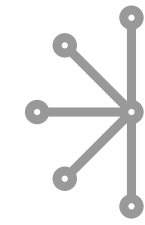




Technische
Universität
Braunschweig



Institut für Betriebssysteme
und Rechnerverbund
Algorithmik

Algorithm Engineering

Lecture 7: LP & MIP

Linear Programming

Reminder:

Reminder:

$$\max \sum_i c_i x_i \text{ s.t.}$$

Reminder:

$$\begin{aligned} &\max \sum_i c_i x_i \text{ s.t.} \\ &1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \end{aligned}$$

Reminder:

$$\begin{aligned} \max \quad & \sum_i c_i x_i \text{ s.t.} \\ & 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ & \ell_i \leq x_i \leq u_i \end{aligned}$$

Reminder:

$$\begin{aligned} \max \quad & \sum_i c_i x_i \text{ s.t.} \\ & 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ & \ell_i \leq x_i \leq u_i \end{aligned}$$

The linear programming problem:

Reminder:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \end{aligned}$$

The linear programming problem:

Given $c_i, b_i, a_{ji}, \ell_i, u_i$, find the optimal $x^* = \{x_1^*, \dots, x_n^*\}$ if it exists.

Reminder:

$$\begin{aligned} \max \quad & \sum_i c_i x_i \text{ s.t.} \\ & 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ & \ell_i \leq x_i \leq u_i \end{aligned}$$

The linear programming problem:

Given $c_i, b_i, a_{ji}, \ell_i, u_i$, find the optimal $x^* = \{x_1^*, \dots, x_n^*\}$ if it exists.

LP is in P (both in theory and practice); several practical algorithms exist.

Reminder:

$$\begin{aligned} \max \quad & \sum_i c_i x_i \text{ s.t.} \\ & 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ & \ell_i \leq x_i \leq u_i \end{aligned}$$

The linear programming problem:

Given $c_i, b_i, a_{ji}, \ell_i, u_i$, find the optimal $x^* = \{x_1^*, \dots, x_n^*\}$ if it exists.

LP is in P (both in theory and practice); several practical algorithms exist.

Runtime depends mainly on n, m and the number of non-zero a_{ji} (sparseness).

Reminder:

$$\begin{aligned} \max \quad & \sum_i c_i x_i \text{ s.t.} \\ & 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ & \ell_i \leq x_i \leq u_i \end{aligned}$$

Mathematische Methoden der Algorithmik

The linear programming problem:

Given $c_i, b_i, a_{ji}, \ell_i, u_i$, find the optimal $x^* = \{x_1^*, \dots, x_n^*\}$ if it exists.

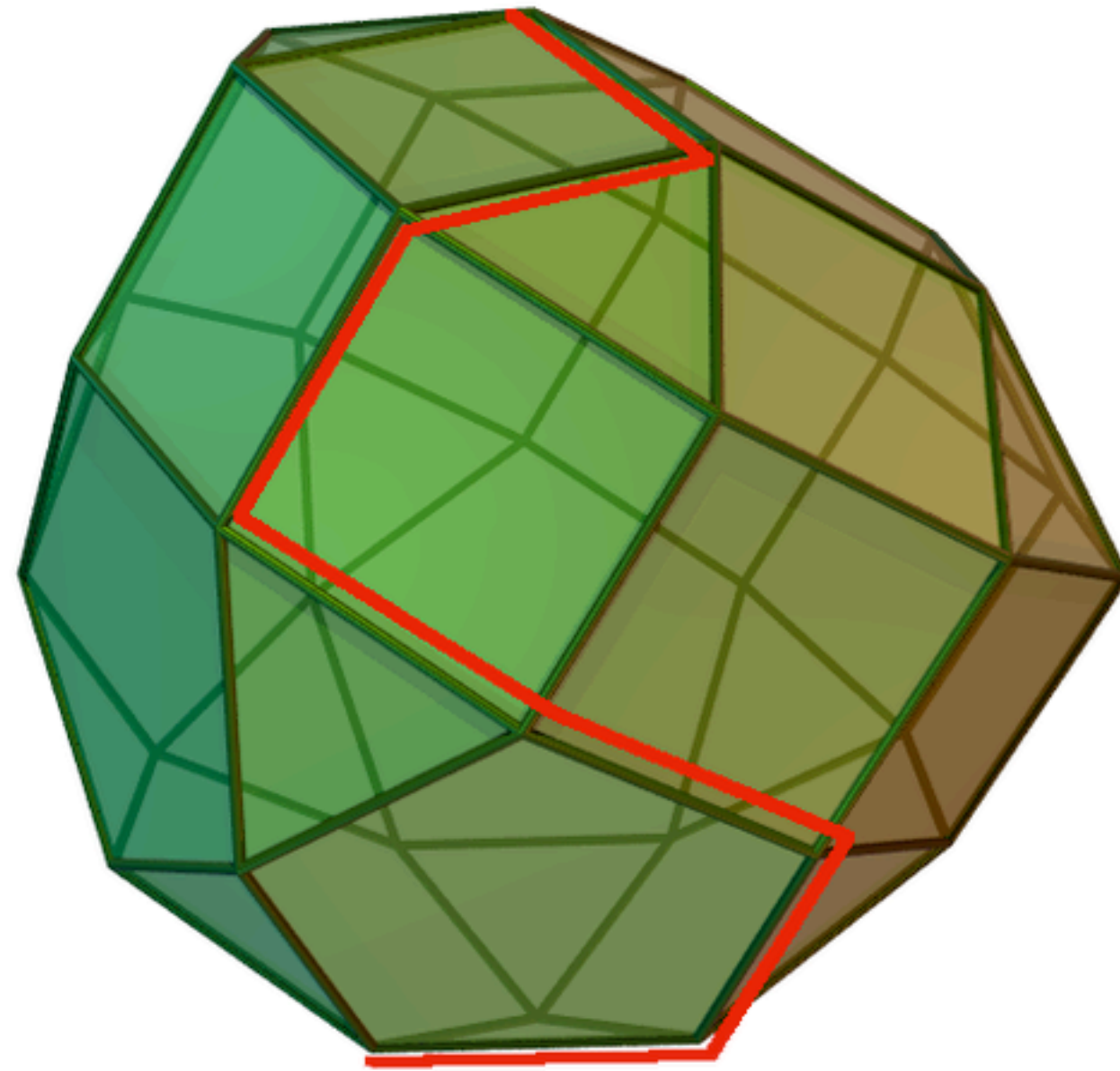
LP is in P (both in theory and practice); several practical algorithms exist.

Runtime depends mainly on n, m and the number of non-zero a_{ji} (sparseness).

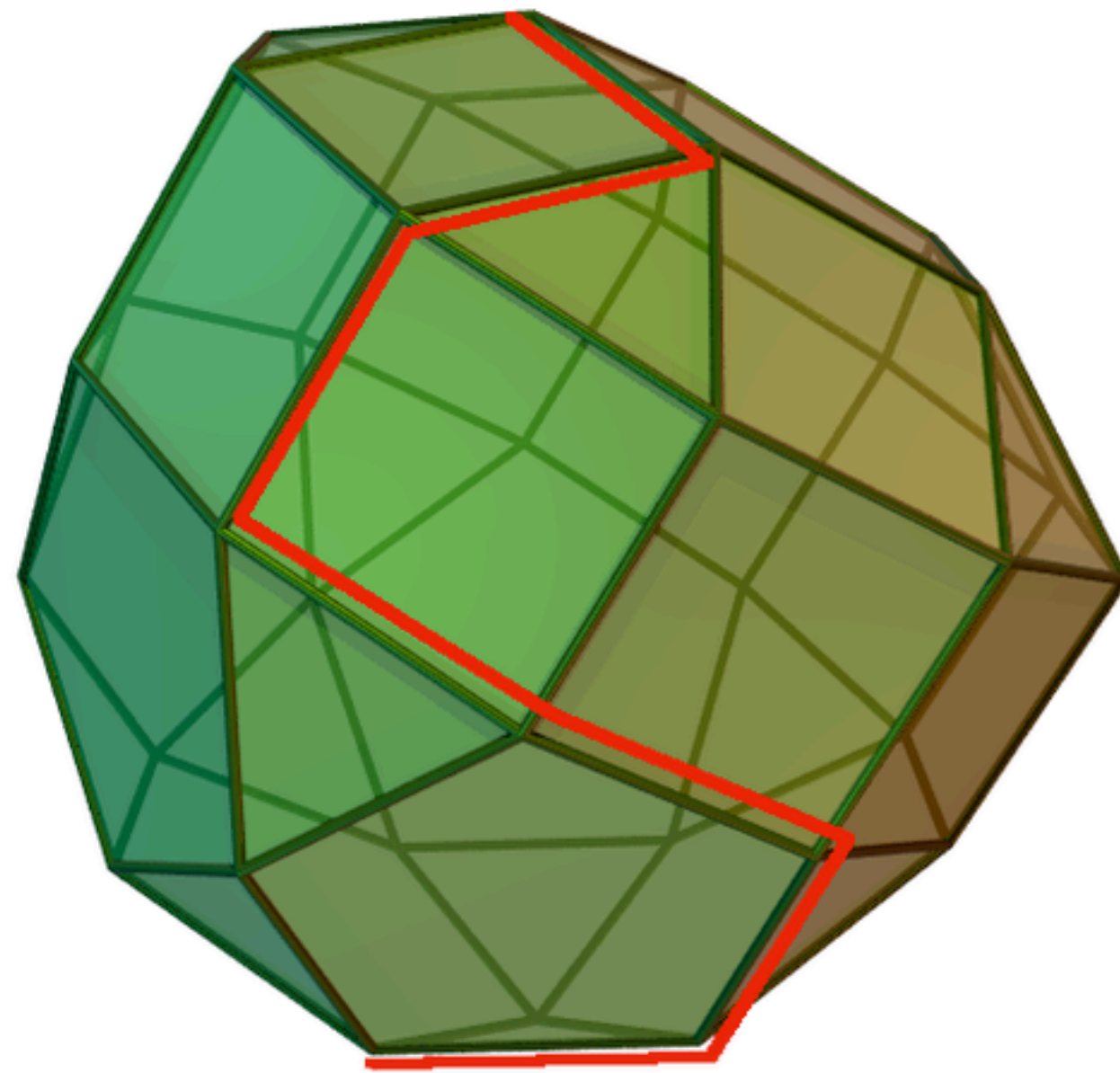
Linear Programming: Practical Algorithms



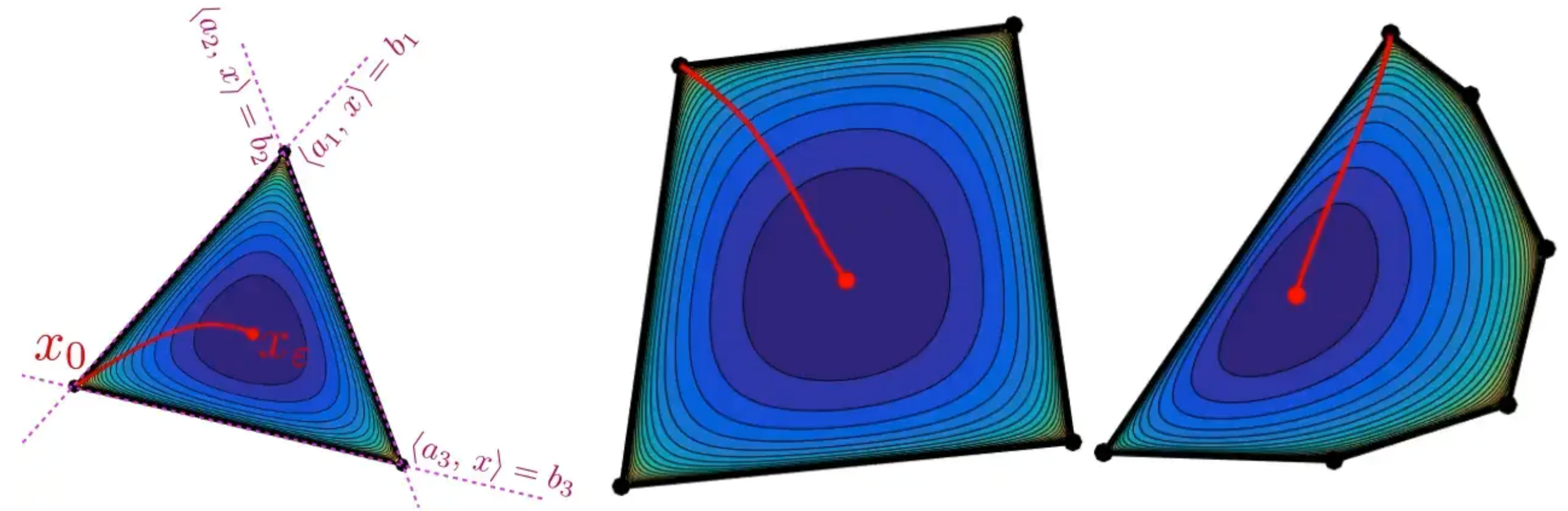
Simplex



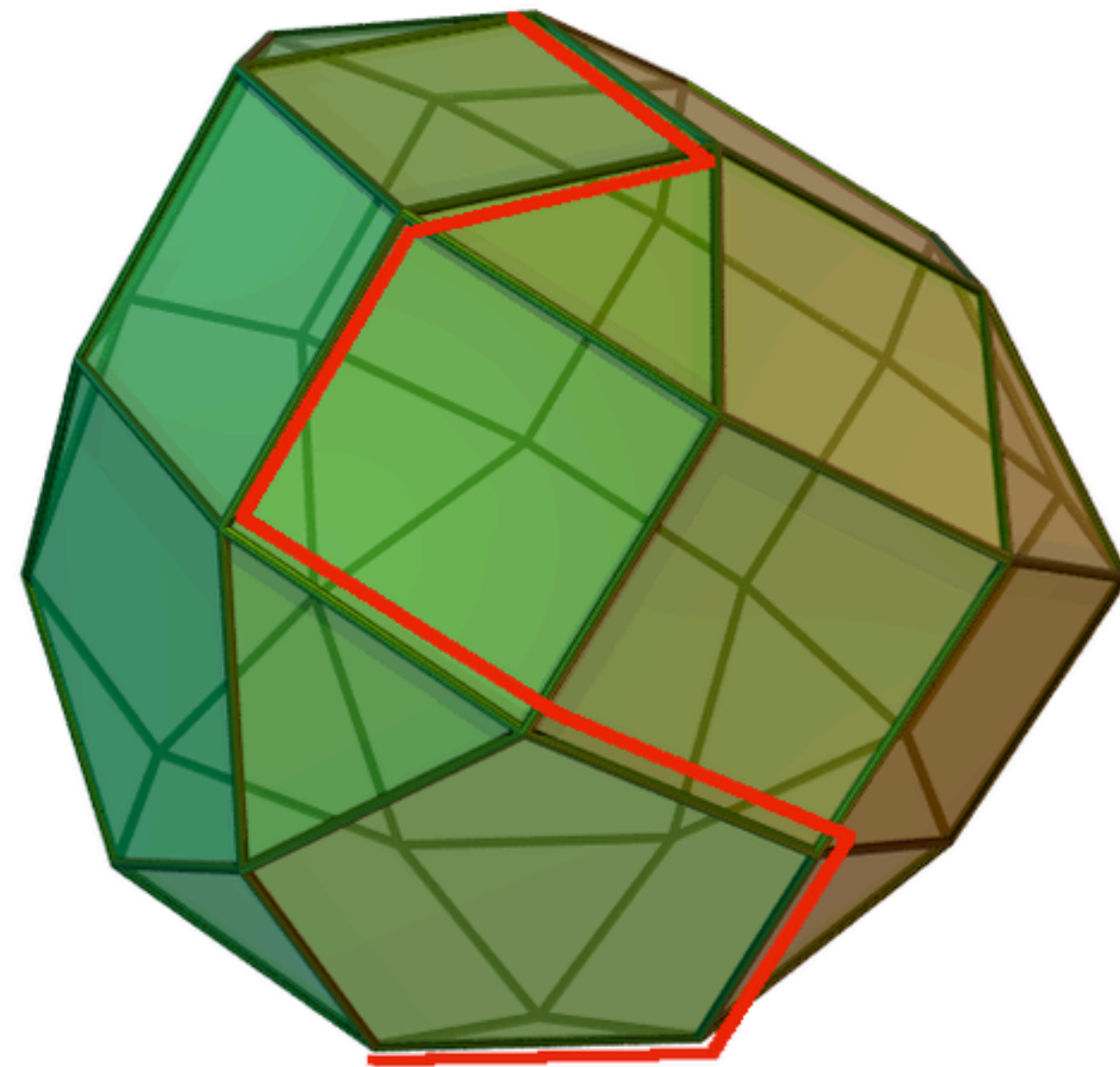
Simplex



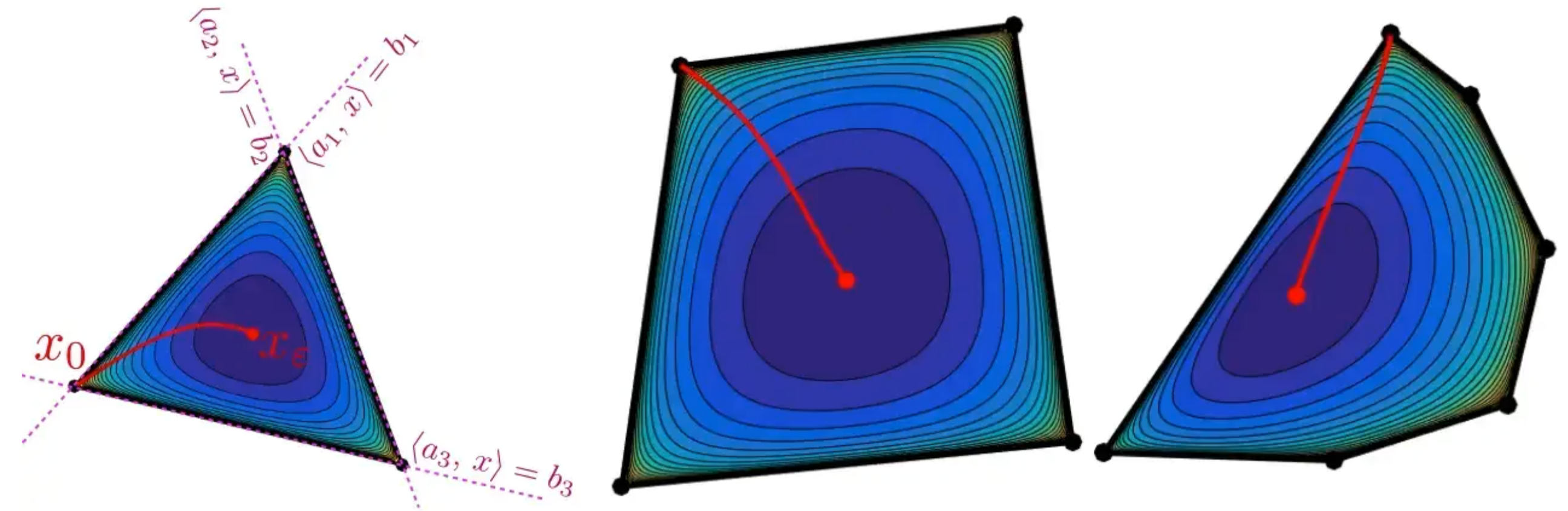
Barrier Interior Point



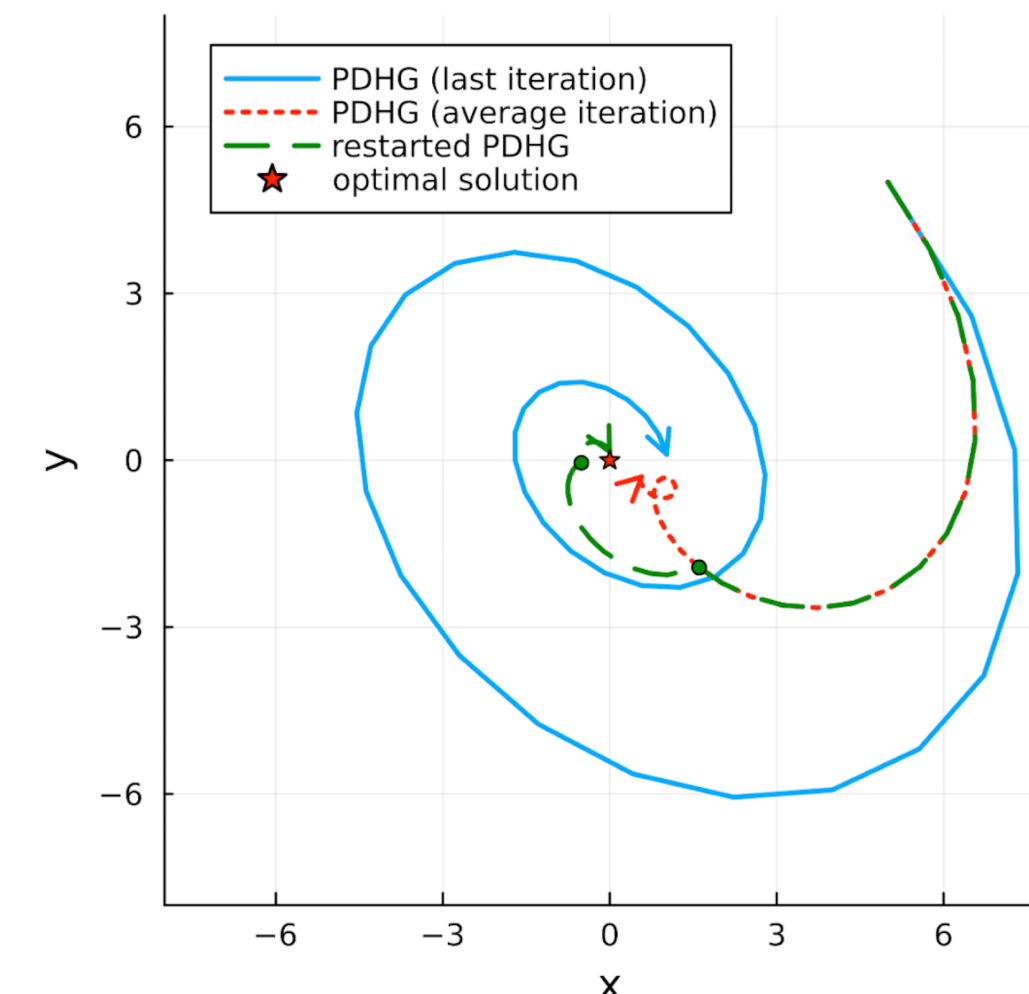
Simplex



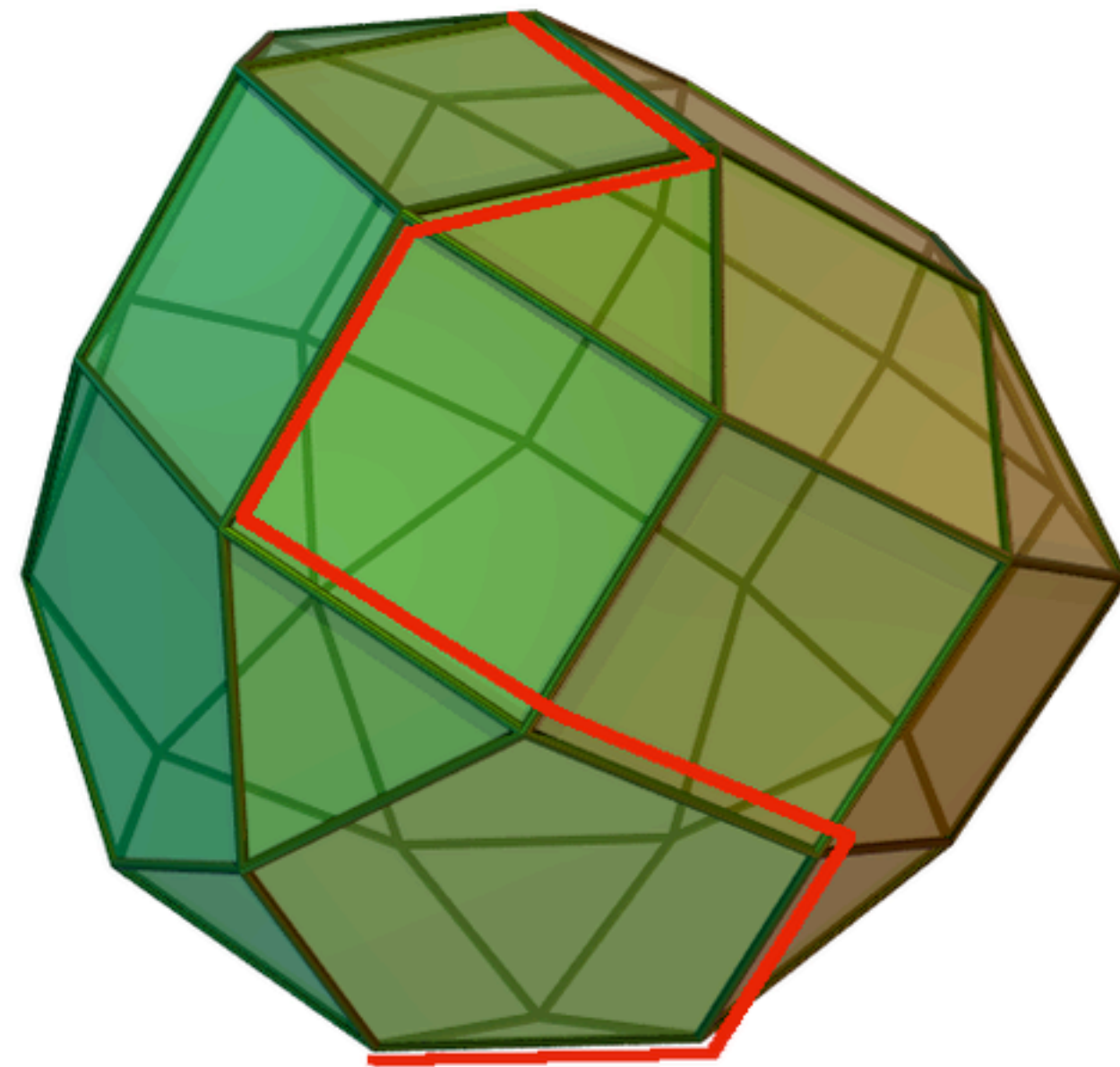
Barrier Interior Point



Primal-Dual Hybrid Gradient (PDHG/PDLP)

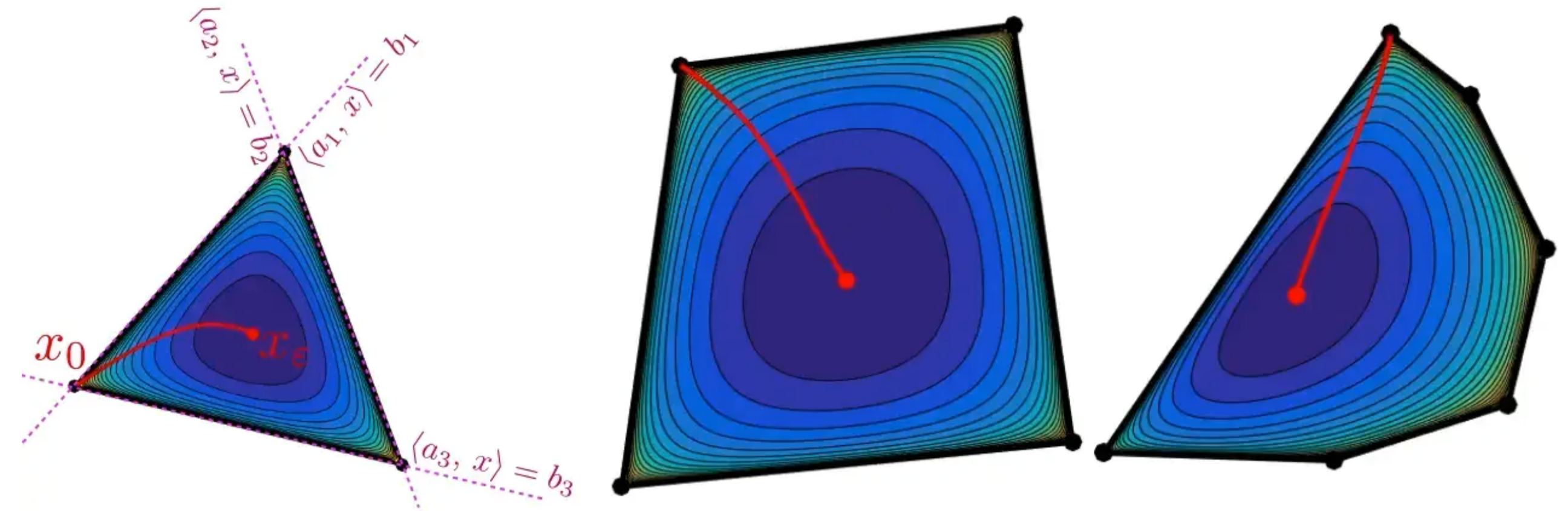


Simplex

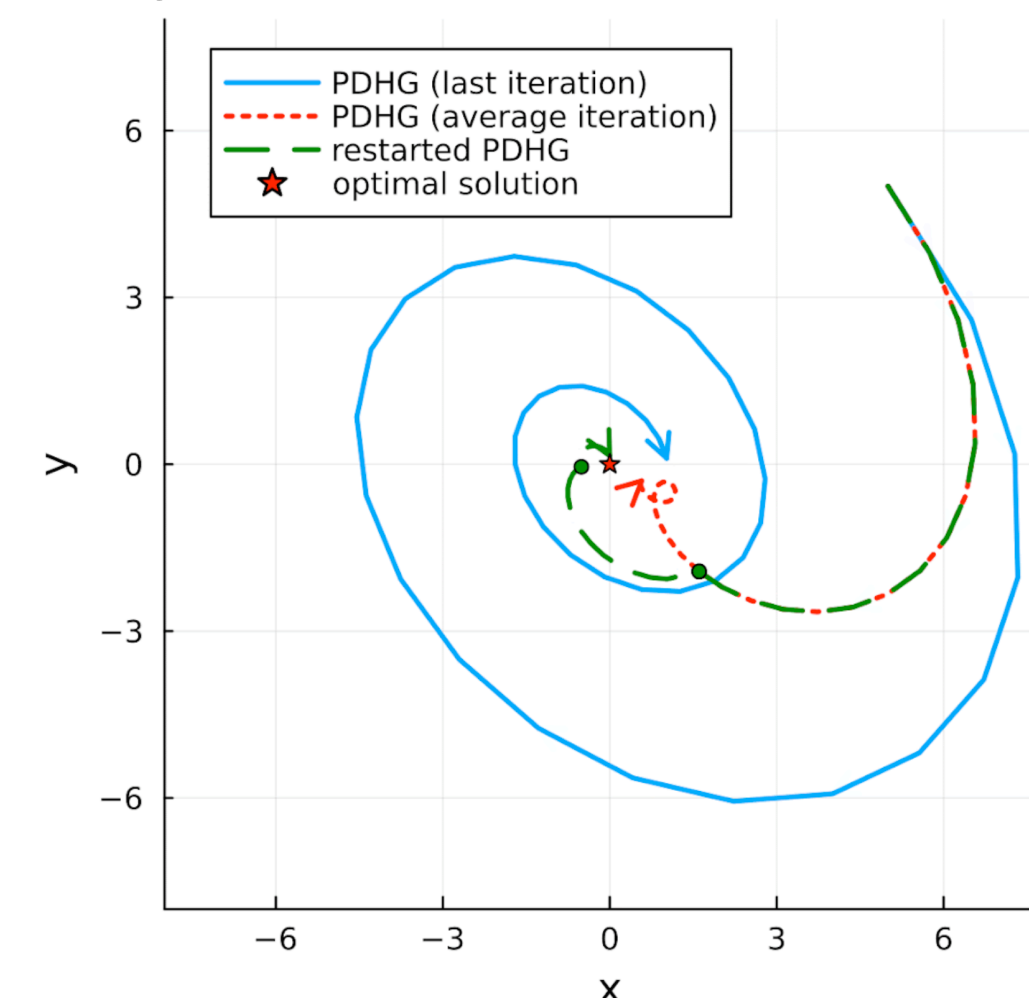


Caveat: looks easier in low dimensions!

