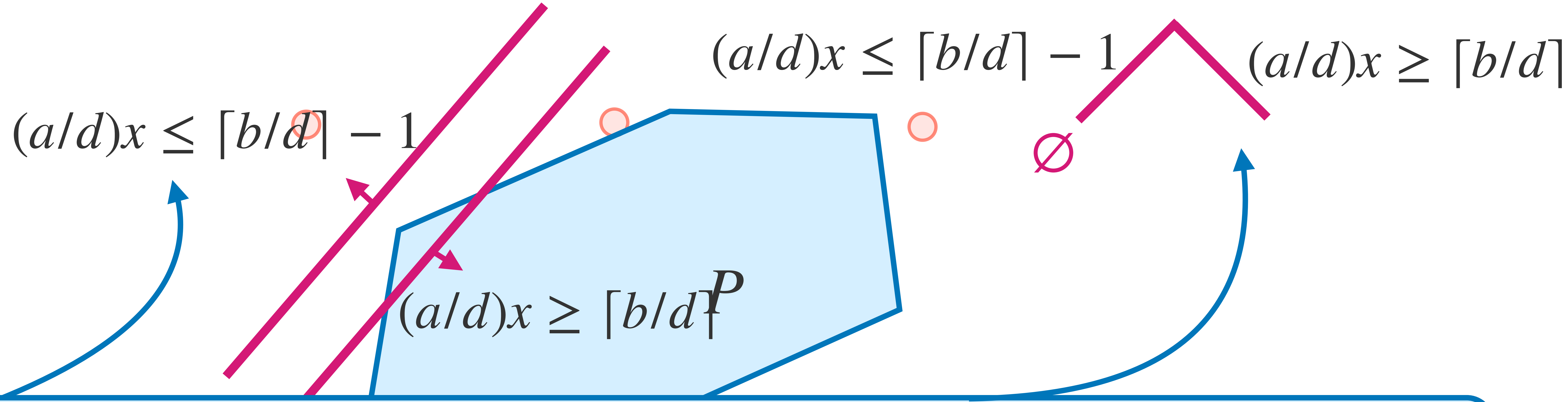
The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Cut:

In SP query:



$P \cap \{(a/d)x \leq \lceil b/d \rceil - 1\}$ is empty! So this only increases the size by 1!

Stabbing Planes

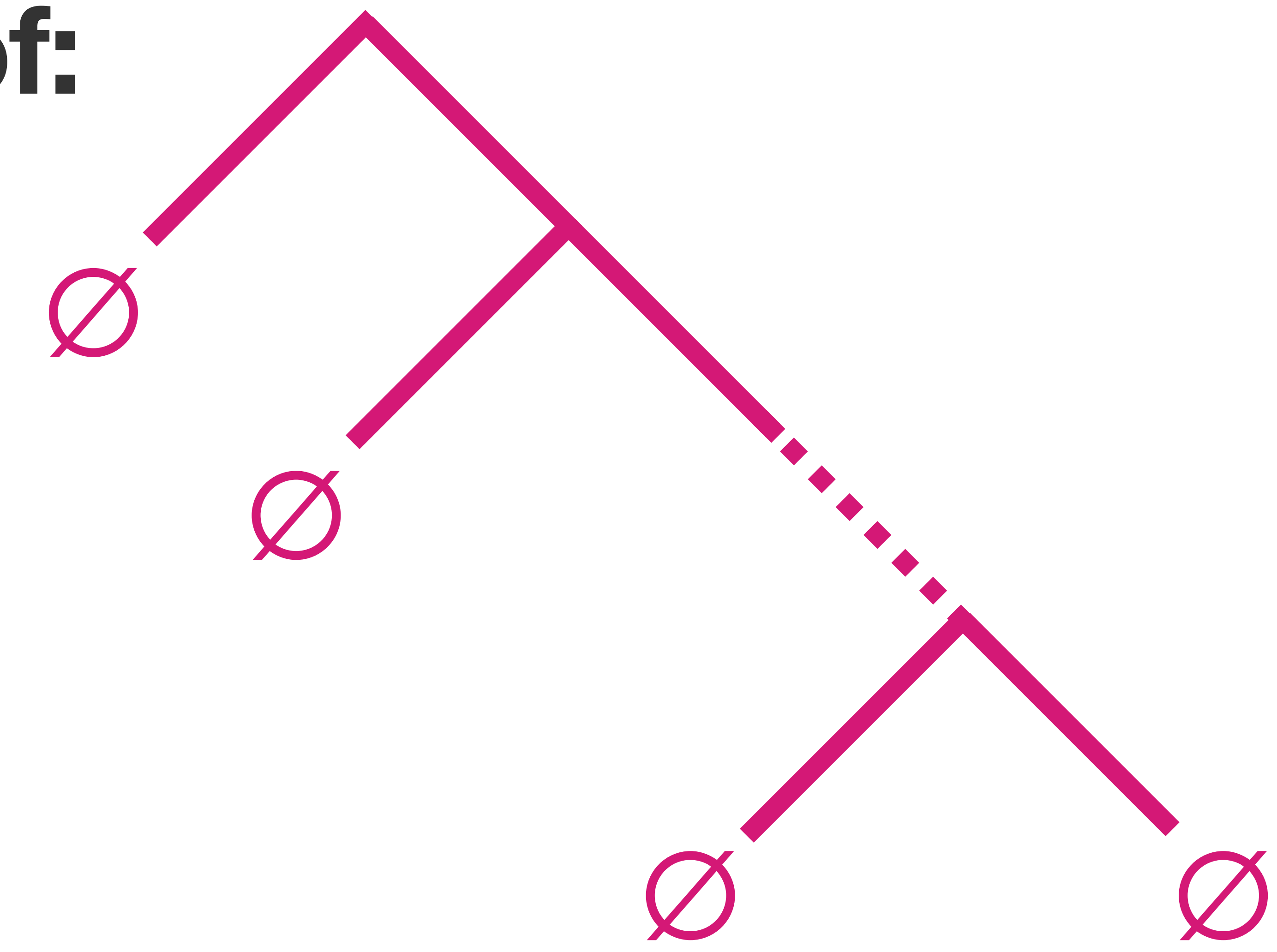
Claim

The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Resulting Stabbing Planes Proof:



Stabbing Planes

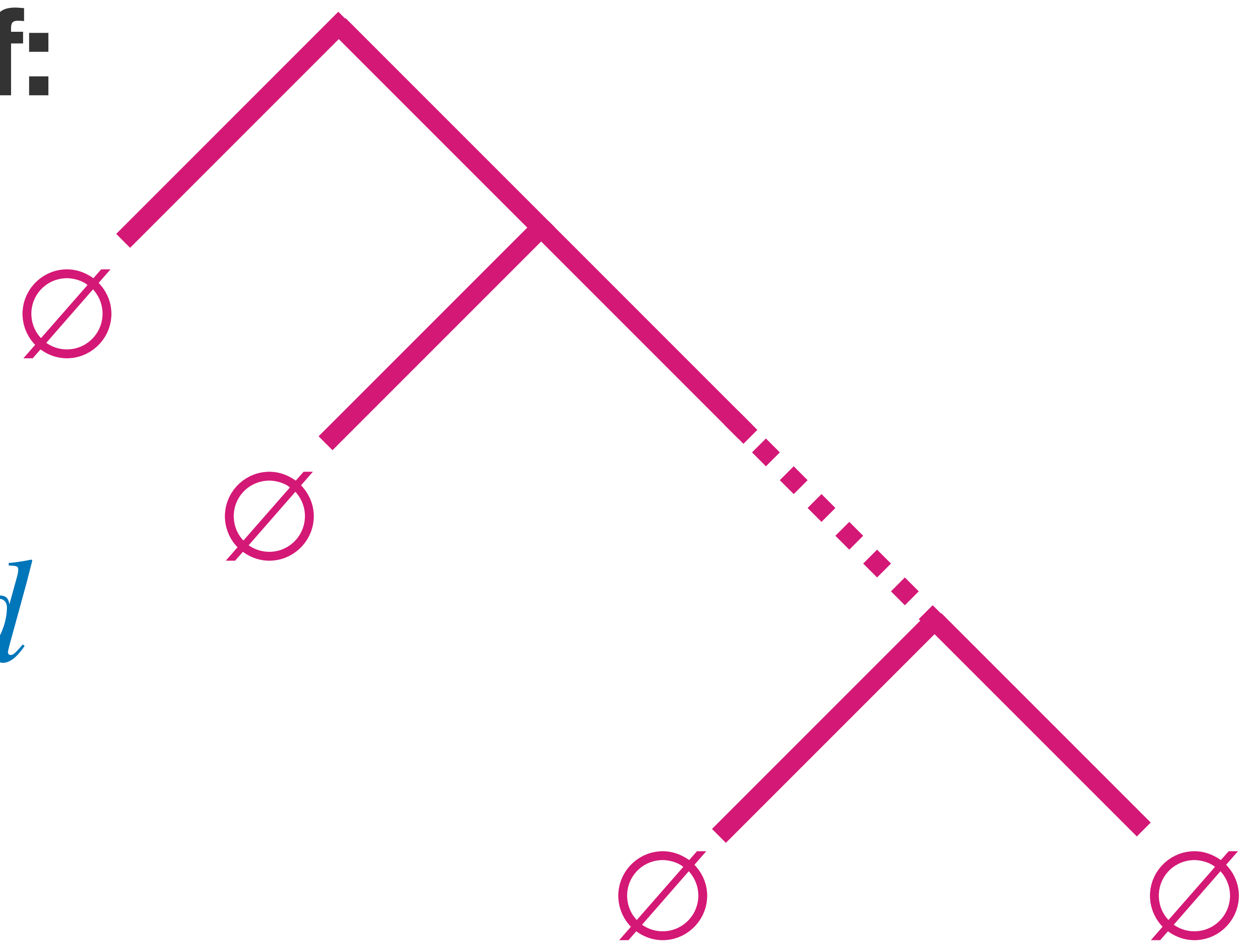
Claim

The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Resulting Stabbing Planes Proof:



If the CP proof had size s, depth d

→ SP proof has size s, depth s

Stabbing Planes

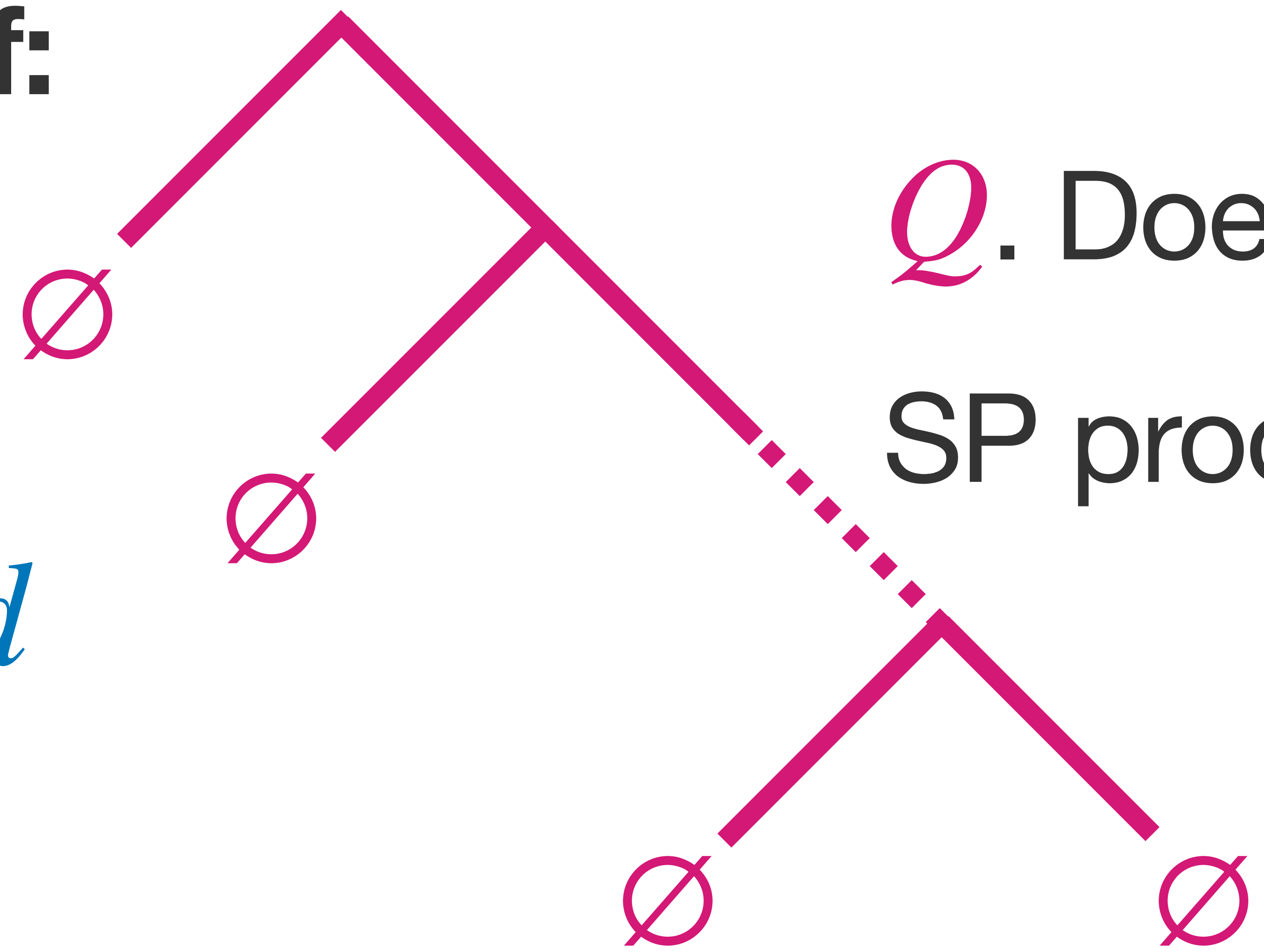
Claim

The execution of a **branch-and-cut** solver produces a **Stabbing Planes proof**.

→ Stabbing Planes rule simulates both branching **and** cutting!

Claim: Stabbing Planes simulates Cutting Planes proofs

Resulting Stabbing Planes Proof:

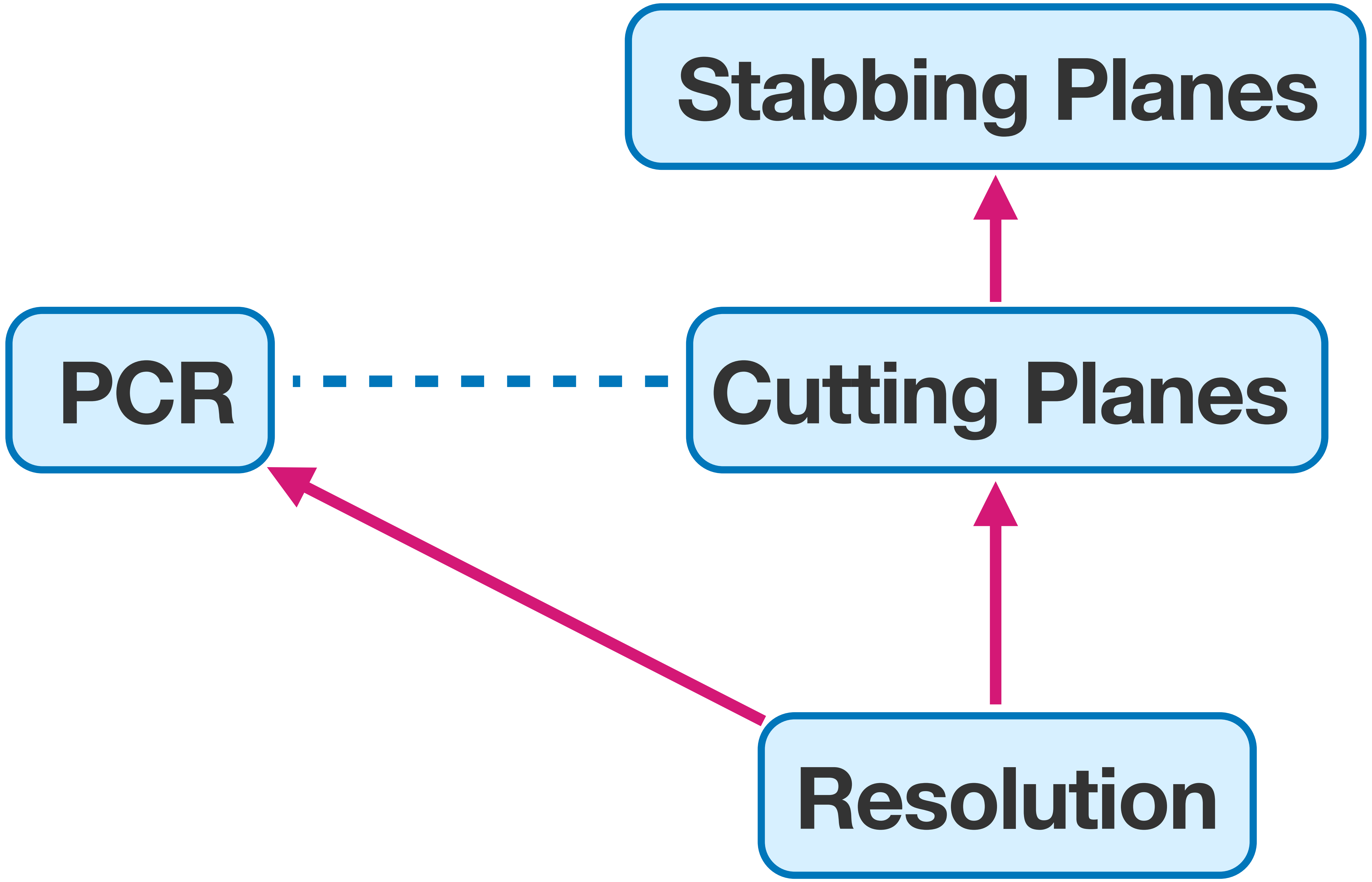


Q. Does there always exist an SP proof of size s and depth d?

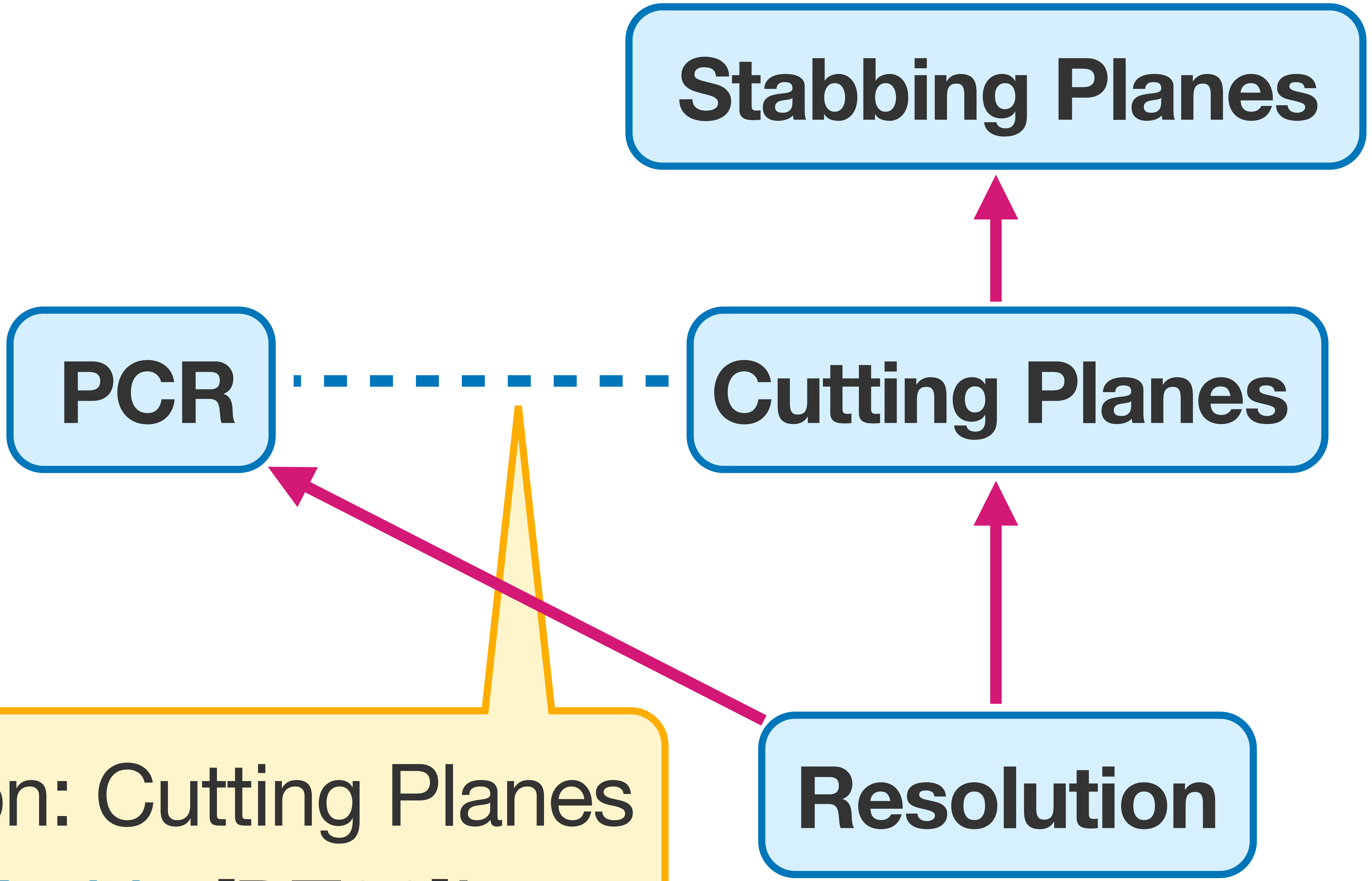
If the CP proof had size s, depth d

→ SP proof has size s, depth s

Comparison of Proof Systems



Comparison of Proof Systems



One direction: Cutting Planes
Can prove Tseitin [DT20]!

Cutting Planes Proves Tseitin!

The Tseitin formulas are the canonical family of formulas hard to prove in many **algebraic** proof systems — e.g. PCR from last time

Cutting Planes Proves Tseitin!

The Tseitin formulas are the canonical family of formulas hard to prove in many **algebraic** proof systems — e.g. PCR from last time

Tseitin Formulas: Let $G = (V, E)$ be a graph, $\sigma : V \to \{0,1\}$ be such that $\Sigma_{v \in V} \sigma(v)$ is odd.

Cutting Planes Proves Tseitin!

The Tseitin formulas are the canonical family of formulas hard to prove in many **algebraic** proof systems — e.g. PCR from last time

Tseitin Formulas: Let $G = (V, E)$ be a graph, $\sigma : V \to \{0,1\}$ be such that $\Sigma_{v \in V} \sigma(v)$ is odd. For each $e \in E$ we have a variable x_e.

Cutting Planes Proves Tseitin!

The Tseitin formulas are the canonical family of formulas hard to prove in many **algebraic** proof systems — e.g. PCR from last time

Tseitin Formulas: Let $G = (V, E)$ be a graph, $\sigma : V \to \{0,1\}$ be such that $\Sigma_{v \in V}\sigma(v)$ is odd. For each $e \in E$ we have a variable x_e.

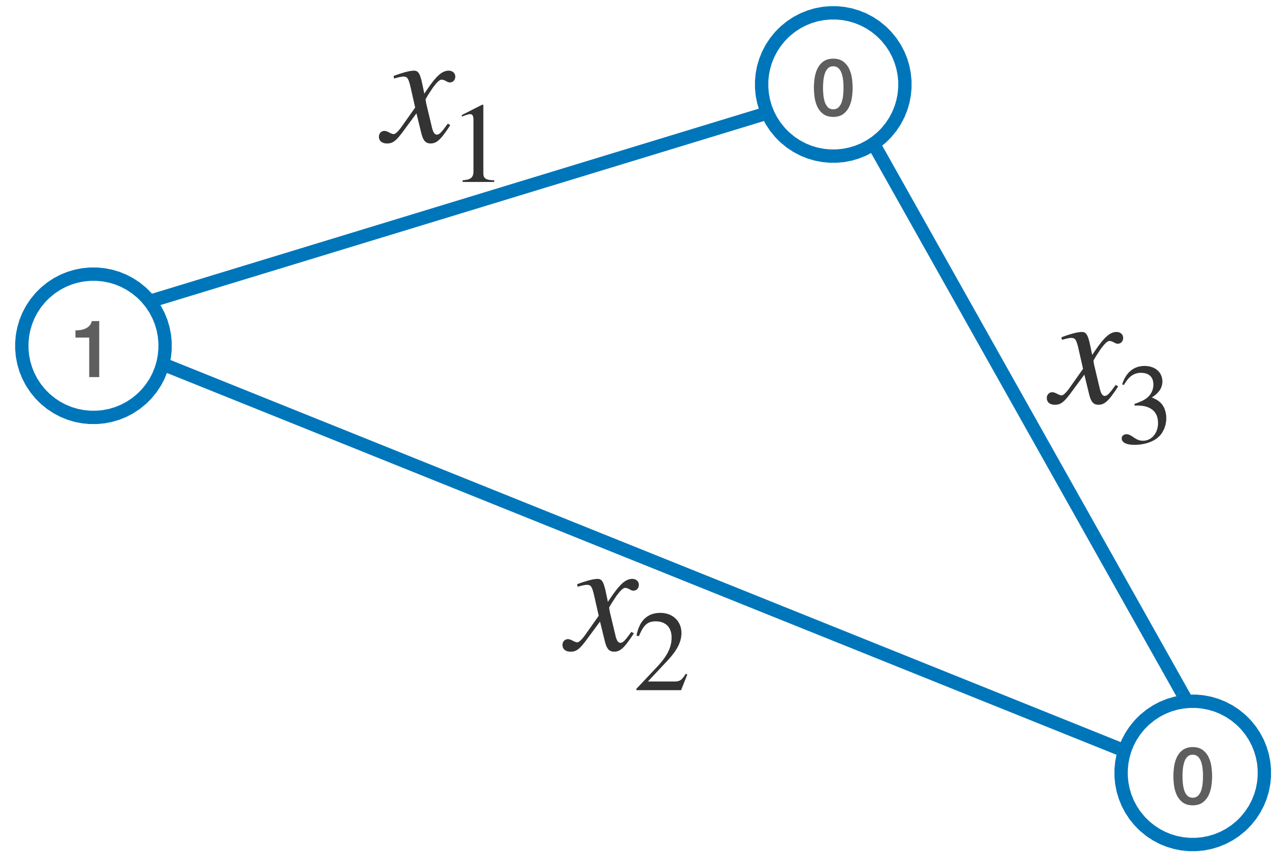
$\text{Tseitin}_{G,\sigma}$: for each vertex $v \in V$, a constraint $\bigoplus_{v \in e} x_e = \sigma(v)$

Cutting Planes Proves Tseitin!

The Tseitin formulas are the canonical family of formulas hard to prove in many **algebraic** proof systems — e.g. PCR from last time

Tseitin Formulas: Let $G = (V, E)$ be a graph, $\sigma : V \to \{0,1\}$ be such that $\Sigma_{v \in V} \sigma(v)$ is odd. For each $e \in E$ we have a variable x_e.

Tseitin$_{G,\sigma}$: for each vertex $v \in V$, a constraint $\bigoplus_{v \in e} x_e = \sigma(v)$

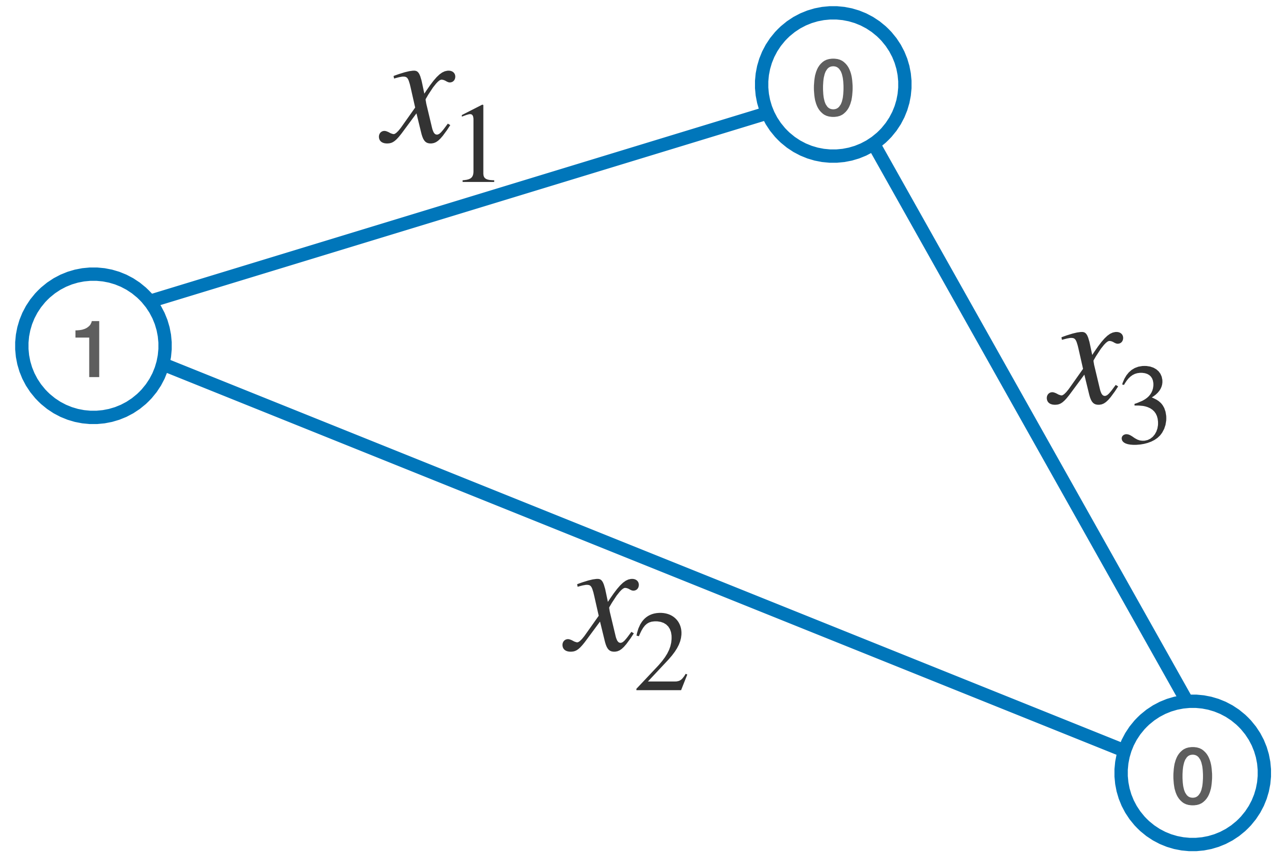


Cutting Planes Proves Tseitin!

The Tseitin formulas are the canonical family of formulas hard to prove in many **algebraic** proof systems — e.g. PCR from last time

Tseitin Formulas: Let $G = (V, E)$ be a graph, $\sigma : V \to \{0,1\}$ be such that $\Sigma_{v \in V} \sigma(v)$ is odd. For each $e \in E$ we have a variable x_e.

Tseitin$_{G,\sigma}$: for each vertex $v \in V$, a constraint $\bigoplus_{v \in e} x_e = \sigma(v)$



$$x_1 \oplus x_2 = 1$$
$$x_1 \oplus x_3 = 0$$
$$x_2 \oplus x_3 = 0$$

Cutting Planes Proves Tseitin!

Thm[DT20]**:** There are quasipolynomial size Cutting Planes proofs of $\text{Tseitin}_{G,\sigma}$

High Level:

1. Exhibit a quasipolynomial size Stabbing Planes proof of Tseitin
2. Translate that proof into Cutting Planes

Stabbing Planes Proves Tseitin

Thm[BFI+18]**:** There are quasipolynomial size Stabbing Planes proofs of $\text{Tseitin}_{G,\sigma}$

Stabbing Planes Proves Tseitin

Thm[BFI+18]**:** There are quasipolynomial size Stabbing Planes proofs of Tseitin$_{G,\sigma}$

1. We describe an algorithm that, given an assignment $y \in \{0,1\}^n$, finds a falsified constraint of Tseitin$_{G,\sigma}(y)$

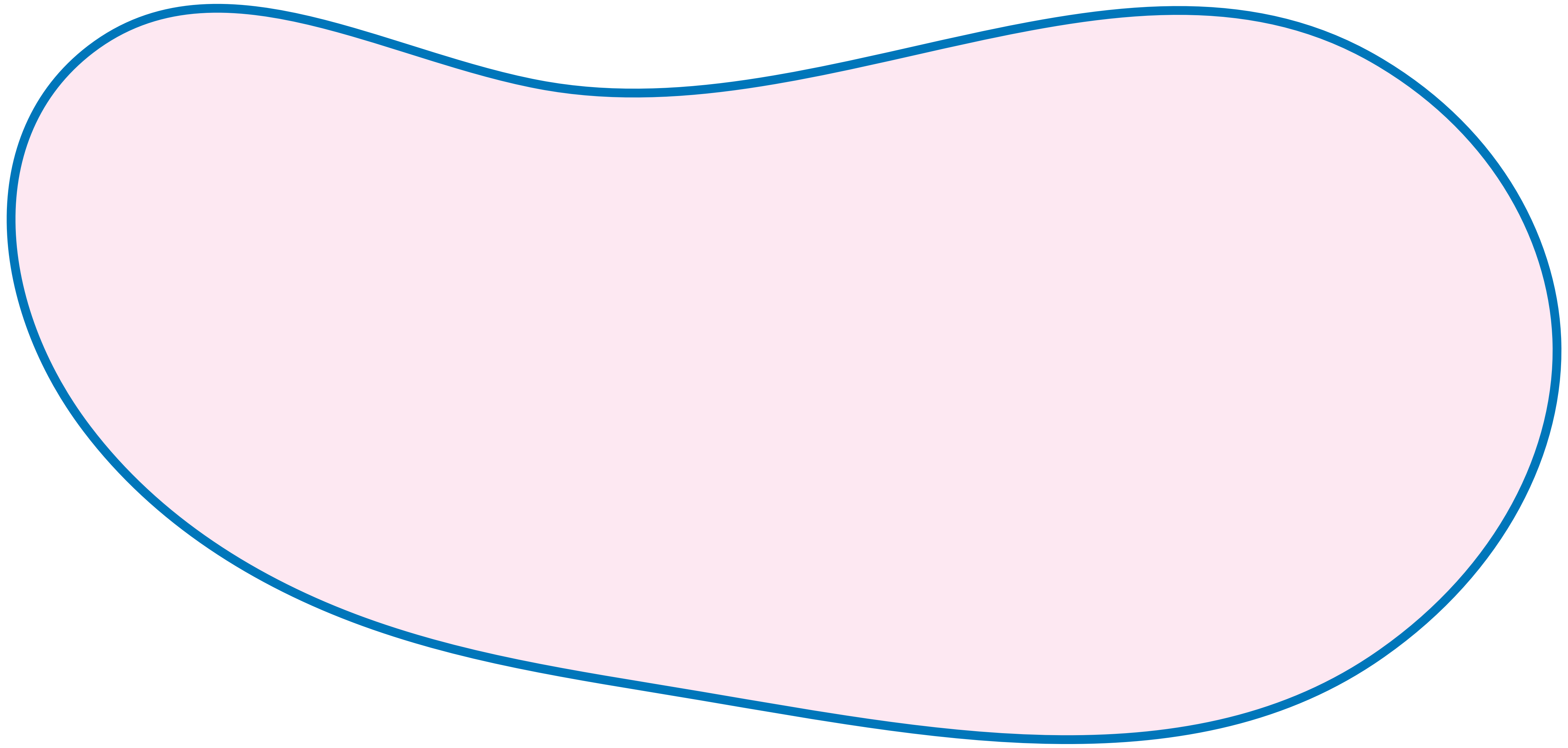
Stabbing Planes Proves Tseitin

Thm[BFI+18]**:** There are quasipolynomial size Stabbing Planes proofs of Tseitin$_{G,\sigma}$

1. We describe an algorithm that, given an assignment $y \in \{0,1\}^n$, finds a falsified constraint of Tseitin$_{G,\sigma}(y)$
2. "Implement" the algorithm in Stabbing Planes

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

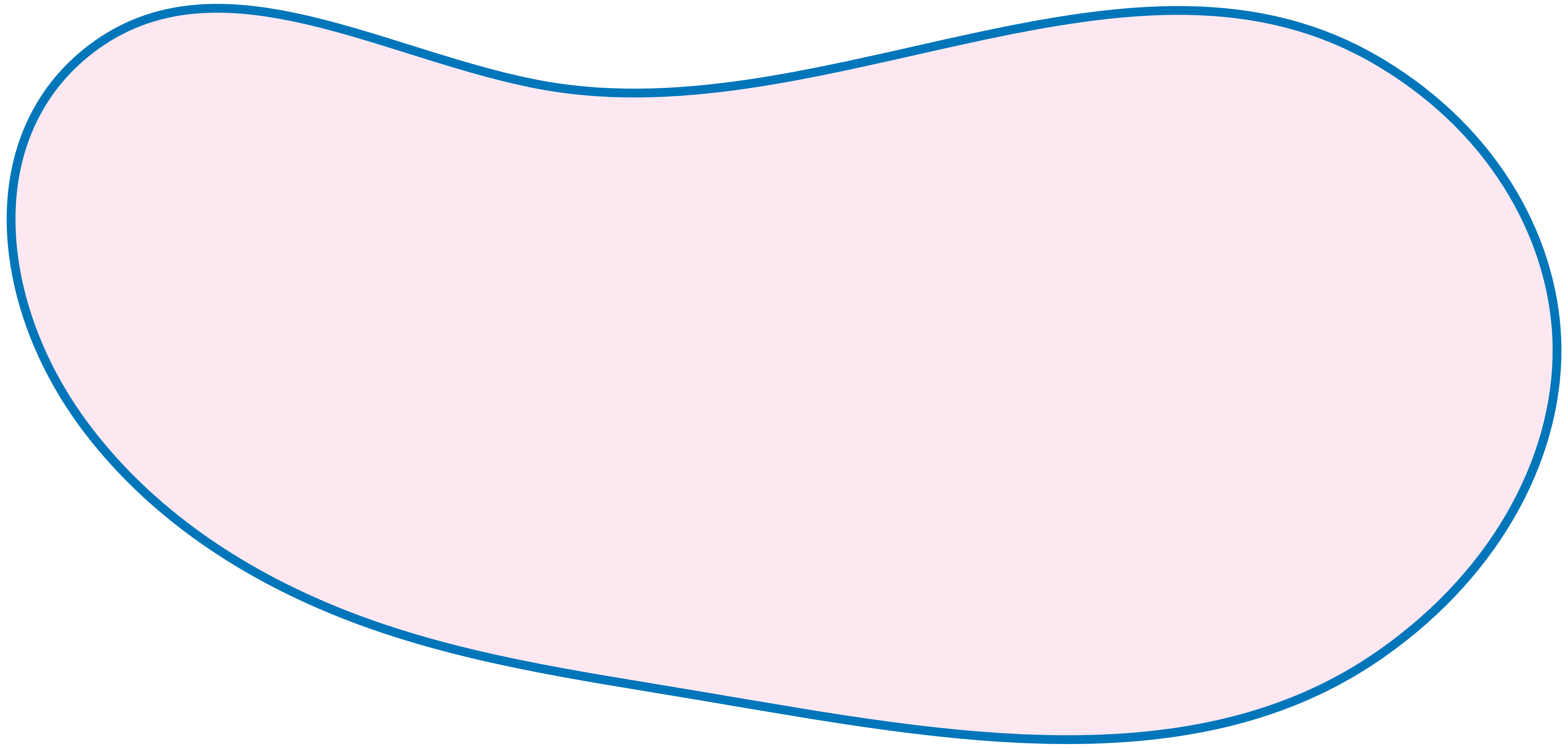


$G = (V, E)$

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint



$G = (V, E)$

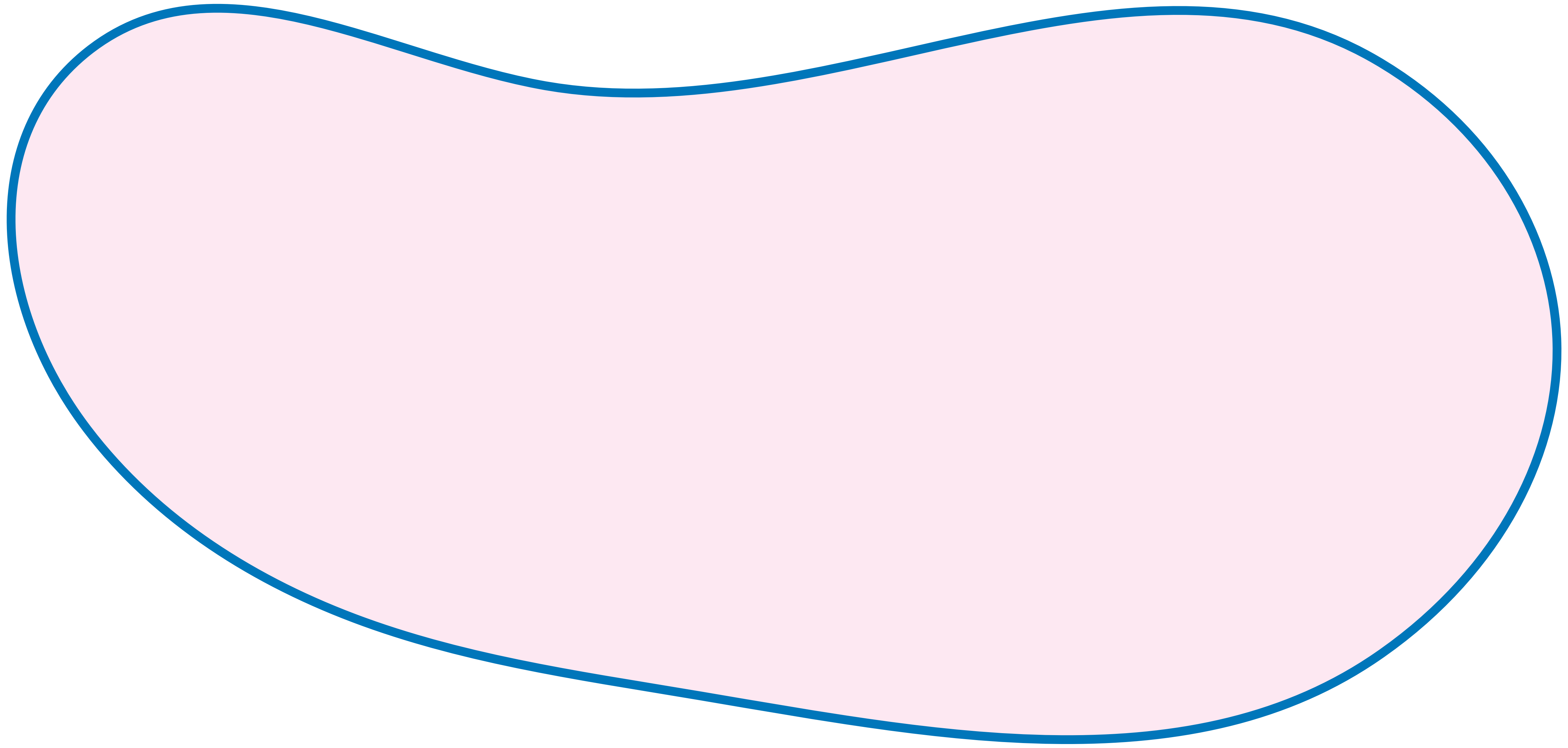
Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

- Each round maintains: $U \subseteq V$



$G = (V, E)$

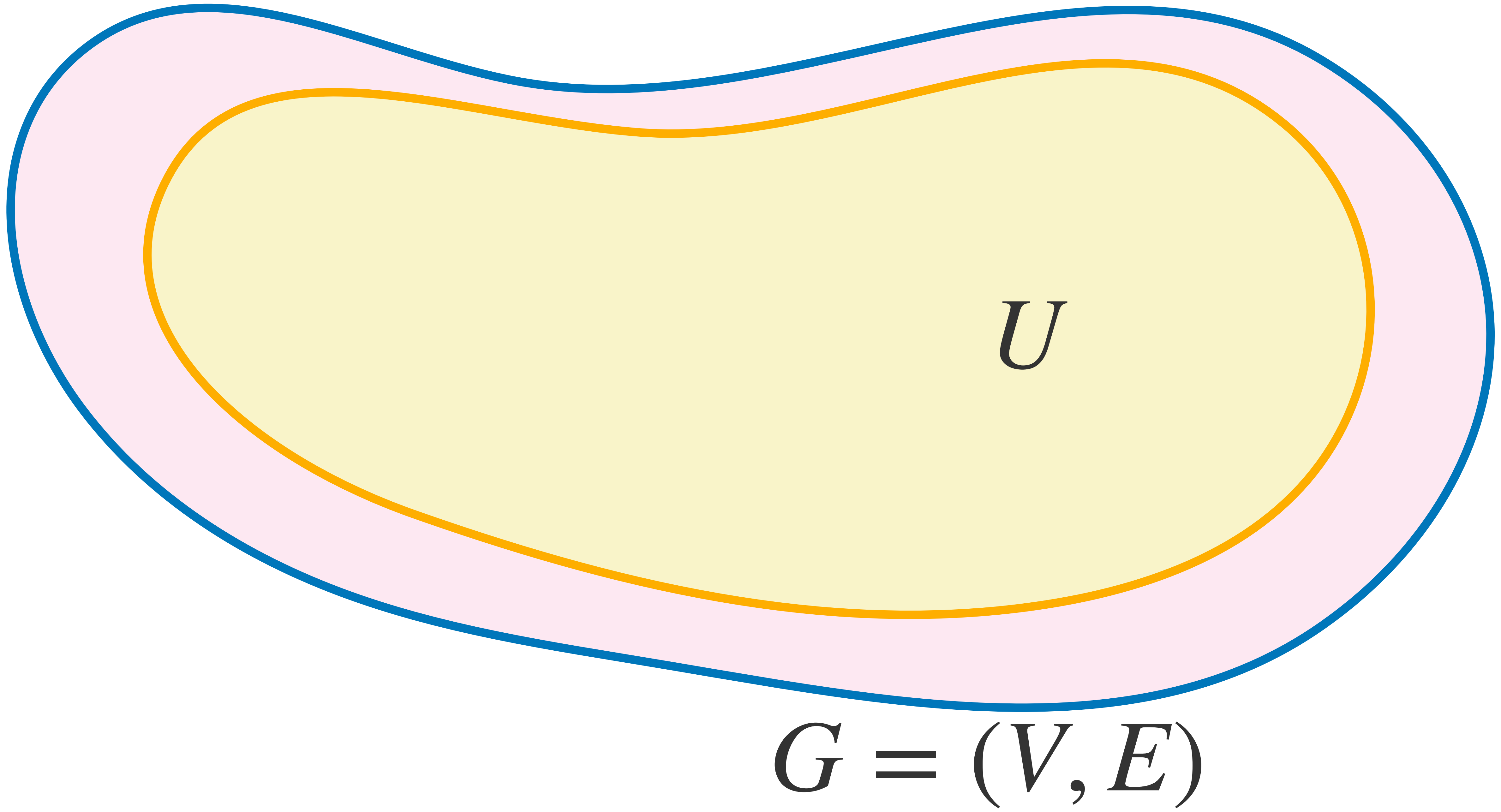
Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ — a falsified constraint