Barrier Interior Point



Primal-Dual Hybrid Gradient (PDHG/PDLP)



Linear Programming: Practical Algorithms

Simplex:

- Many cheap iterations
- Not known to be polynomial
- Uses warm starts well
- Hard to parallelize effectively
- Natively produces basic solutions
- Can struggle on real-world instances

Simplex:

- Many cheap iterations
- Not known to be polynomial
- Uses warm starts well
- Hard to parallelize effectively
- Natively produces basic solutions
- Can struggle on real-world instances

Barrier:

- Fewer, more expensive iterations
- Parallelizable
- Uses warm starts badly
- Can be significantly faster
- Known to be polynomial
- Solves more powerful problems
- Crossover to basic solution can dominate

Simplex:

- Many cheap iterations
- Not known to be polynomial
- Uses warm starts well
- Hard to parallelize effectively
- Natively produces basic solutions
- Can struggle on real-world instances

Barrier:

- Fewer, more expensive iterations
- Parallelizable
- Uses warm starts badly
- Can be significantly faster
- Known to be polynomial
- Solves more powerful problems
- Crossover to basic solution can dominate

PDLP:

- Many cheap iterations
- Parallelizable (even profits from GPUs)
- Warm starts: not great, not terrible
- No matrix-matrix operations (only matrix-vector)
- Can be significantly faster (at least to moderate precision)
- Not known to be polynomial (different issues than simplex)
- Solves more powerful problems
- Crossover to basic solution can dominate

LP:

$$\begin{aligned} \max \quad & \sum_i c_i x_i \text{ s.t.} \\ & 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ & \ell_i \leq x_i \leq u_i \end{aligned}$$

(Mixed) Integer Programming

LP:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \end{aligned}$$



MIP:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \\ \forall i \in I : x_i \in \mathbb{Z} \end{aligned}$$

LP:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \end{aligned} \quad \longrightarrow$$

MIP:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \\ \forall i \in I : x_i \in \mathbb{Z} \end{aligned}$$

Only integer variables: Integer Program

LP:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \end{aligned}$$



MIP:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \\ \forall i \in I : x_i \in \mathbb{Z} \end{aligned}$$

Only integer variables: Integer Program

Only $\{0,1\}$ -variables: Binary Program/ $\{0,1\}$ Integer Program

LP:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \end{aligned}$$



MIP:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \\ \forall i \in I : x_i \in \mathbb{Z} \end{aligned}$$

Only integer variables: Integer Program

Only $\{0,1\}$ -variables: Binary Program/ $\{0,1\}$ Integer Program

MIP, IP, BP are all known NP-complete problems.

LP:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \end{aligned}$$



MIP:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \\ \forall i \in I : x_i \in \mathbb{Z} \end{aligned}$$

Only integer variables: Integer Program

Only $\{0,1\}$ -variables: Binary Program/ $\{0,1\}$ Integer Program

MIP, IP, BP are all known NP-complete problems.

Practically solved by MIP solvers (Branch & Cut + Extensions)

MIP Example: TSP

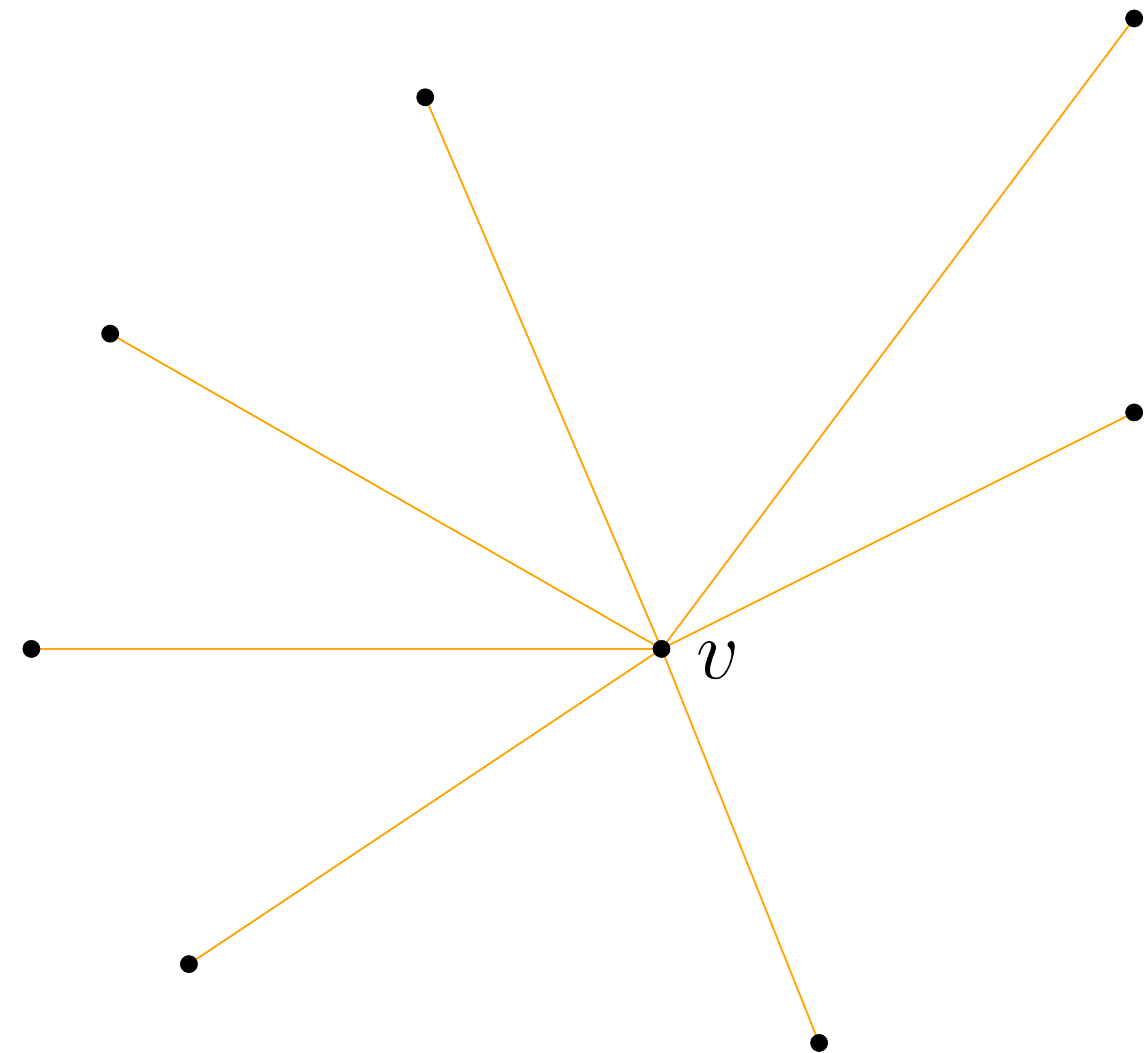
TSP formulation as IP (Dantzig, Fulkerson, Johnson):

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

**TSP formulation as IP
(Dantzig, Fulkerson, Johnson):**

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V: \sum_{e \in \delta(\{v\})} x_e = 2$$



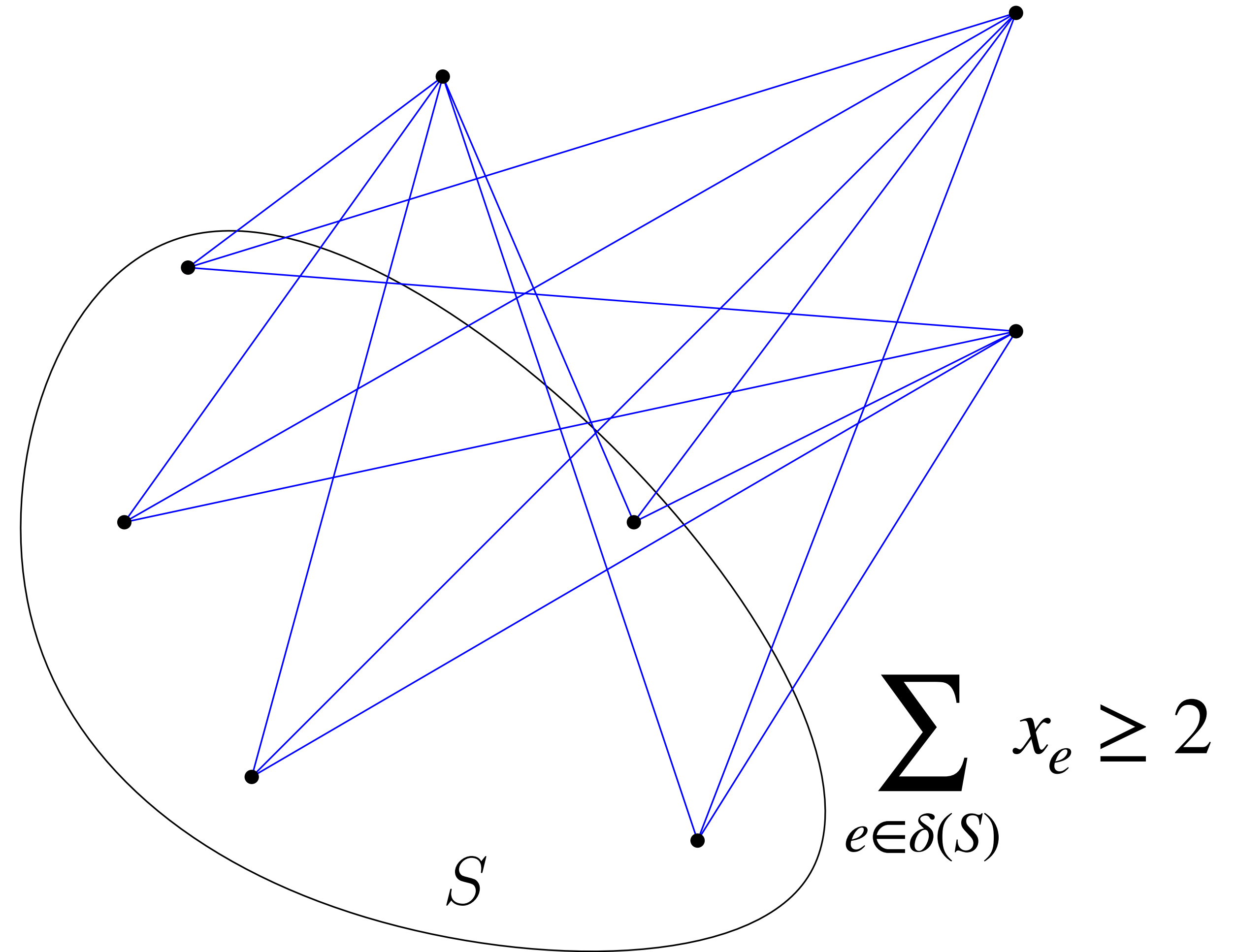
$$\sum_{e \in \delta(\{v\})} x_e = 2$$

TSP formulation as IP (Dantzig, Fulkerson, Johnson):

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V: \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset: \sum_{e \in \delta(S)} x_e \geq 2$$



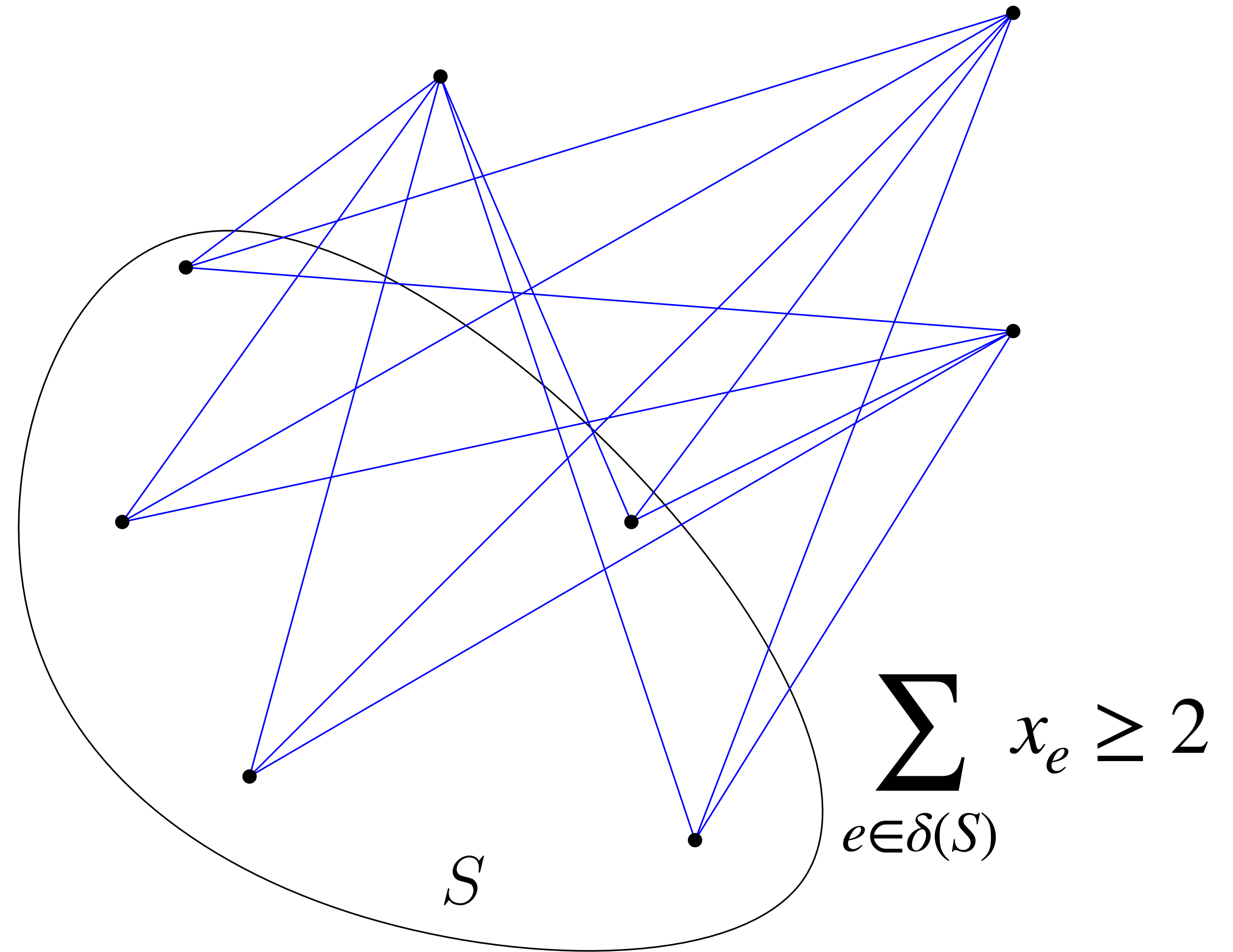
TSP formulation as IP (Dantzig, Fulkerson, Johnson):

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V: \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset: \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \in \{0,1\}$$



TSP formulation as IP (Dantzig, Fulkerson, Johnson):

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V : \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset : \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \in \{0,1\}$$

**TSP formulation as IP
(Dantzig, Fulkerson, Johnson):**

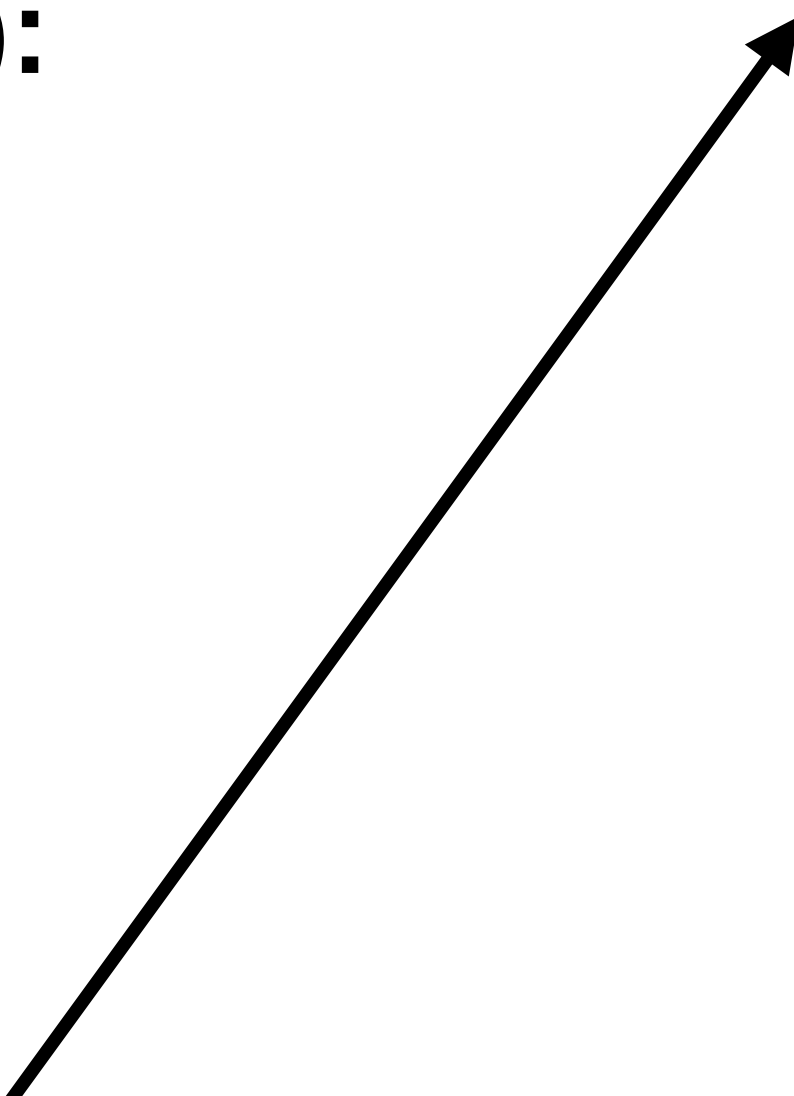
$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V : \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset : \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \in \{0,1\}$$

What's the issue with these?



**TSP formulation as IP
(Dantzig, Fulkerson, Johnson):**

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V : \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset : \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \in \{0,1\}$$

What's the issue with these?

- Exponentially many!

TSP formulation as IP (Dantzig, Fulkerson, Johnson):

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V : \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset : \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \in \{0,1\}$$

What's the issue with these?

- Exponentially many!
- We add them when needed!

TSP formulation as IP (Dantzig, Fulkerson, Johnson):

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V: \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset: \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \in \{0,1\}$$

What's the issue with these?

- Exponentially many!
- We add them when needed!
- Hopefully, we don't need too many.

TSP formulation as IP (Dantzig, Fulkerson, Johnson):

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V: \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset: \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \in \{0,1\}$$

What's the issue with these?

- Exponentially many!
- We add them when needed!
- Hopefully, we don't need too many.
- Not proven, but works in practice.

TSP formulation as IP (Dantzig, Fulkerson, Johnson):

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V: \sum_{e \in \delta(\{v\})} x_e = 2$$

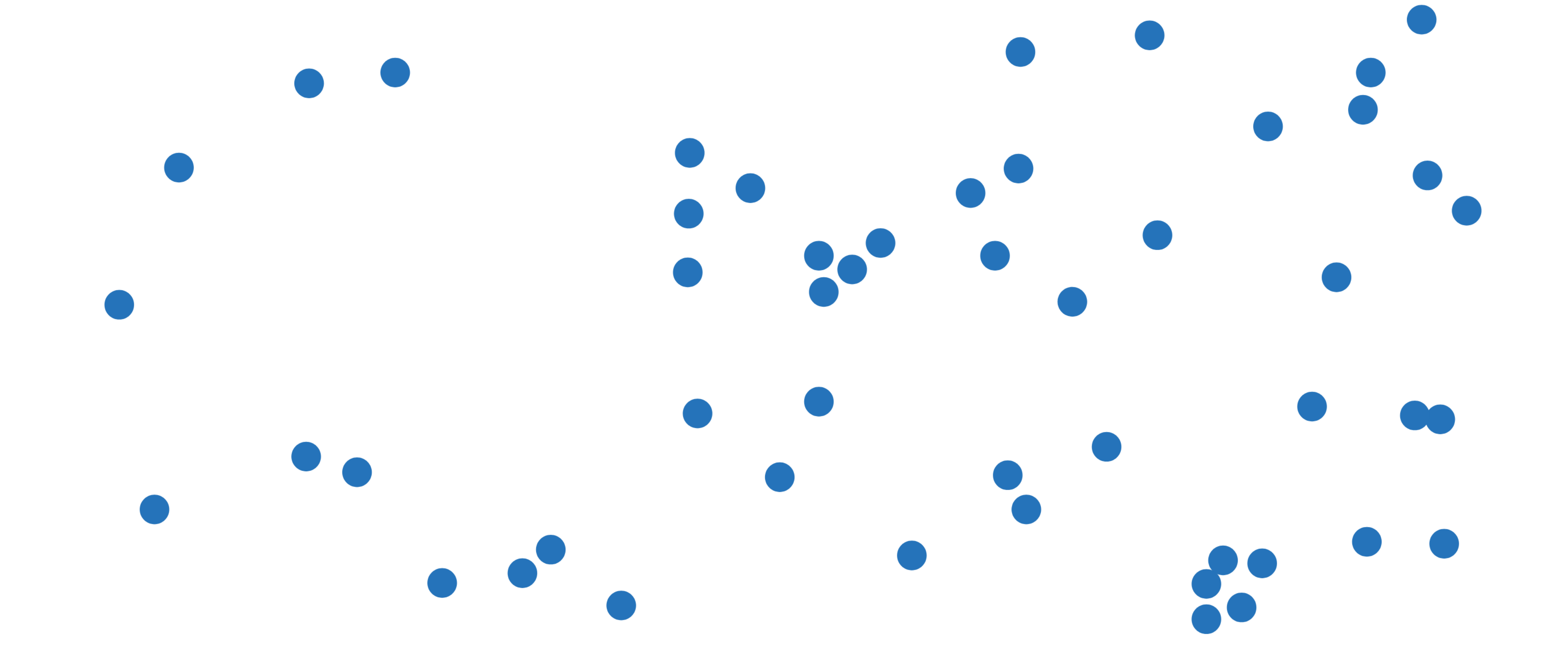
$$\forall S \subsetneq V, S \neq \emptyset: \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \in \{0,1\}$$

What's the issue with these?

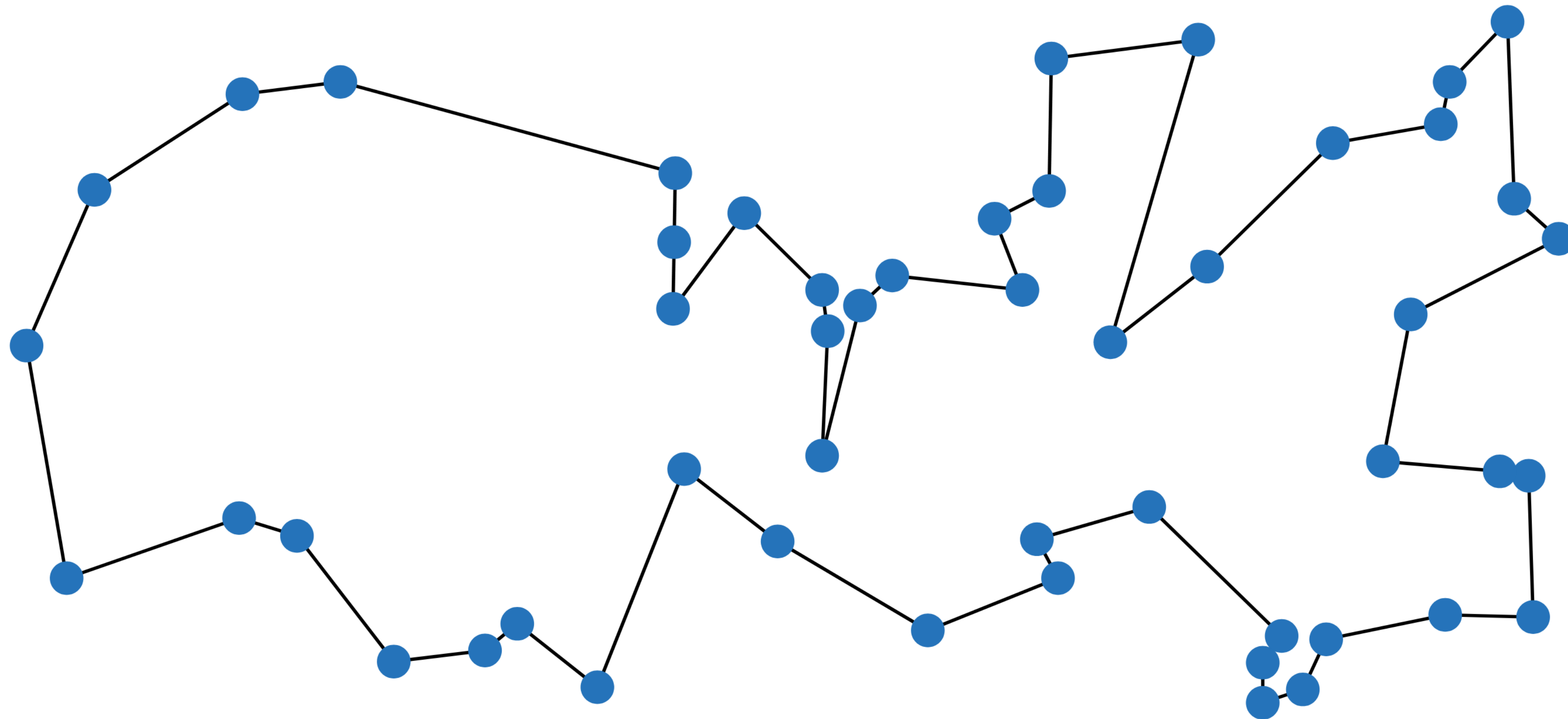
- Exponentially many!
 - We add them when needed!
 - Hopefully, we don't need too many.
 - Not proven, but works in practice.
- ➔ Lazy constraints.

Example



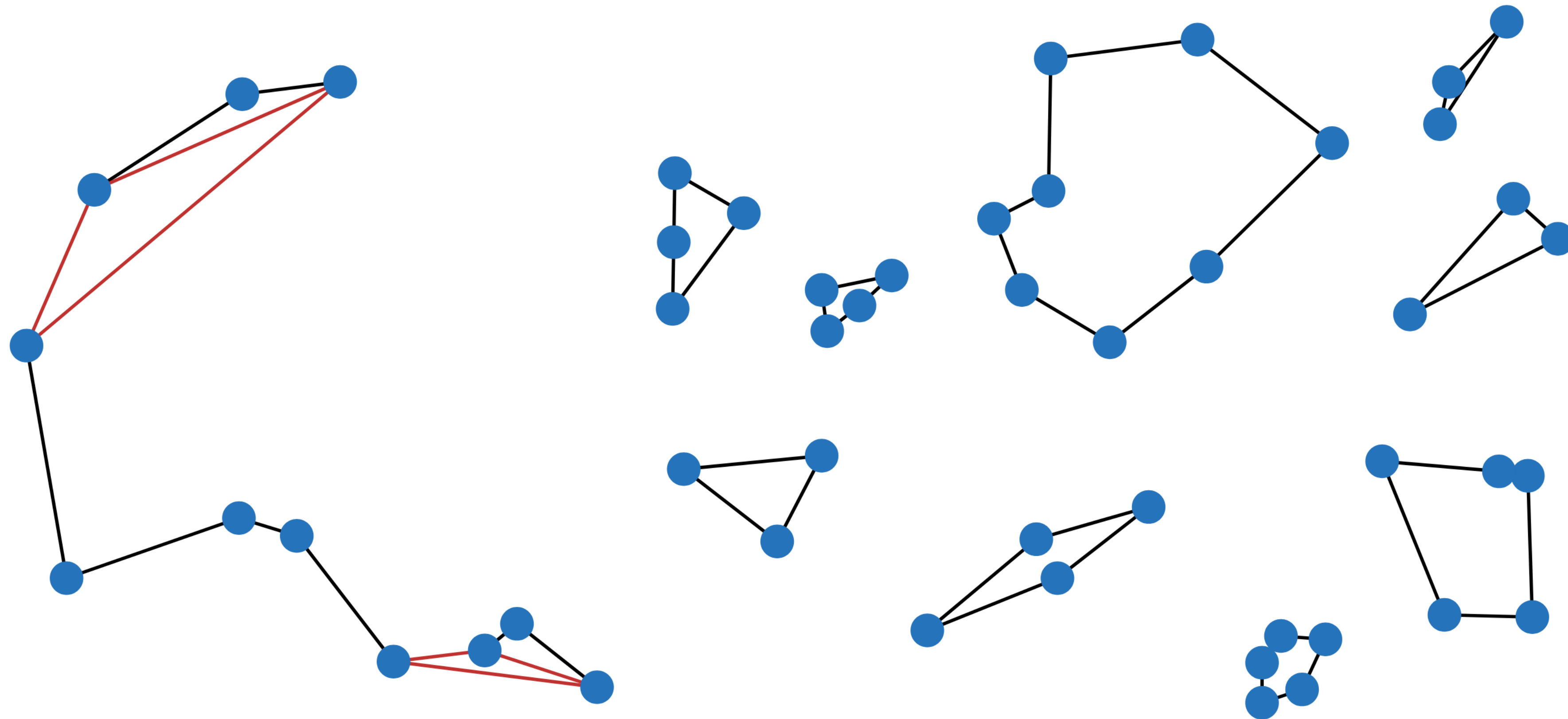
Random 50-city TSP instance (Euklidean distances).