Algorithm proceeds in rounds:

- Each round maintains: $U \subseteq V$



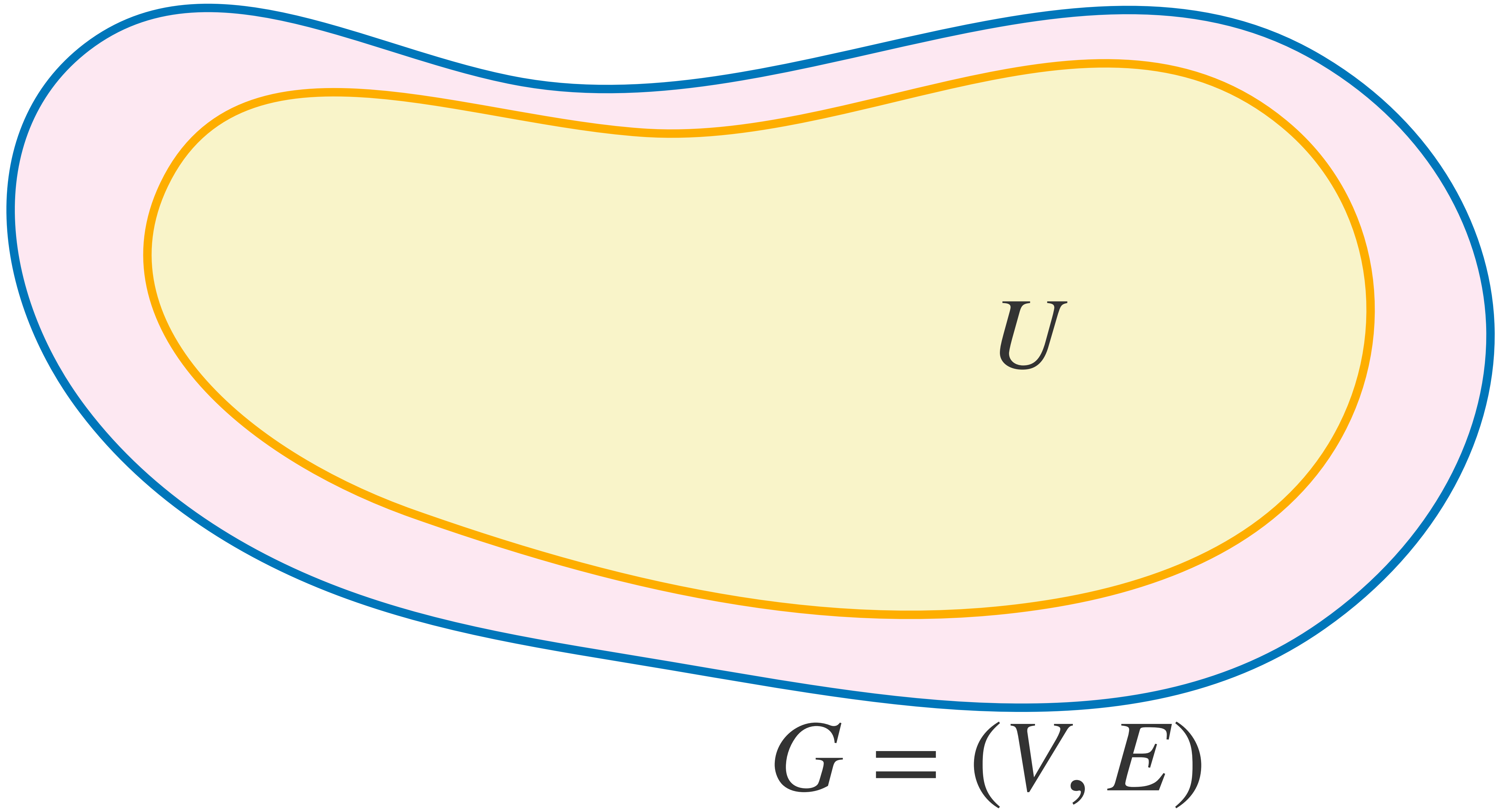
Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \setminus U]} \, y_e$



Algorithm for Finding Falsified Clause

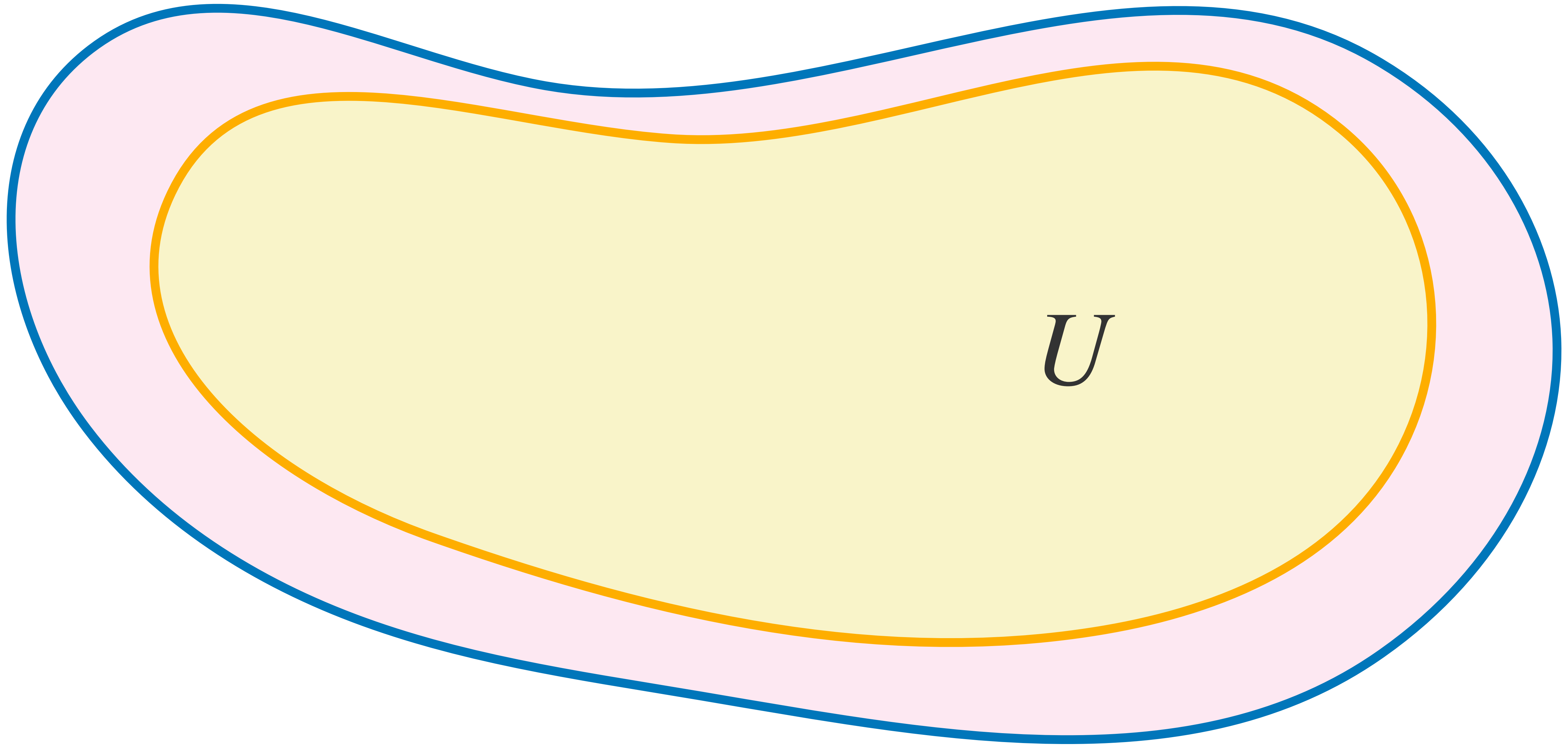
Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \backslash U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \backslash U]} y_e$



$G = (V, E)$

Algorithm for Finding Falsified Clause

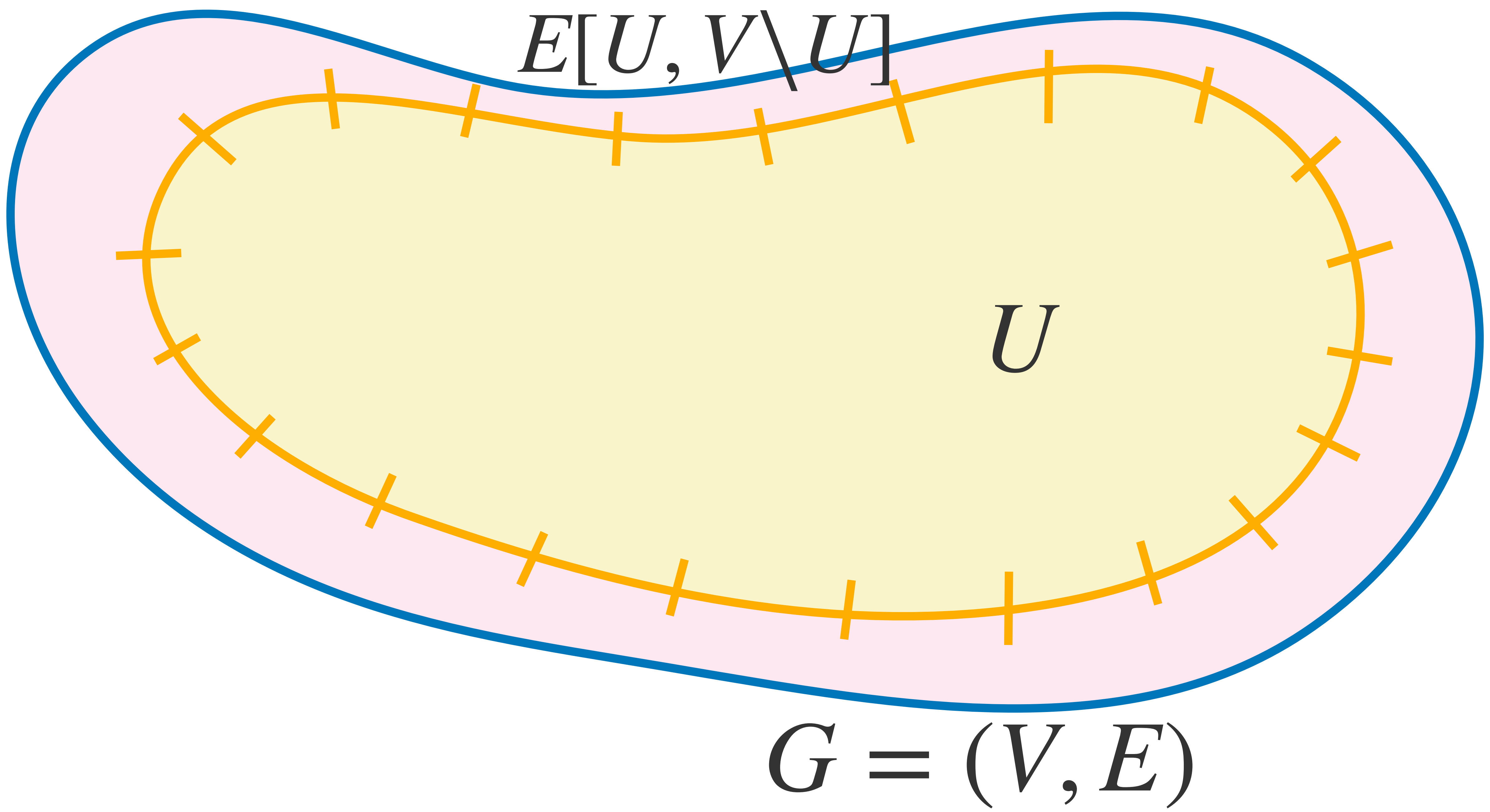
Given: $y \in \{0,1\}^n$ to the variables of Tseitin$_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \setminus U]}\, y_e$



Algorithm for Finding Falsified Clause

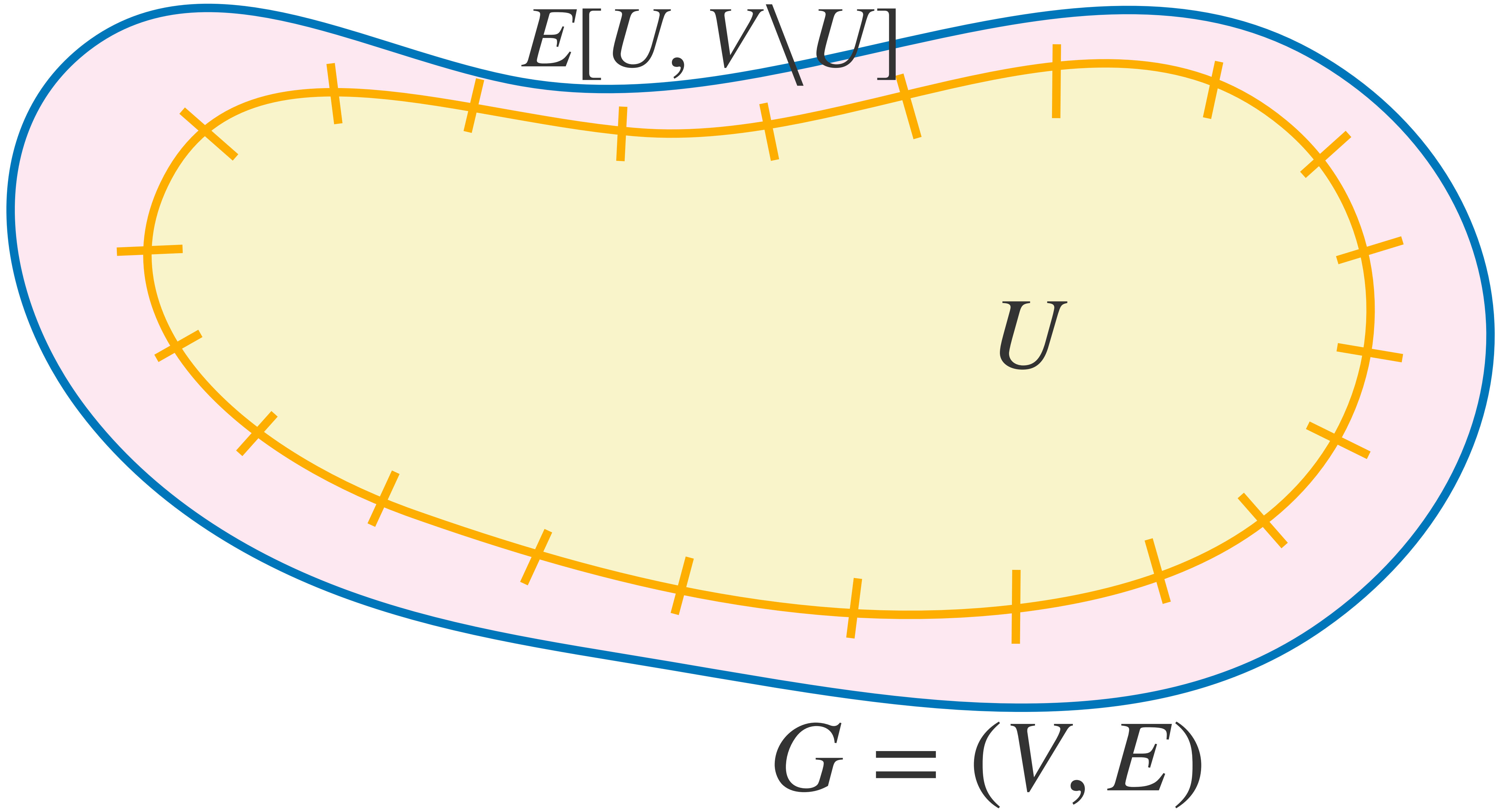
Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \sum_{e \in E[U, V\setminus U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$



Algorithm for Finding Falsified Clause

$\sigma(U) := \Sigma_{v \in U} \sigma(v)$

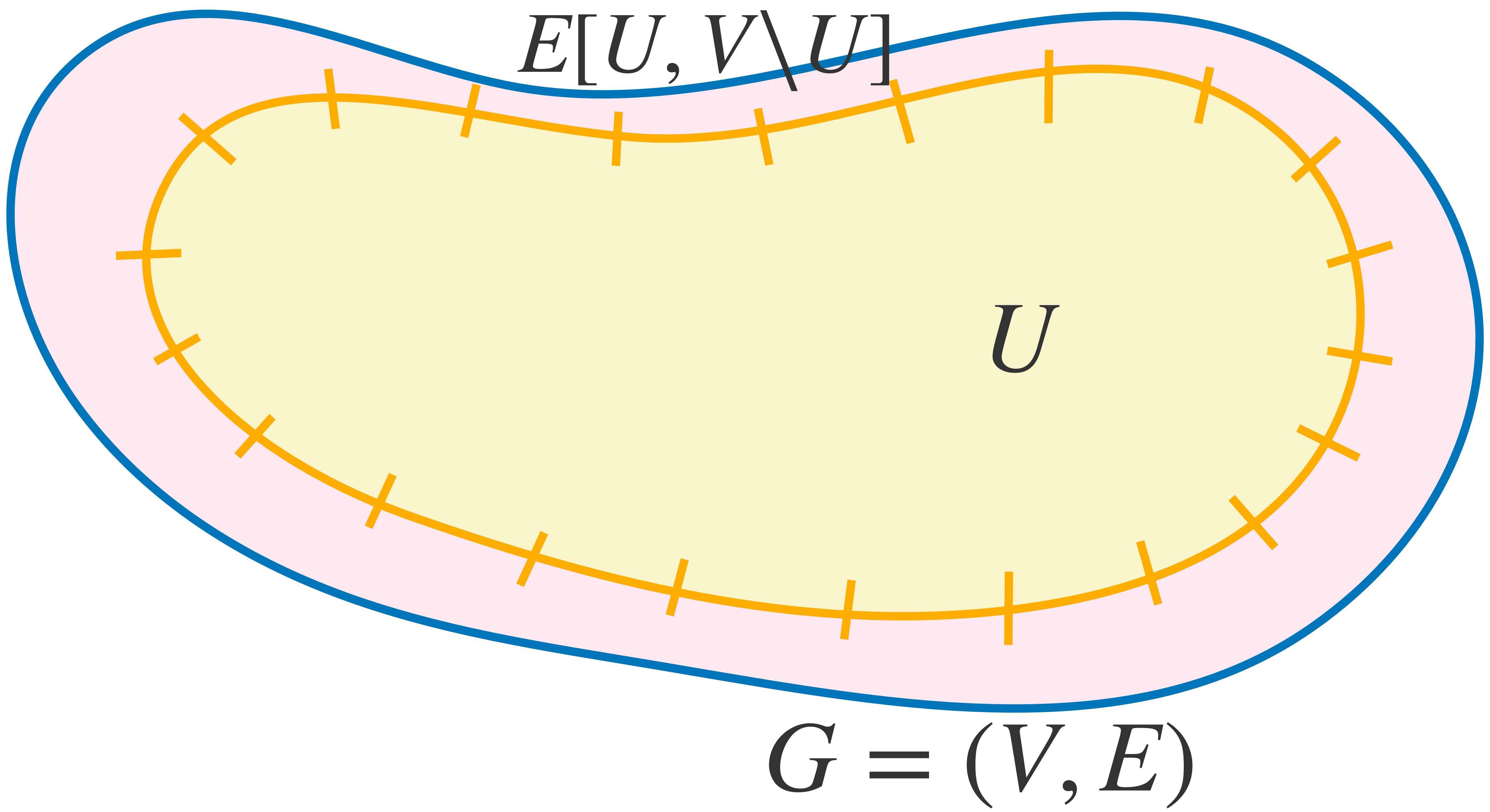
Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \setminus U]} \, y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$



Algorithm for Finding Falsified Clause

$\sigma(U) := \Sigma_{v \in U} \sigma(v)$

Given: $y \in \{0,1\}^n$ to the variables of Tseitin$_{G,\sigma}$

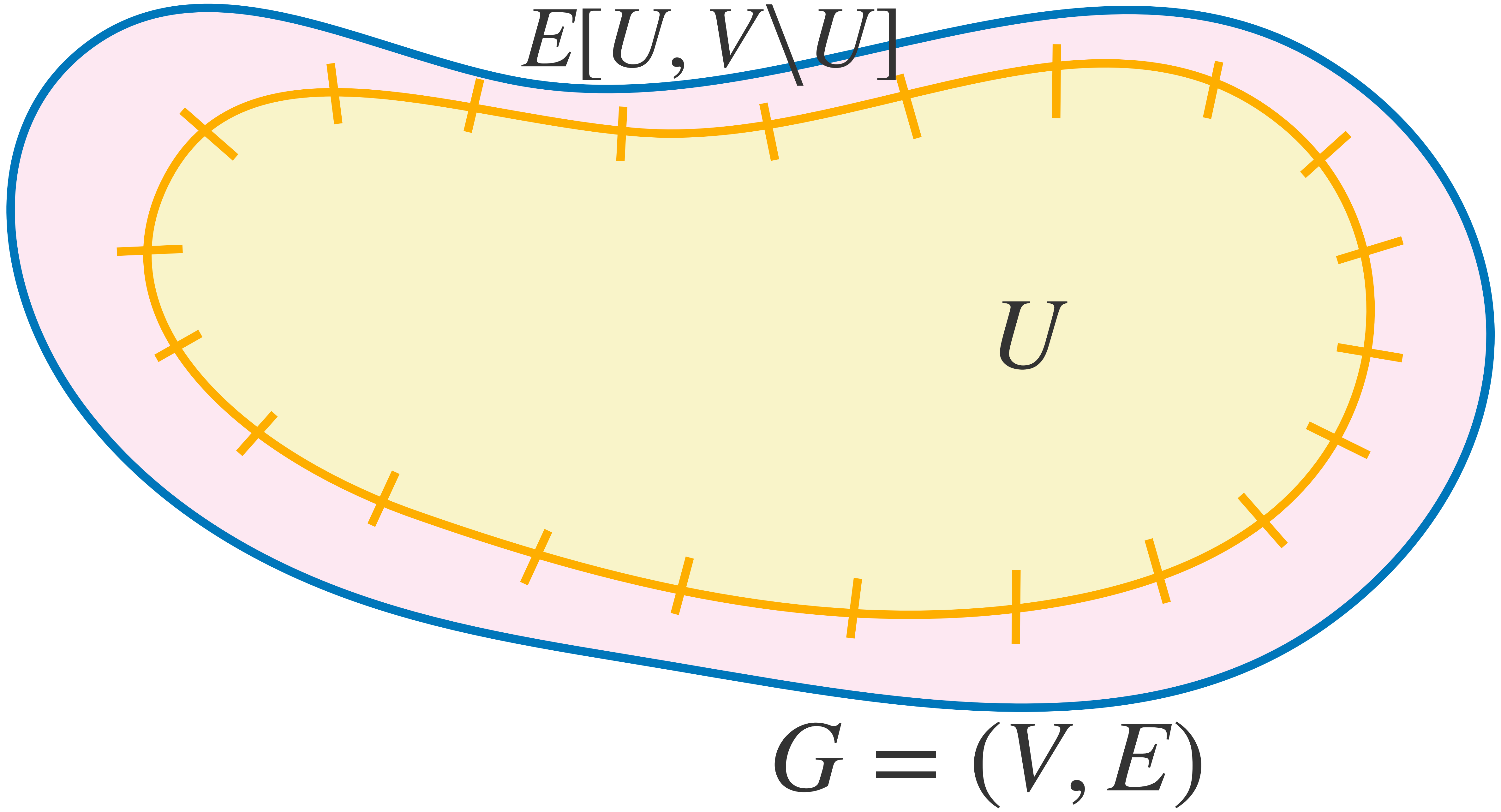
Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \backslash U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U,V \backslash U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

$\Longrightarrow$ There is a falsified constraint in U!