Initialize: Heuristic Solution



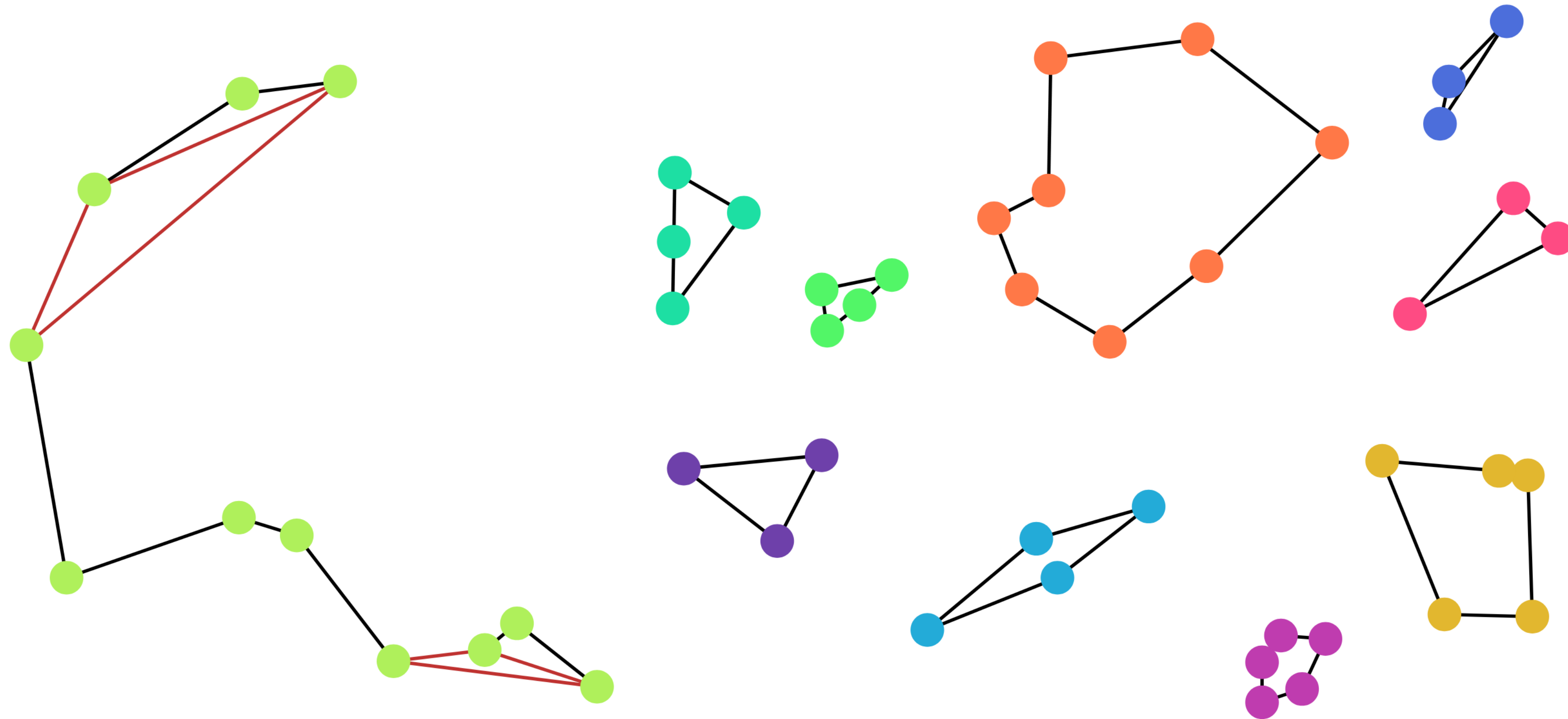
Heuristic solution; total distance: 54277, new best (new UB).

Solve Root Relaxation



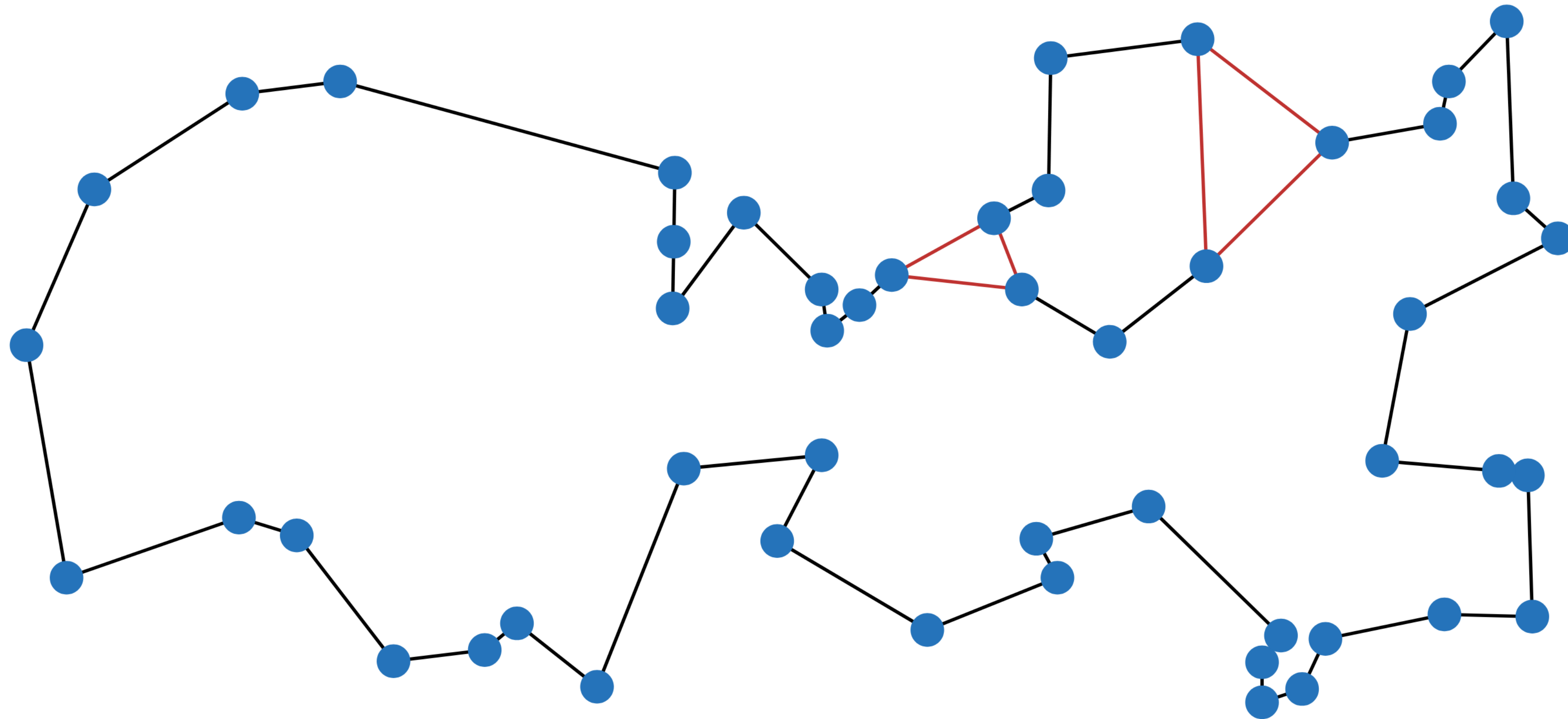
LP relaxation (no subtour constraints): 49125, best UB: 54277.

Solve Root Relaxation



LP relaxation (no subtour constraints): 49125, best UB: 54277.

Solve Root Relaxation



LP relaxation (10 subtour constraints): 51766 (+2641), best UB: 54277.

All subtour constraints are satisfied here!

Cutting Planes

Scenario: we have a fractional solution x^* ; all linear constraints are satisfied.

But: The solution is not integral; what (except for branching) can we do?

Cutting Planes

Scenario: we have a fractional solution x^* ; all linear constraints are satisfied.

But: The solution is not integral; what (except for branching) can we do?

Idea: look for additional constraints (cutting planes) that:

Cutting Planes

Scenario: we have a fractional solution x^* ; all linear constraints are satisfied.

But: The solution is not integral; what (except for branching) can we do?

Idea: look for additional constraints (cutting planes) that:

- are linear,

Cutting Planes

Scenario: we have a fractional solution x^* ; all linear constraints are satisfied.

But: The solution is not integral; what (except for branching) can we do?

Idea: look for additional constraints (cutting planes) that:

- are linear,
- are satisfied by **all** (or at least all potentially optimal) integer solutions,

Cutting Planes

Scenario: we have a fractional solution x^* ; all linear constraints are satisfied.

But: The solution is not integral; what (except for branching) can we do?

Idea: look for additional constraints (cutting planes) that:

- are linear,
- are satisfied by **all** (or at least all potentially optimal) integer solutions,
- are violated (by more than just some minor ε) by x^* .

Cutting Planes

Scenario: we have a fractional solution x^* ; all linear constraints are satisfied.

But: The solution is not integral; what (except for branching) can we do?

Idea: look for additional constraints (cutting planes) that:

- are linear,
- are satisfied by **all** (or at least all potentially optimal) integer solutions,
- are violated (by more than just some minor ε) by x^* .

We can **add** such constraints and re-solve:

Cutting Planes

Scenario: we have a fractional solution x^* ; all linear constraints are satisfied.

But: The solution is not integral; what (except for branching) can we do?

Idea: look for additional constraints (cutting planes) that:

- are linear,
- are satisfied by **all** (or at least all potentially optimal) integer solutions,
- are violated (by more than just some minor ε) by x^* .

We can **add** such constraints and re-solve:

- we retain an LP, i.e., efficiently solvable problem,

Cutting Planes

Scenario: we have a fractional solution x^* ; all linear constraints are satisfied.

But: The solution is not integral; what (except for branching) can we do?

Idea: look for additional constraints (cutting planes) that:

- are linear,
- are satisfied by **all** (or at least all potentially optimal) integer solutions,
- are violated (by more than just some minor ε) by x^* .

We can **add** such constraints and re-solve:

- we retain an LP, i.e., efficiently solvable problem,
- we do not lose any integer solution (or any potentially optimal one),

Cutting Planes

Scenario: we have a fractional solution x^* ; all linear constraints are satisfied.

But: The solution is not integral; what (except for branching) can we do?

Idea: look for additional constraints (cutting planes) that:

- are linear,
- are satisfied by **all** (or at least all potentially optimal) integer solutions,
- are violated (by more than just some minor ε) by x^* .

We can **add** such constraints and re-solve:

- we retain an LP, i.e., efficiently solvable problem,
- we do not lose any integer solution (or any potentially optimal one),
- we get a new solution (because the new constraint **cuts off** x^*).

Cutting Planes

Scenario: we have a fractional solution x^* ; all linear constraints are satisfied.

But: The solution is not integral; what (except for branching) can we do?

Idea: look for additional constraints (cutting planes) that:

- are linear,
- are satisfied by **all** (or at least all potentially optimal) integer solutions,
- are violated (by more than just some minor ε) by x^* .

We can **add** such constraints and re-solve:

- we retain an LP, i.e., efficiently solvable problem,
- we do not lose any integer solution (or any potentially optimal one),
- we get a new solution (because the new constraint **cuts off** x^*).

The problem of looking for a new constraint is called **separation problem**.

Cutting Planes

Scenario: we have a fractional solution x^* ; all linear constraints are satisfied.

But: The solution is not integral; what (except for branching) can we do?

Idea: look for additional constraints (cutting planes) that:

- are linear,
- are satisfied by **all** (or at least all potentially optimal) integer solutions,
- are violated (by more than just some minor ε) by x^* .

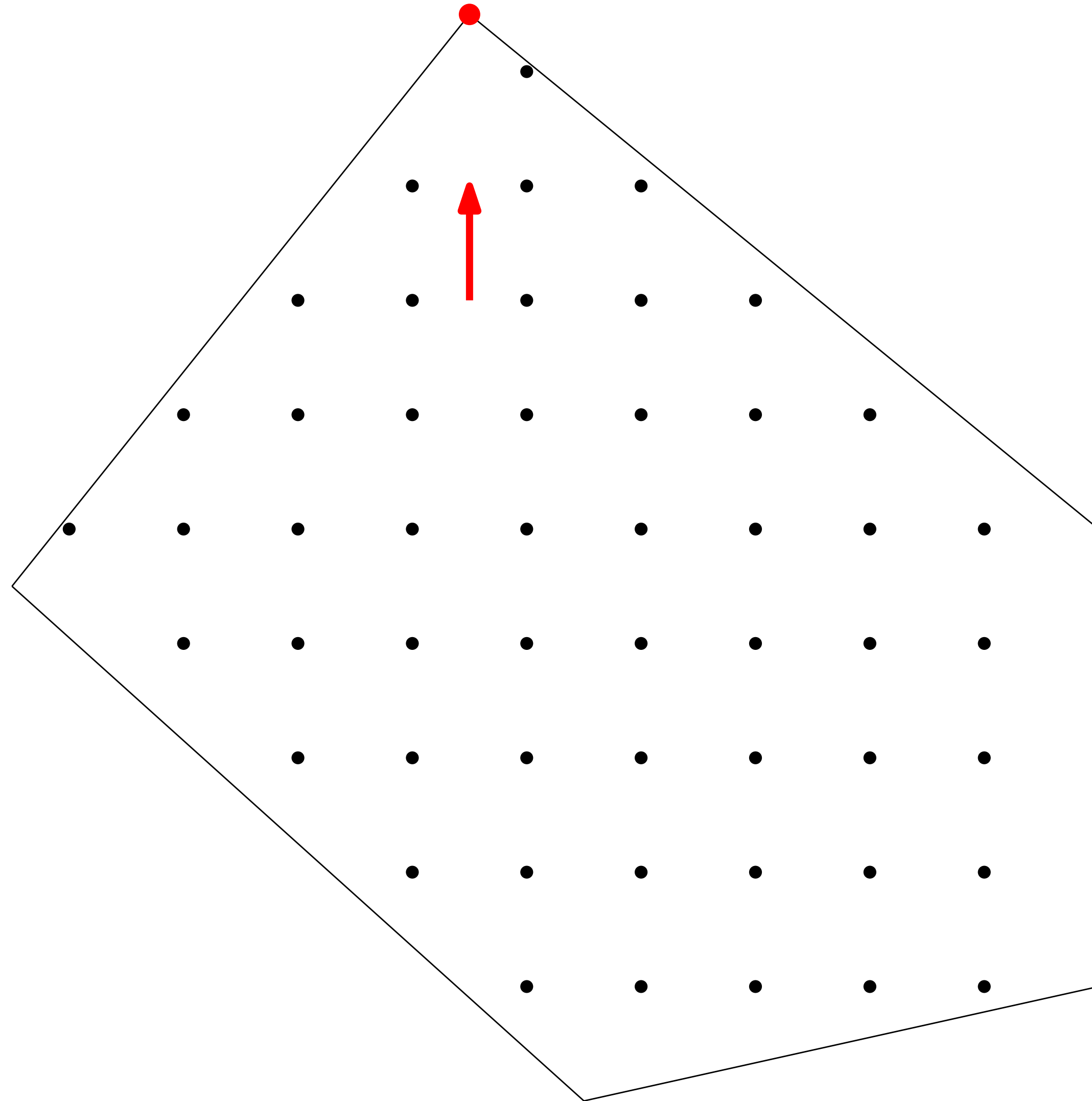
We can **add** such constraints and re-solve:

- we retain an LP, i.e., efficiently solvable problem,
- we do not lose any integer solution (or any potentially optimal one),
- we get a new solution (because the new constraint **cuts off** x^*).

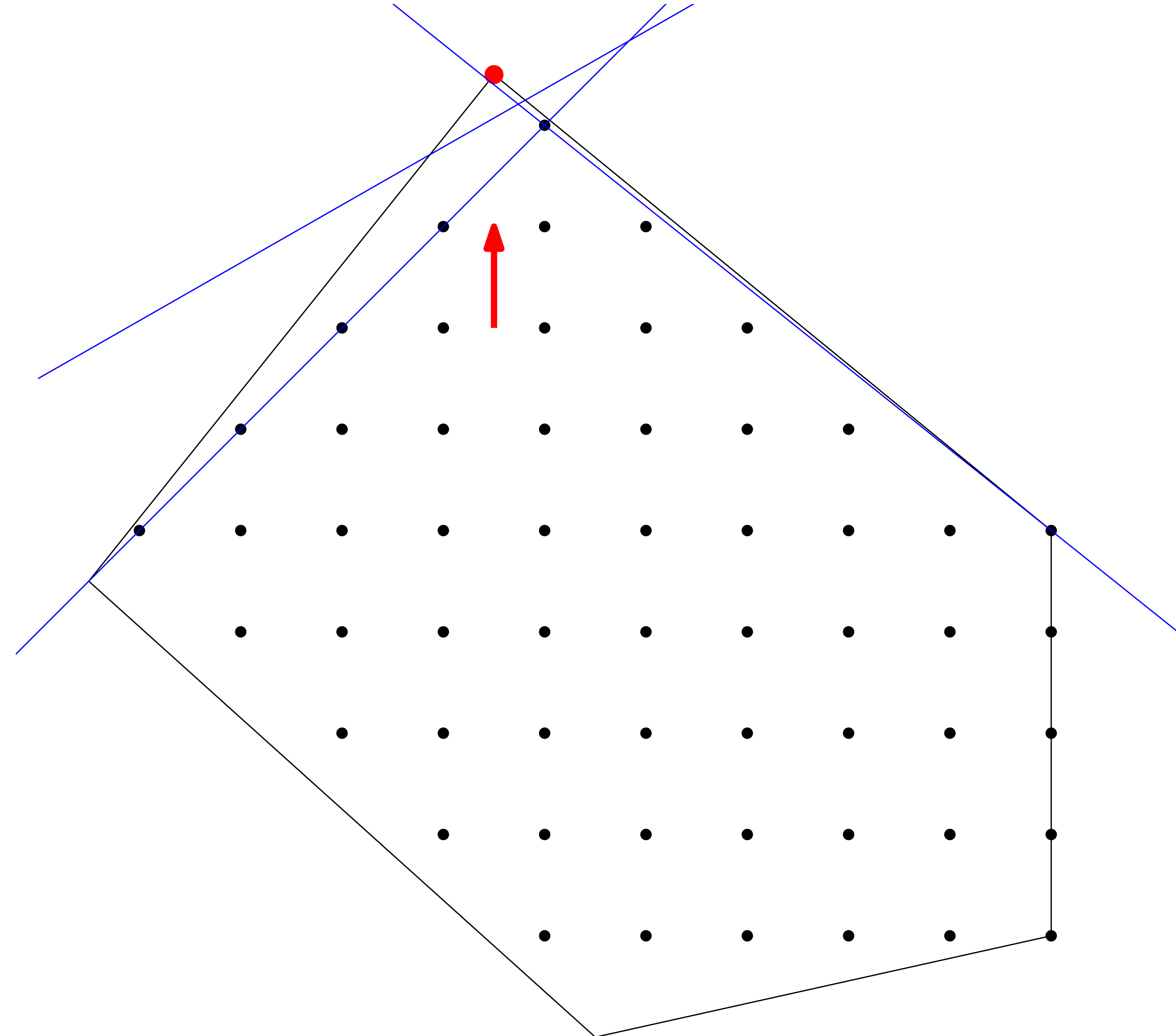
The problem of looking for a new constraint is called **separation problem**.

In most cases, it is solved by a heuristic (that can fail to find a meaningful cut).