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

$\sigma(U) := \Sigma_{v \in U} \sigma(v)$

Goal: find $v \in V$ such that $\bigoplus_{e \ni v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

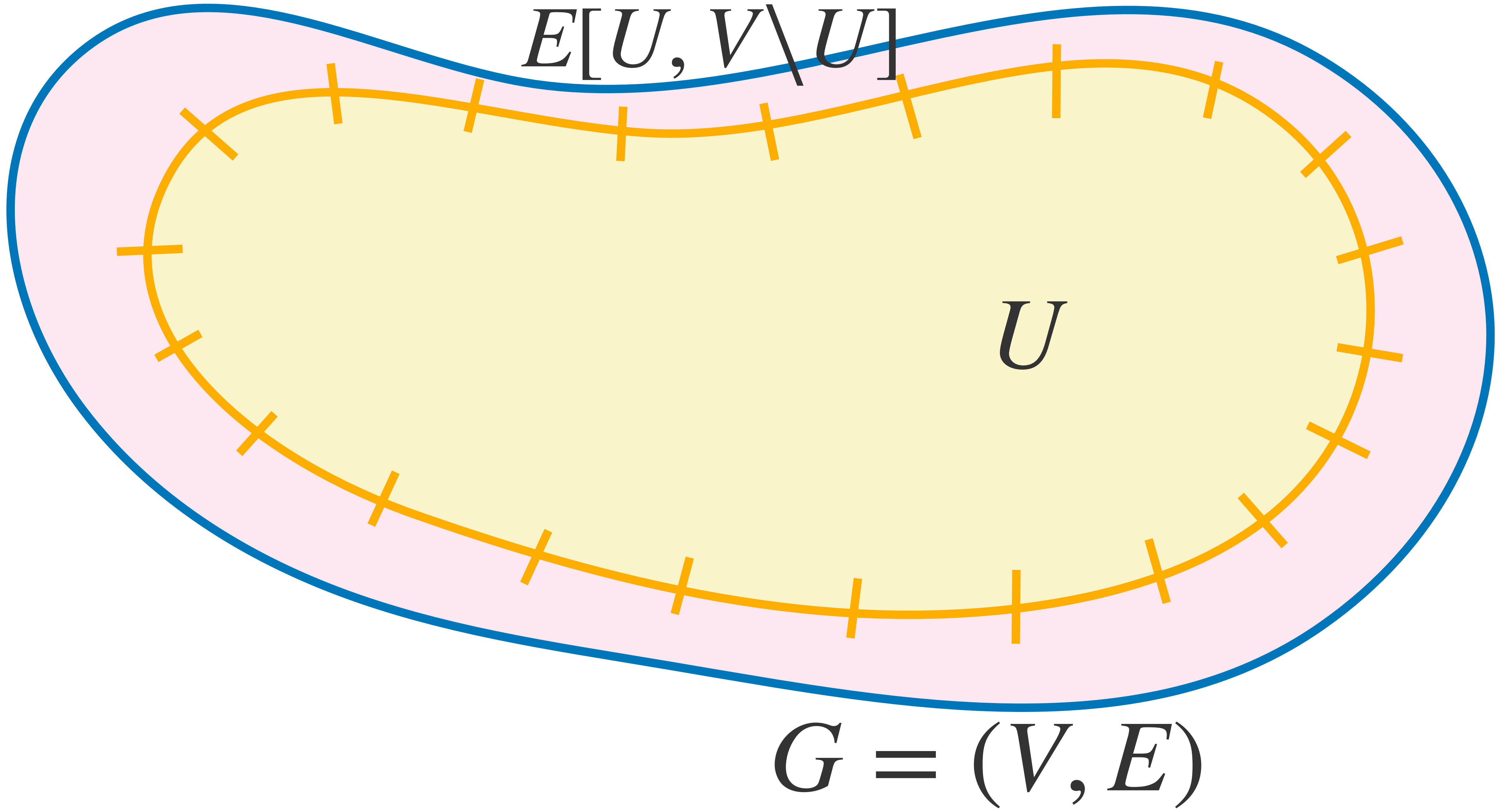
Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V\setminus U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

$\Longrightarrow$ There is a falsified constraint in U!

For U to be satisfiable we need

$\Sigma_{v \in U} \sigma(v) \equiv \Sigma_{v \in U} \Sigma_{v \in e} y_e \bmod 2$



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

$\sigma(U) := \Sigma_{v \in U} \sigma(v)$

Goal: find $v \in V$ such that $\bigoplus_{e \ni v} y_e \neq \sigma(v)$ — a falsified constraint

Algorithm proceeds in rounds:

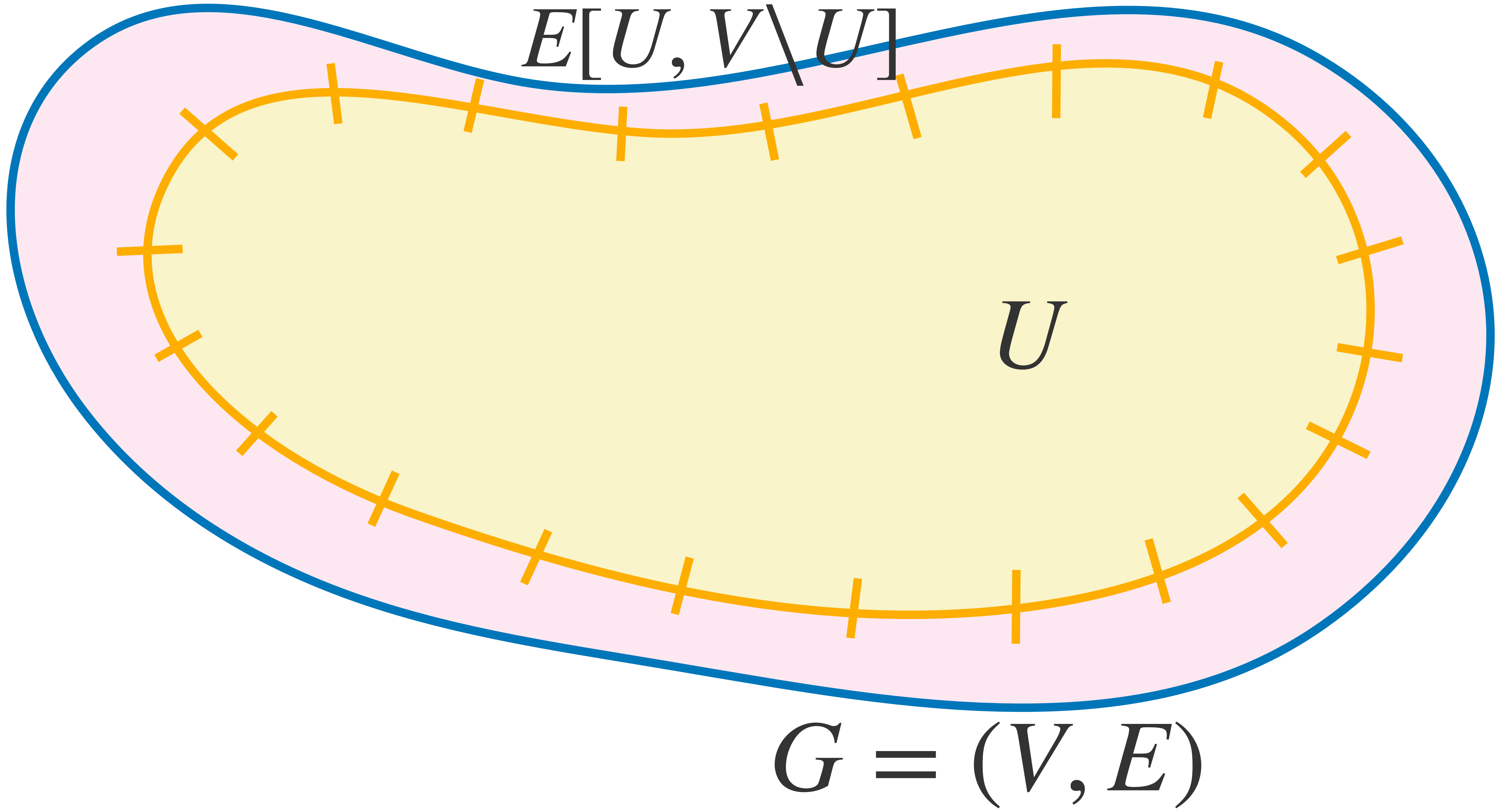
Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V\setminus U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

$\Longrightarrow$ There is a falsified constraint in U!

For U to be satisfiable we need

$\sigma(U) \equiv \Sigma_{v \in U} \Sigma_{v \in e} y_e \bmod 2$



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

$\sigma(U) := \Sigma_{v \in U} \sigma(v)$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

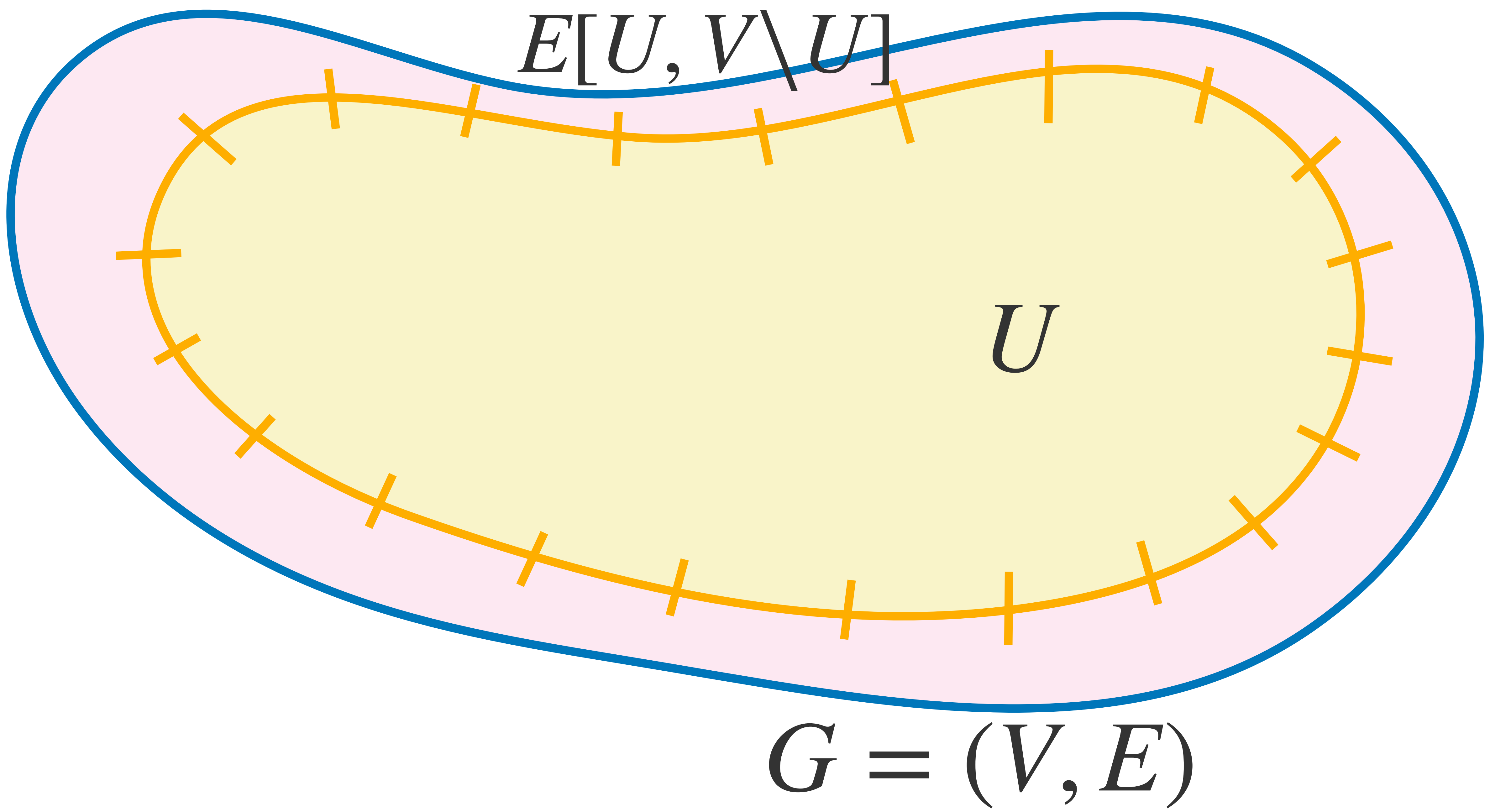
- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V\setminus U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

$\Longrightarrow$ There is a falsified constraint in U!

For U to be satisfiable we need

$$\sigma(U) \equiv \Sigma_{v \in U} \Sigma_{v \in e} y_e \bmod 2$$

$$\equiv 2 \sum_{u,v \in E} y_{uv} + \sum_{e \in E[U, V\setminus U]} y_e \bmod 2$$



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of Tseitin$_{G,\sigma}$

$\sigma(U) := \Sigma_{v \in U} \sigma(v)$

Goal: find $v \in V$ such that $\bigoplus_{e \ni v} y_e \neq \sigma(v)$ — a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V\setminus U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

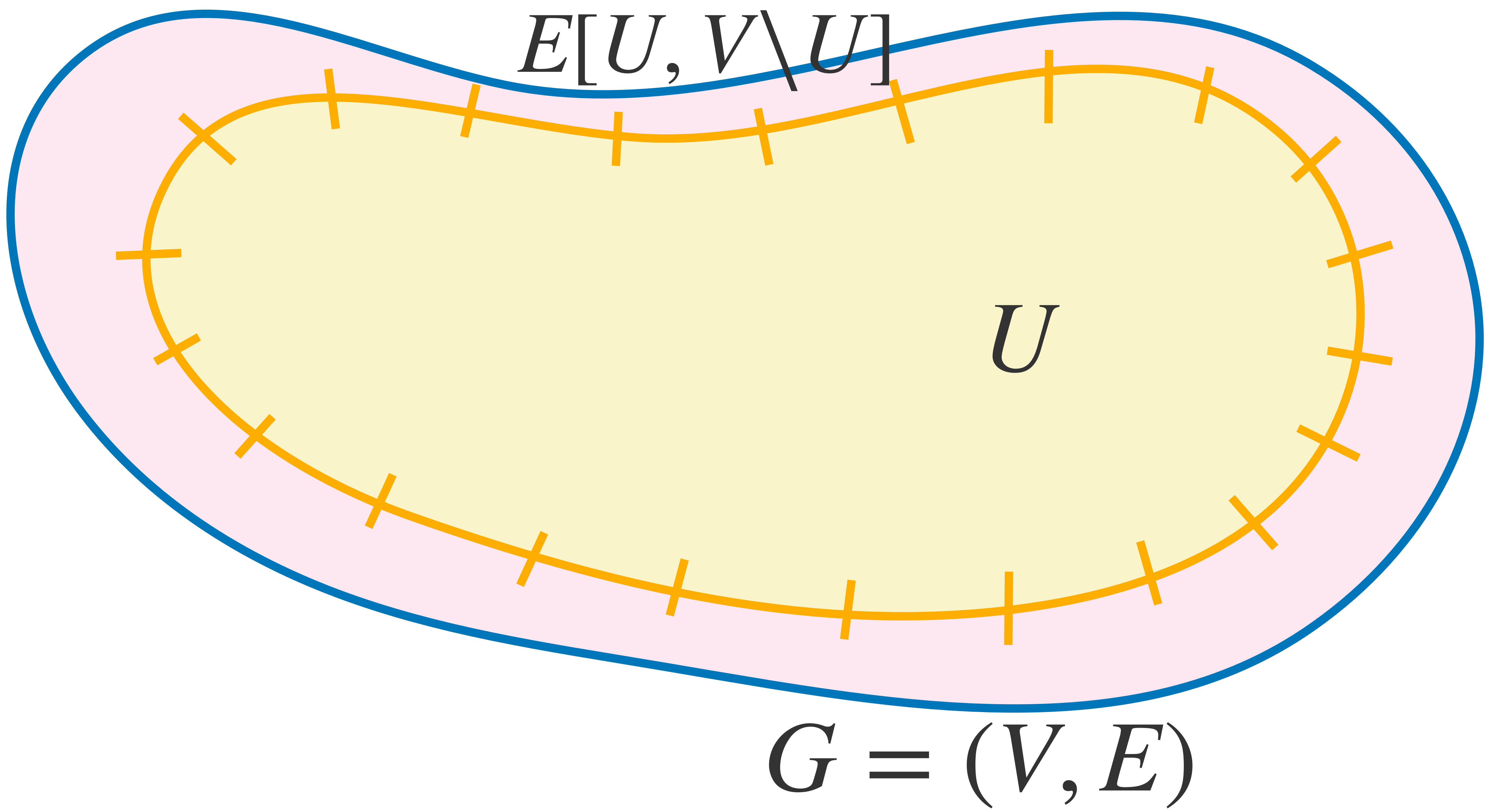
$\Longrightarrow$ There is a falsified constraint in U!

For U to be satisfiable we need

$$\sigma(U) \equiv \Sigma_{v \in U} \Sigma_{v \in e} y_e \bmod 2$$

$$\equiv 2 \sum_{u,v \in E} y_{uv} + \sum_{e \in E[U, V\setminus U]} y_e \bmod 2$$

$$\equiv 0 + \sum_{e \in E[U, V\setminus U]} y_e \bmod 2$$



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of Tseitin$_{G,\sigma}$

$\sigma(U) := \Sigma_{v \in U} \sigma(v)$

Goal: find $v \in V$ such that $\bigoplus_{e \ni v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V\setminus U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

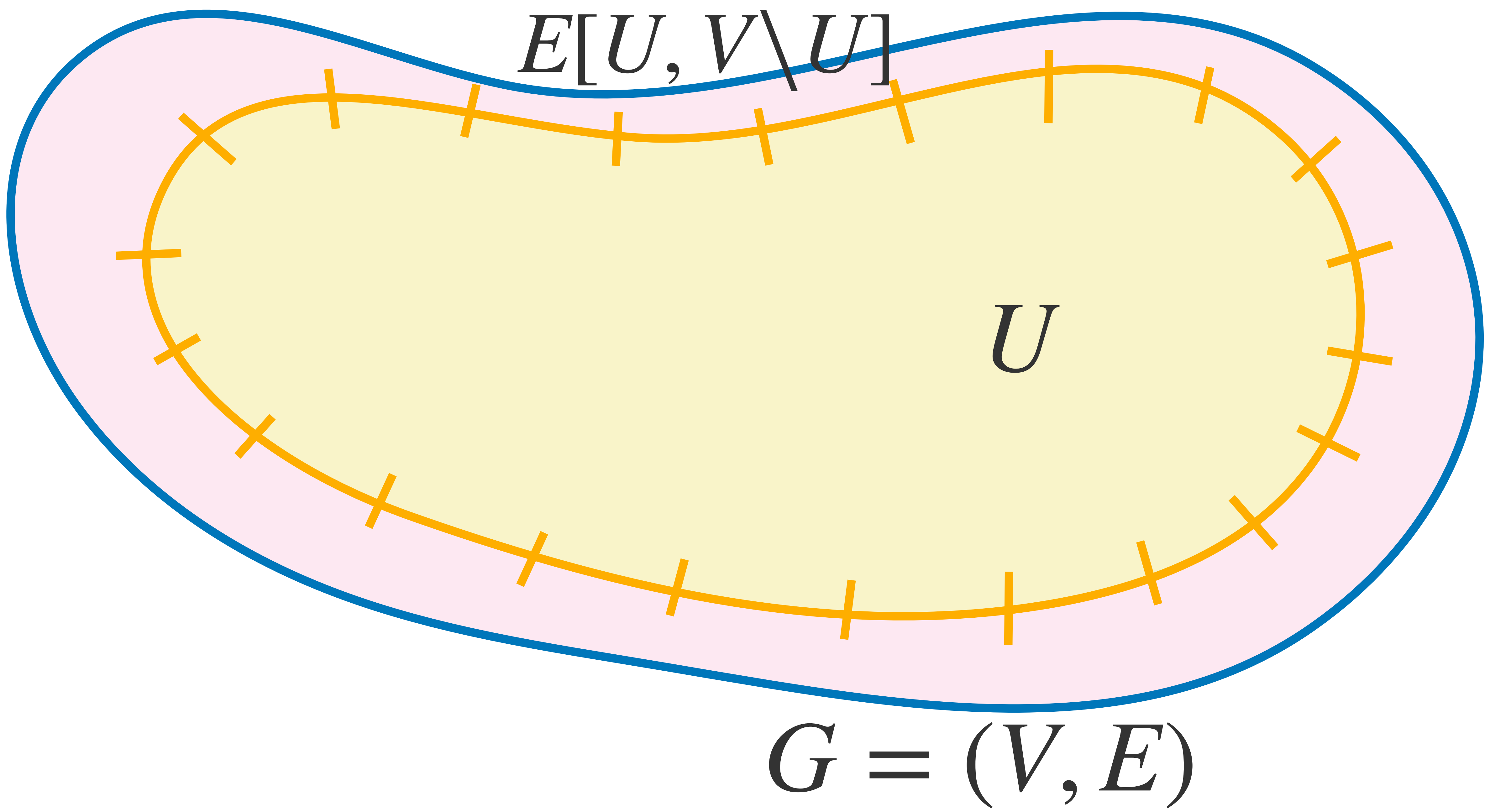
$\Longrightarrow$ There is a falsified constraint in U!

For U to be satisfiable we need

$$\sigma(U) \equiv \Sigma_{v \in U} \Sigma_{v \in e} y_e \bmod 2$$

$$\equiv 2 \sum_{u,v \in E} y_{uv} + \sum_{e \in E[U,V\setminus U]} y_e \bmod 2$$

$$\equiv 0 + \kappa_U \bmod 2$$



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

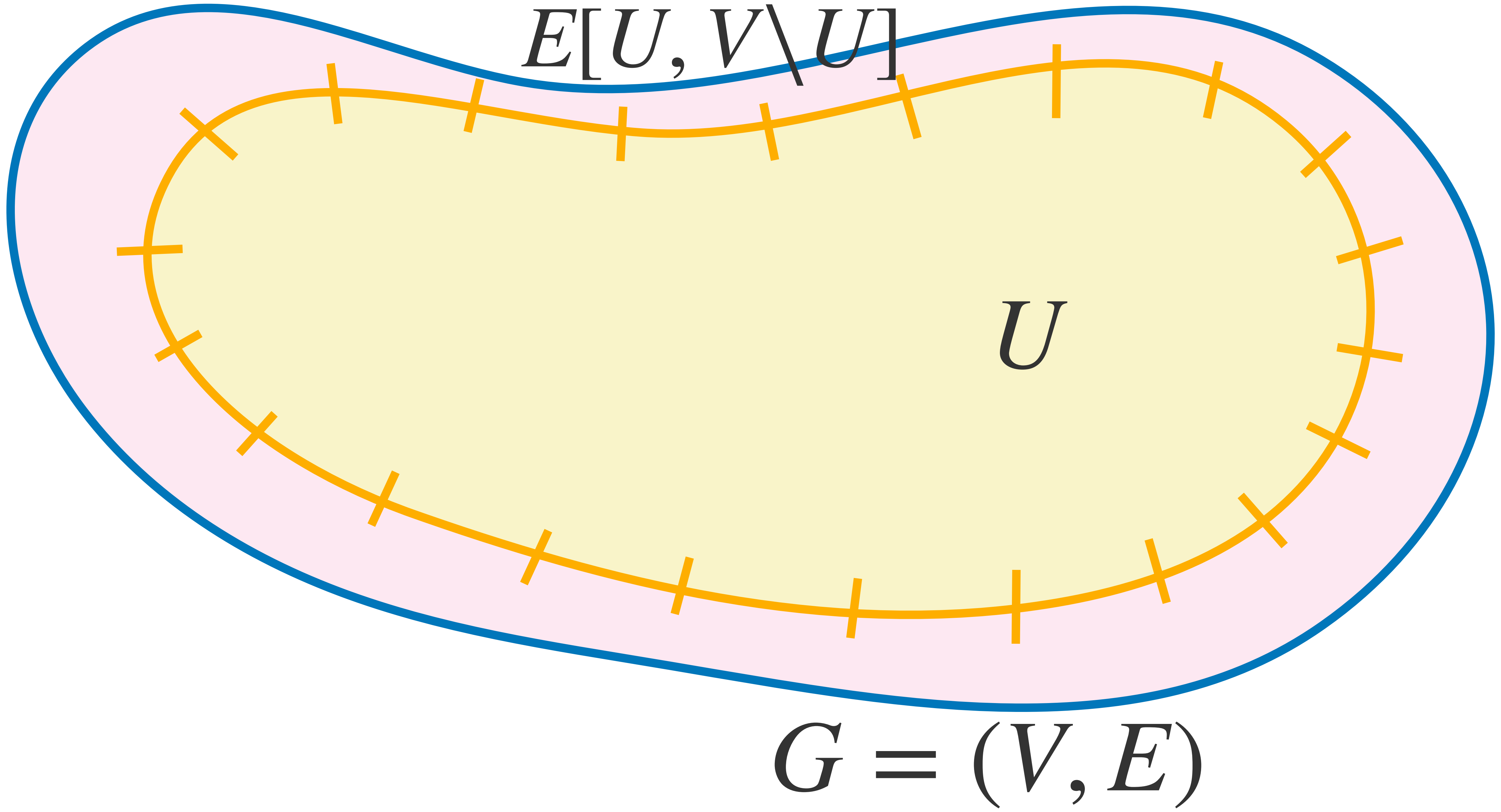
$\sigma(U) := \Sigma_{v \in U} \sigma(v)$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \backslash U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \backslash U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$
- Initially $U = V$ and $\kappa_U = 0$



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

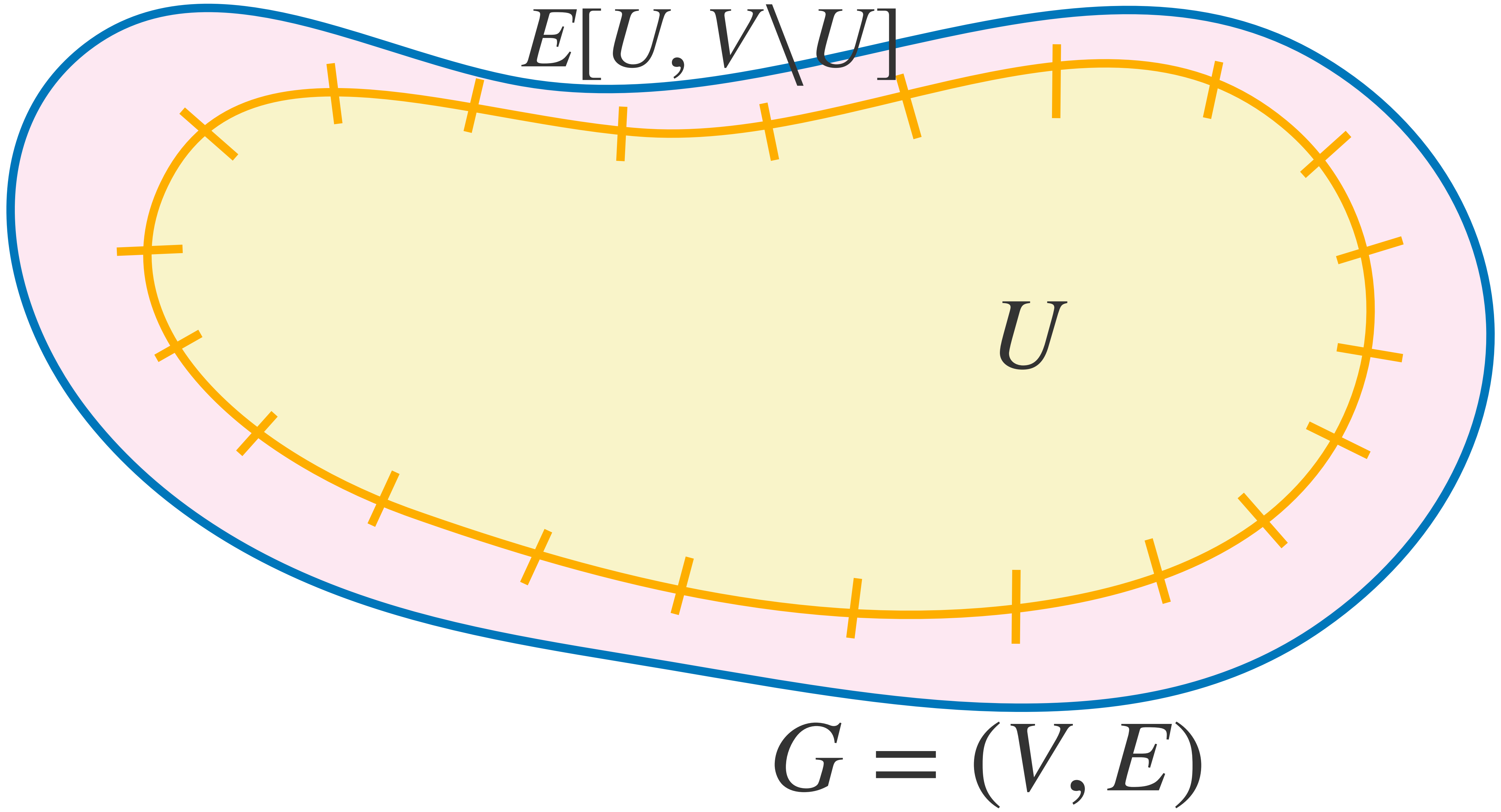
Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \sum_{e \in E[U, V \setminus U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$
- Initially $U = V$ and $\kappa_U = 0$

→ Each round we divide U in half



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\mathrm{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

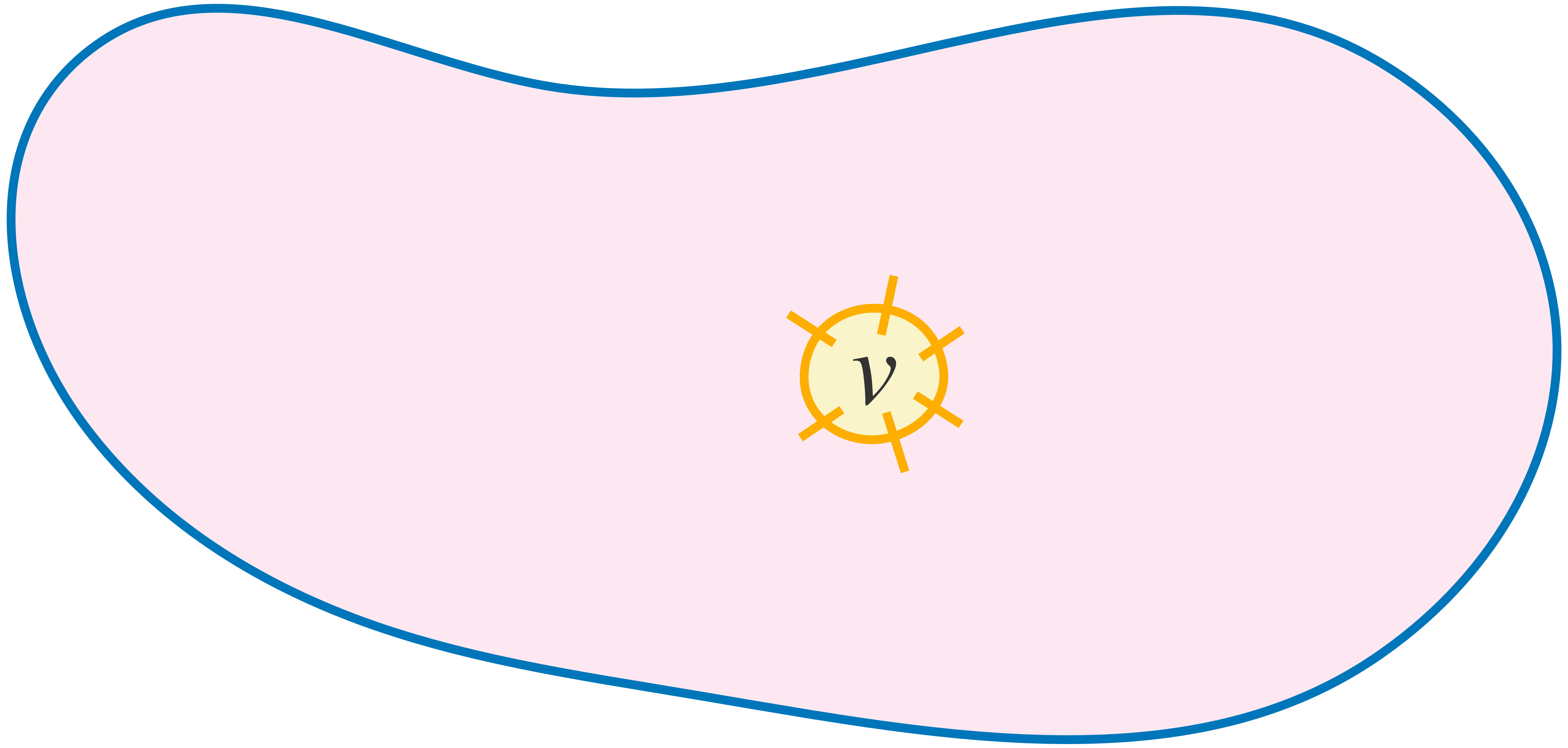
Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \sum_{e \in E[U, V\setminus U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$
- Initially $U = V$ and $\kappa_U = 0$

→ Each round we divide U in half

→ Once $U = \{v\}$



$G = (V, E)$

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \ni v} y_e \neq \sigma(v)$ – a falsified constraint

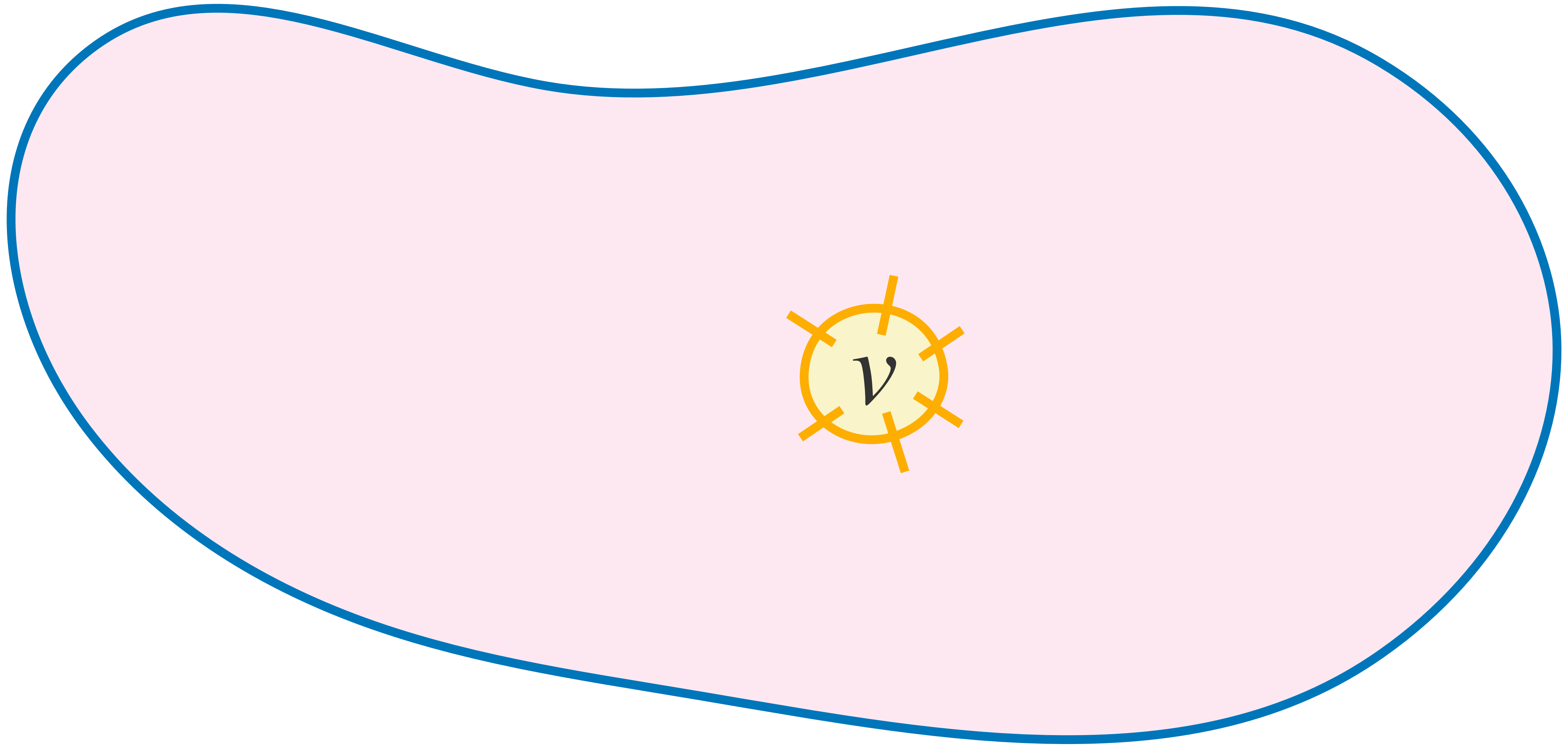
Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V\setminus U]}\, y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$
- Initially $U = V$ and $\kappa_U = 0$

→ Each round we divide U in half

→ Once $U = \{v\}$ we have a vertex such that $\sigma(v) \neq (\kappa_v = \Sigma_{e: v \in e} y_e) \bmod 2$, a falsified constraint!



$G = (V, E)$

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

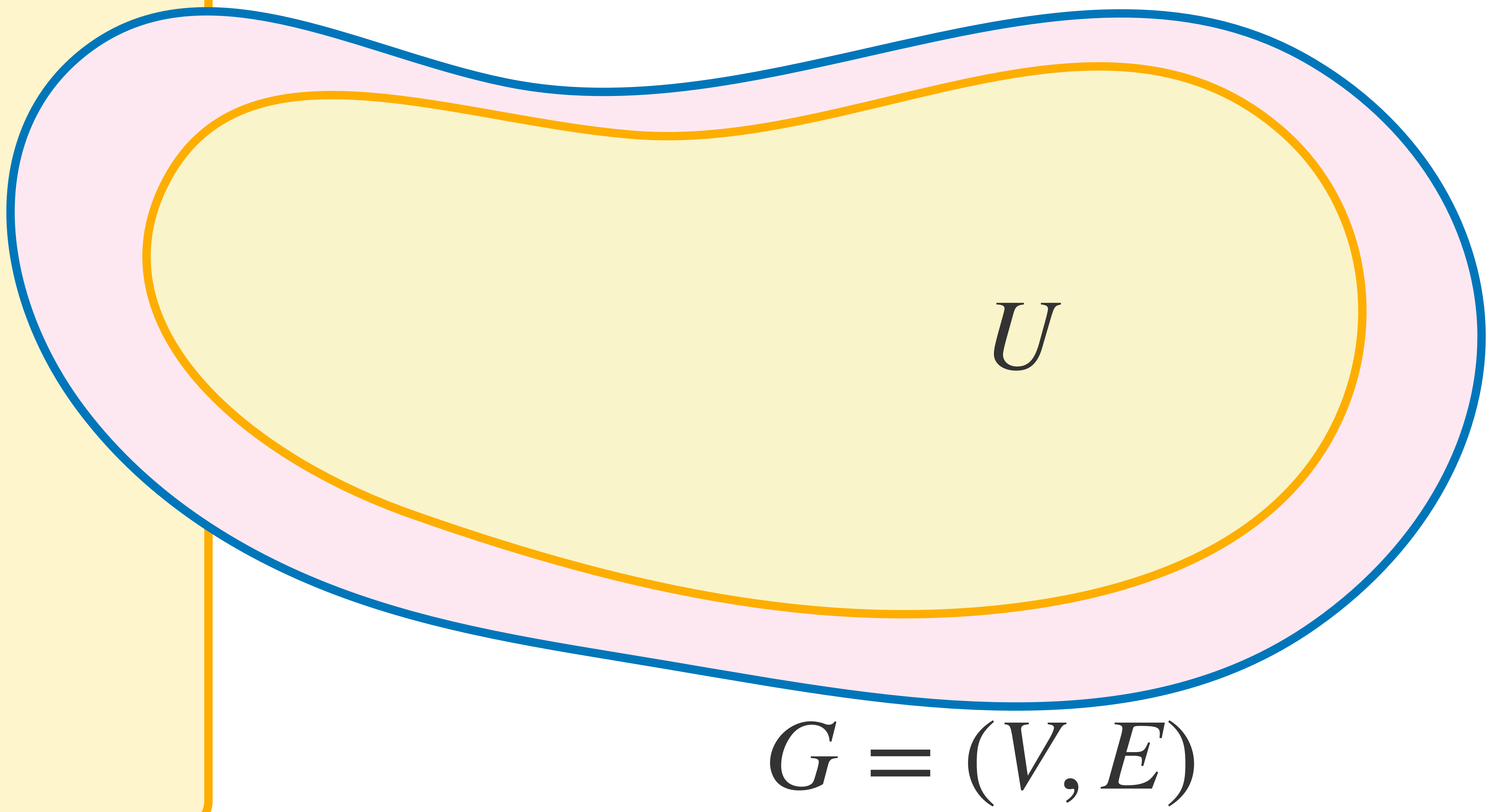
Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \setminus U]}\, y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1.



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of Tseitin$_{G,\sigma}$

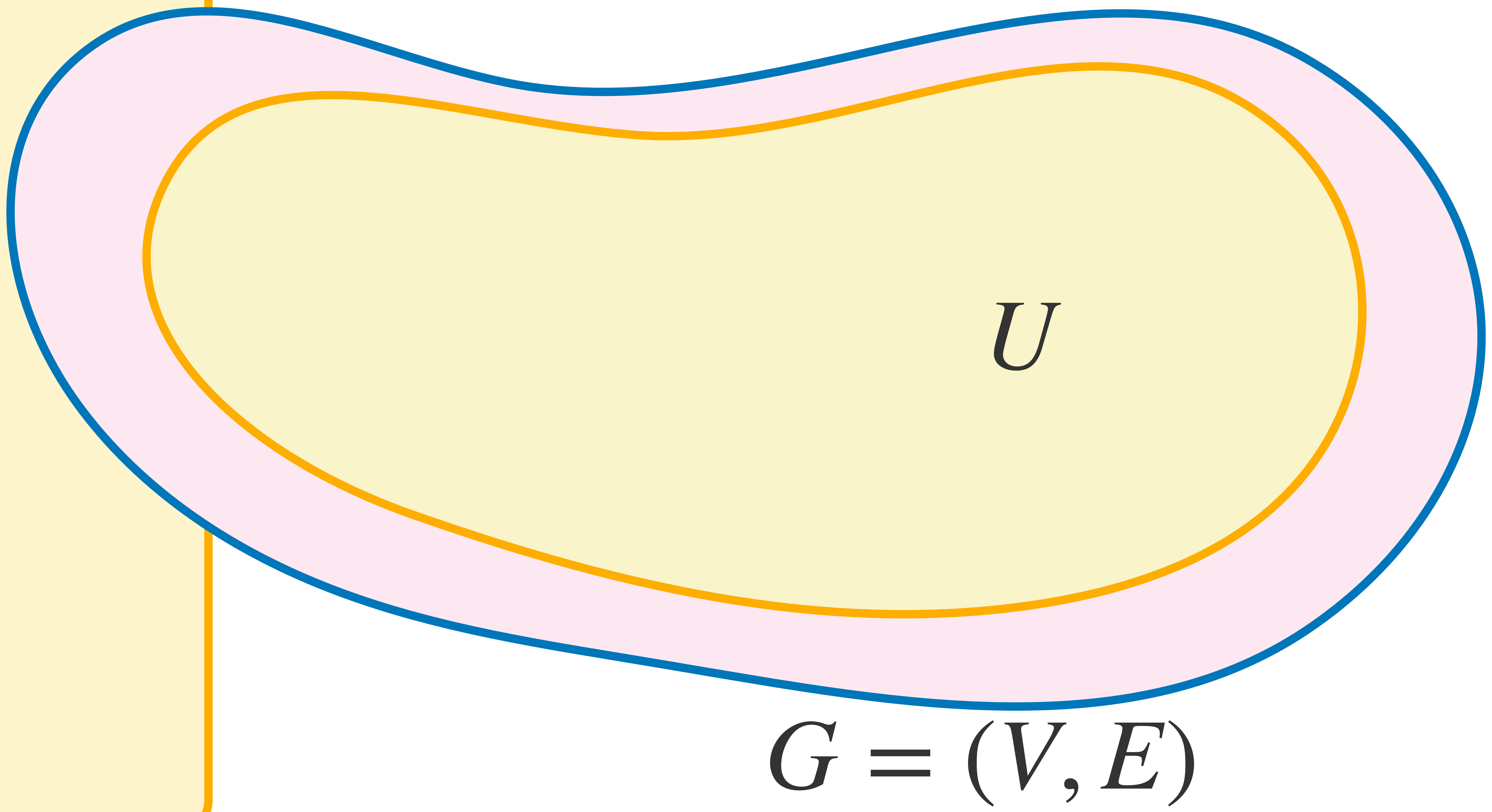
Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ — a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \backslash U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V\backslash U]}\, y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1. Pick a balanced partition $U = U_1 \cup U_2$