Cutting Planes: Geometric Intuition

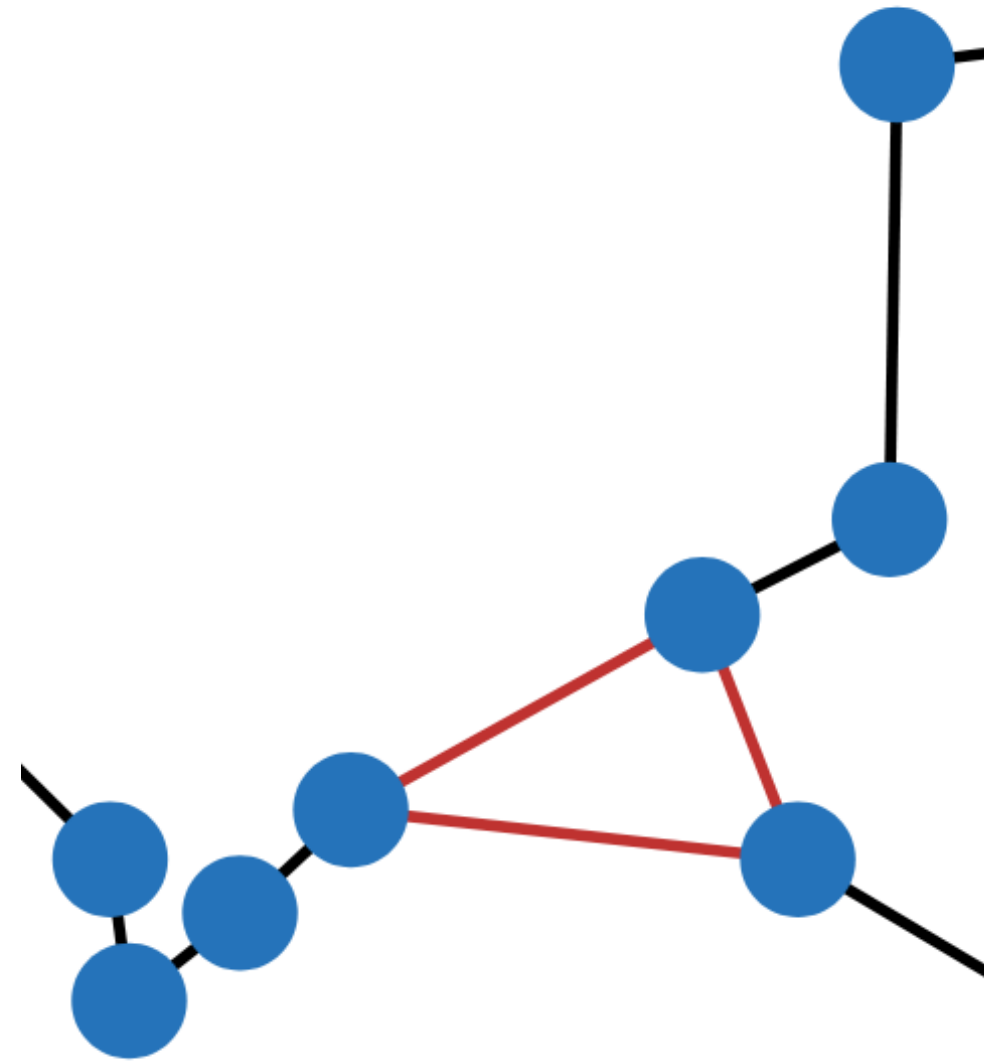


Cutting Planes: Geometric Intuition

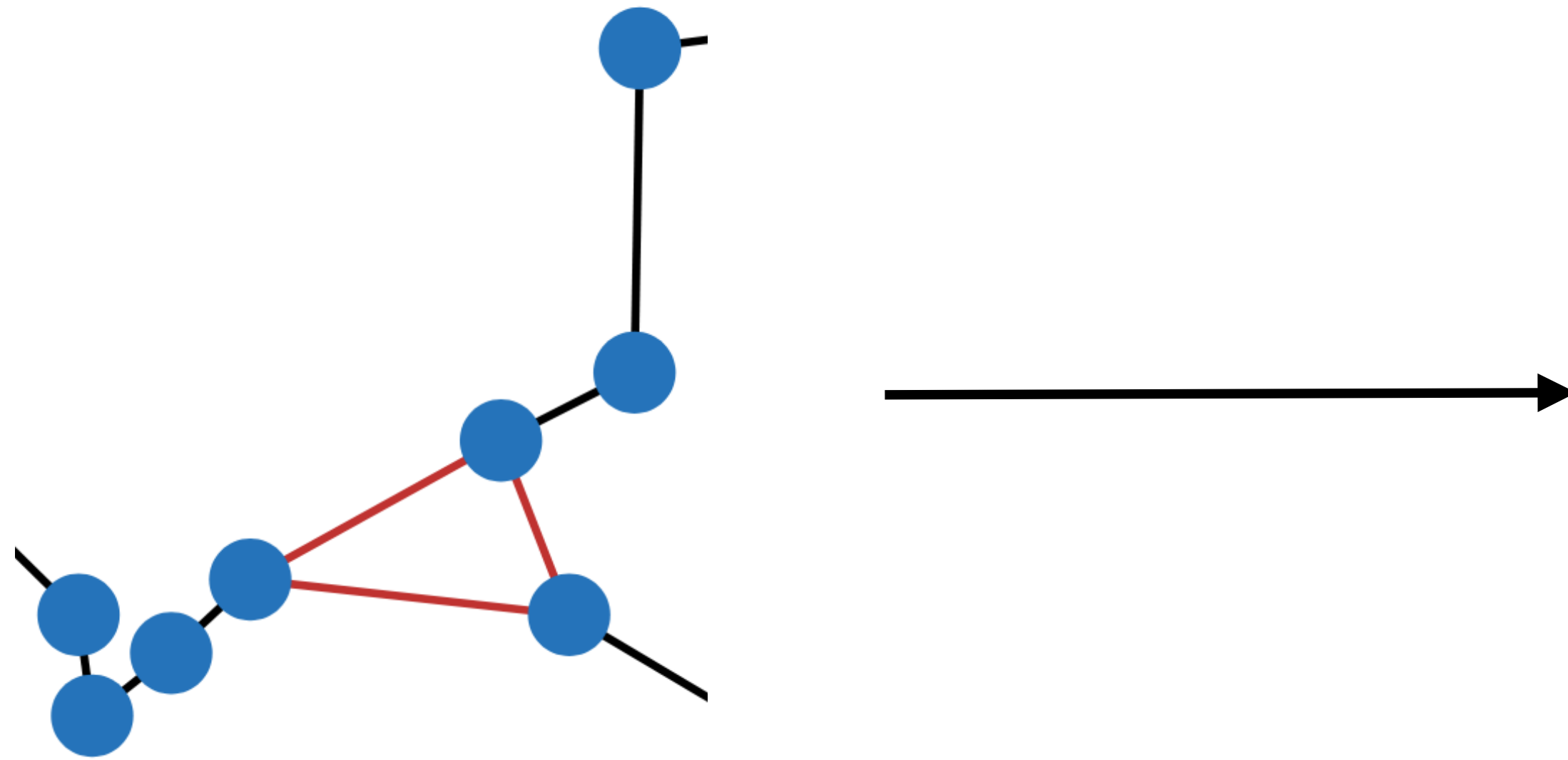


More Cutting Planes: Combs

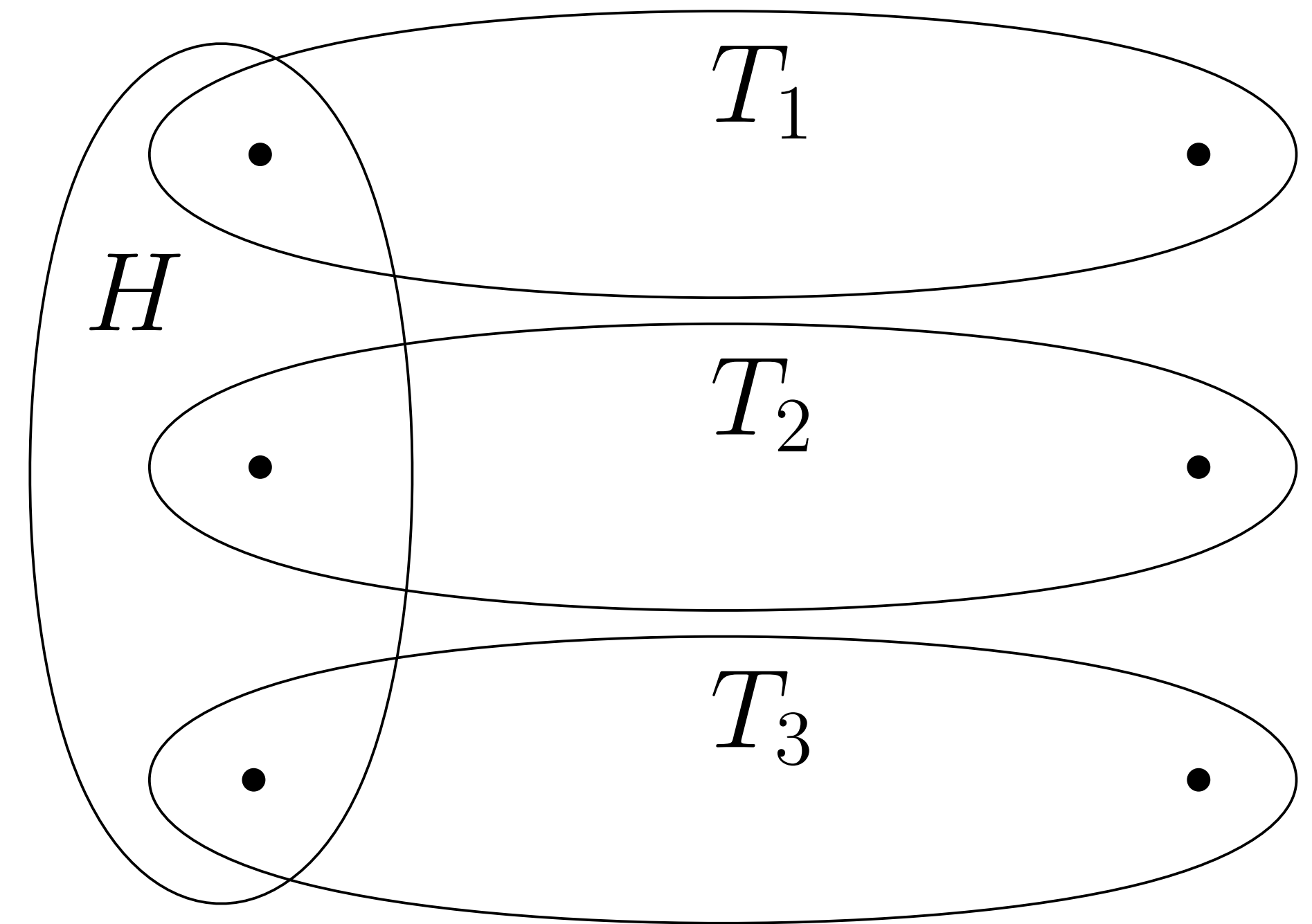
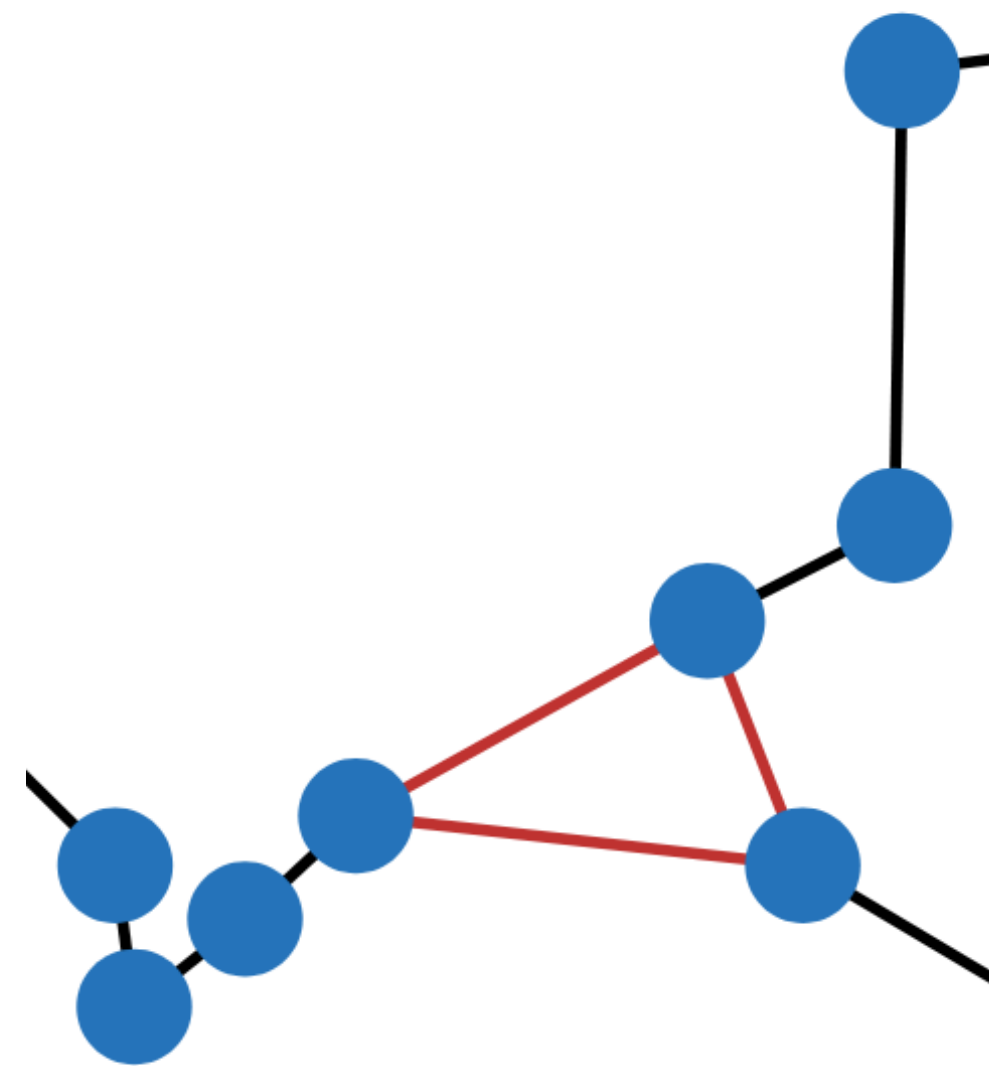
More Cutting Planes: Combs



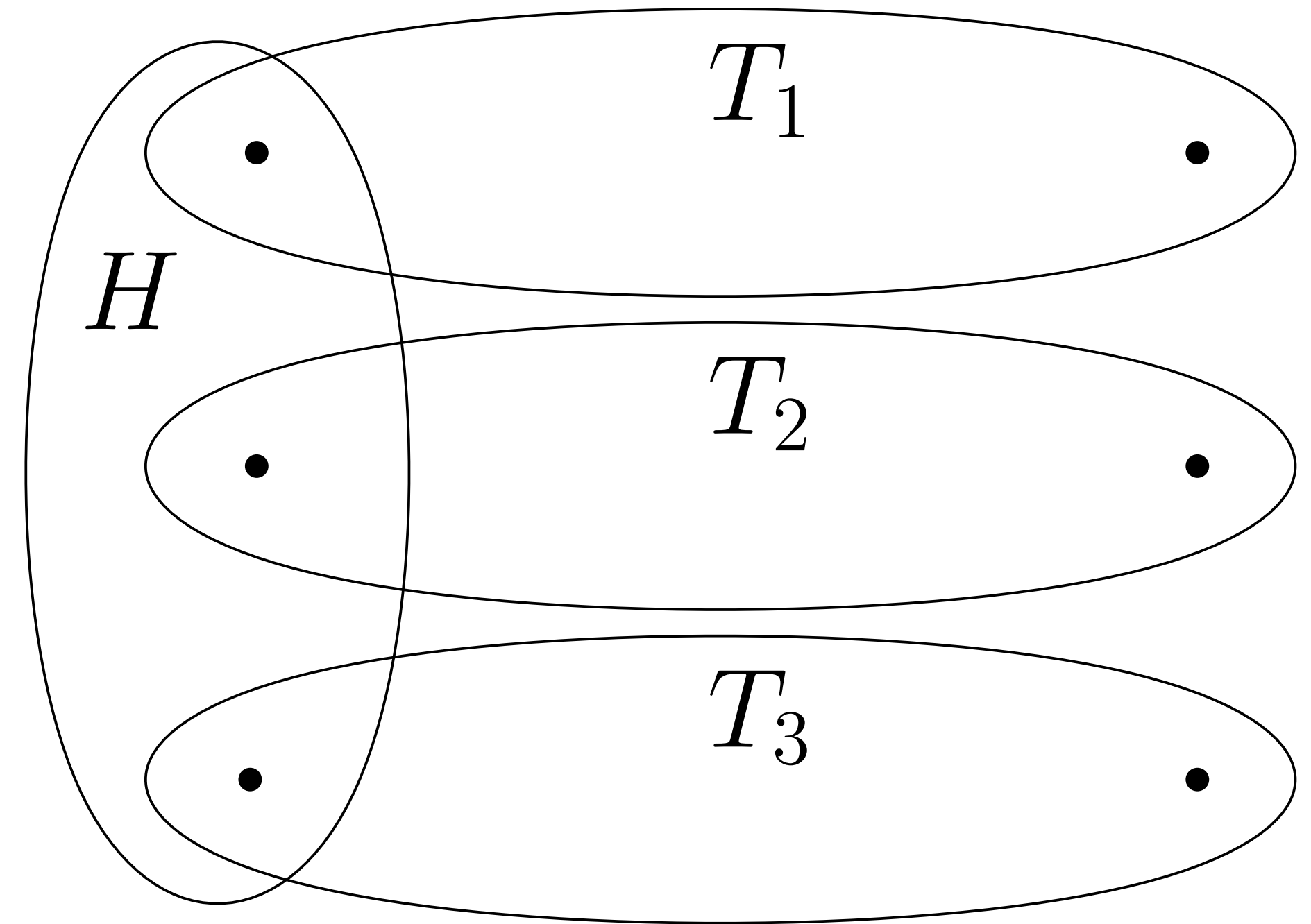
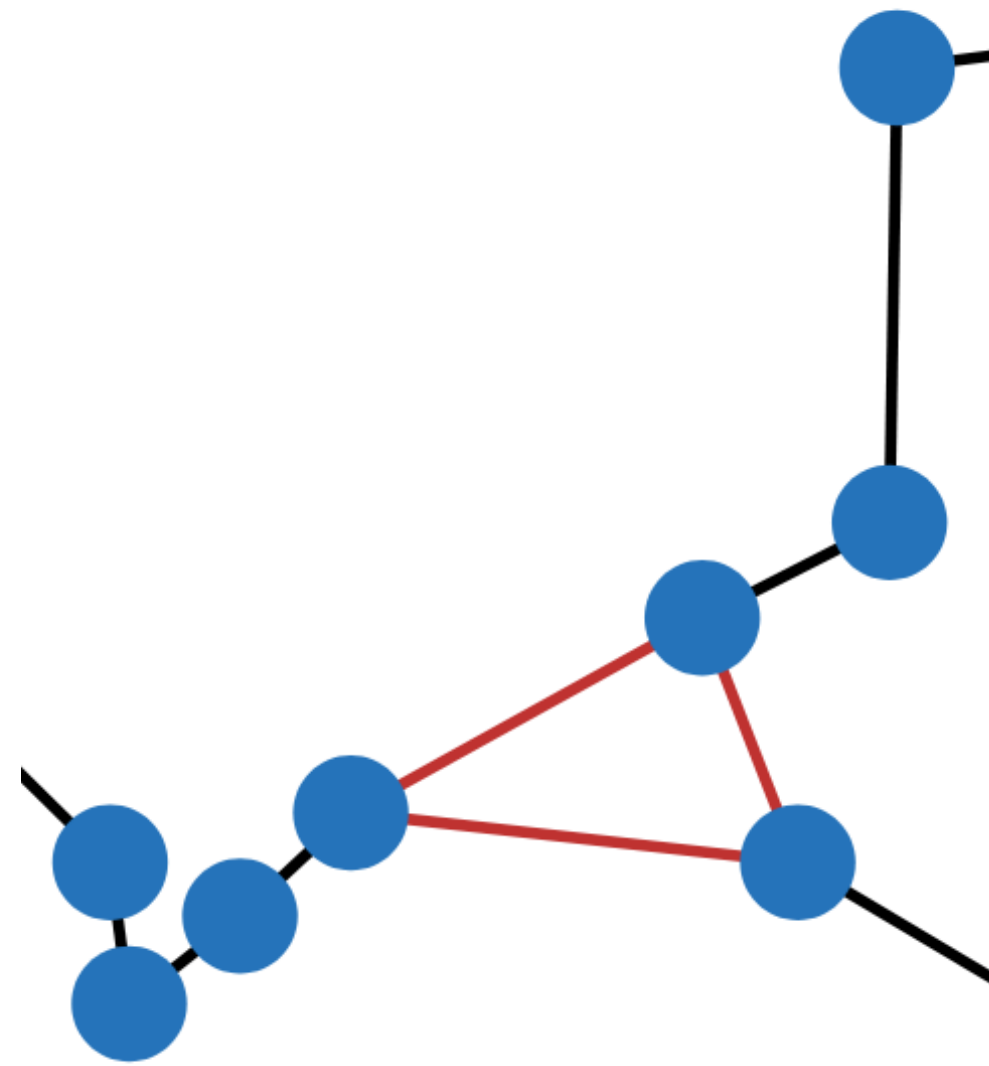
More Cutting Planes: Combs



More Cutting Planes: Combs

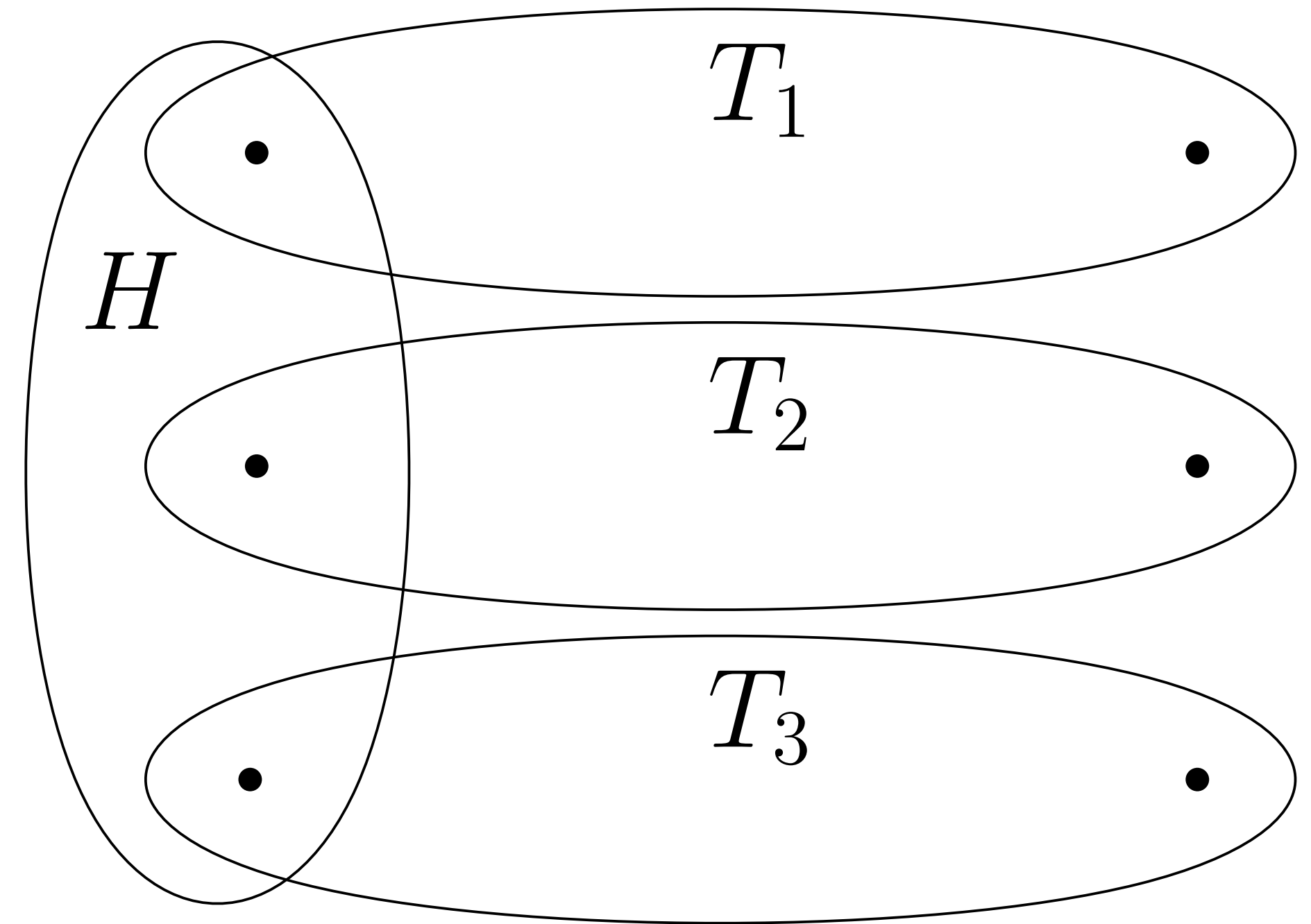
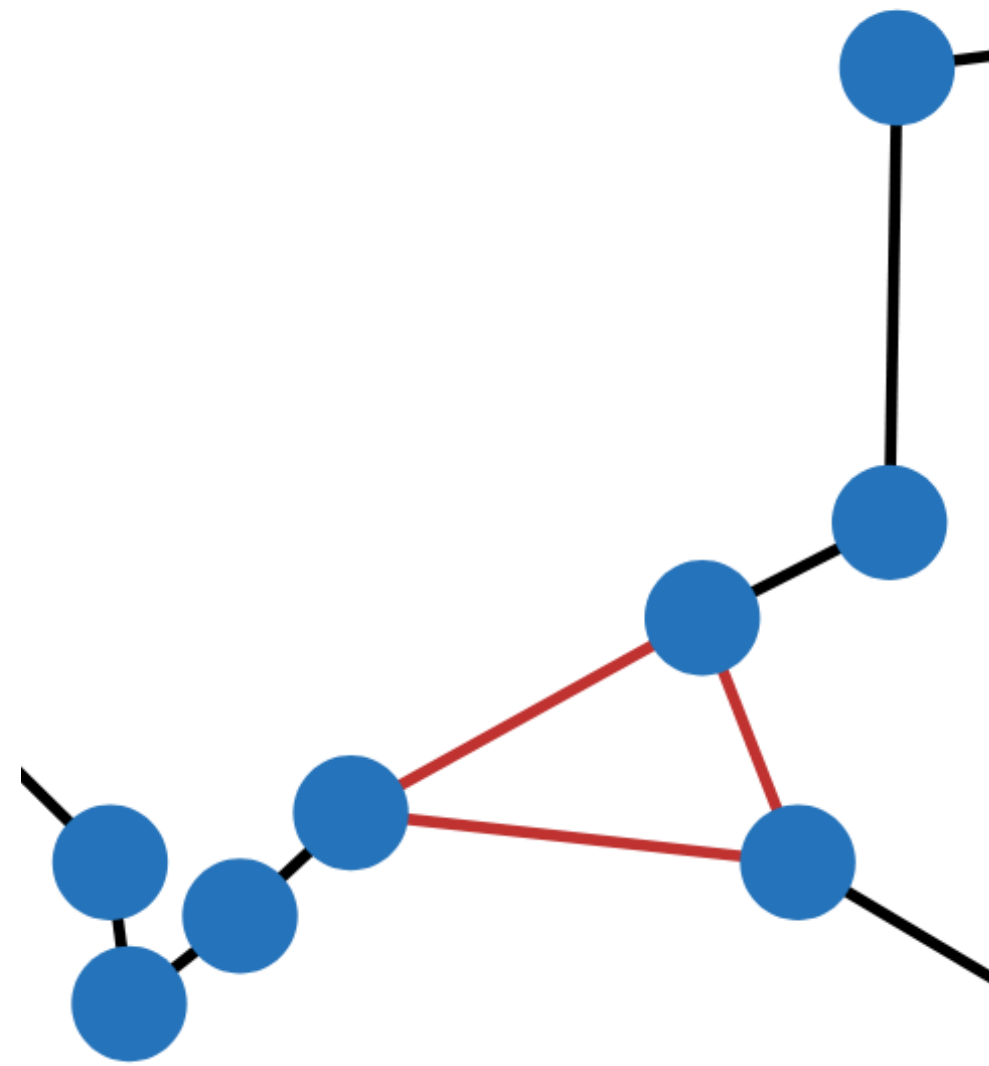


More Cutting Planes: Combs



$H, T_1, \dots, T_k, k \geq 3, k \text{ odd}$

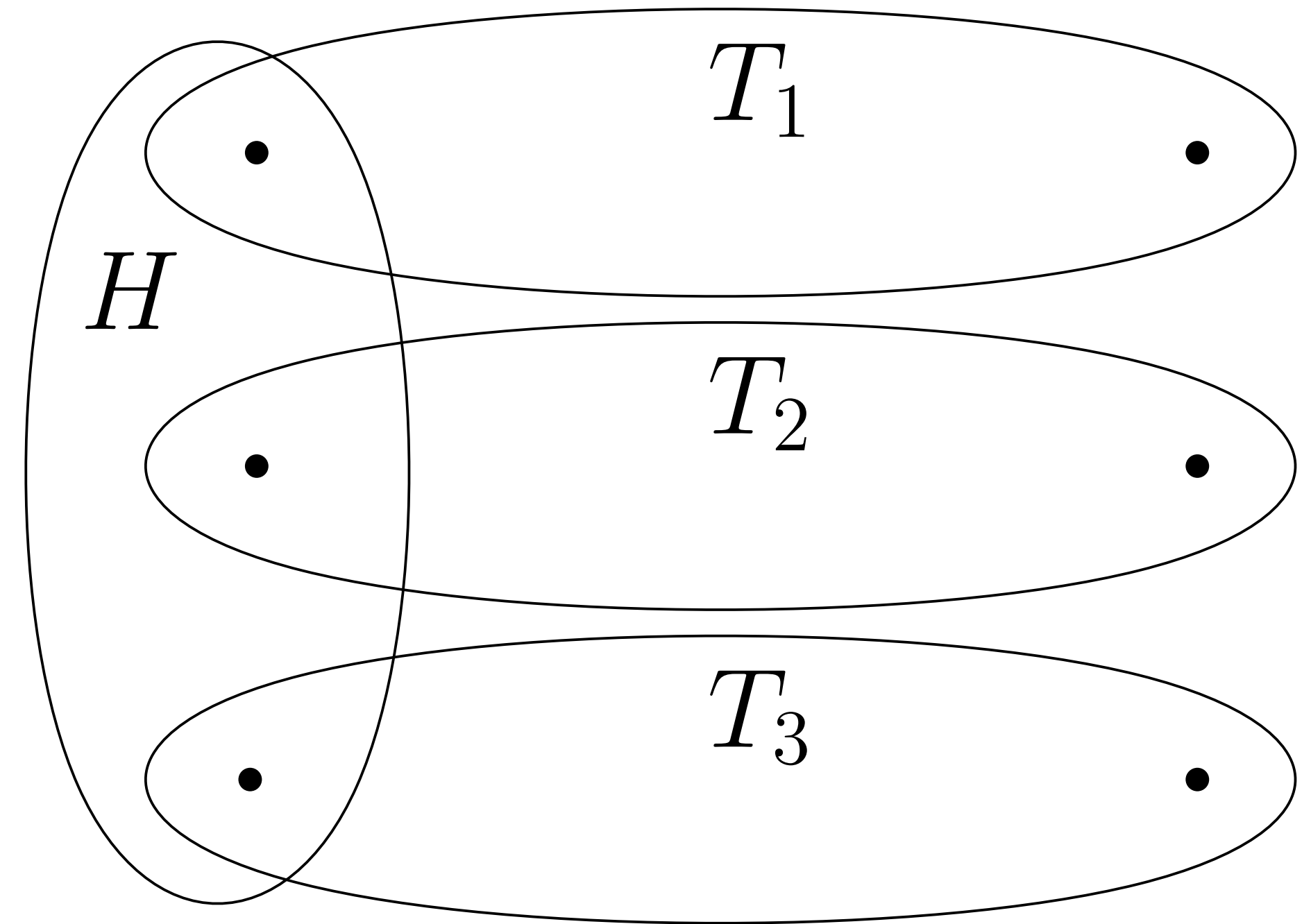
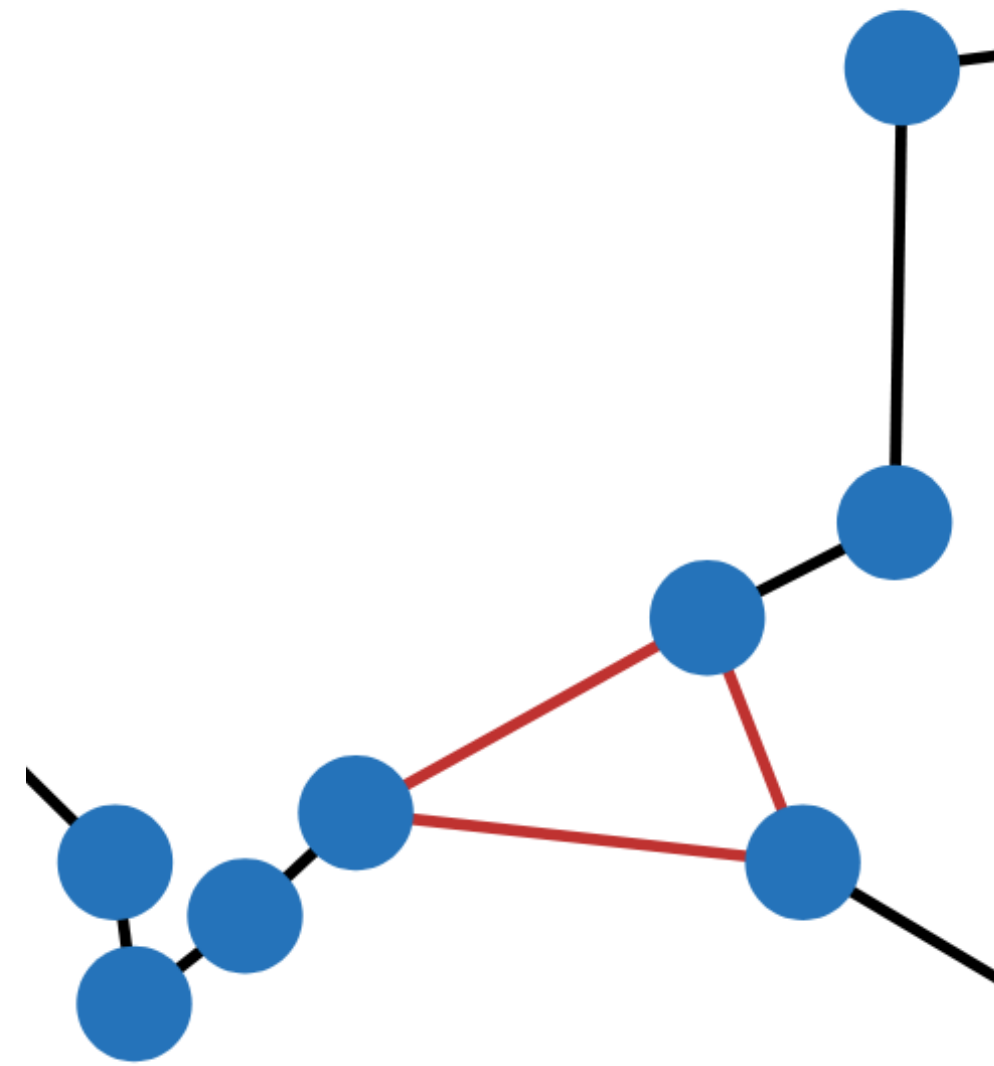
More Cutting Planes: Combs



$H, T_1, \dots, T_k, k \geq 3, k \text{ odd}$

$$T_i \cap T_j = \emptyset$$

More Cutting Planes: Combs

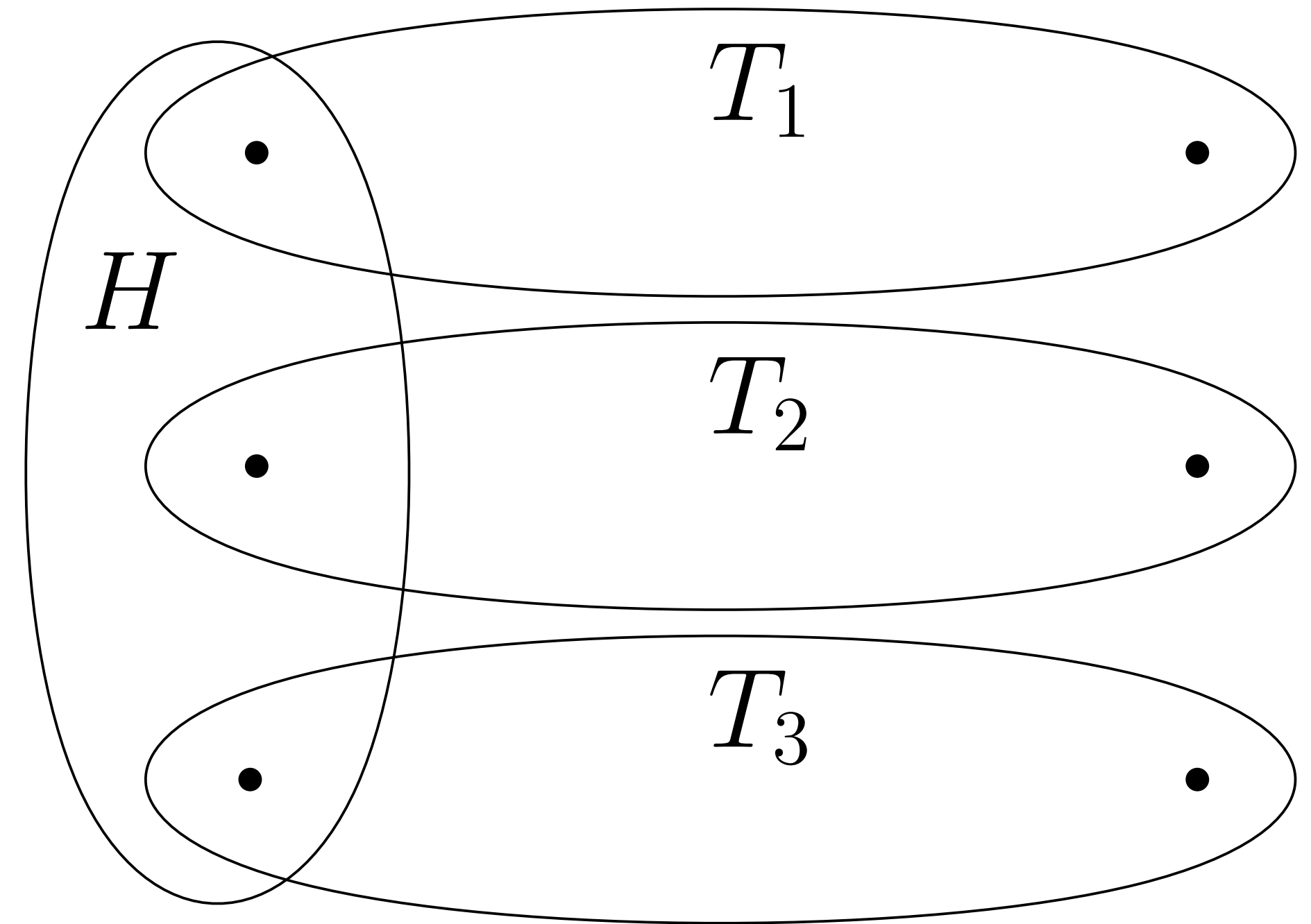
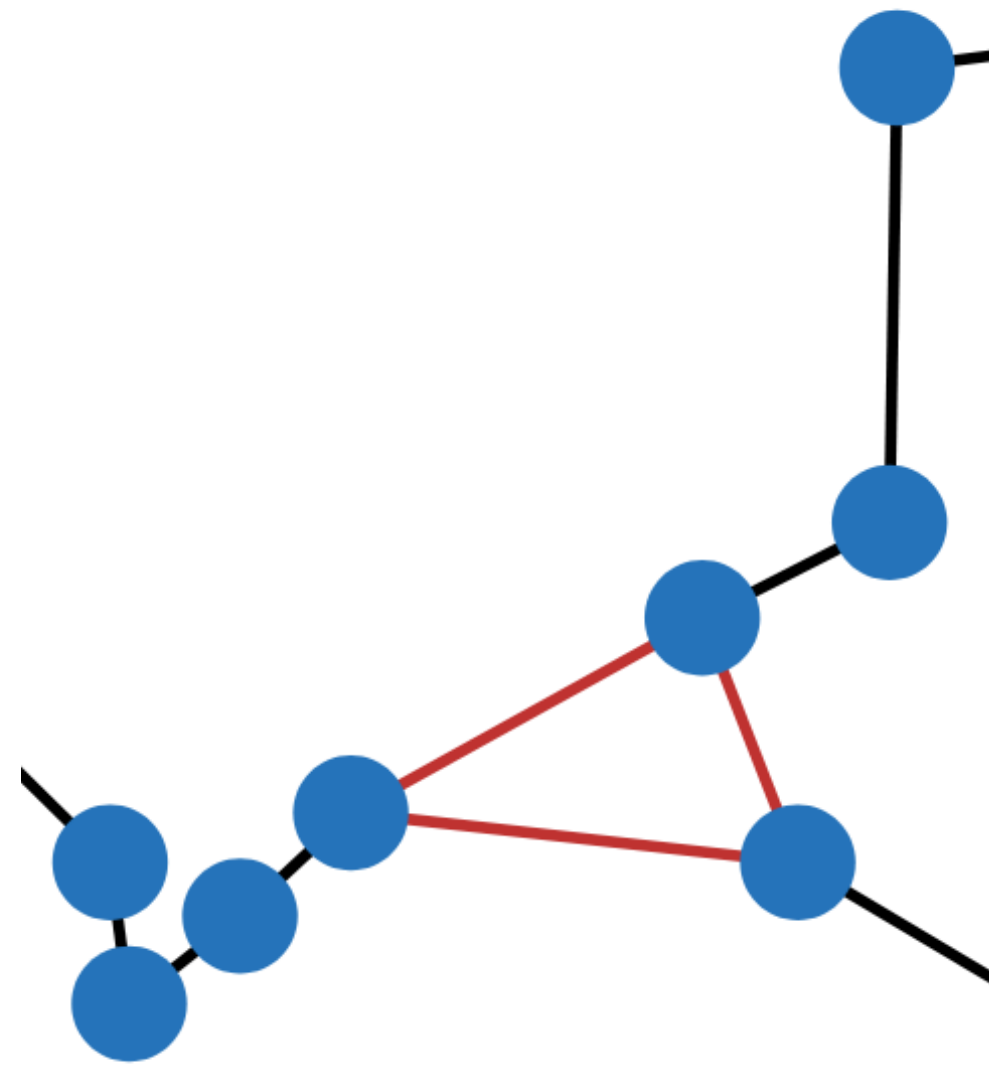