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

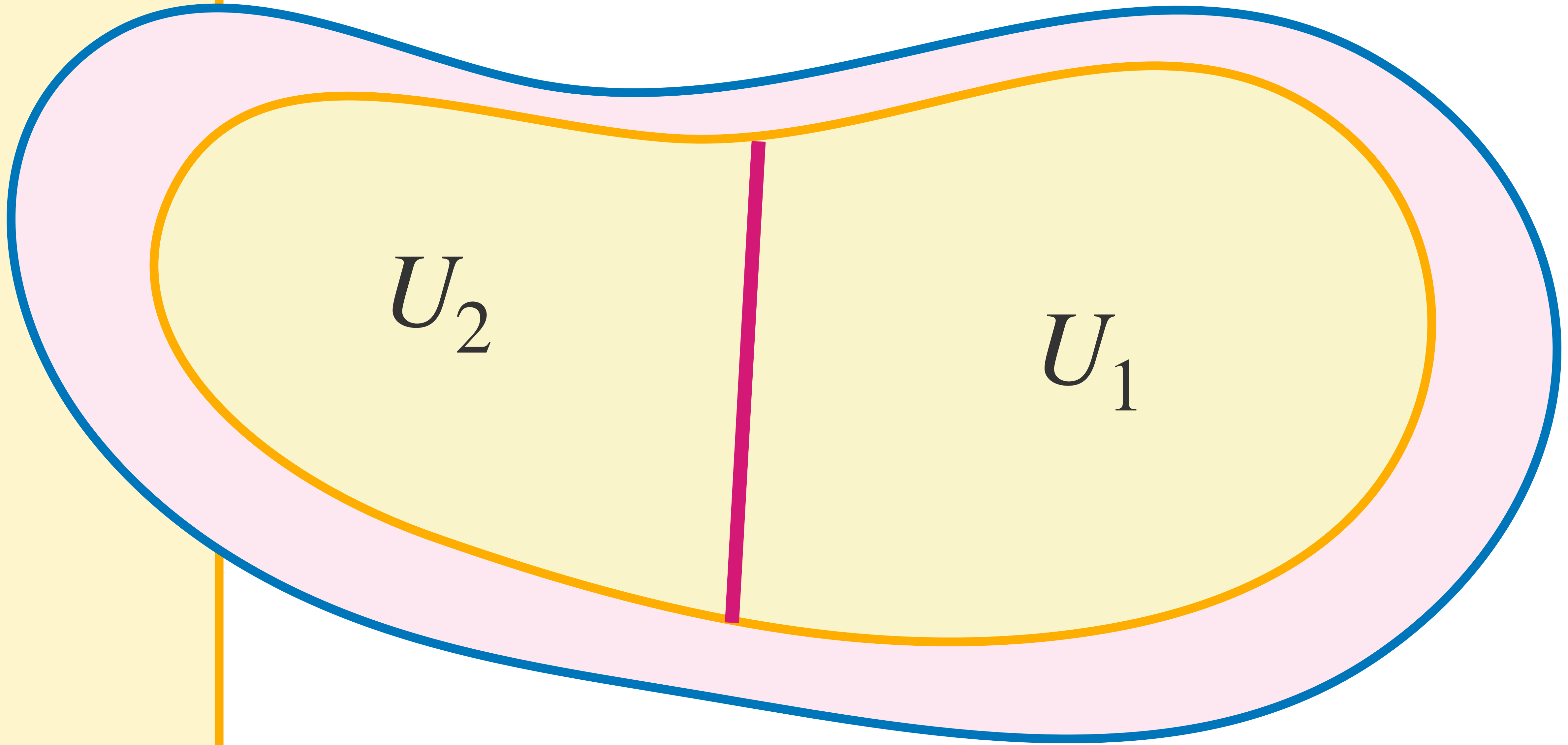
Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ — a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \sum_{e \in E[U, V \setminus U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1. Pick a balanced partition $U = U_1 \cup U_2$



$G = (V, E)$

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ — a falsified constraint

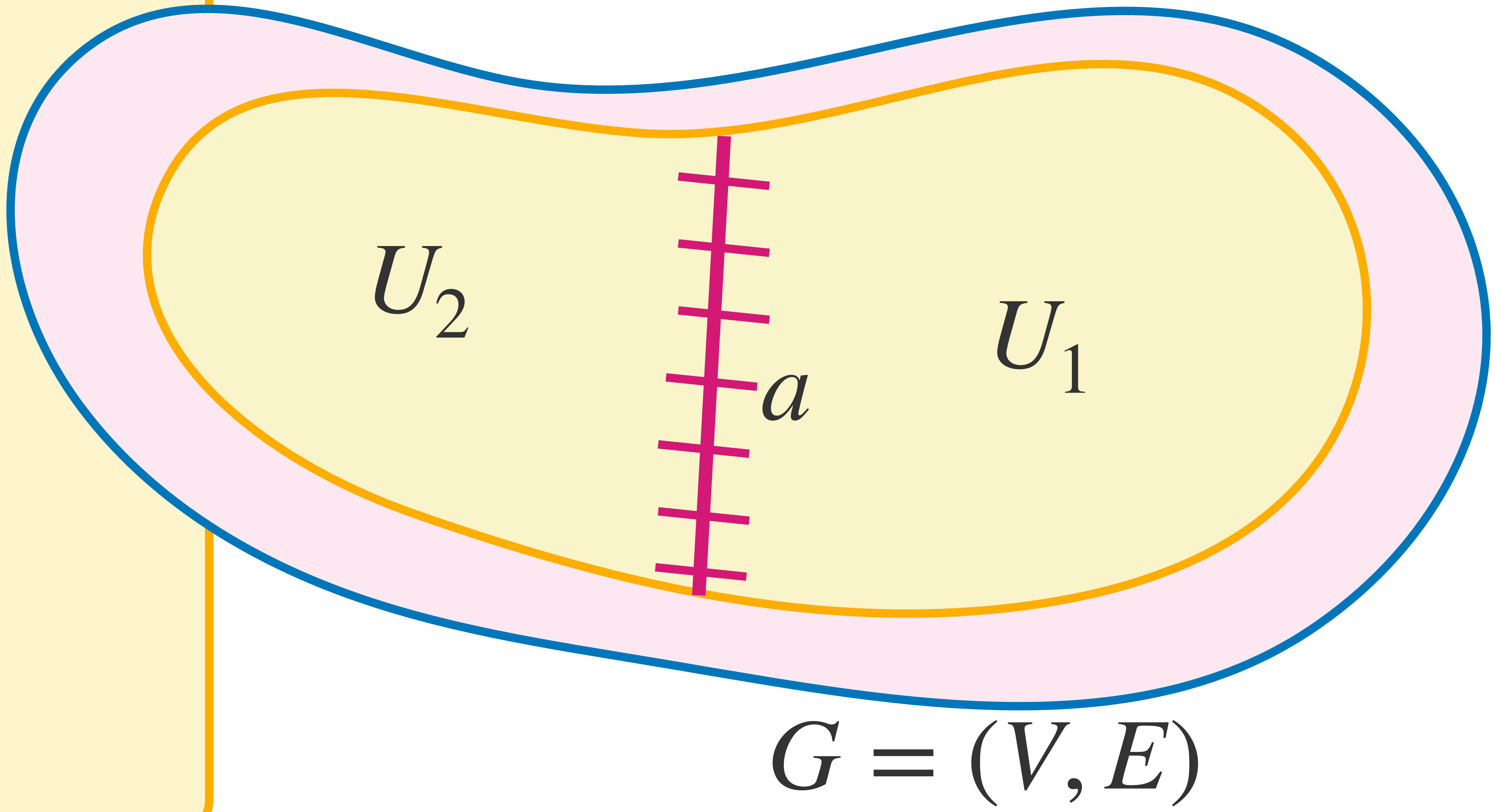
Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \backslash U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \backslash U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$a = \Sigma_{e \in [U_1, U_2]} y_e$



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ — a falsified constraint

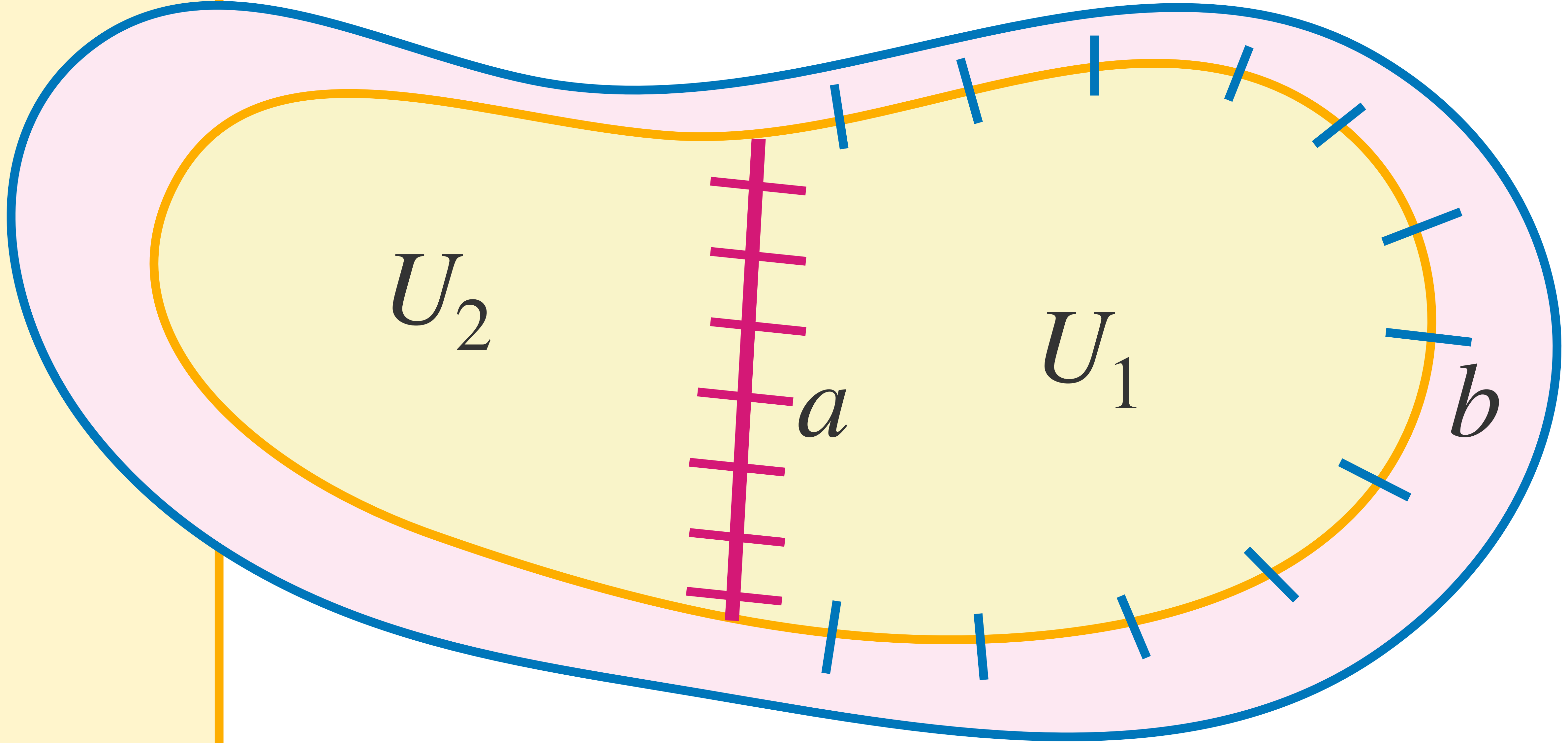
Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \setminus U]}\ y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$a = \Sigma_{e \in [U_1, U_2]} y_e$ $\qquad$ $b = \Sigma_{e \in [U_1, V \setminus U]} y_e$



$G = (V, E)$

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ — a falsified constraint

Algorithm proceeds in rounds:

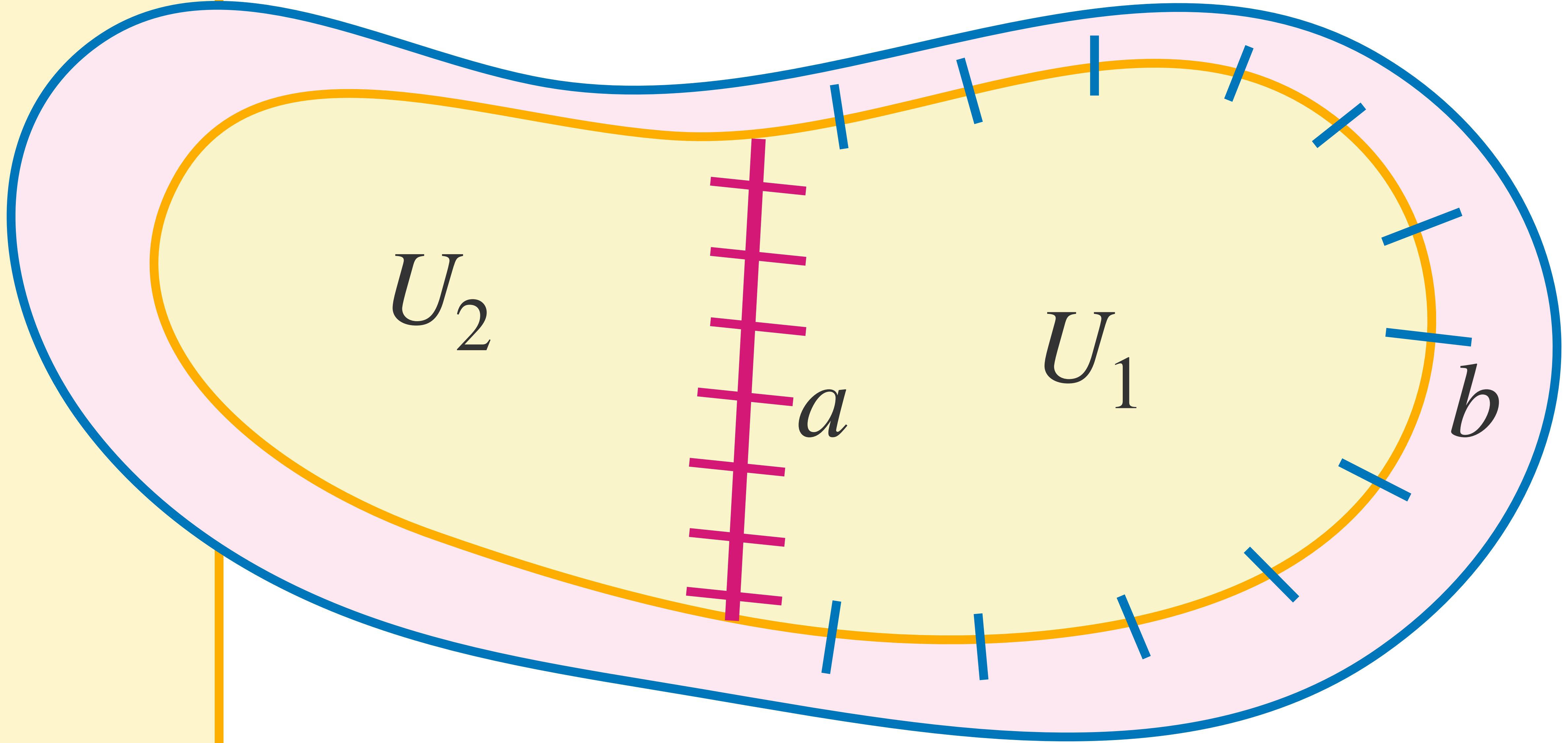
Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \setminus U]}\, y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$a = \Sigma_{e \in [U_1, U_2]} y_e$ $\qquad$ $b = \Sigma_{e \in [U_1, V \setminus U]} y_e$

$\kappa_{U_1} = a + b$



$G = (V, E)$

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

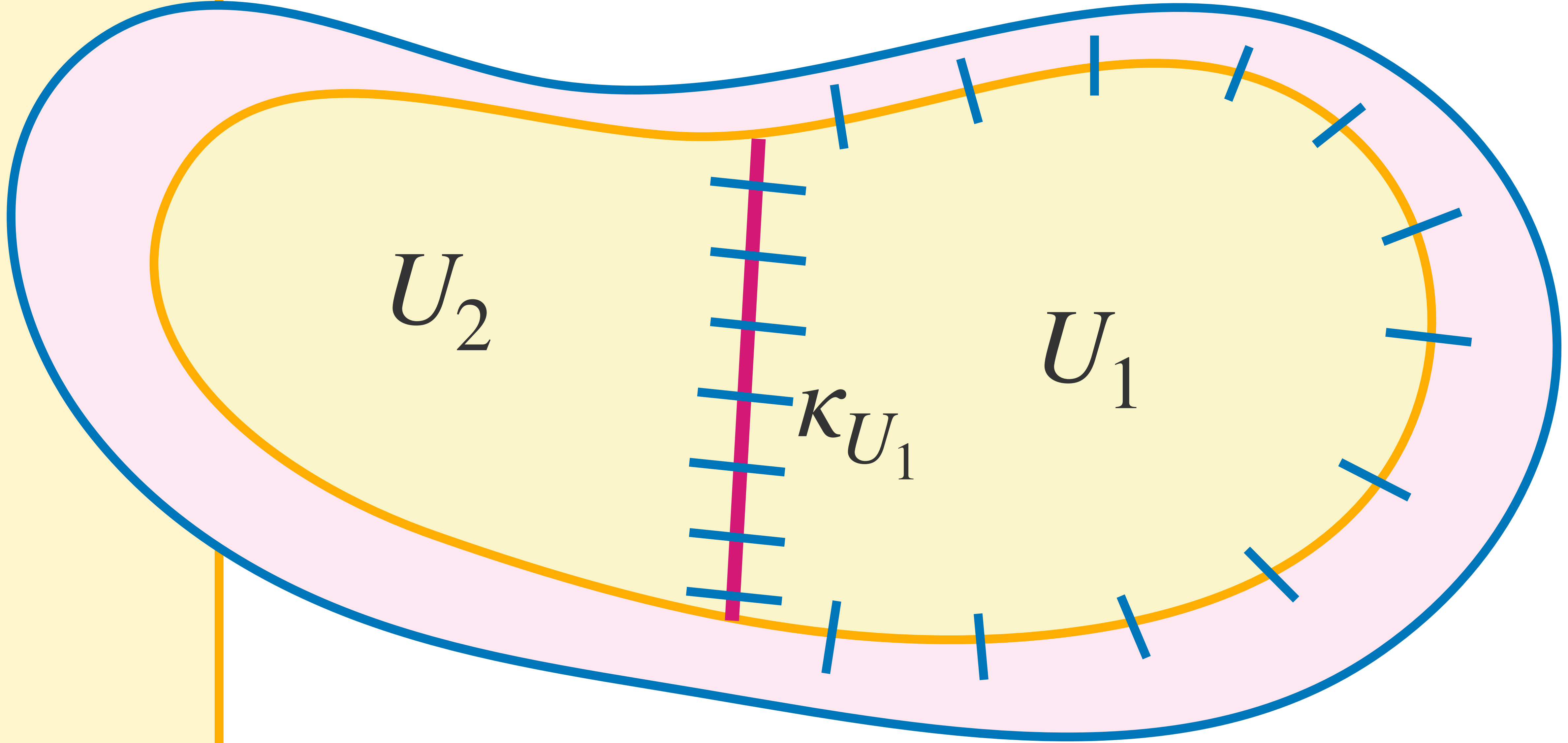
Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V\setminus U]}\, y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$a = \Sigma_{e \in [U_1, U_2]} y_e$ $\quad$ $b = \Sigma_{e \in [U_1, V\setminus U]} y_e$

$\kappa_{U_1} = a + b$



$G = (V, E)$

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of Tseitin$_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ — a falsified constraint

Algorithm proceeds in rounds:

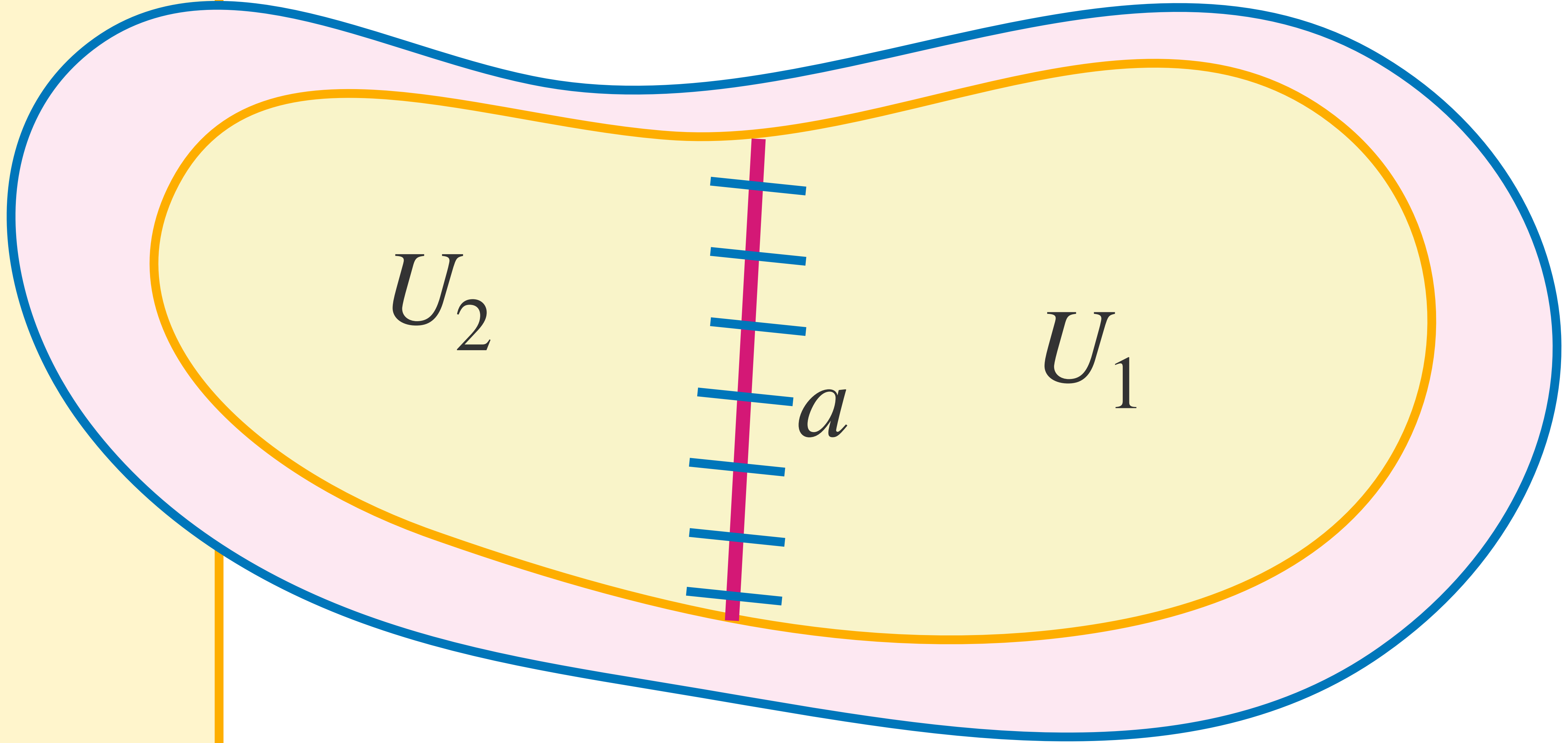
Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \setminus U]}\, y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$a = \Sigma_{e \in [U_1, U_2]} y_e$ $b = \Sigma_{e \in [U_1, V \setminus U]} y_e$