$H, T_1, \dots, T_k, k \geq 3, k \text{ odd}$

$$T_i \cap T_j = \emptyset$$

$$T_i \setminus H \neq \emptyset$$

More Cutting Planes: Combs



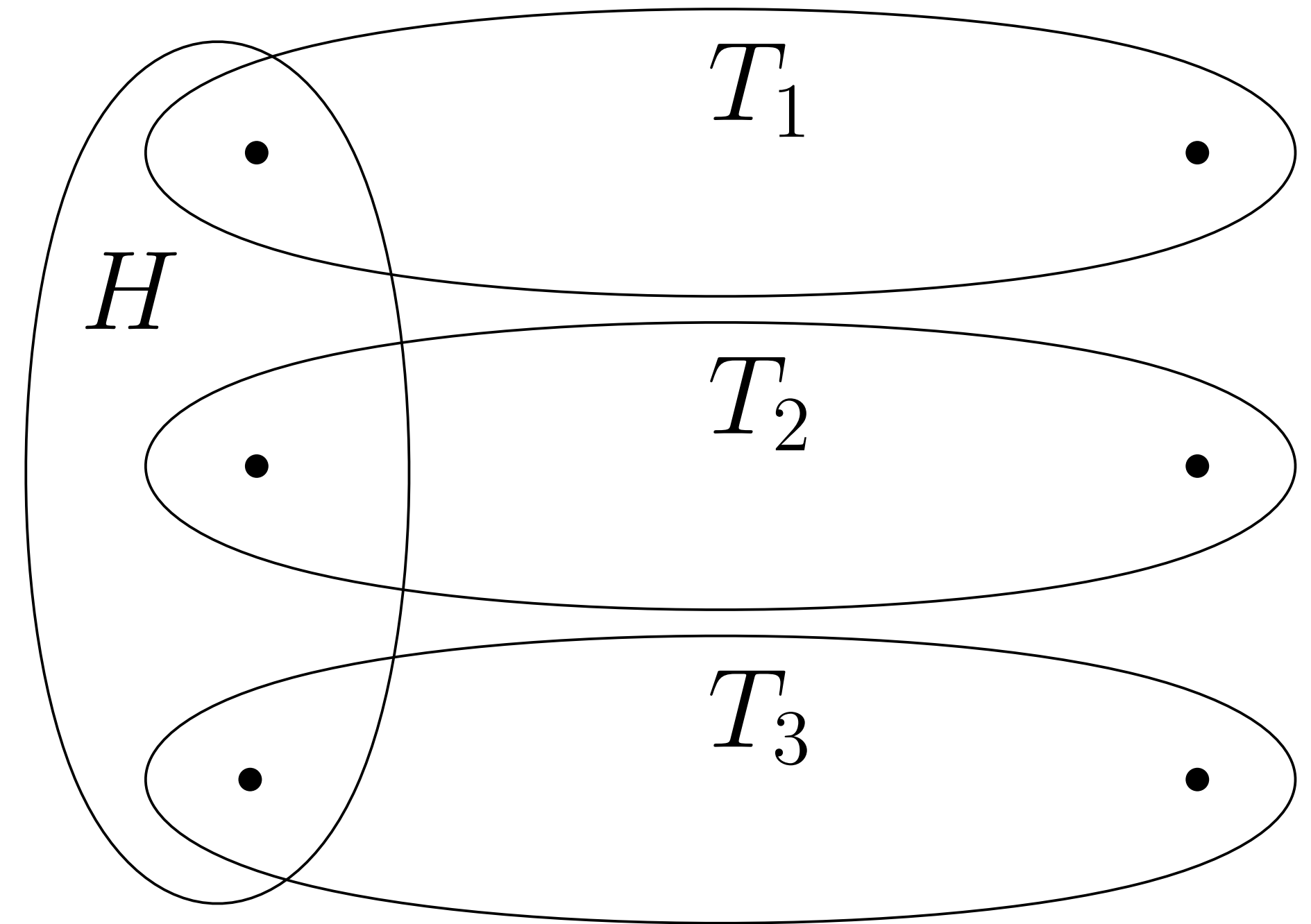
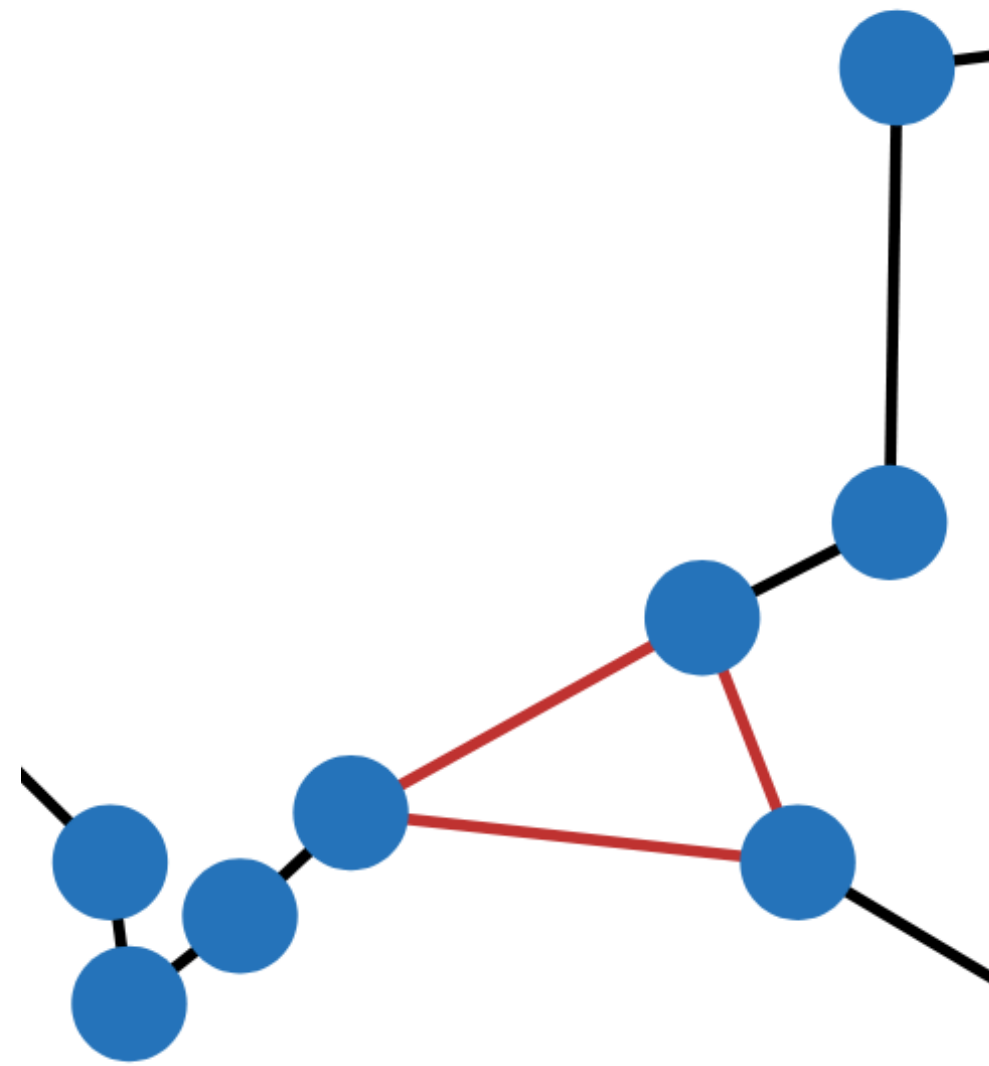
$H, T_1, \dots, T_k, k \geq 3, k \text{ odd}$

$$T_i \cap T_j = \emptyset$$

$$T_i \setminus H \neq \emptyset$$

$$H \cap T_i \neq \emptyset$$

More Cutting Planes: Combs



$H, T_1, \dots, T_k, k \geq 3, k \text{ odd}$

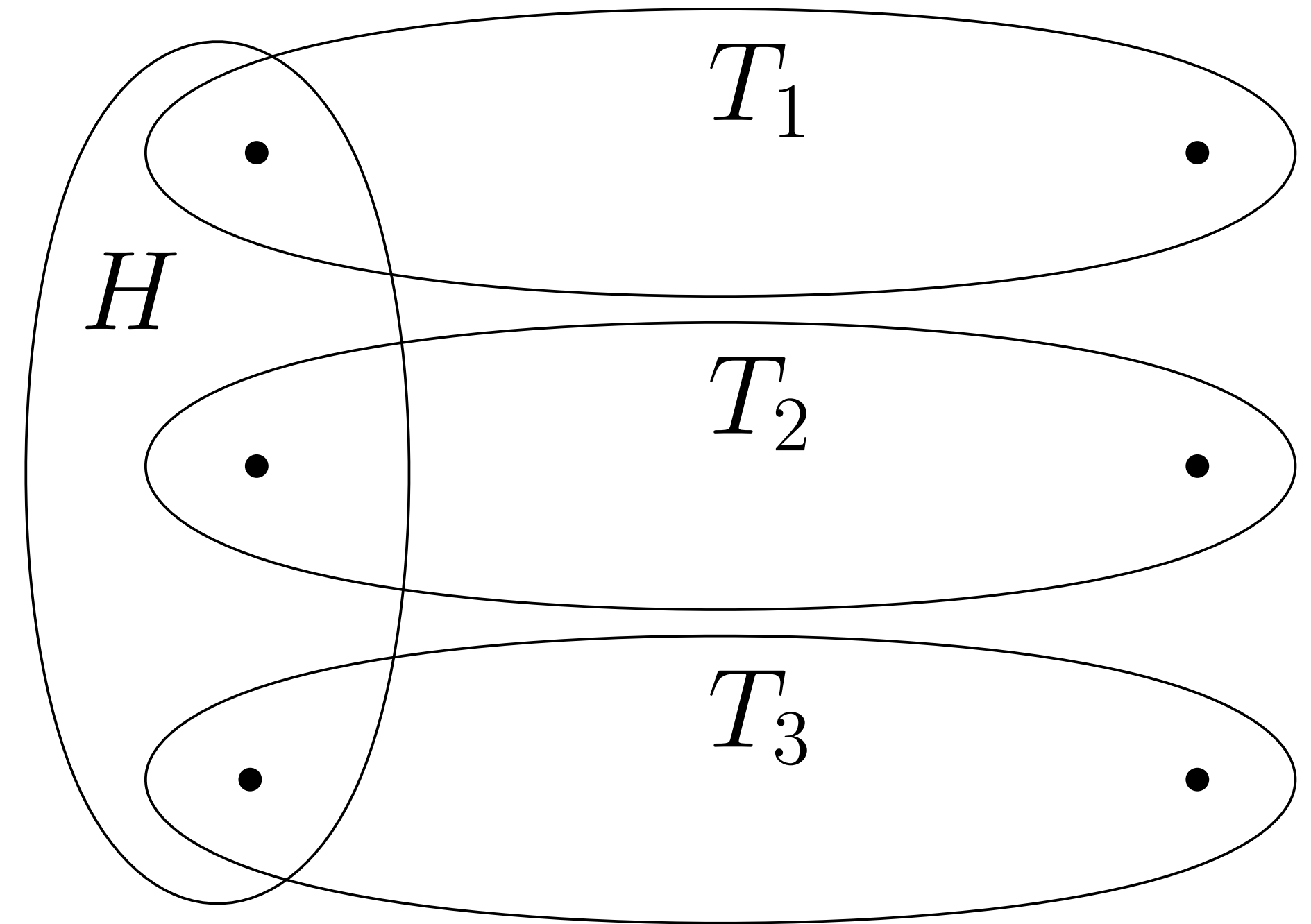
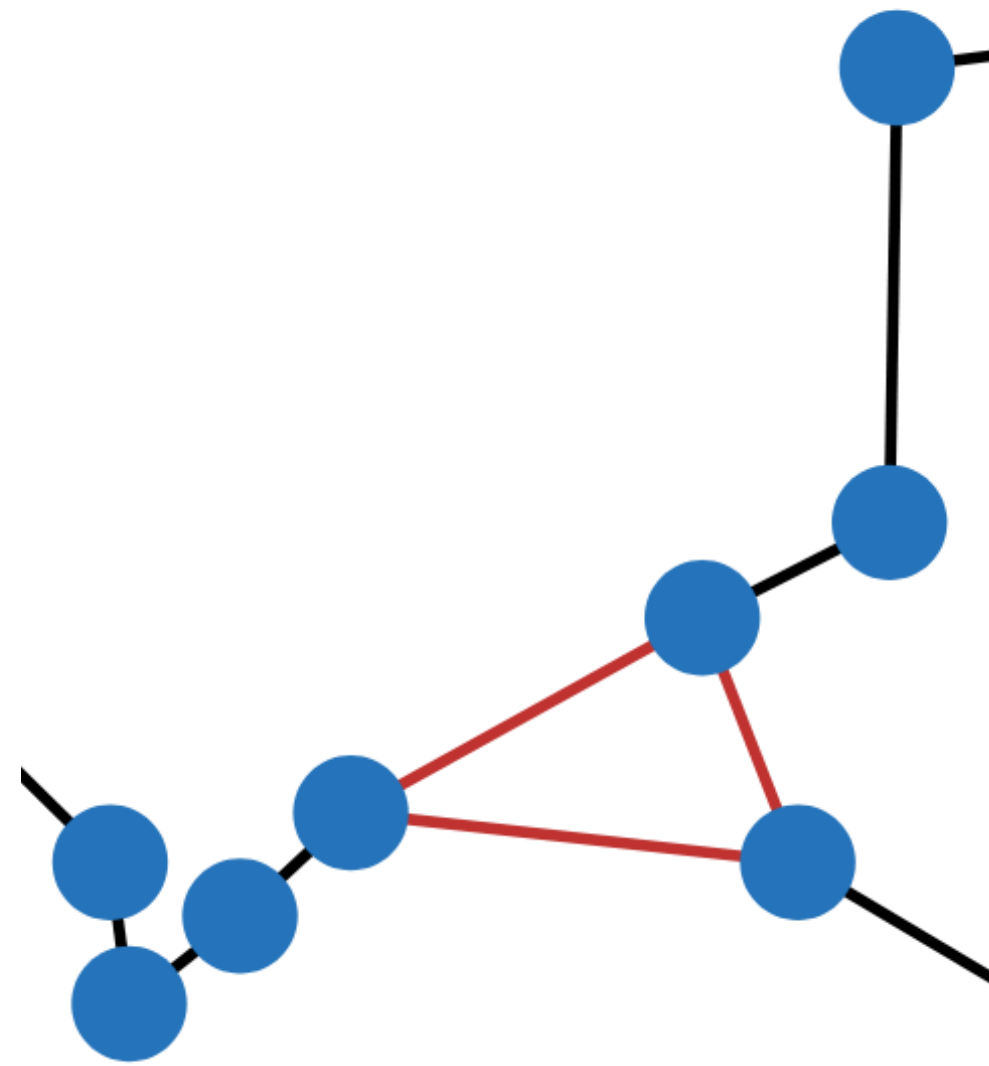
$$T_i \cap T_j = \emptyset$$

$$T_i \setminus H \neq \emptyset$$

$$H \cap T_i \neq \emptyset$$

$$\sum_{e \in \delta(H)} x_e + \sum_{i=1}^k \sum_{e \in \delta(\{T_i\})} x_e \geq ?$$

More Cutting Planes: Combs



$H, T_1, \dots, T_k, k \geq 3, k \text{ odd}$

$$T_i \cap T_j = \emptyset$$

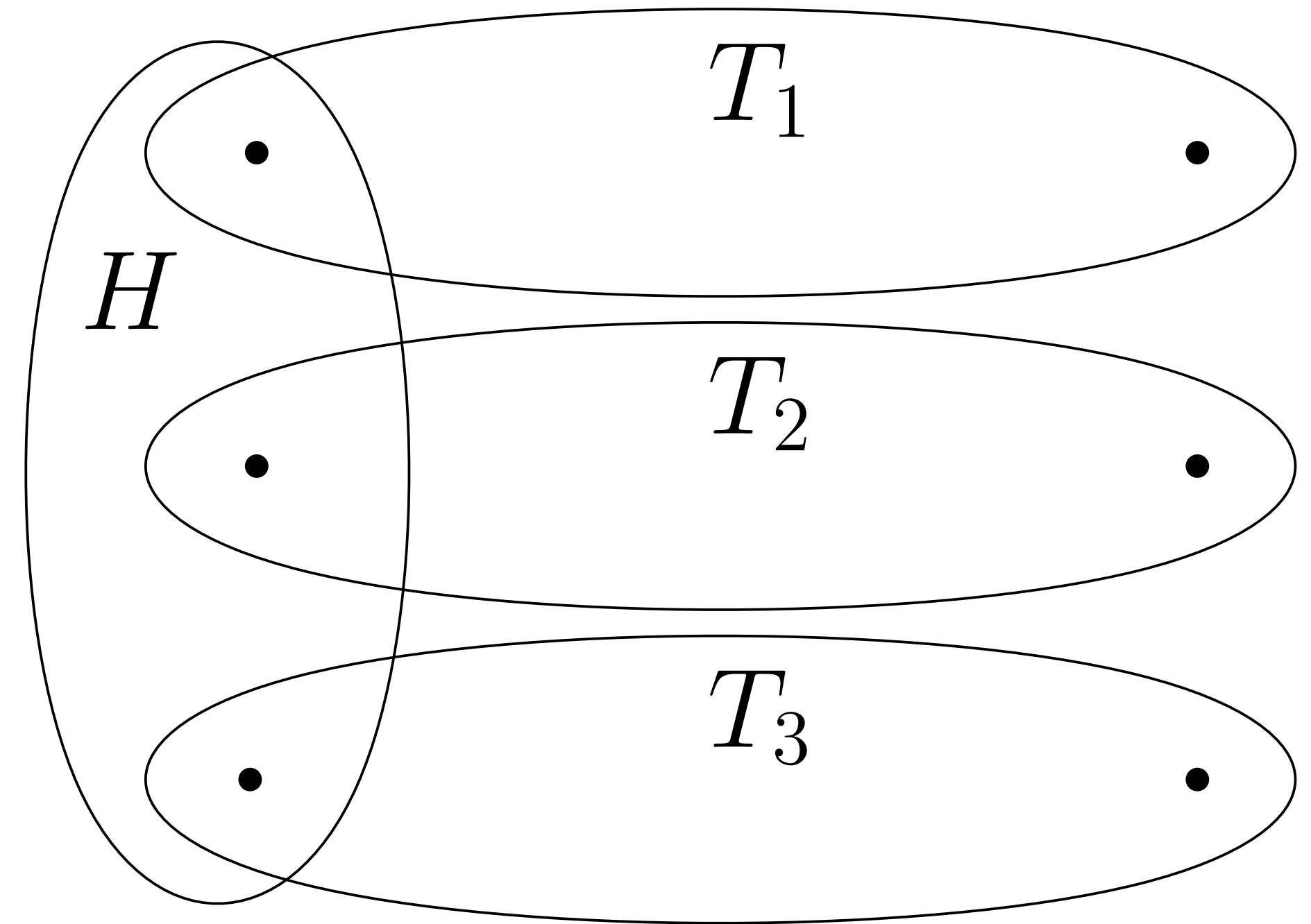
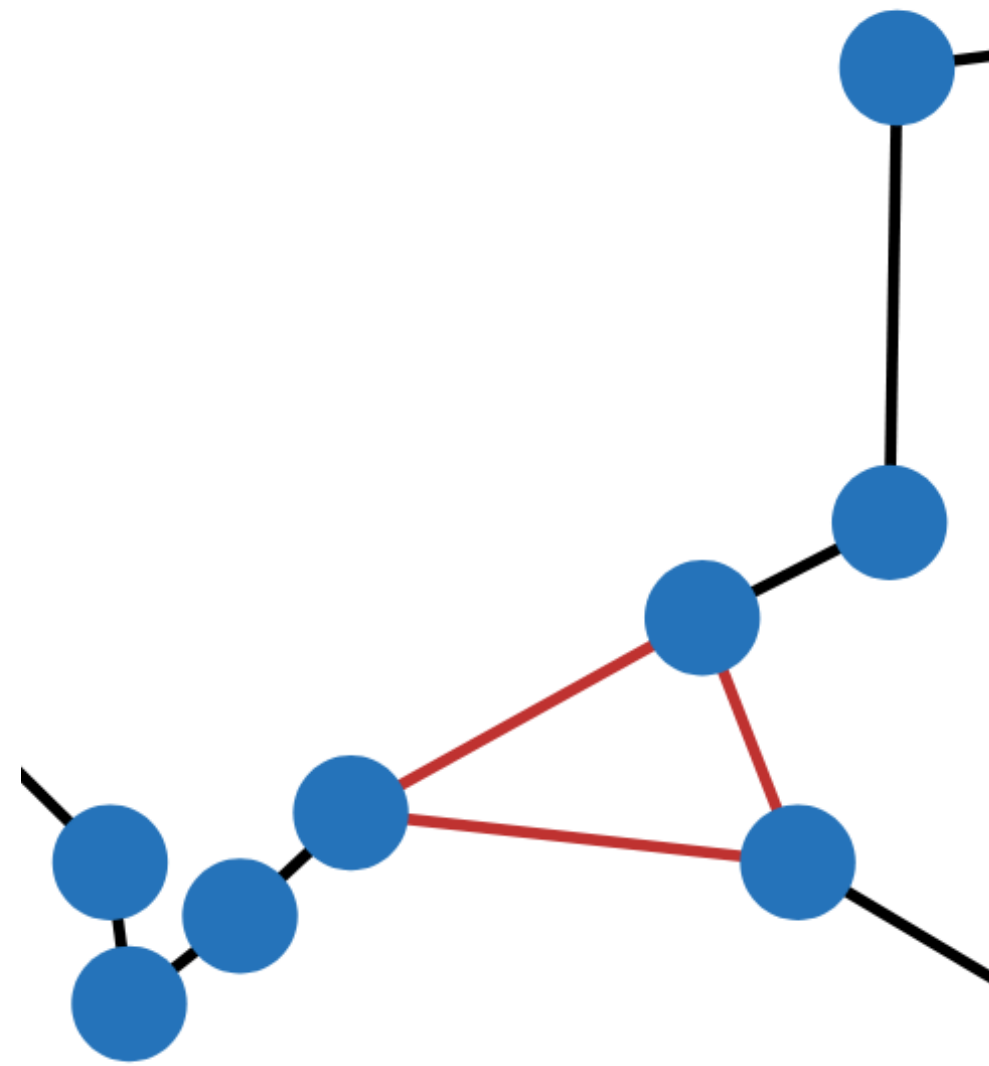
$$T_i \setminus H \neq \emptyset$$

$$H \cap T_i \neq \emptyset$$

$$\sum_{e \in \delta(H)} x_e + \sum_{i=1}^k \sum_{e \in \delta(\{T_i\})} x_e \geq ?$$

How much does each tooth T_i contribute in any tour?

More Cutting Planes: Combs



$H, T_1, \dots, T_k, k \geq 3, k \text{ odd}$

$$T_i \cap T_j = \emptyset$$

$$T_i \setminus H \neq \emptyset$$

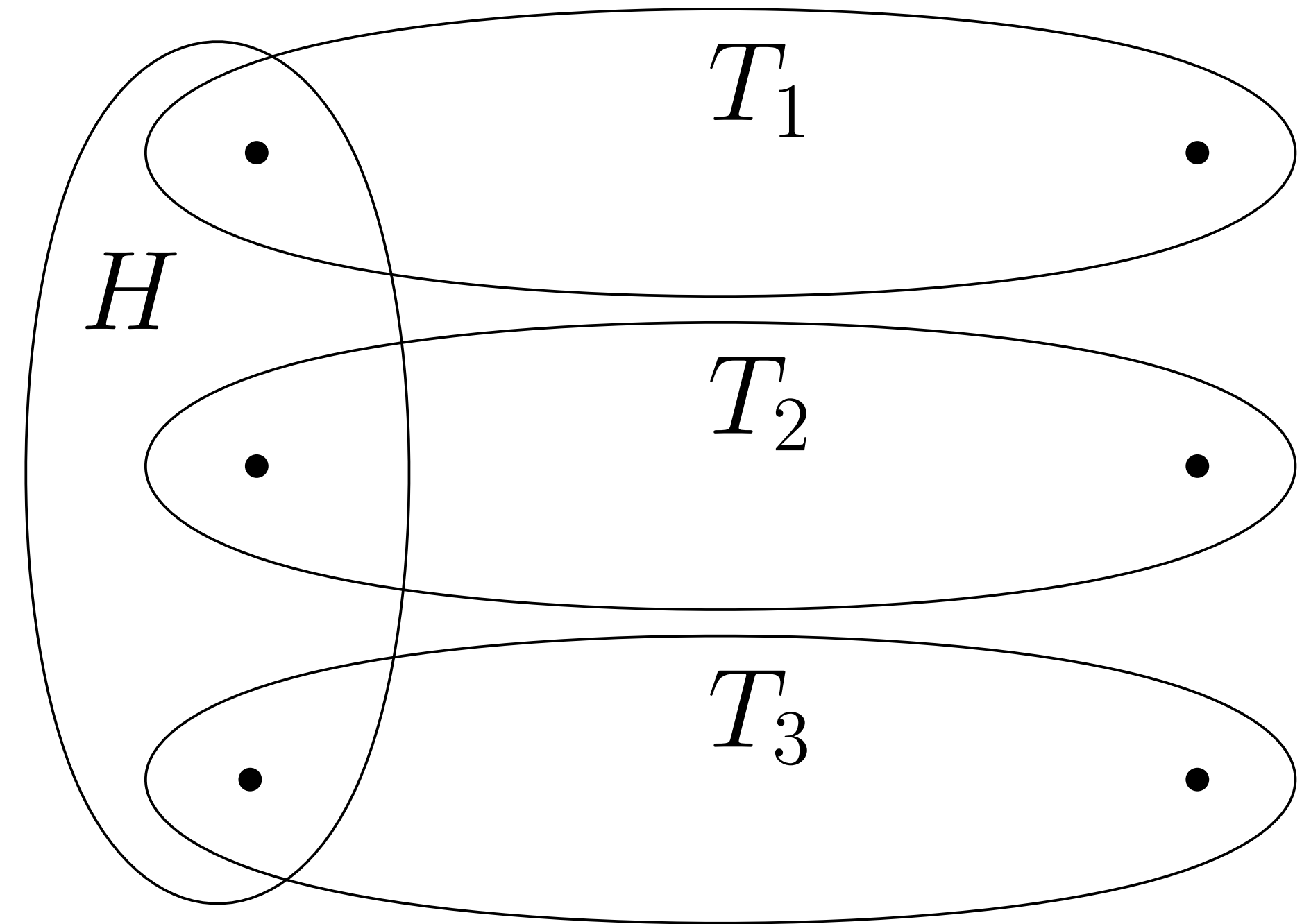
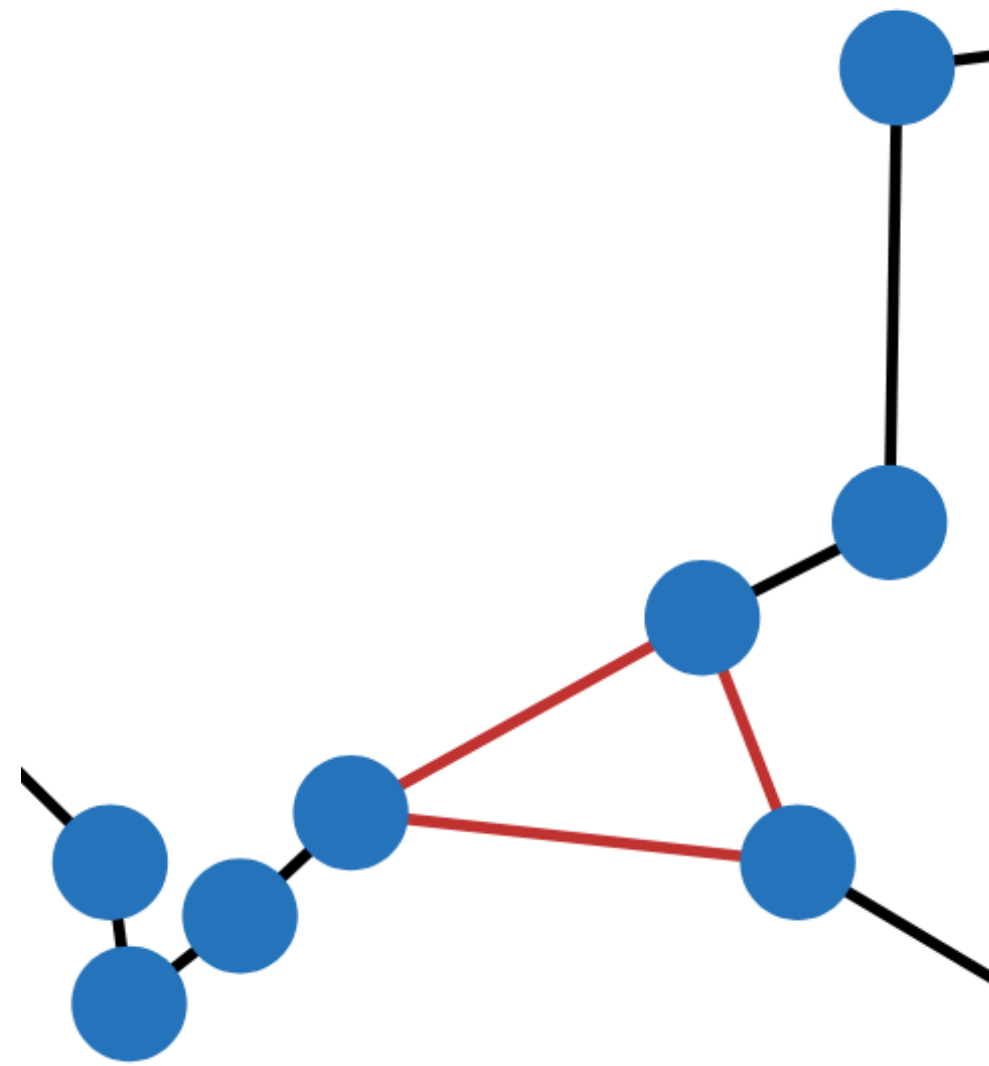
$$H \cap T_i \neq \emptyset$$

$$\sum_{e \in \delta(H)} x_e + \sum_{i=1}^k \sum_{e \in \delta(\{T_i\})} x_e \geq ?$$

How much does each tooth T_i contribute in any tour?