$\kappa_{U_1} = a + b$ and $\kappa_{U_2} = a + (\kappa_U - b)$



$G = (V, E)$

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

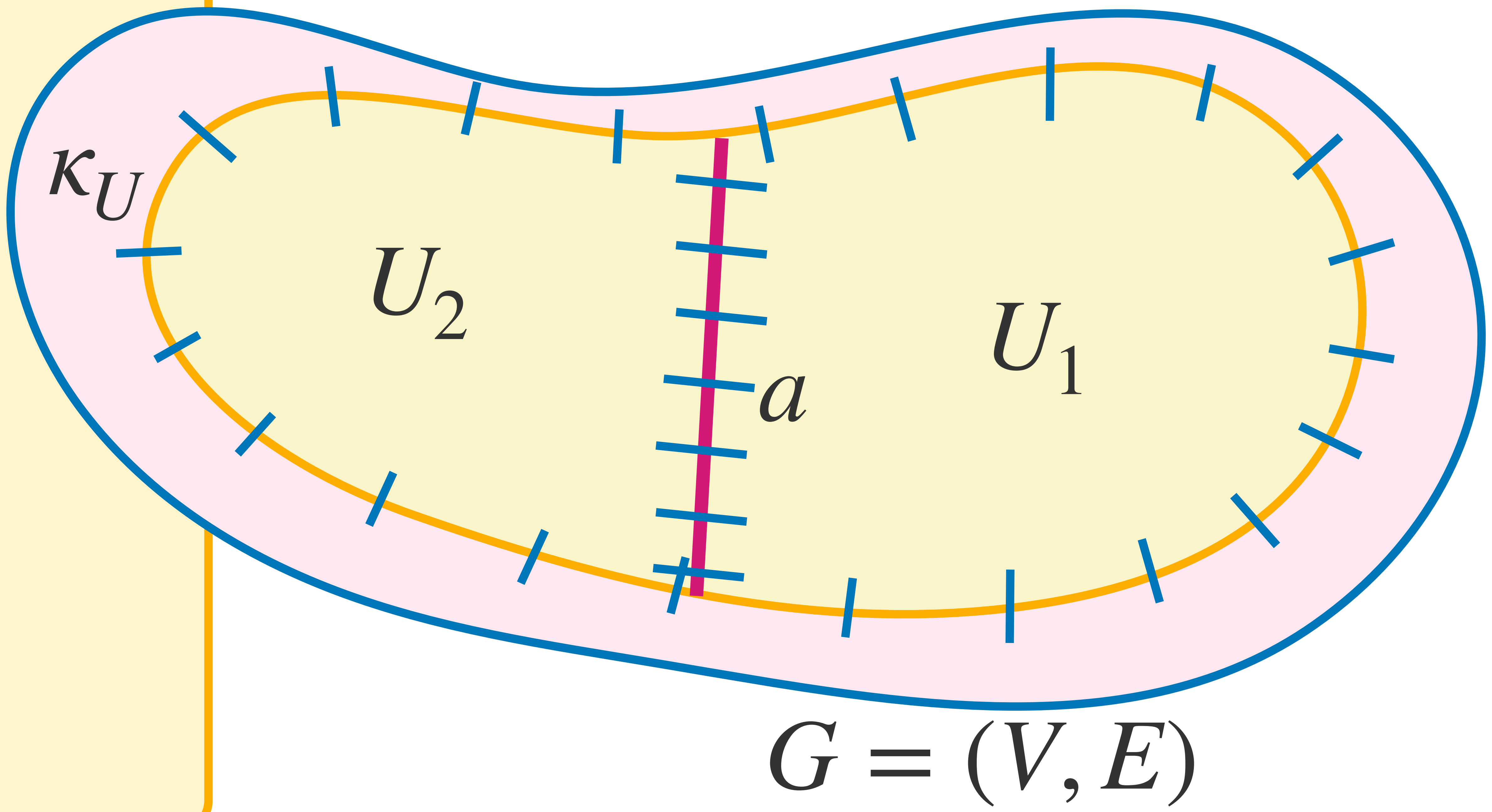
Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \setminus U]}\, y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$a = \Sigma_{e \in [U_1, U_2]} y_e$ $\quad$ $b = \Sigma_{e \in [U_1, V \setminus U]} y_e$

$\kappa_{U_1} = a + b$ and $\kappa_{U_2} = a + (\kappa_U - b)$



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

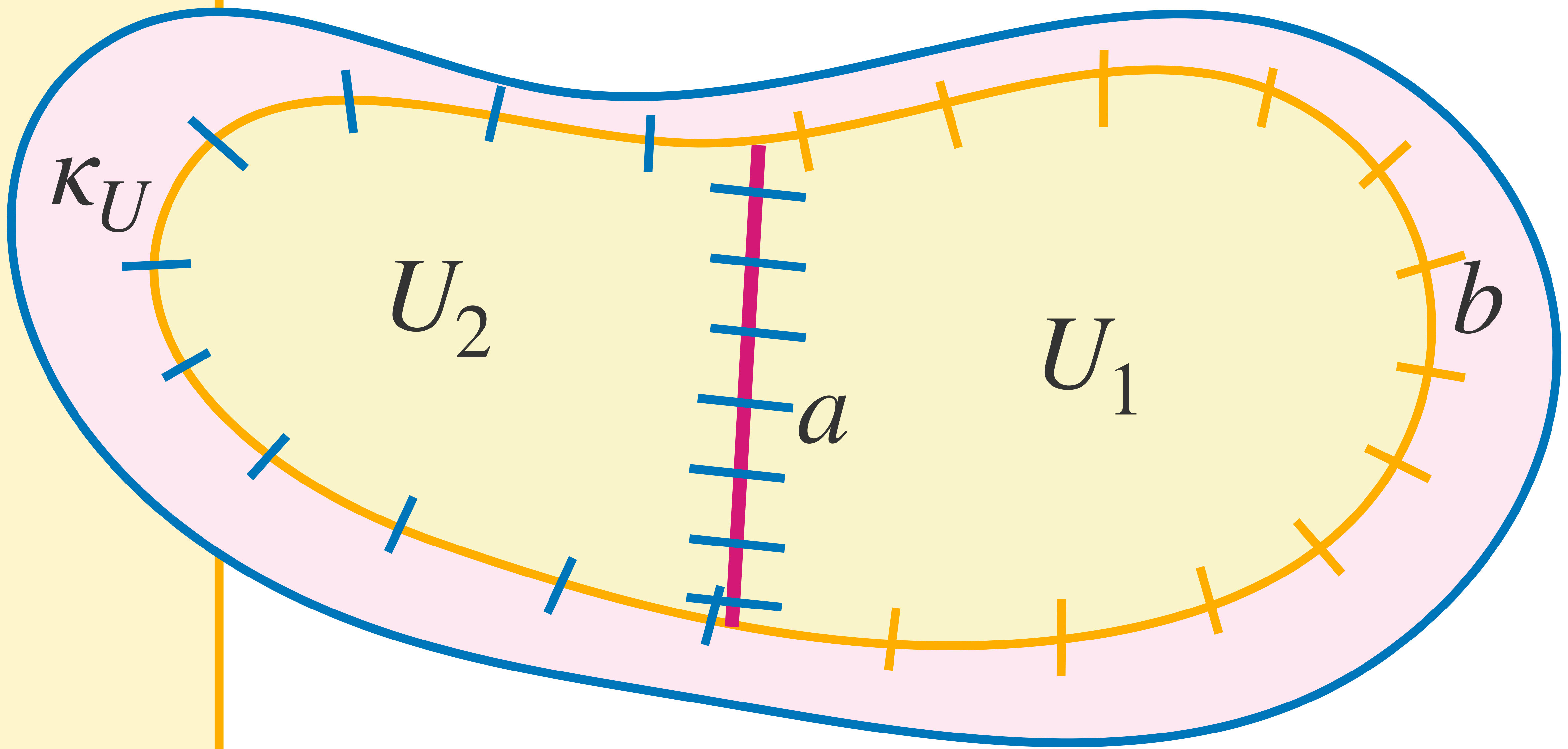
Algorithm proceeds in rounds: Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \setminus U]} \, y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$\boxed{a = \Sigma_{e \in [U_1, U_2]} y_e}$ $\boxed{b = \Sigma_{e \in [U_1, V \setminus U]} y_e}$

$\kappa_{U_1} = a + b$ and $\kappa_{U_2} = a + (\kappa_U - b)$



$G = (V, E)$

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

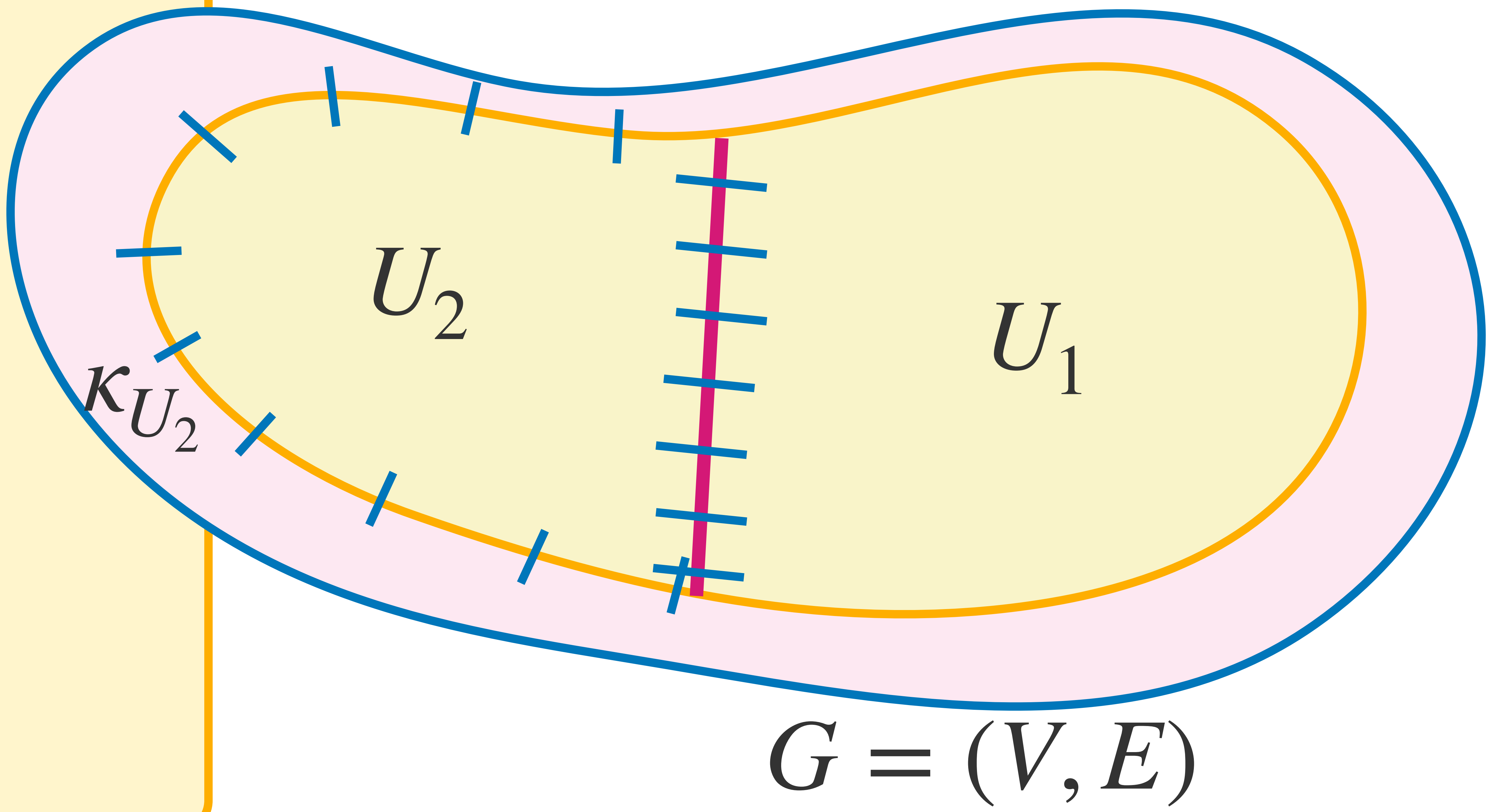
Edges with one endpoint in U one in $V \backslash U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \backslash U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$a = \Sigma_{e \in [U_1, U_2]} y_e$ $\qquad$ $b = \Sigma_{e \in [U_1, V \backslash U]} y_e$

$\kappa_{U_1} = a + b$ and $\kappa_{U_2} = a + (\kappa_U - b)$



Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

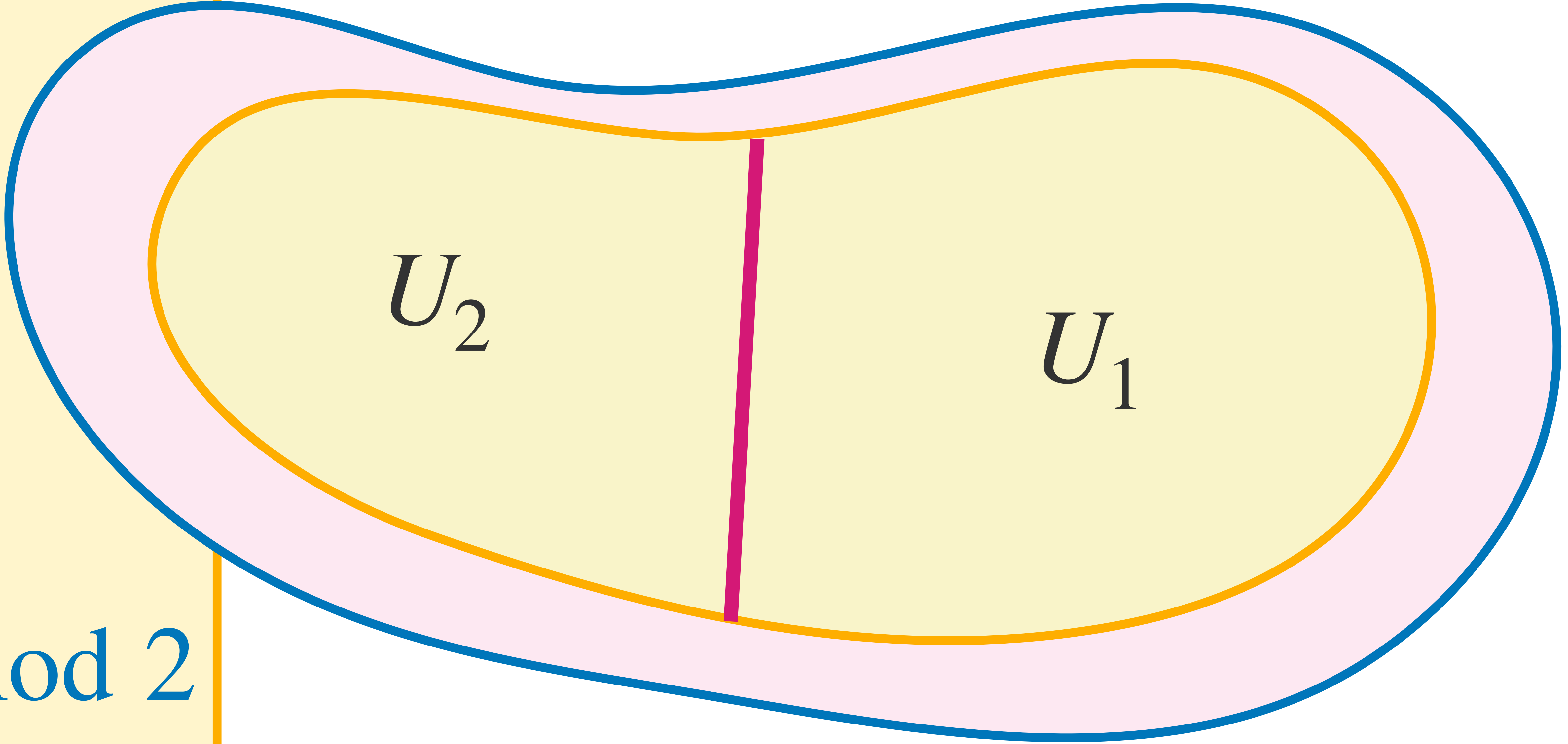
- Each round maintains: $U \subseteq V$ and $\kappa_U = \sum_{e \in E[U, V\setminus U]} y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

   $\boxed{a = \sum_{e \in [U_1, U_2]} y_e}$ $\boxed{b = \sum_{e \in [U_1, V\setminus U]} y_e}$

   $\kappa_{U_1} = a + b$ and $\kappa_{U_2} = a + (\kappa_U - b)$
3. Because $\sigma(U) \neq \kappa_U \bmod 2$, either

   $\sigma(U_1) \neq \kappa_{U_1} \bmod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \bmod 2$



$G = (V, E)$

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \ni v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \setminus U]}\, y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

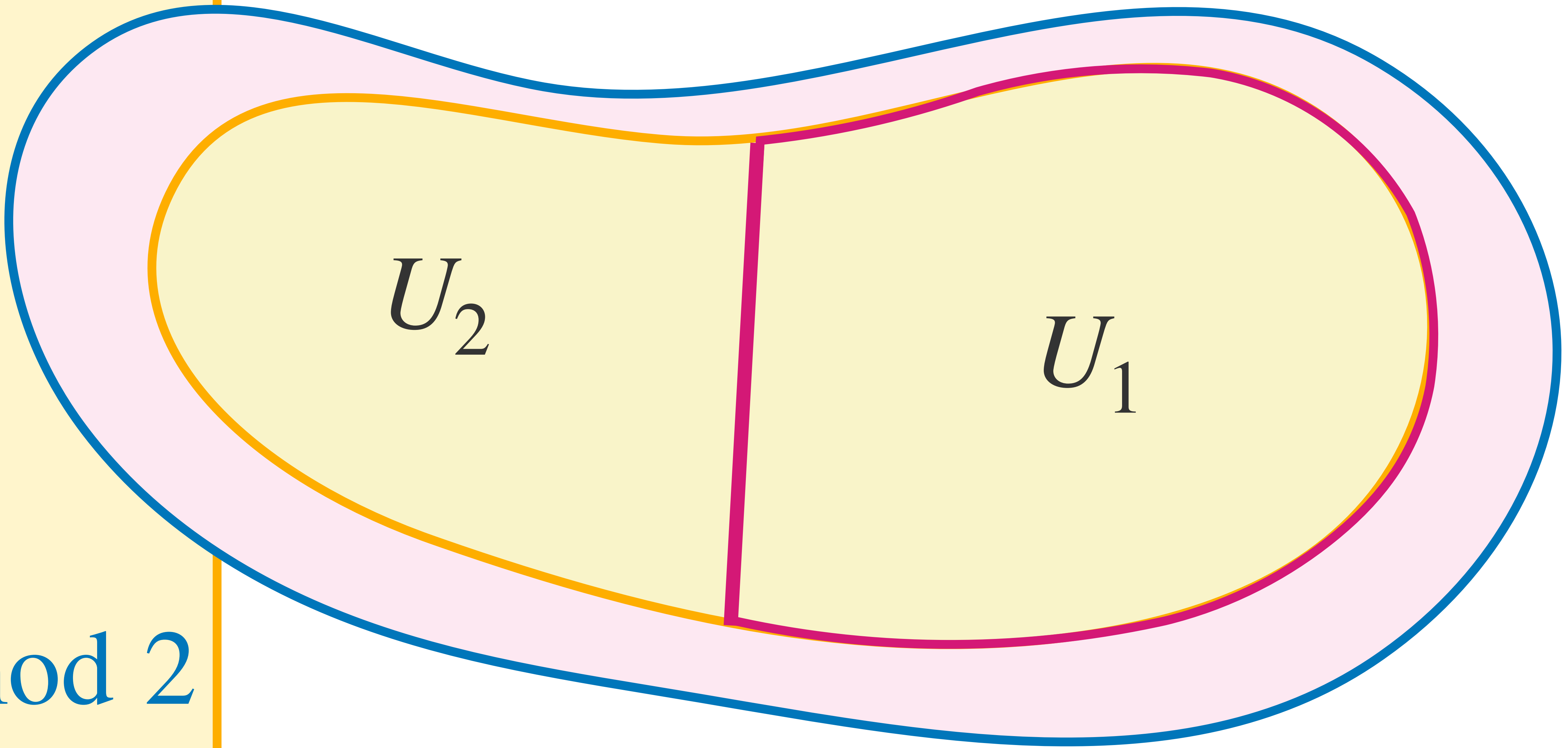
1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

   $a = \Sigma_{e \in [U_1, U_2]} y_e$ $\quad$ $b = \Sigma_{e \in [U_1, V \setminus U]} y_e$

   $\kappa_{U_1} = a + b$ and $\kappa_{U_2} = a + (\kappa_U - b)$
3. Because $\sigma(U) \neq \kappa_U \bmod 2$, either

   $\sigma(U_1) \neq \kappa_{U_1} \bmod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \bmod 2$

   $\rightarrow$ recurse on that one.



$G = (V, E)$

Algorithm for Finding Falsified Clause

Given: $y \in \{0,1\}^n$ to the variables of $\text{Tseitin}_{G,\sigma}$

Goal: find $v \in V$ such that $\bigoplus_{e \in v} y_e \neq \sigma(v)$ – a falsified constraint

Algorithm proceeds in rounds:

Edges with one endpoint in U one in $V \setminus U$

- Each round maintains: $U \subseteq V$ and $\kappa_U = \Sigma_{e \in E[U, V \setminus U]}\, y_e$ s.t. $\sigma(U) \neq \kappa_U \bmod 2$

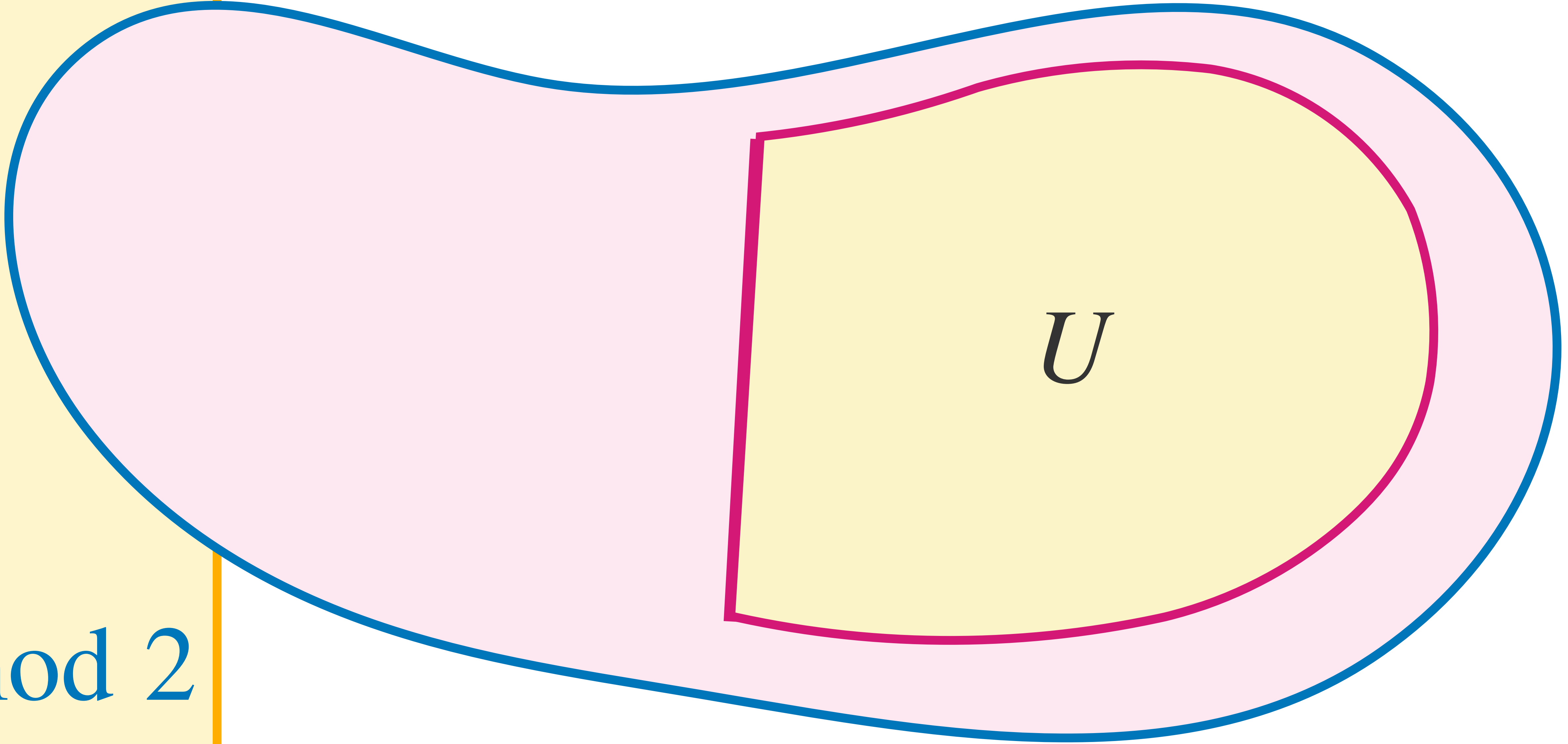
1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

   $\boxed{a = \Sigma_{e \in [U_1, U_2]} y_e}$ $\boxed{b = \Sigma_{e \in [U_1, V \setminus U]} y_e}$

   $\kappa_{U_1} = a + b$ and $\kappa_{U_2} = a + (\kappa_U - b)$
3. Because $\sigma(U) \neq \kappa_U \bmod 2$, either

   $\sigma(U_1) \neq \kappa_{U_1} \bmod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \bmod 2$

   $\rightarrow$ recurse on that one.