Edge connecting $H \cap T_i$ and $T_i \setminus H$: 3, otherwise: 4

More Cutting Planes: Combs



$H, T_1, \dots, T_k, k \geq 3, k \text{ odd}$

$$T_i \cap T_j = \emptyset$$

$$T_i \setminus H \neq \emptyset$$

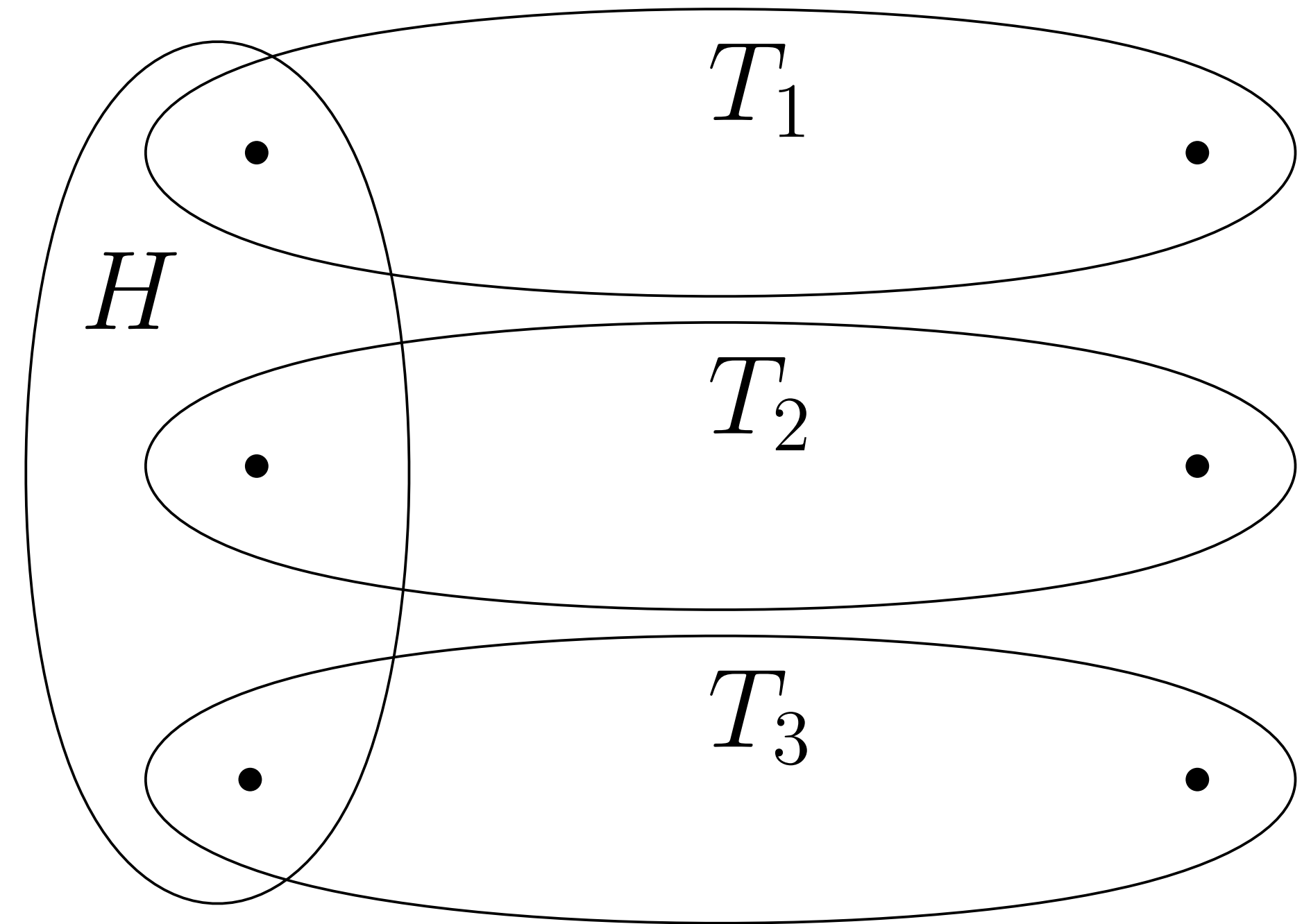
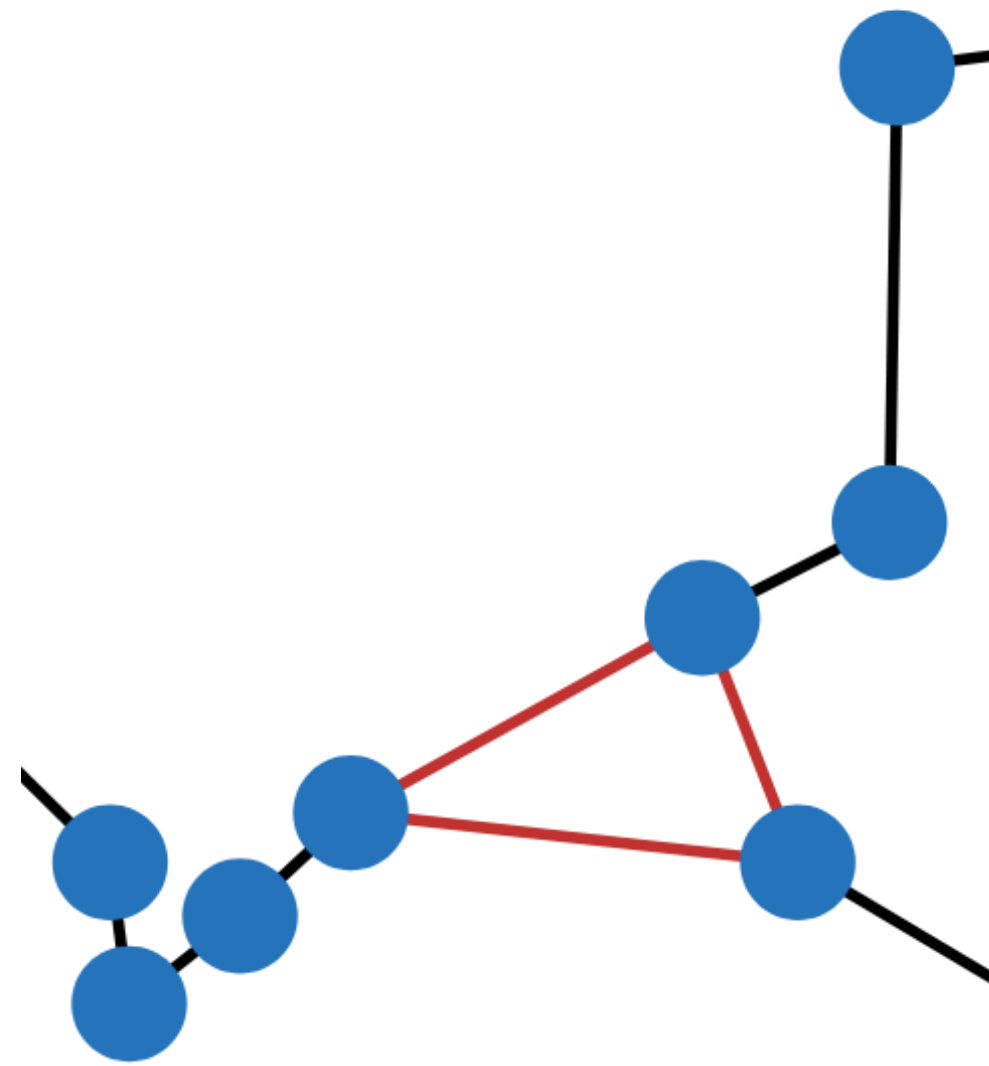
$$H \cap T_i \neq \emptyset$$

$$\sum_{e \in \delta(H)} x_e + \sum_{i=1}^k \sum_{e \in \delta(\{T_i\})} x_e \geq 3k$$

How much does each tooth T_i contribute in any tour?

Edge connecting $H \cap T_i$ and $T_i \setminus H$: 3, otherwise: 4

More Cutting Planes: Combs



$H, T_1, \dots, T_k, k \geq 3, k \text{ odd}$

$$T_i \cap T_j = \emptyset$$

$$T_i \setminus H \neq \emptyset$$

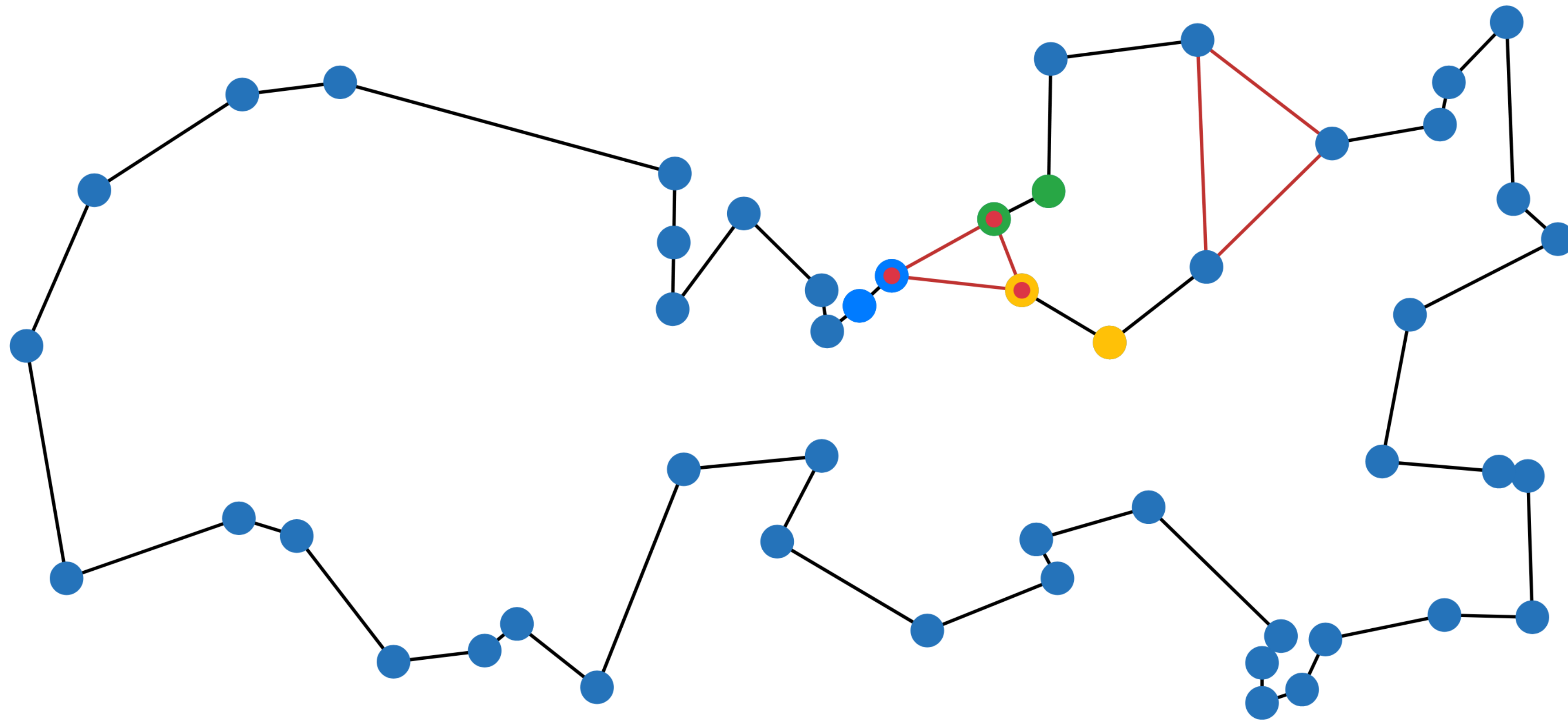
$$H \cap T_i \neq \emptyset$$

$$\sum_{e \in \delta(H)} x_e + \sum_{i=1}^k \sum_{e \in \delta(\{T_i\})} x_e \geq 3k + 1$$

How much does each tooth T_i contribute in any tour?

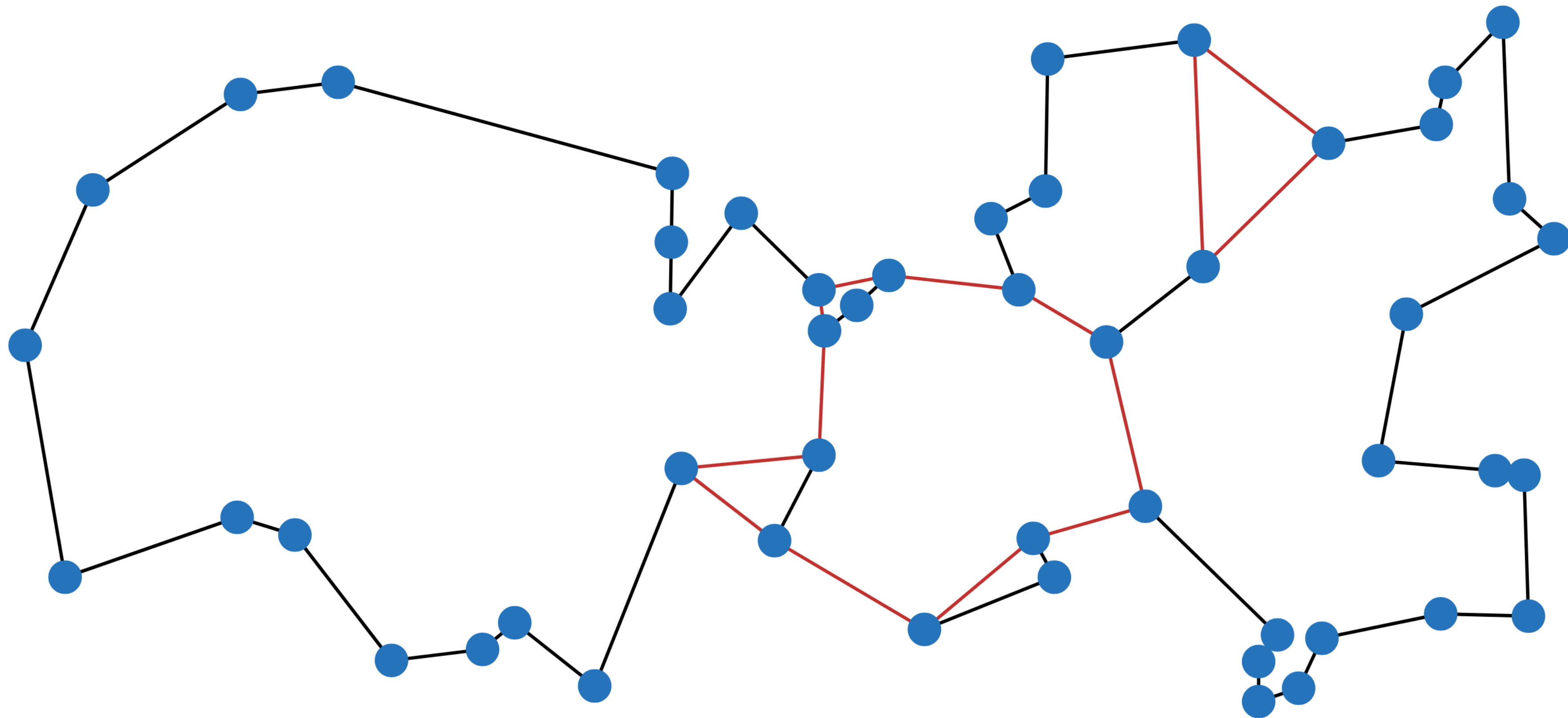
Edge connecting $H \cap T_i$ and $T_i \setminus H$: 3, otherwise: 4

Add Comb Inequality



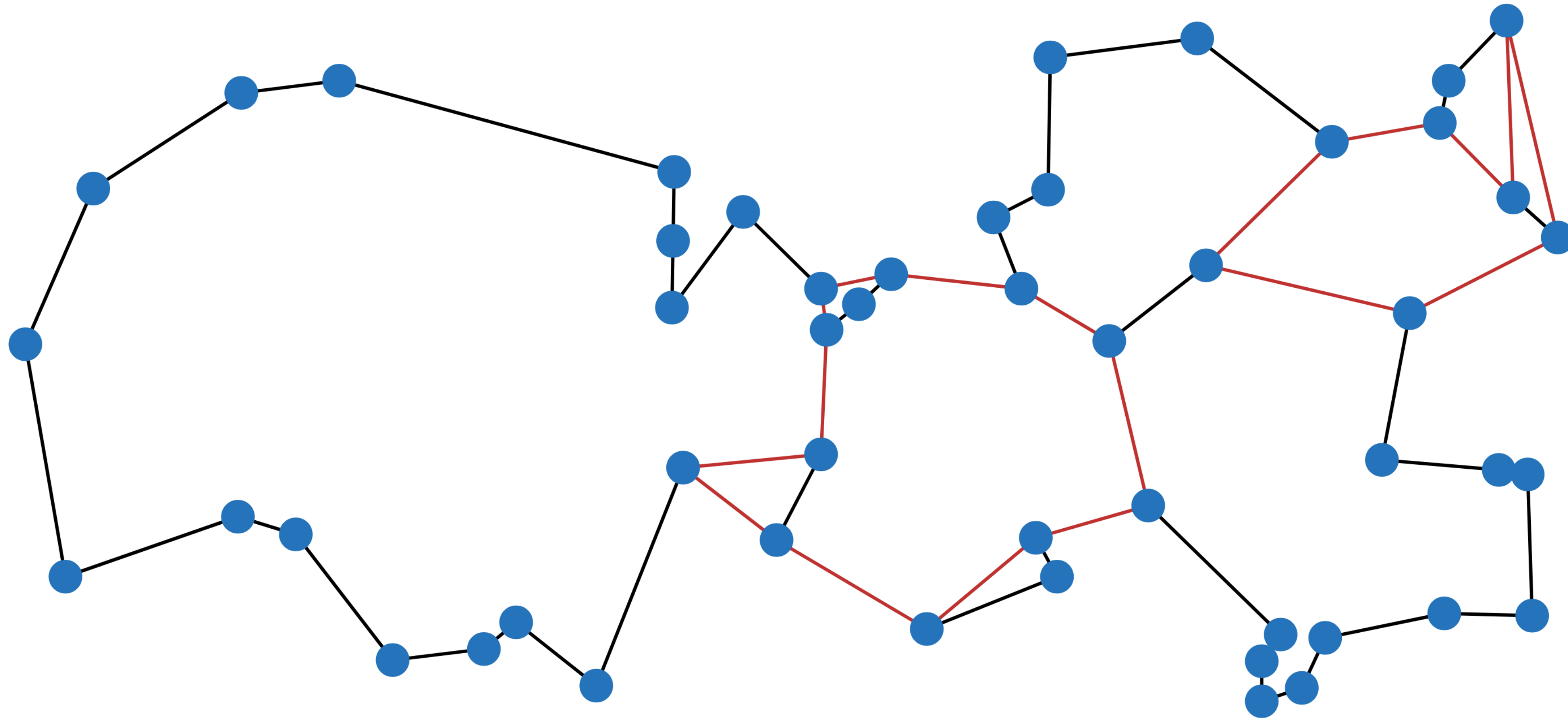
LP relaxation: 51766, best UB: 54277.

Add Comb Inequality



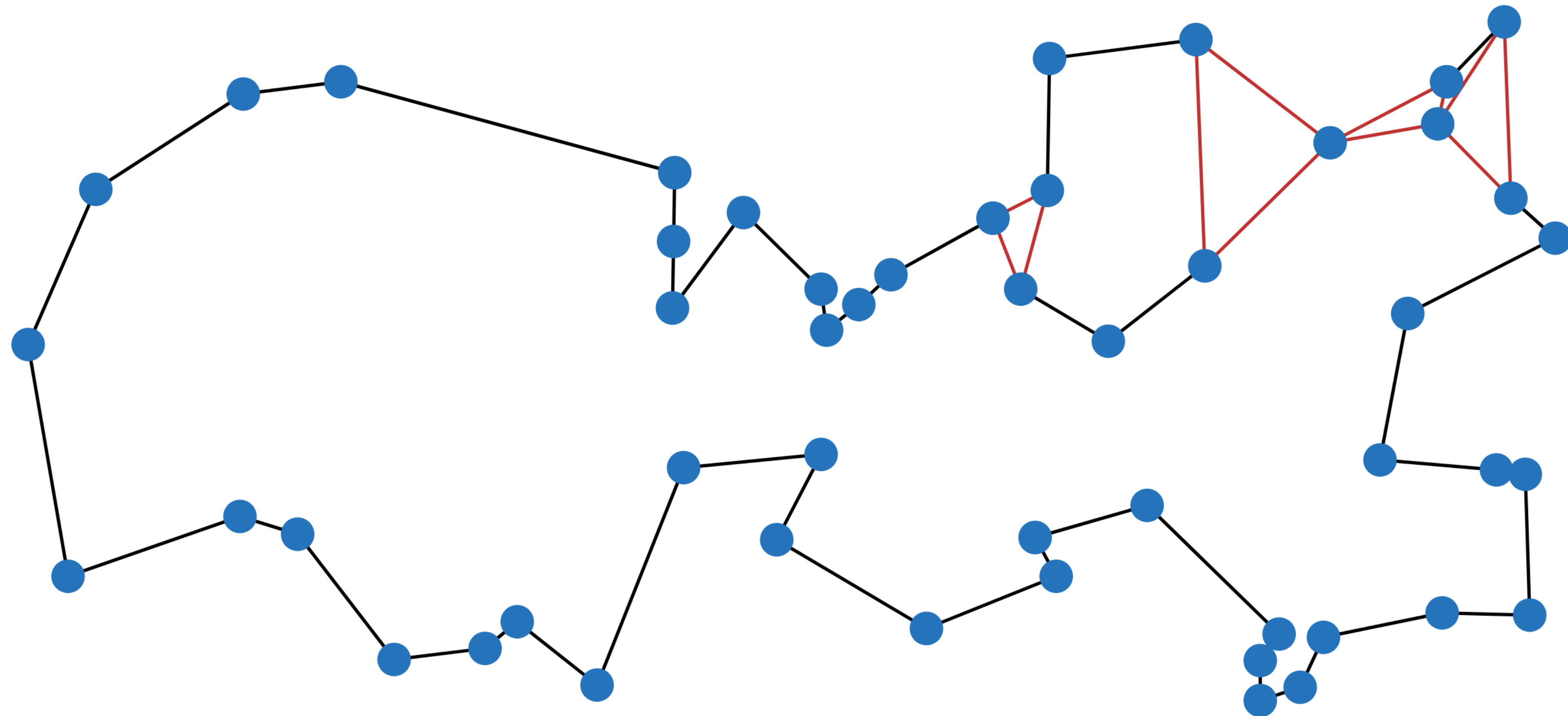
LP relaxation with comb: 51817 (+51), best UB: 54277.

Add Another Comb Inequality



LP relaxation with 2 combs: 51936 (+119), best UB: 54277.

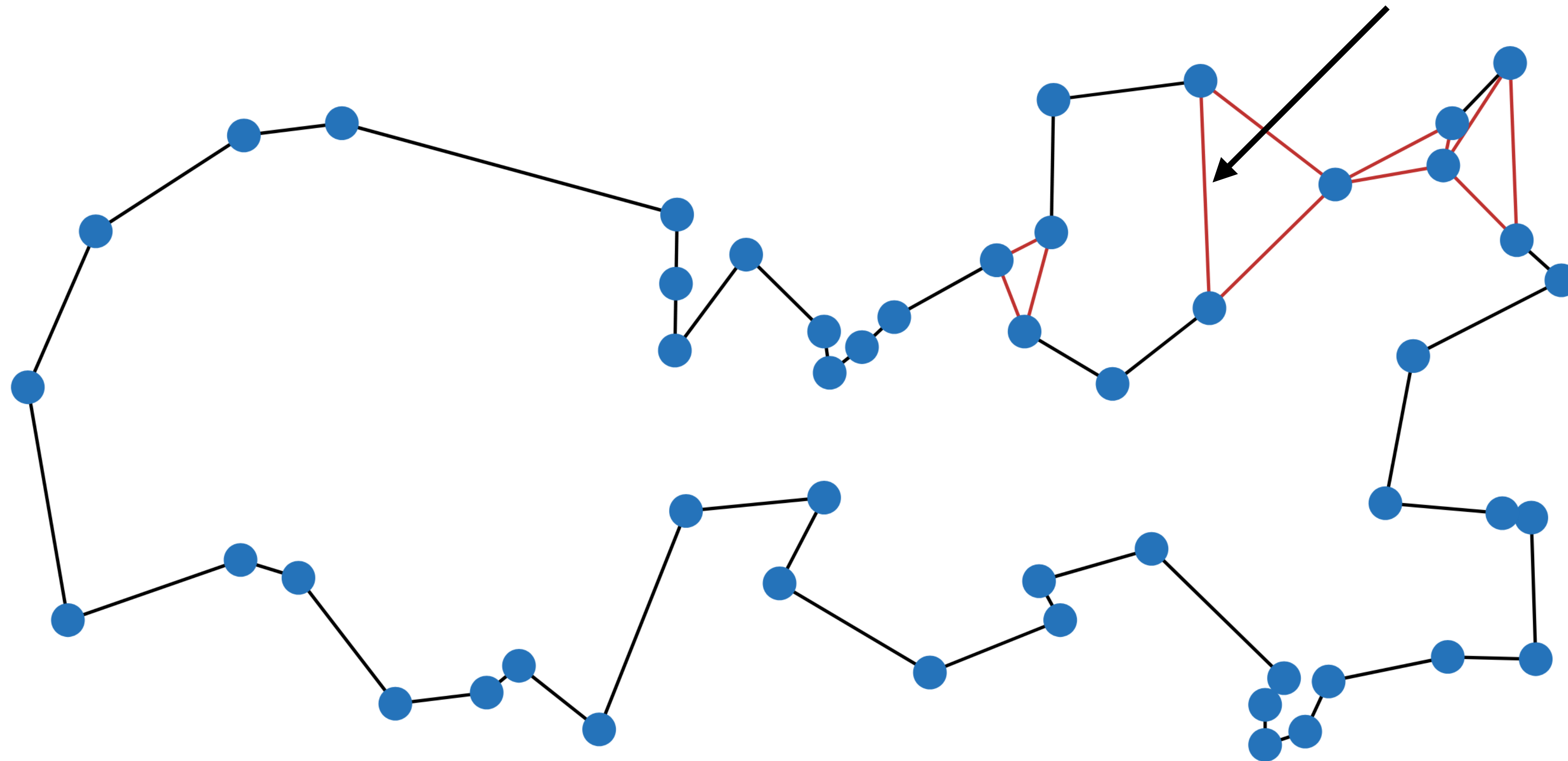
Add Subtour Inequality



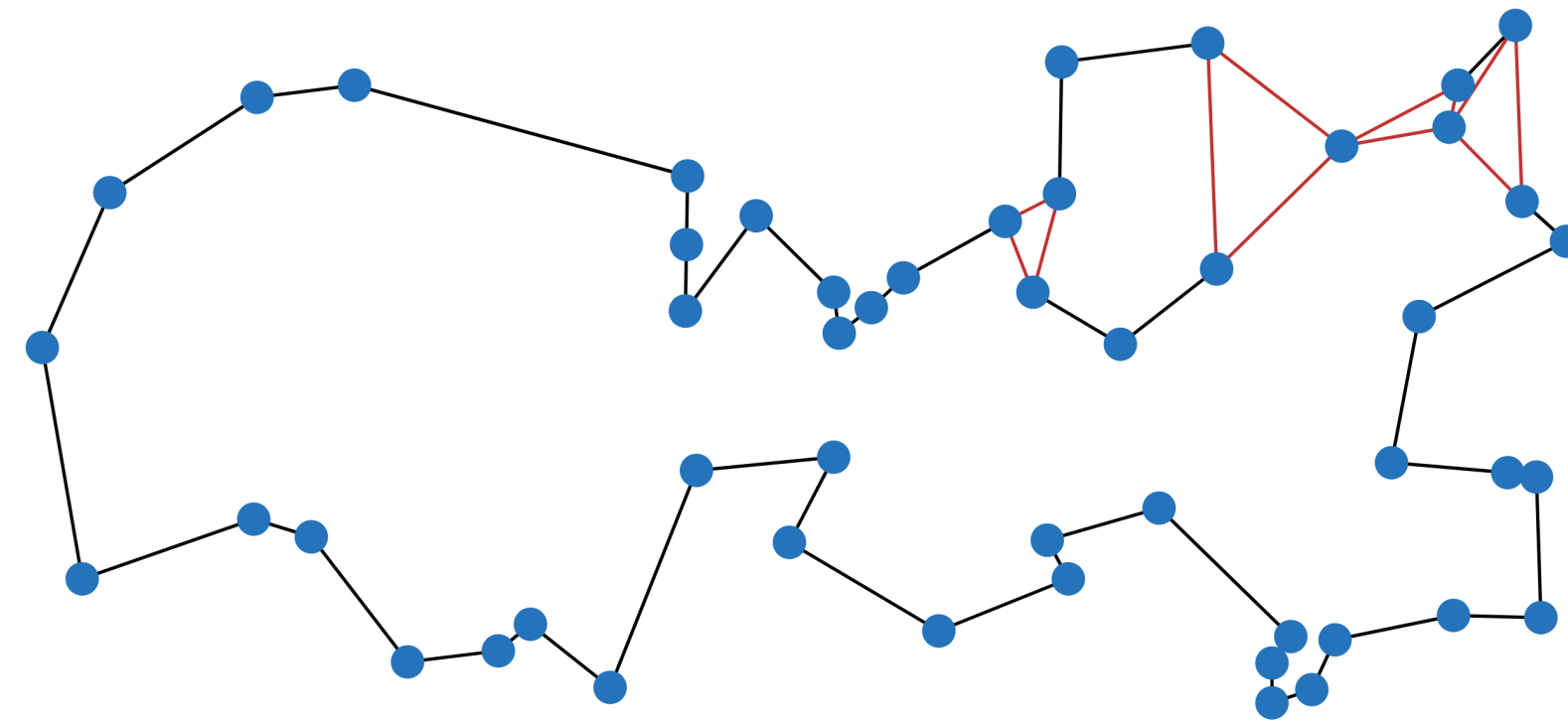
New LP relaxation: 52019 (+83), best UB: 54277.

Let's Branch

Branch variable

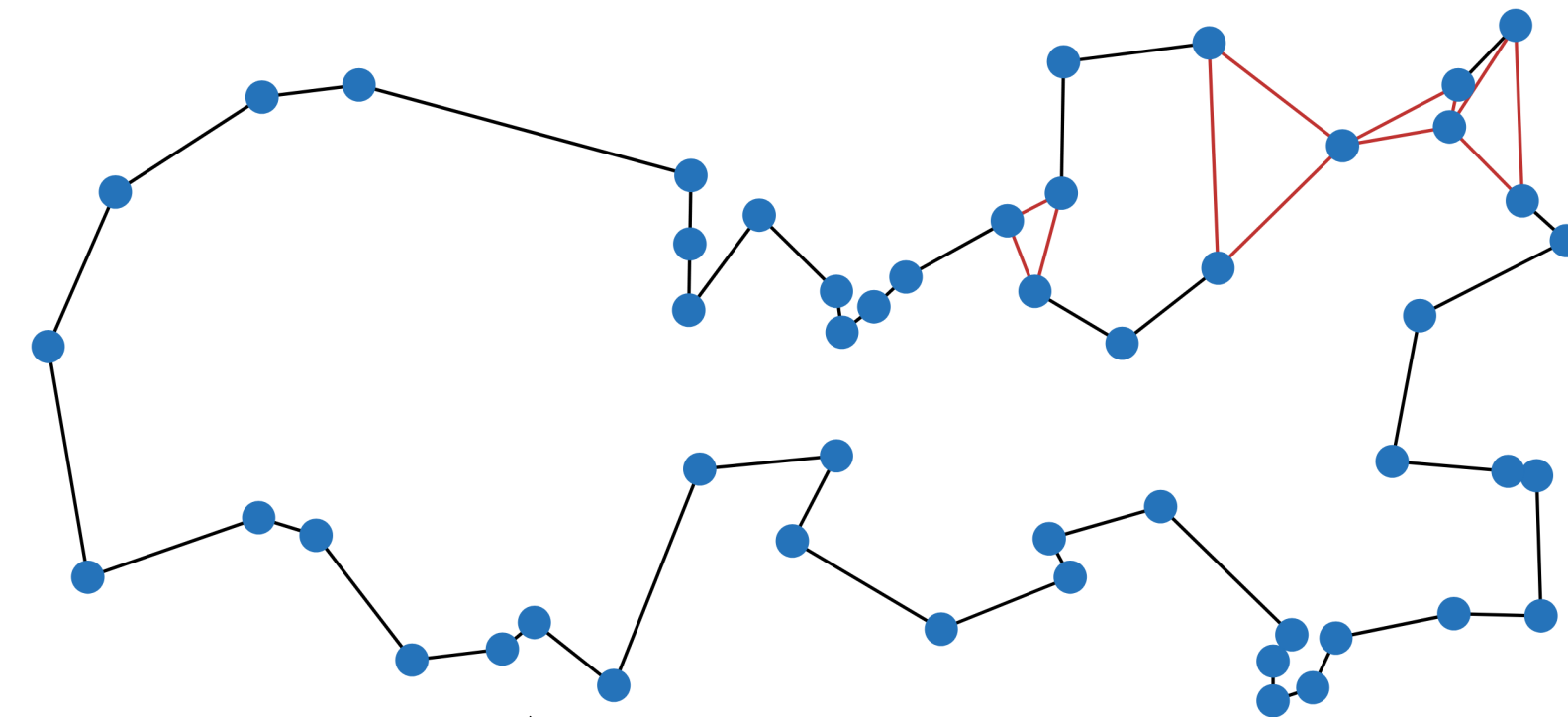


Let's Branch

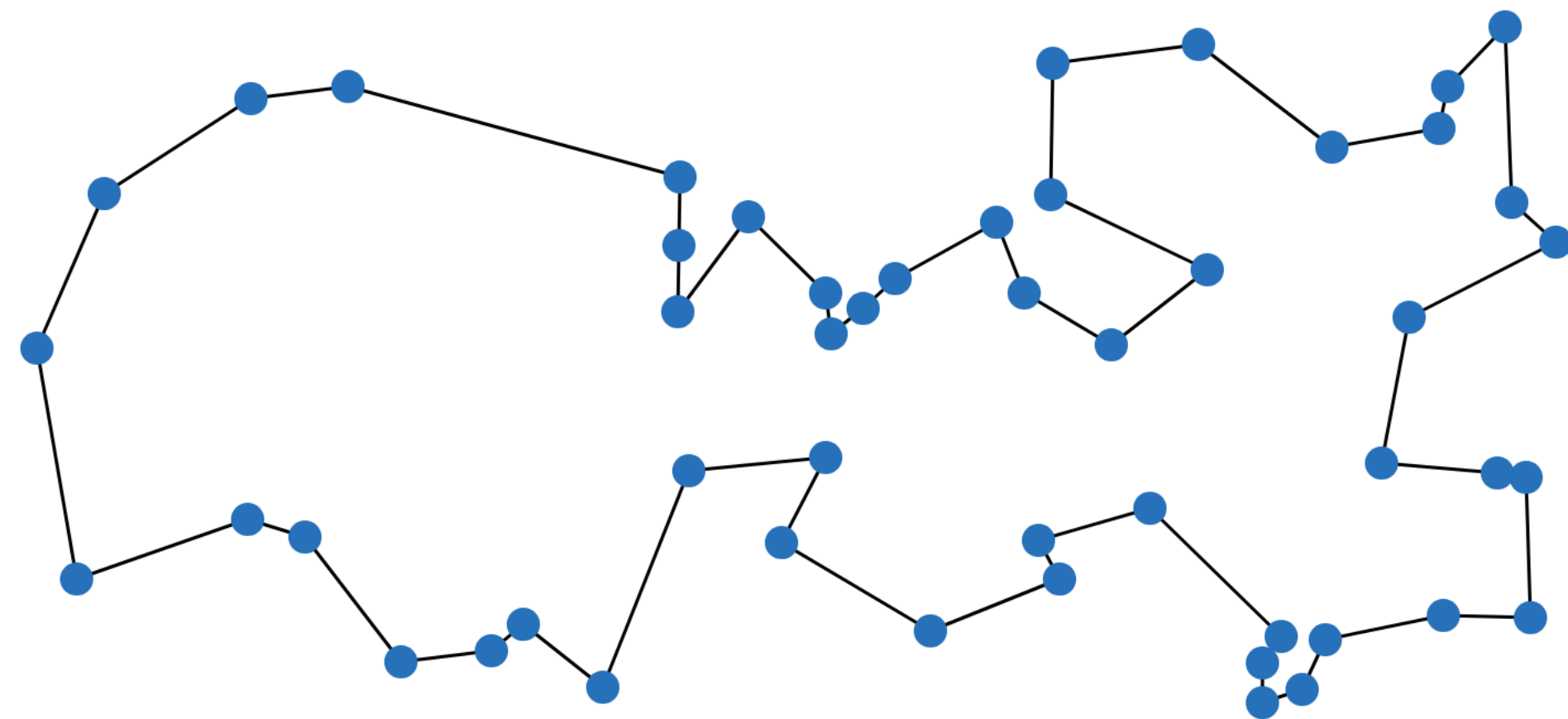
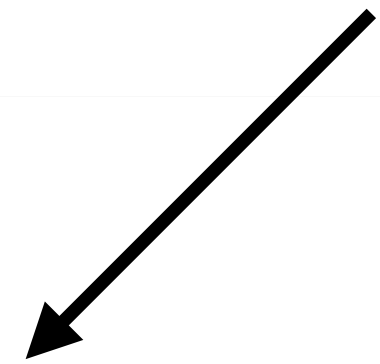


52019 (UB: 54277)

Let's Branch

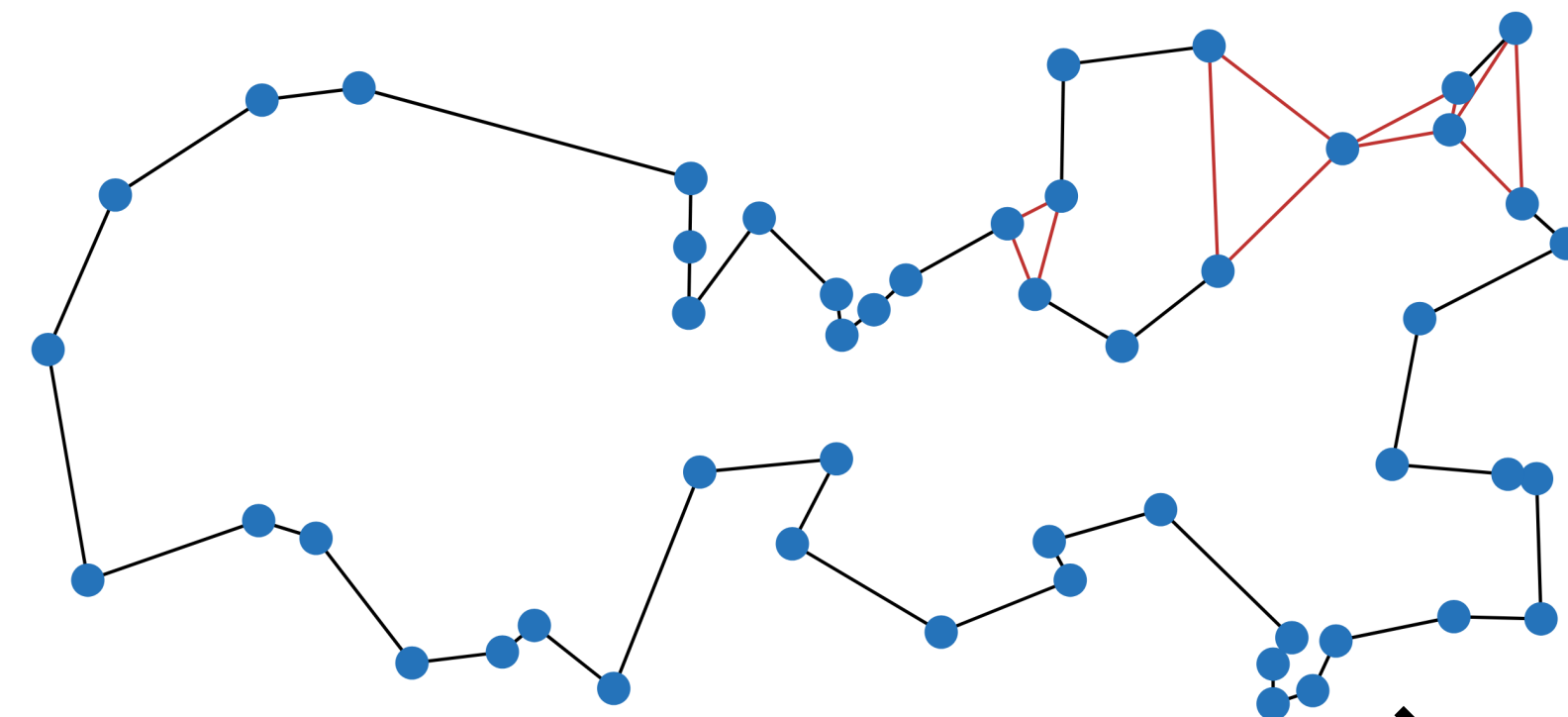


52019 (UB: 54277)

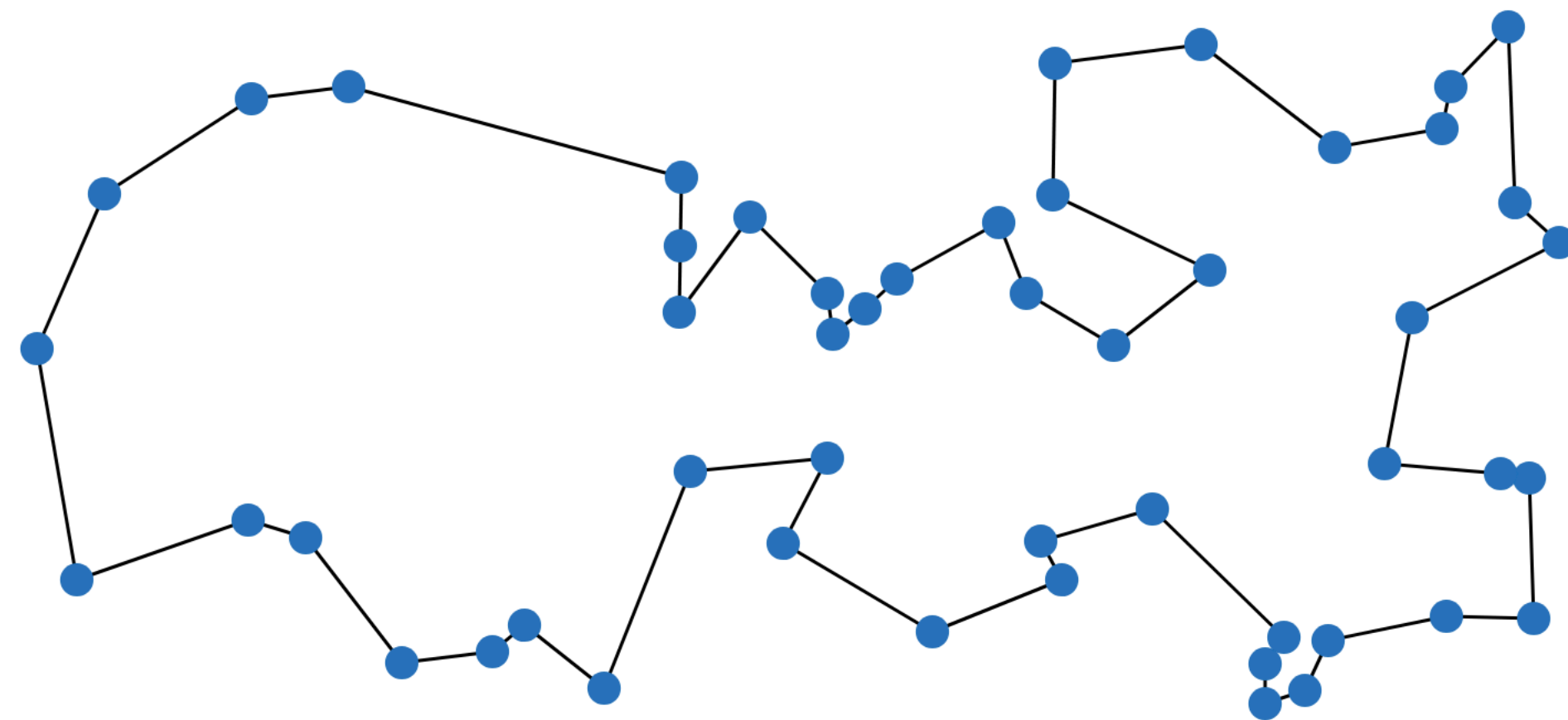
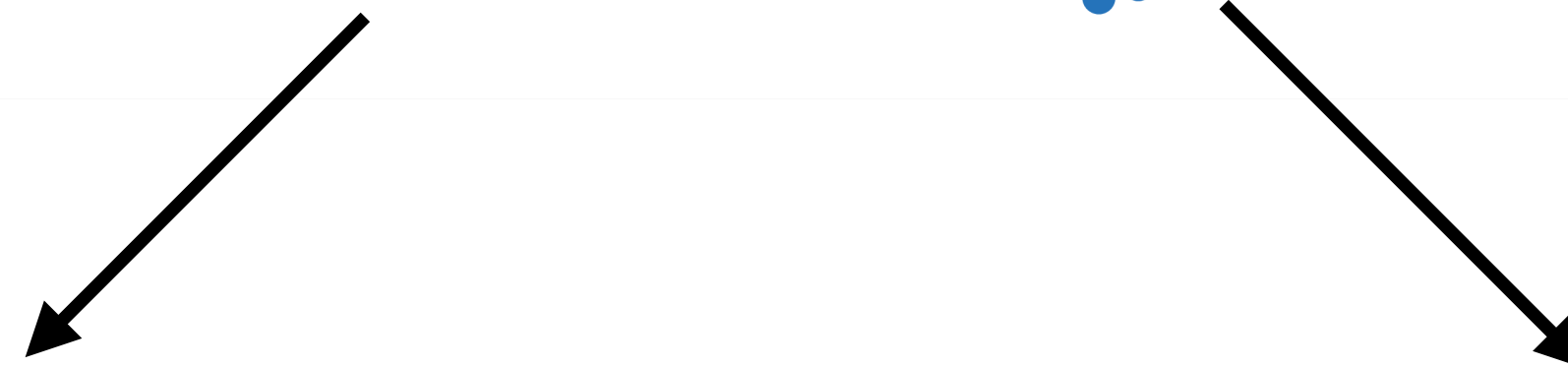


52021 (UB: 52021)

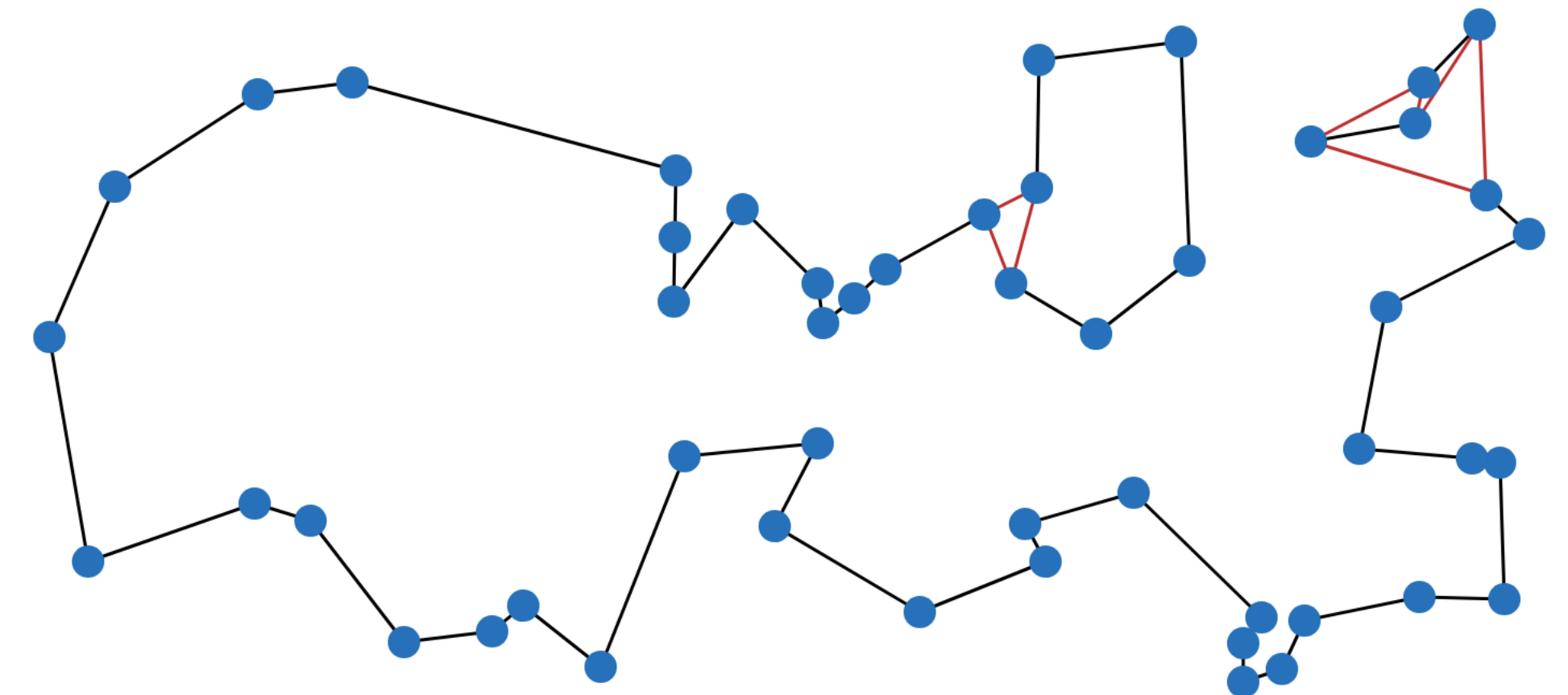
Let's Branch



52019 (UB: 54277)

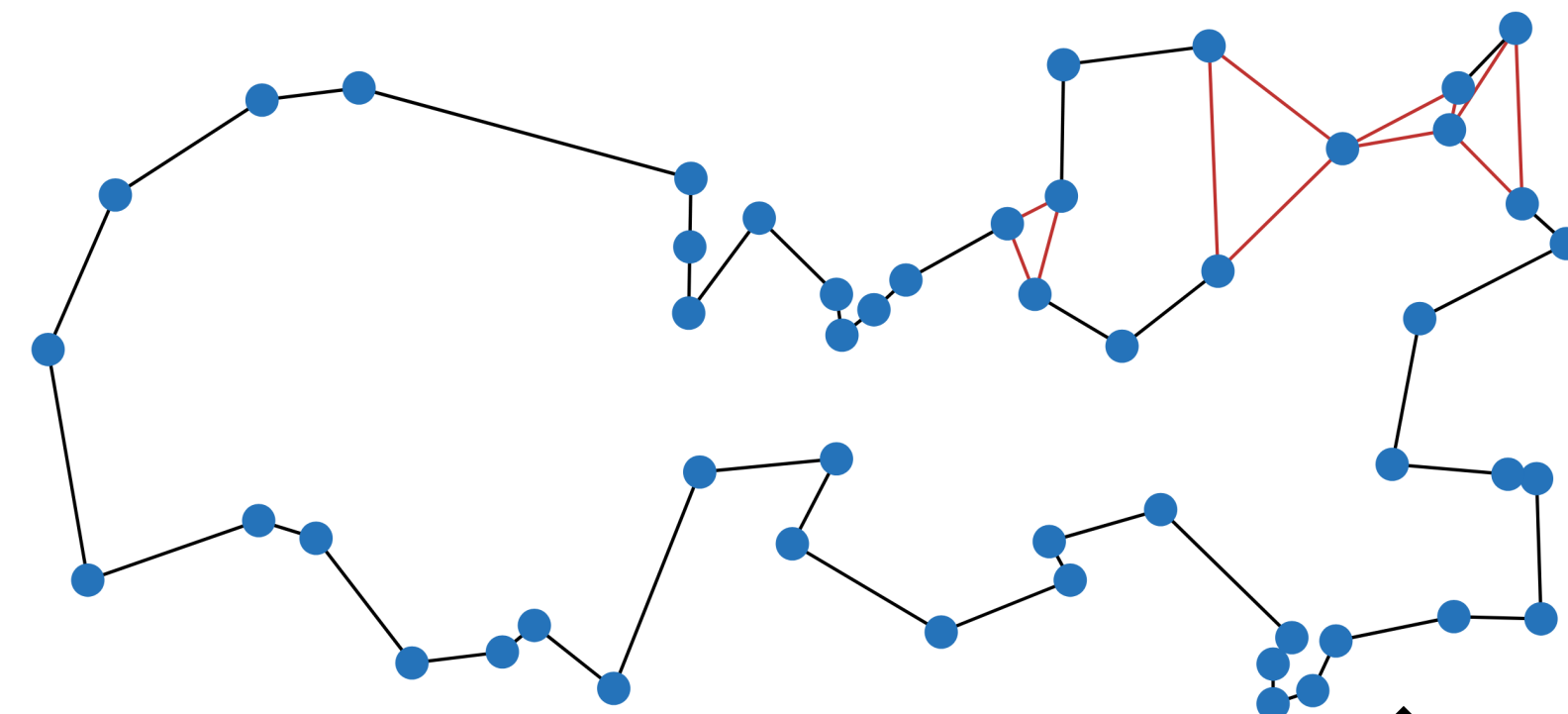


52021 (UB: 52021)

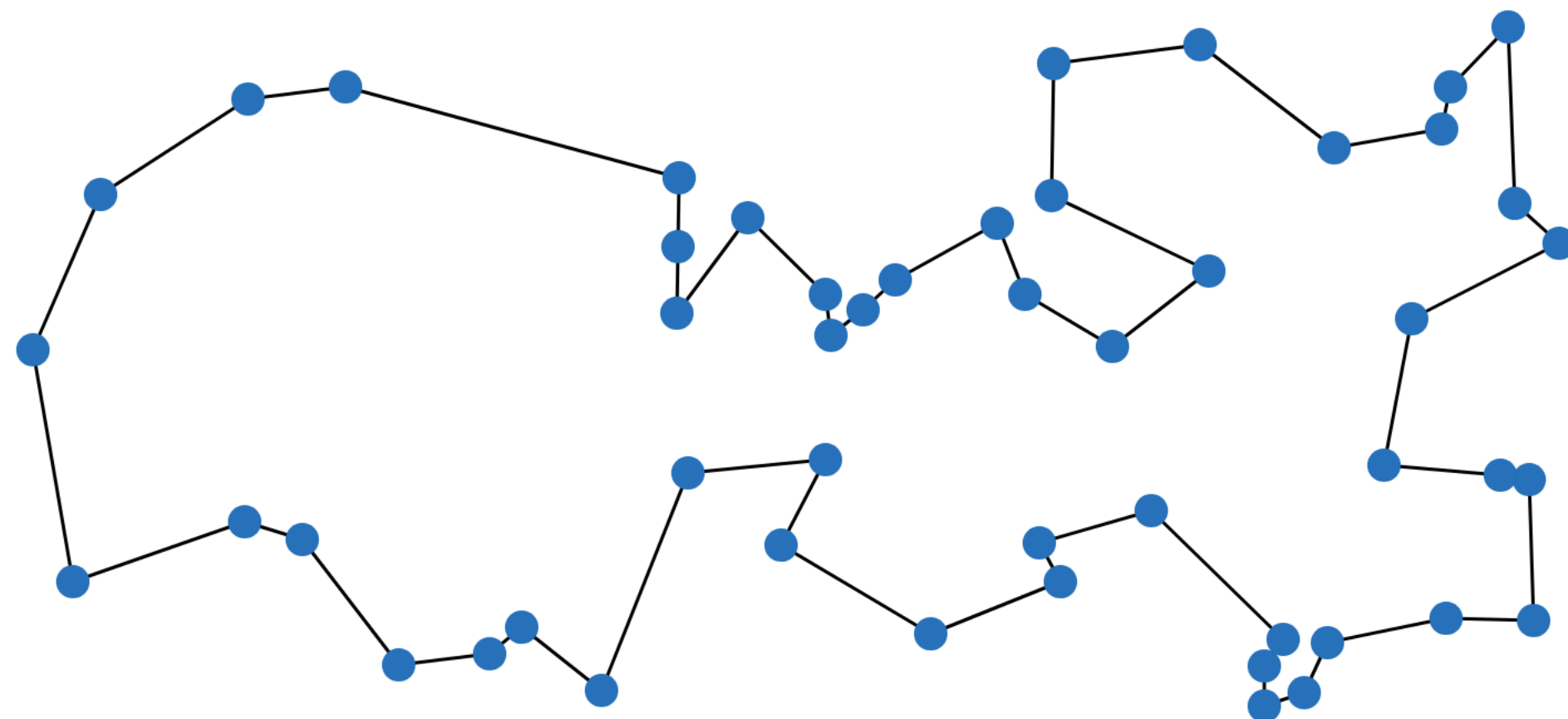
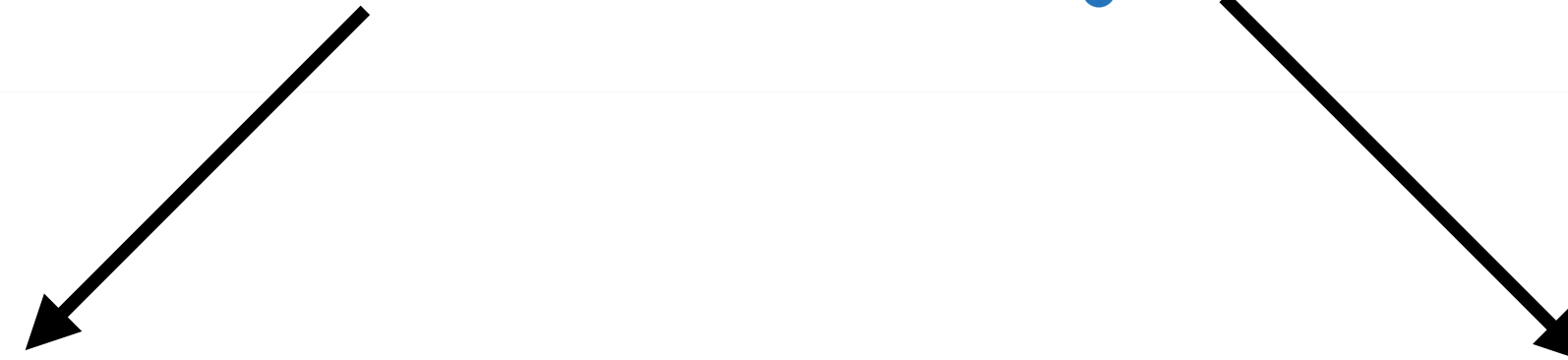