$G = (V, E)$

Implementation in Stabbing Planes

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$\boxed{a = \Sigma_{e \in [U_1, U_2]} y_e}$ $\boxed{b = \Sigma_{e \in [U_1, V \setminus U]} y_e}$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \mod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \mod 2$; recurse on that one.

Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$\boxed{a = \Sigma_{e \in [U_1, U_2]} y_e}$ $\boxed{b = \Sigma_{e \in [U_1, V \setminus U]} y_e}$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \mod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \mod 2$; recurse on that one.

Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

→ Observe that the possible values of a and b are in $\{0,\ldots,|E|\}$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$\boxed{a = \Sigma_{e\in[U_1,U_2]} y_e}$ $\boxed{b = \Sigma_{e\in[U_1,V\setminus U]} y_e}$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \mod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \mod 2$; recurse on that one.

Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

→ Observe that the possible values of a and b are in $\{0,\ldots,|E|\}$

→ We can determine the value of a and b in $\log|E|$ depth SP trees

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$$\boxed{a = \Sigma_{e\in[U_1,U_2]} y_e} \qquad \boxed{b = \Sigma_{e\in[U_1,V\setminus U]} y_e}$$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \bmod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \bmod 2$; recurse on that one.

Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

→ Observe that the possible values of a and b are in $\{0,\ldots,|E|\}$

→ We can determine the value of a and b in $\log|E|$ depth SP trees

$a \leq |E|/2 - 1$ $\quad$ $a \geq |E|/2$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$\boxed{a = \Sigma_{e\in[U_1,U_2]} y_e}$ $\quad$ $\boxed{b = \Sigma_{e\in[U_1,V\setminus U]} y_e}$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \mod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \mod 2$; recurse on that one.

Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

→ Observe that the possible values of a and b are in $\{0,\ldots,|E|\}$

→ We can determine the value of a and b in $\log|E|$ depth SP trees

$a \leq |E|/2 - 1$ $\quad$ $a \geq |E|/2$

$a \leq 3|E|/4 - 1$ $\quad$ $a \geq 3|E|/4$

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

   $a = \Sigma_{e\in[U_1,U_2]} y_e$ $\qquad$ $b = \Sigma_{e\in[U_1,V\setminus U]} y_e$

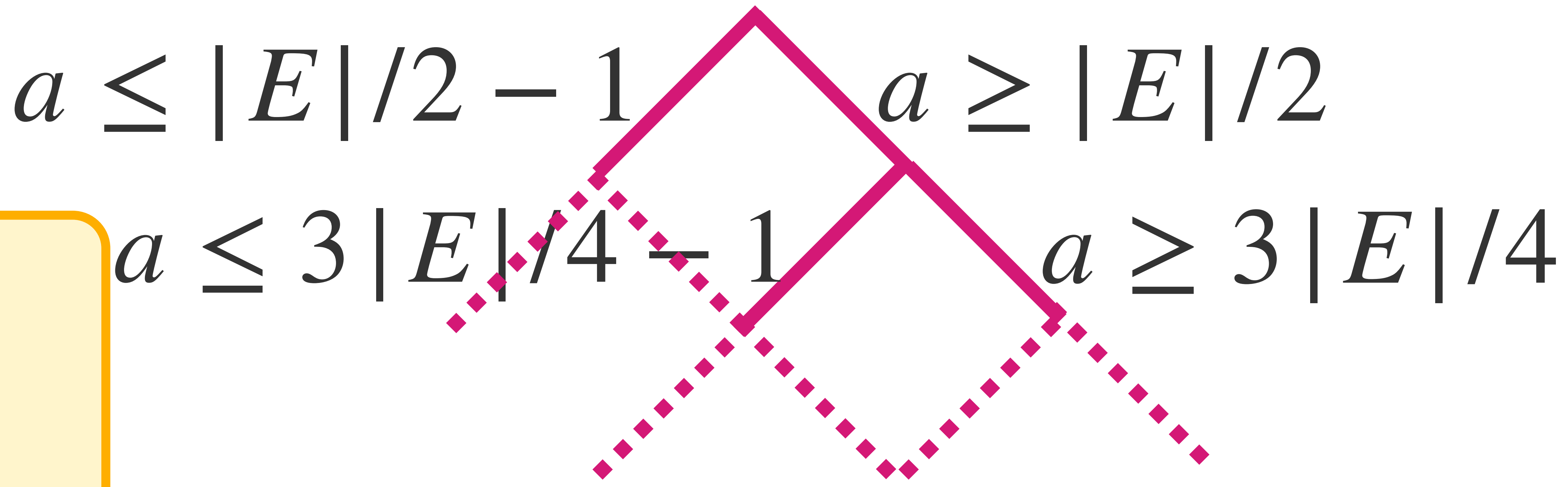
3. Either $\sigma(U_1) \neq \kappa_{U_1} \mod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \mod 2$; recurse on that one.

Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

→ Observe that the possible values of a and b are in $\{0,\ldots,|E|\}$

→ We can determine the value of a and b in $\log|E|$ depth SP trees



1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

   $\boxed{a = \Sigma_{e \in [U_1, U_2]} y_e}$ $\boxed{b = \Sigma_{e \in [U_1, V \setminus U]} y_e}$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \bmod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \bmod 2$; recurse on that one.

Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

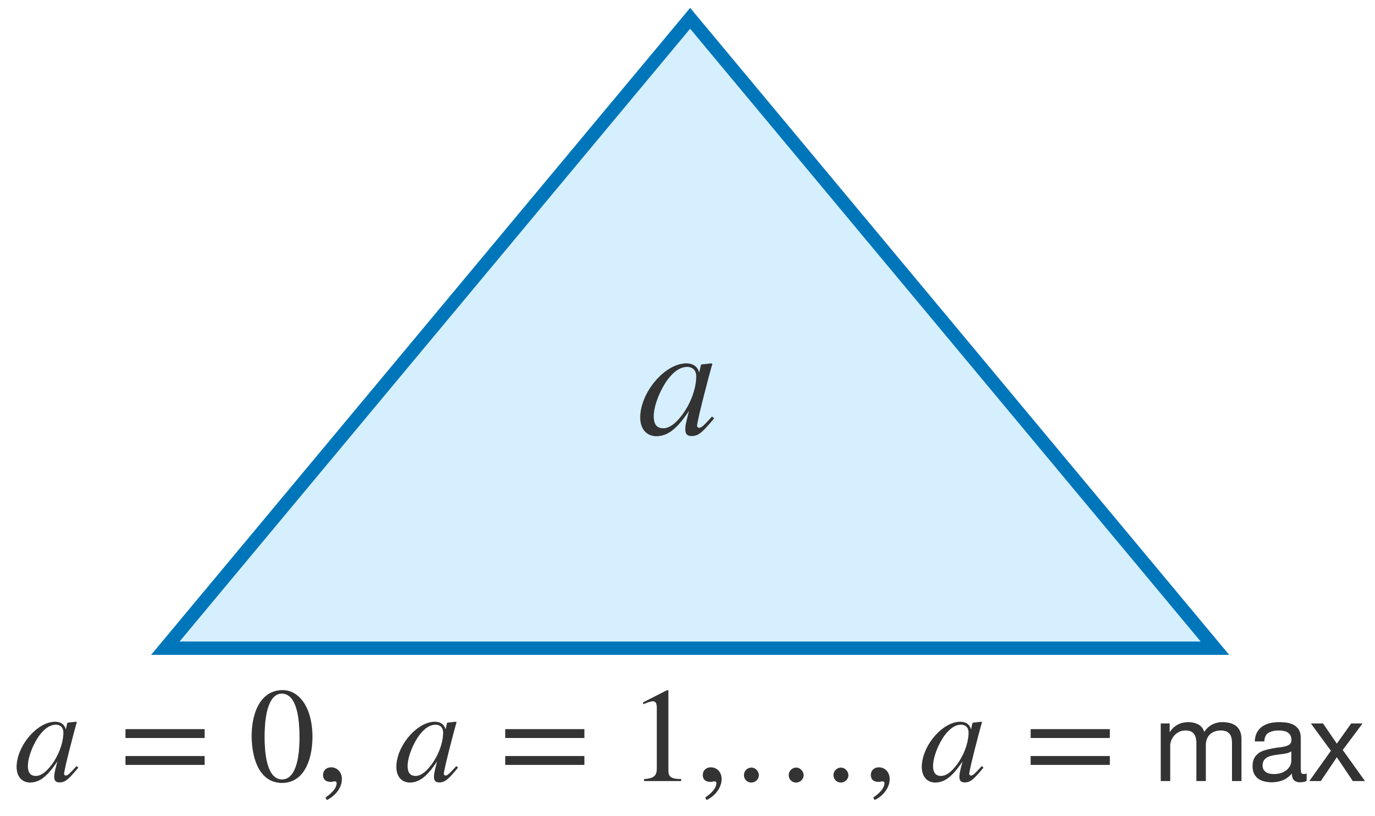
→ Observe that the possible values of a and b are in $\{0,\ldots,|E|\}$

→ We can determine the value of a and b in $\log|E|$ depth trees

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$$\boxed{a = \Sigma_{e\in[U_1,U_2]} y_e} \qquad \boxed{b = \Sigma_{e\in[U_1,V\setminus U]} y_e}$$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \bmod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \bmod 2$; recurse on that one.



Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

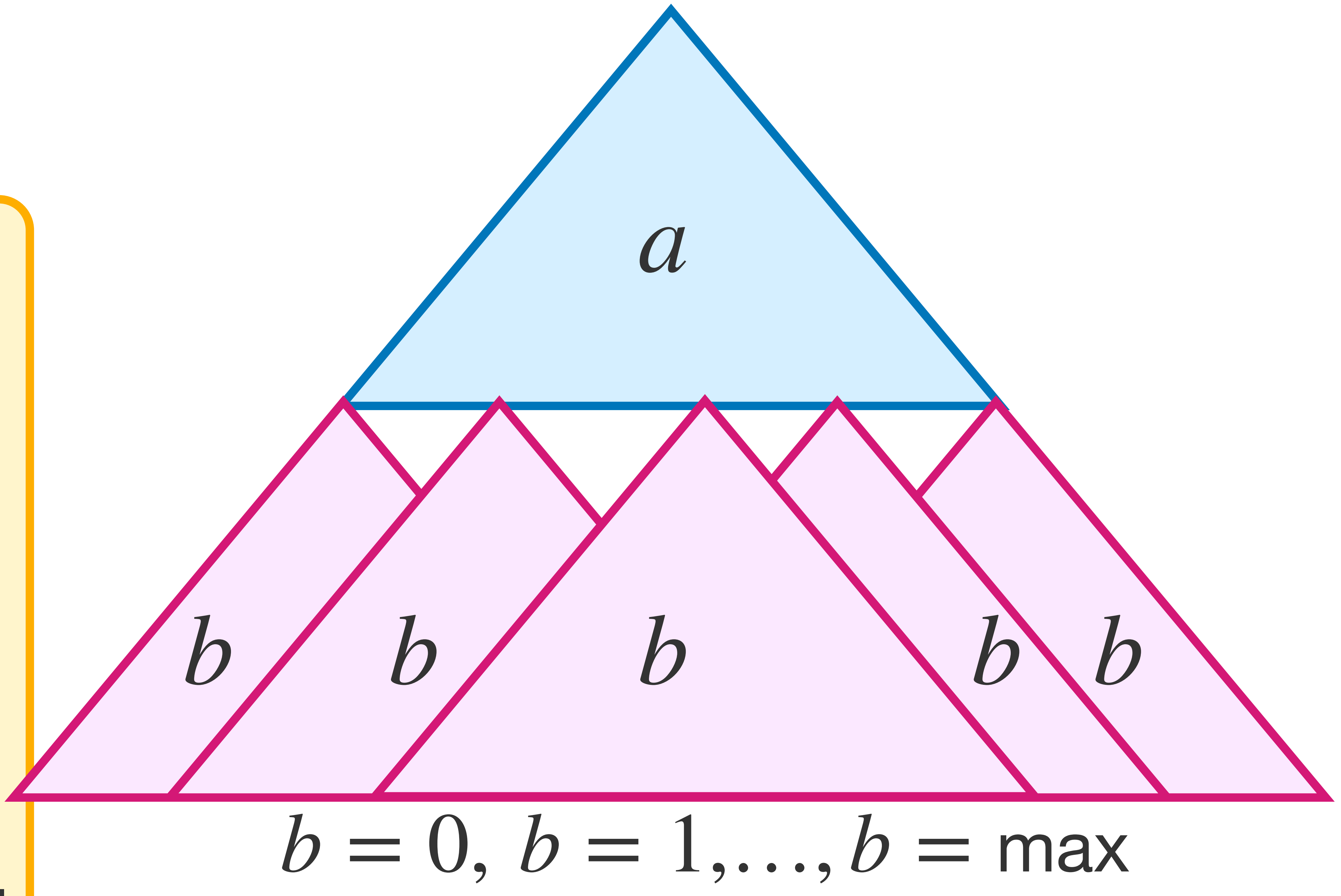
→ Observe that the possible values of a and b are in $\{0,\ldots,|E|\}$

→ We can determine the value of a and b in $\log|E|$ depth trees

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

   $\boxed{a = \Sigma_{e\in[U_1,U_2]} y_e}$ $\boxed{b = \Sigma_{e\in[U_1,V\setminus U]} y_e}$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \bmod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \bmod 2$; recurse on that one.



Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

→ Observe that the possible values of a and b are in $\{0, \ldots, |E|\}$

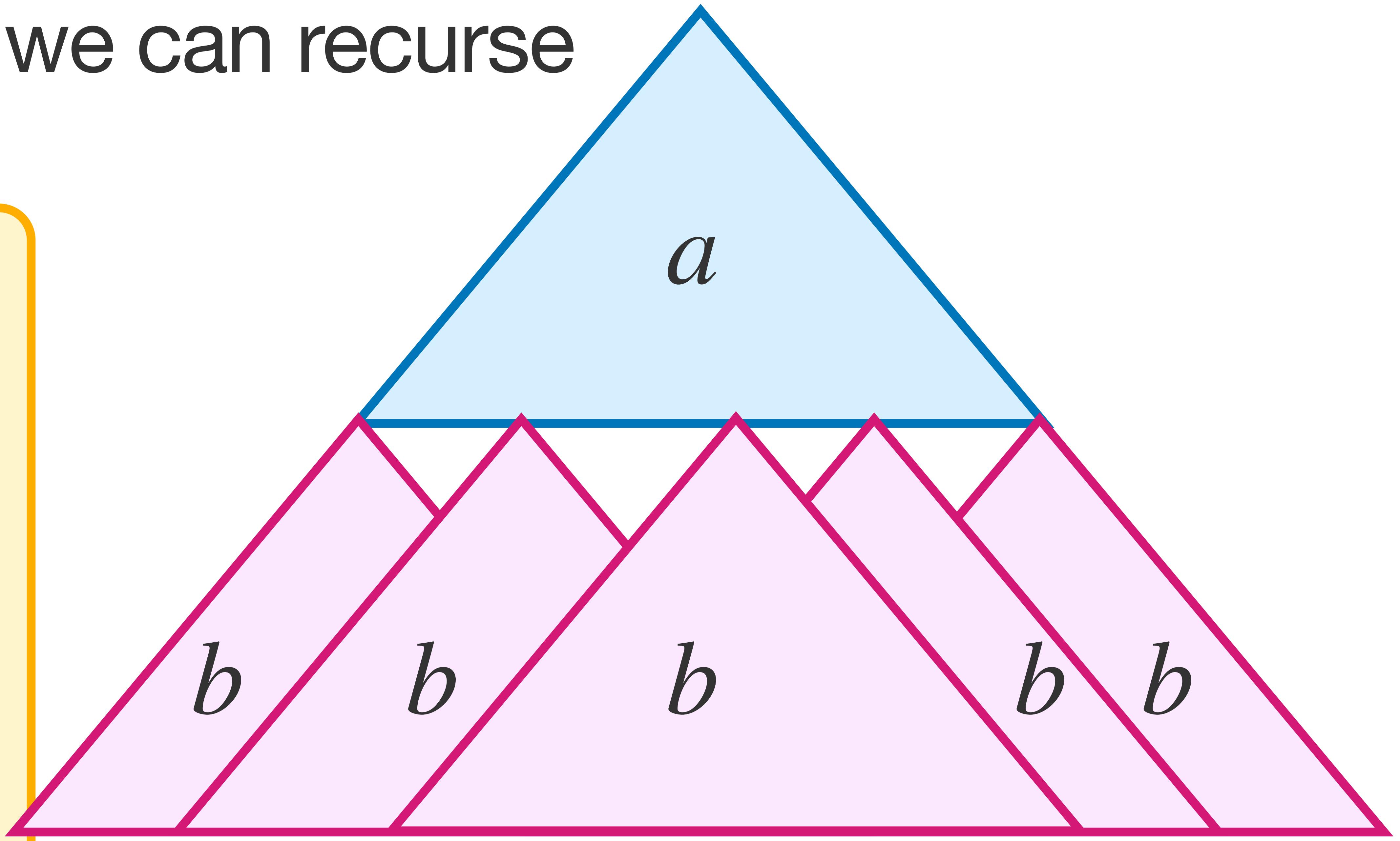
→ We can determine the value of a and b in $\log|E|$ depth trees

→ At each leaf we know value of a and b, so we can recurse

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

   $a = \Sigma_{e \in [U_1, U_2]} y_e$ $\qquad$ $b = \Sigma_{e \in [U_1, V \setminus U]} y_e$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \mod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \mod 2$; recurse on that one.



$b = 0,\ b = 1, \ldots, b =$ max

Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

→ Observe that the possible values of a and b are in $\{0,\ldots,|E|\}$

→ We can determine the value of a and b in $\log|E|$ depth trees

→ At each leaf we know value of a and b, so we can recurse

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

$$\boxed{a = \Sigma_{e\in[U_1,U_2]} y_e} \qquad \boxed{b = \Sigma_{e\in[U_1,V\setminus U]} y_e}$$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \bmod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \bmod 2$; recurse on that one.

Complexity:

- $\log|V|$ rounds

Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

→ Observe that the possible values of a and b are in $\{0,\ldots,|E|\}$

→ We can determine the value of a and b in $\log|E|$ depth trees

→ At each leaf we know value of a and b, so we can recurse

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

   $a = \Sigma_{e\in[U_1,U_2]} y_e$ $\quad$ $b = \Sigma_{e\in[U_1,V\setminus U]} y_e$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \mod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \mod 2$; recurse on that one.

Complexity:

- $\log|V|$ rounds
- Each round takes two depth $\leq \log|E|$ trees

Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

→ Observe that the possible values of a and b are in $\{0,\ldots,|E|\}$

→ We can determine the value of a and b in $\log|E|$ depth trees

→ At each leaf we know value of a and b, so we can recurse

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

   $\boxed{a = \Sigma_{e\in[U_1,U_2]} y_e}$ $\boxed{b = \Sigma_{e\in[U_1,V\setminus U]} y_e}$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \mod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \mod 2$; recurse on that one.

Complexity:

- $\log|V|$ rounds
- Each round takes two depth $\leq \log|E|$ trees

→ total **depth:** $2\log|E|\log|V| = O(\log^2 n)$

Implementation in Stabbing Planes

To implement in SP we need to perform the queries a and b

→ Observe that the possible values of a and b are in $\{0,\ldots,|E|\}$

→ We can determine the value of a and b in $\log|E|$ depth trees

→ At each leaf we know value of a and b, so we can recurse

1. Pick a balanced partition $U = U_1 \cup U_2$
2. Query:

   $\boxed{a = \Sigma_{e\in[U_1,U_2]} y_e}$ $\boxed{b = \Sigma_{e\in[U_1,V\setminus U]} y_e}$

3. Either $\sigma(U_1) \neq \kappa_{U_1} \mod 2$ or $\sigma(U_2) \neq \kappa_{U_2} \mod 2$; recurse on that one.

Complexity:

SP proofs are binary trees

depth $O(\log^2 n) \rightarrow$ size $2^{O(\log^2 n)}$

Cutting Planes Proves Tseitin!

Thm: There are quasipolynomial size Cutting Planes proofs of Tseitin

High Level:

1. Exhibit a quasipolynomial size Stabbing Planes proof of Tseitin
2. Translate that proof into Cutting Planes

In fact, almost every SP proof can be translated into CP!

Cutting Planes Proves Tseitin!

Thm: There are quasipolynomial size Cutting Planes proofs of Tseitin

High Level:

1. Exhibit a quasipolynomial size Stabbing Planes proof of Tseitin
2. Translate that proof into Cutting Planes

Thm [FGI+21]

Any Stabbing Planes proof with coefficients at most $2^{\mathrm{polylog} n}$ (SP*) can be translated into Cutting Planes with a quasi-polynomial blow-up in the size.

Cutting Planes Proves Tseitin!

Thm [FGI+21]

Any Stabbing Planes proof with coefficients at most $2^{\text{polylog } n}$ (SP*) can be translated into Cutting Planes with a quasi-polynomial blow-up in the size.

Idea:

1. Turn the proof SP* into a facelike SP proof — one that branches on the faces of the current polytope

Cutting Planes Proves Tseitin!

Thm [FGI+21]

Any Stabbing Planes proof with coefficients at most $2^{\text{polylog } n}$ (SP*) can be translated into Cutting Planes with a quasi-polynomial blow-up in the size.

Idea:

1. Turn the proof SP* into a facelike SP proof — one that branches on the faces of the current polytope (causes a quasipolynomial blow-up)

Cutting Planes Proves Tseitin!

Thm [FGI+21]

Any Stabbing Planes proof with coefficients at most $2^{\text{polylog } n}$ (SP*) can be translated into Cutting Planes with a quasi-polynomial blow-up in the size.

Idea:

1. Turn the proof SP* into a facelike SP proof — one that branches on the faces of the current polytope (causes a quasipolynomial blow-up)
2. Show that facelike SP proofs are equivalent to Cutting Planes proofs