52360 (UB: 52021, pruned)

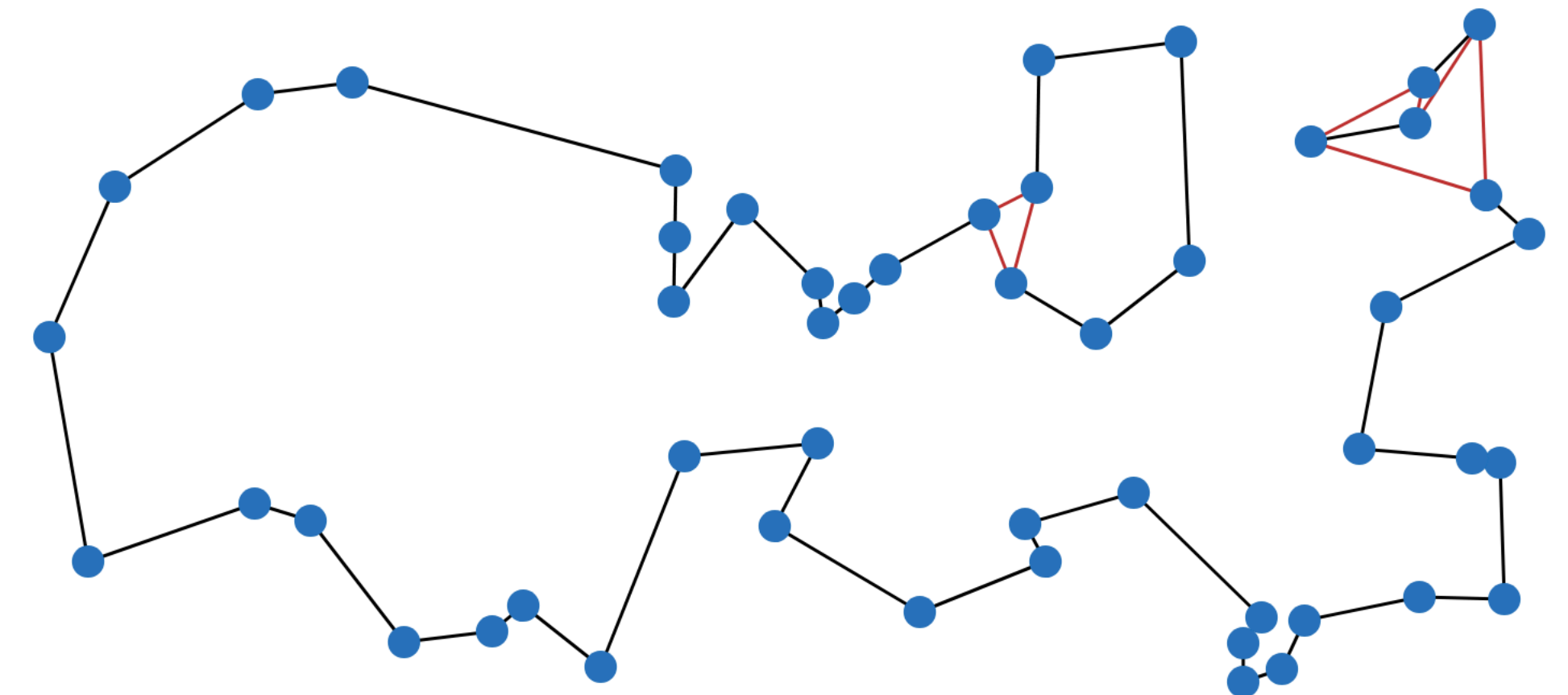
Let's Branch



52019 (UB: 54277)



52021 (UB: 52021)
Optimal!



52360 (UB: 52021, pruned)

How can we bound LP solution values? First, let's simplify.

General LP:

$$\begin{aligned} \max \quad & \sum_i c_i x_i \text{ s.t.} \\ & 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ & \ell_i \leq x_i \leq u_i \end{aligned}$$

How can we bound LP solution values? First, let's simplify.

General LP:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : d_j \leq \sum_i a_{ji} x_i \leq b_j \\ \ell_i \leq x_i \leq u_i \end{aligned}$$

Standard form:

$$\begin{aligned} \max \sum_i c_i x_i \text{ s.t.} \\ 1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j \\ x_i \geq 0 \end{aligned}$$

Standard form:

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$$x_i \geq 0$$

Standard form:

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$$x_i \geq 0$$

We want: $\sum_{i=1}^n c_i x_i \leq \xi$; how can we do that?

Standard form:

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$

$$x_i \geq 0$$

We want: $\sum_{i=1}^n c_i x_i \leq \xi$; how can we do that?

Adding up multiples of constraints ($y_i \geq 0$):

Standard form:

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$
$$x_i \geq 0$$

We want: $\sum_{i=1}^n c_i x_i \leq \xi$; how can we do that?

Adding up multiples of constraints ($y_i \geq 0$):

$$y_1 \cdot \left(\sum_i a_{1i} x_i \right) \leq y_1 b_1$$

Standard form:

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$
$$x_i \geq 0$$

We want: $\sum_{i=1}^n c_i x_i \leq \xi$; how can we do that?

Adding up multiples of constraints ($y_i \geq 0$):

$$y_1 \cdot \left(\sum_i a_{1i} x_i \right) \leq y_1 b_1$$
$$+ y_2 \cdot \left(\sum_i a_{2i} x_i \right) \leq y_2 b_2$$

Standard form:

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$
$$x_i \geq 0$$

We want: $\sum_{i=1}^n c_i x_i \leq \xi$; how can we do that?

Adding up multiples of constraints ($y_i \geq 0$):

$$y_1 \cdot \left(\sum_i a_{1i} x_i \right) \leq y_1 b_1$$
$$+ y_2 \cdot \left(\sum_i a_{2i} x_i \right) \leq y_2 b_2$$
$$+ \dots$$
$$+ y_m \cdot \left(\sum_i a_{mi} x_i \right) \leq y_m b_m$$

Standard form:

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$
$$x_i \geq 0$$

We want: $\sum_{i=1}^n c_i x_i \leq \xi$; how can we do that?

Adding up multiples of constraints ($y_i \geq 0$):

$$y_1 \cdot \left(\sum_i a_{1i} x_i \right) \leq y_1 b_1$$

$$+ y_2 \cdot \left(\sum_i a_{2i} x_i \right) \leq y_2 b_2$$

+ ...

$$+ y_m \cdot \left(\sum_i a_{mi} x_i \right) \leq y_m b_m$$

$$\rightarrow \sum_{i=1}^n \left(\sum_{j=1}^m a_{ji} y_j \right) x_i \leq \sum_{j=1}^m y_j b_j$$

Standard form:

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$
$$x_i \geq 0$$

We want: $\sum_{i=1}^n c_i x_i \leq \xi$; how can we do that?

Adding up multiples of constraints ($y_i \geq 0$):

$$\begin{aligned} & y_1 \cdot \left(\sum_i a_{1i} x_i \right) \leq y_1 b_1 \\ & + y_2 \cdot \left(\sum_i a_{2i} x_i \right) \leq y_2 b_2 \\ & + \dots \\ & + y_m \cdot \left(\sum_i a_{mi} x_i \right) \leq y_m b_m \\ & \rightarrow \sum_{i=1}^n \left(\sum_{j=1}^m a_{ji} y_j \right) x_i \leq \sum_{j=1}^m y_j b_j \end{aligned}$$
$$\sum_{i=1}^n c_i x_i \stackrel{!}{\leq} \sum_{i=1}^n \left(\sum_{j=1}^m a_{ji} y_j \right) x_i \leq \sum_{j=1}^m y_j b_j$$

Important LP Concepts: Duality

Standard form:

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$
$$x_i \geq 0$$

We want: $\sum_{i=1}^n c_i x_i \leq \xi$; how can we do that?

Adding up multiples of constraints ($y_i \geq 0$):

$$\begin{aligned} & y_1 \cdot \left(\sum_i a_{1i} x_i \right) \leq y_1 b_1 \\ & + y_2 \cdot \left(\sum_i a_{2i} x_i \right) \leq y_2 b_2 \\ & + \dots \\ & + y_m \cdot \left(\sum_i a_{mi} x_i \right) \leq y_m b_m \\ & \rightarrow \sum_{i=1}^n \left(\sum_{j=1}^m a_{ji} y_j \right) x_i \leq \sum_{j=1}^m y_j b_j \end{aligned}$$
$$\sum_{i=1}^n c_i x_i \leq \sum_{i=1}^n \left(\sum_{j=1}^m a_{ji} y_j \right) x_i \leq \sum_{j=1}^m y_j b_j$$

Important LP Concepts: Duality

Standard form:

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$
$$x_i \geq 0$$

We want: $\sum_{i=1}^n c_i x_i \leq \xi$; how can we do that?

Adding up multiples of constraints ($y_i \geq 0$):

$$y_1 \cdot \left(\sum_i a_{1i} x_i \right) \leq y_1 b_1$$

$$+ y_2 \cdot \left(\sum_i a_{2i} x_i \right) \leq y_2 b_2$$

+ ...

$$+ y_m \cdot \left(\sum_i a_{mi} x_i \right) \leq y_m b_m$$

$$\rightarrow \sum_{i=1}^n \left(\sum_{j=1}^m a_{ji} y_j \right) x_i \leq \sum_{j=1}^m y_j b_j$$

$$\sum_{i=1}^n c_i x_i \leq \sum_{i=1}^n \left(\sum_{j=1}^m a_{ji} y_j \right) x_i \leq \sum_{j=1}^m y_j b_j$$

Due to $x_i \geq 0$: holds if $\sum_{j=1}^m a_{ji} y_j \geq c_i$

Important LP Concepts: Duality

Standard form:

$$\max \sum_i c_i x_i \text{ s.t.}$$

$$1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j$$
$$x_i \geq 0$$

Dual:

$$\min \sum_j b_j y_j \text{ s.t.}$$

$$1 \leq i \leq n : \sum_j a_{ji} x_i \geq c_i$$
$$y_j \geq 0$$

We want: $\sum_{i=1}^n c_i x_i \leq \xi$; how can we do that?

Adding up multiples of constraints ($y_i \geq 0$):

$$y_1 \cdot \left(\sum_i a_{1i} x_i \right) \leq y_1 b_1$$

$$+ y_2 \cdot \left(\sum_i a_{2i} x_i \right) \leq y_2 b_2$$

+ ...

$$+ y_m \cdot \left(\sum_i a_{mi} x_i \right) \leq y_m b_m$$

$$\rightarrow \sum_{i=1}^n \left(\sum_{j=1}^m a_{ji} y_j \right) x_i \leq \sum_{j=1}^m y_j b_j$$

$$\sum_{i=1}^n c_i x_i \leq \sum_{i=1}^n \left(\sum_{j=1}^m a_{ji} y_j \right) x_i \leq \sum_{j=1}^m y_j b_j$$

Due to $x_i \geq 0$: holds if $\sum_{j=1}^m a_{ji} y_j \geq c_i$

Primal:

$$\begin{aligned} & \max \sum_i c_i x_i \text{ s.t.} \\ & 1 \leq j \leq m : \sum_i a_{ji} x_i \leq b_j \\ & x_i \geq 0 \end{aligned}$$

By construction: $\sum_i c_i x_i \leq \sum_j b_j y_j$ for any feasible x, y .

Dual:

$$\begin{aligned} & \min \sum_j b_j y_j \text{ s.t.} \\ & 1 \leq i \leq n : \sum_j a_{ji} x_i \geq c_i \\ & y_j \geq 0 \end{aligned}$$

Strong duality theorem:

If the primal LP has an optimal solution x^* , the dual also has an optimal solution y^* and $\sum_{i=1}^n c_i x_i^* = \sum_{j=1}^m b_j y_j^*$.

Primal max, Dual min

$x_i \geq 0 \rightarrow \geq$ -constraint

x_i free $\rightarrow =$ -constraint

$x_i \leq 0 \rightarrow \leq$ -constraint

$\leq$ -constraint $\rightarrow y_j \geq 0$

$=$ -constraint $\rightarrow y_j$ free

$\geq$ -constraint $\rightarrow y_j \leq 0$

Dualization Rules

Primal max, Dual min

$x_i \geq 0 \rightarrow \geq$ -constraint

x_i free $\rightarrow =$ -constraint

$x_i \leq 0 \rightarrow \leq$ -constraint

$\leq$ -constraint $\rightarrow y_j \geq 0$

$=$ -constraint $\rightarrow y_j$ free

$\geq$ -constraint $\rightarrow y_j \leq 0$

Primal min, Dual max

$x_i \geq 0 \rightarrow \leq$ -constraint

x_i free $\rightarrow =$ -constraint

$x_i \leq 0 \rightarrow \geq$ -constraint

$\leq$ -constraint $\rightarrow y_j \leq 0$

$=$ -constraint $\rightarrow y_j$ free

$\geq$ -constraint $\rightarrow y_j \geq 0$

TSP formulation LP relaxation:

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V : \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset : \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \geq 0$$

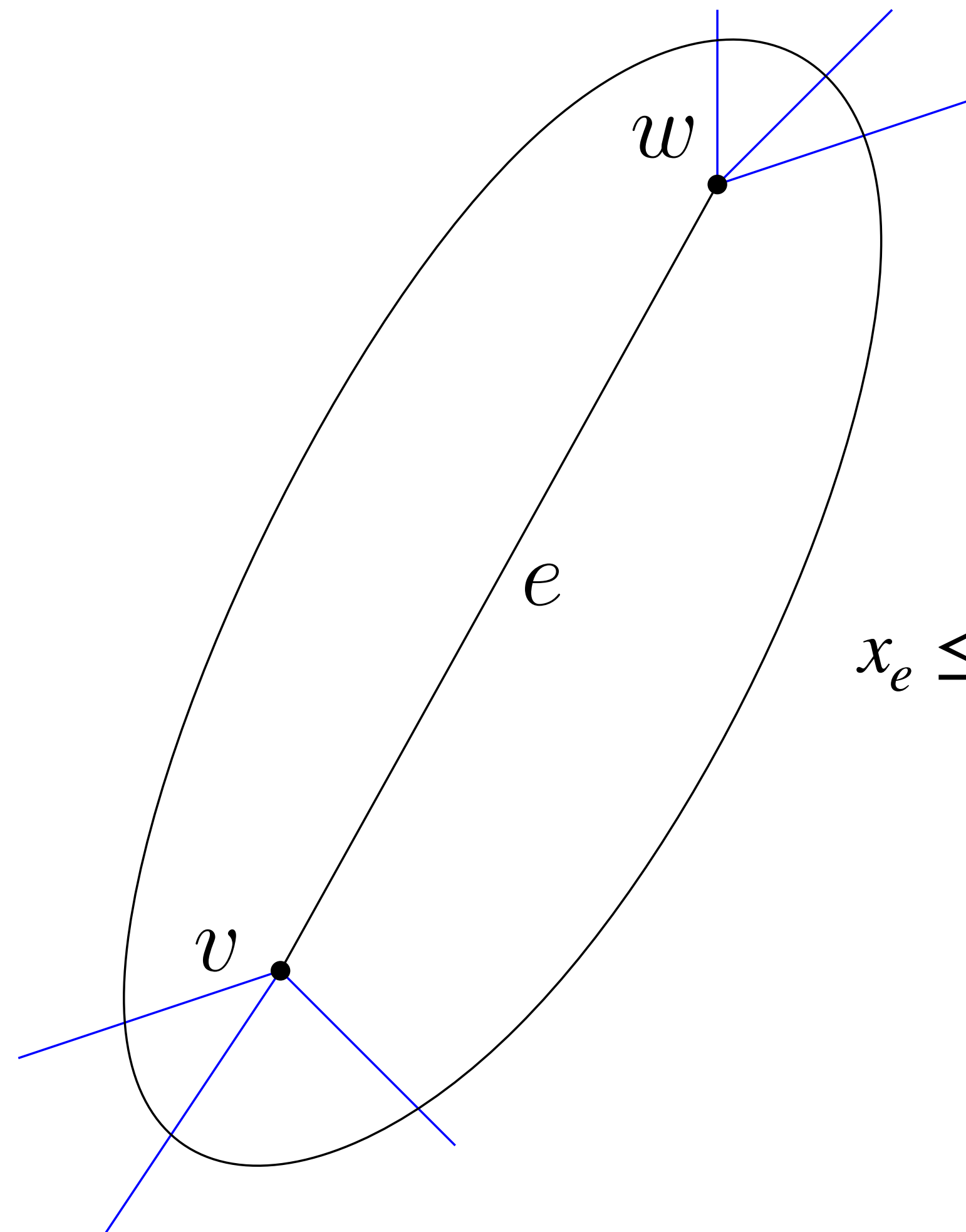
TSP formulation LP relaxation:

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V: \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset: \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \geq 0$$



$$x_e \leq 1 \Leftrightarrow \sum_{f \in \delta(\{v,w\})} x_f \geq 2$$

TSP formulation LP relaxation:

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V : \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset : \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \geq 0$$

Duality Example: TSP

TSP formulation LP relaxation:

$$\min \sum_{e \in E} c_e x_e \text{ s.t.}$$

$$\forall v \in V: \sum_{e \in \delta(\{v\})} x_e = 2$$

$$\forall S \subsetneq V, S \neq \emptyset: \sum_{e \in \delta(S)} x_e \geq 2$$

$$x_e \geq 0$$

Dual:

$$\max 2 \sum_{v \in V} y_v + 2 \sum_S y_S \text{ s.t.}$$

$$\forall e = vw \in E: y_v + y_w + \sum_{S: e \in \delta(S)} y_S \leq c_e$$

$$y_S \geq 0, y_v \text{ free}$$

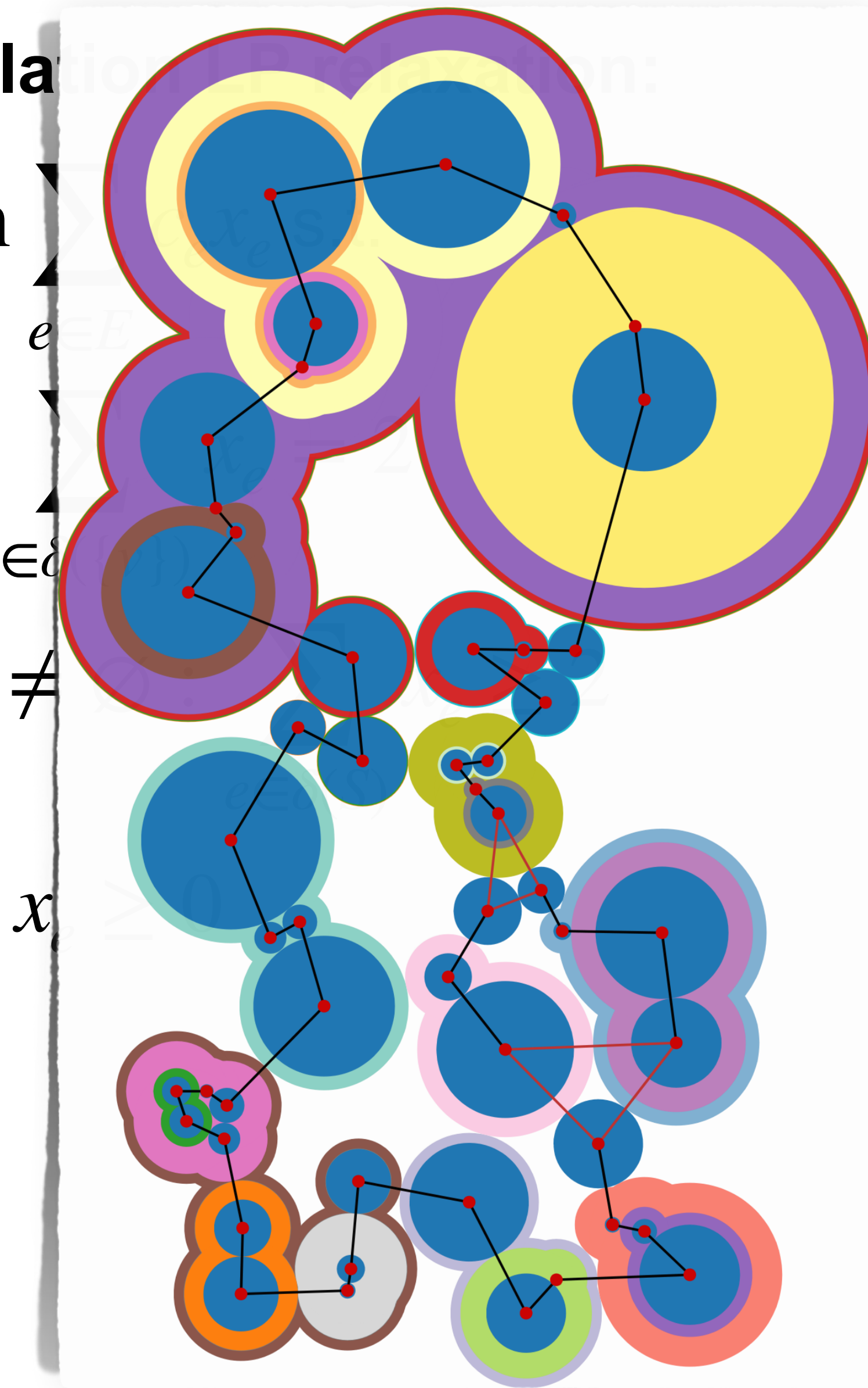
Duality Example: TSP

TSP formula

$\min$

$\forall v \in V:$

$\forall S \subsetneq V, S \neq \emptyset$



Dual:

$$\max 2 \sum_{v \in V} y_v + 2 \sum_S y_S \text{ s.t.}$$

$$\forall e = vw \in E: y_v + y_w + \sum_{S: e \in \delta(S)} y_S \leq c_e$$

$$y_S \geq 0, y_v \text{ free}$$

- **Efficient LP solver**
 - Root relaxation
 - Later on: warm starting
- **Branch & Bound**
 - Exploration order
 - Branch variable selection
 - Propagation & conflict analysis
- **Cutting planes**
 - Cutting plane templates & heuristics
 - Cut management (Selection, Scoring, Forgetting, ...)
- **Presolve**
 - Problem size reduction (eliminating variables/constraints, ...)
 - Tightening (strengthening bounds, symmetry breaking, ...)
 - Numerical preprocessing (scaling, ...)
 - Exposing structure (helps cutting planes & heuristics, alternative solution strategies)
- **Primal heuristics**

- **Efficient LP solver**
 - Root relaxation
 - Later on: warm starting
- **Branch & Bound**
 - Exploration order
 - Branch variable selection
 - Propagation & conflict analysis
- **Cutting planes**
 - Cutting plane templates & heuristics
 - Cut management (Selection, Scoring, Forgetting, ...)
- **Presolve**
 - Problem size reduction (eliminating variables/constraints, ...)
 - Tightening (strengthening bounds, symmetry breaking, ...)
 - Numerical preprocessing (scaling, ...)
 - Exposing structure (helps cutting planes & heuristics, alternative solution strategies)
- **Primal heuristics**

Parts of a MIP Solver: LP Solving & Exploration Order

Balance between finding better solutions & improving bounds.

**Balance between finding better solutions & improving bounds.
Exploration order influences warm-starting! Several options exist:**

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)
 - Switching a few variable bounds typically needs only a few simplex iterations

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)
 - Switching a few variable bounds typically needs only a few simplex iterations
 - Saves memory

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)
 - Switching a few variable bounds typically needs only a few simplex iterations
 - Saves memory
 - Far 'jumps' in the tree during exploration can be very expensive

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)
 - Switching a few variable bounds typically needs only a few simplex iterations
 - Saves memory
 - Far 'jumps' in the tree during exploration can be very expensive
 - Discourages non-DFS-like exploration patterns

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)
 - Switching a few variable bounds typically needs only a few simplex iterations
 - Saves memory
 - Far 'jumps' in the tree during exploration can be very expensive
 - Discourages non-DFS-like exploration patterns
 - Cuts can be kept/discarded easily and apply globally

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)
 - Switching a few variable bounds typically needs only a few simplex iterations
 - Saves memory
 - Far 'jumps' in the tree during exploration can be very expensive
 - Discourages non-DFS-like exploration patterns
 - Cuts can be kept/discarded easily and apply globally
2. Store extra information on search nodes

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)
 - Switching a few variable bounds typically needs only a few simplex iterations
 - Saves memory
 - Far 'jumps' in the tree during exploration can be very expensive
 - Discourages non-DFS-like exploration patterns
 - Cuts can be kept/discarded easily and apply globally
2. Store extra information on search nodes
 - E.g., an optimal basis of the parent/some other warm-starting basis

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)
 - Switching a few variable bounds typically needs only a few simplex iterations
 - Saves memory
 - Far 'jumps' in the tree during exploration can be very expensive
 - Discourages non-DFS-like exploration patterns
 - Cuts can be kept/discarded easily and apply globally
2. Store extra information on search nodes
 - E.g., an optimal basis of the parent/some other warm-starting basis
 - Or even more information to avoid heavy matrix operations

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)
 - Switching a few variable bounds typically needs only a few simplex iterations
 - Saves memory
 - Far 'jumps' in the tree during exploration can be very expensive
 - Discourages non-DFS-like exploration patterns
 - Cuts can be kept/discarded easily and apply globally
2. Store extra information on search nodes
 - E.g., an optimal basis of the parent/some other warm-starting basis
 - Or even more information to avoid heavy matrix operations
 - Trade-off between memory and runtime

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)
 - Switching a few variable bounds typically needs only a few simplex iterations
 - Saves memory
 - Far 'jumps' in the tree during exploration can be very expensive
 - Discourages non-DFS-like exploration patterns
 - Cuts can be kept/discarded easily and apply globally
2. Store extra information on search nodes
 - E.g., an optimal basis of the parent/some other warm-starting basis
 - Or even more information to avoid heavy matrix operations
 - Trade-off between memory and runtime
 - Makes handling cuts more involved (there may be old bases without them)

Balance between finding better solutions & improving bounds.

Exploration order influences warm-starting! Several options exist:

1. Work with a single copy of the LP data structure (or one per thread)
 - Switching a few variable bounds typically needs only a few simplex iterations
 - Saves memory
 - Far 'jumps' in the tree during exploration can be very expensive
 - Discourages non-DFS-like exploration patterns
 - Cuts can be kept/discarded easily and apply globally
2. Store extra information on search nodes
 - E.g., an optimal basis of the parent/some other warm-starting basis
 - Or even more information to avoid heavy matrix operations
 - Trade-off between memory and runtime
 - Makes handling cuts more involved (there may be old bases without them)
3. Some hybrid (e.g., based on memory consumption, some initial phase, ...)

Parts of a MIP Solver: Branch Variable Selection

- **Efficient LP solver**
 - Root relaxation
 - Later on: warm starting
- **Branch & Bound**
 - Exploration order
 - Branch variable selection
 - Propagation & conflict analysis
- **Cutting planes**
 - Cutting plane templates & heuristics
 - Cut management (Selection, Scoring, Forgetting, ...)
- **Presolve**
 - Problem size reduction (eliminating variables/constraints, ...)
 - Tightening (strengthening bounds, symmetry breaking, ...)
 - Numerical preprocessing (scaling, ...)
 - Exposing structure (helps cutting planes & heuristics, alternative solution strategies)
- **Primal heuristics**

Branch Variable Selection

Which fractional variable do we branch on?

Which fractional variable do we branch on?

Classical options:

Which fractional variable do we branch on?

Classical options:

- Most Fractional

Which fractional variable do we branch on?

Classical options:

- Most Fractional
- Pseudo-Cost Branching

Which fractional variable do we branch on?

Classical options:

- Most Fractional
- Pseudo-Cost Branching
- Strong Branching

Which fractional variable do we branch on?

Classical options:

- Most Fractional
- Pseudo-Cost Branching
- Strong Branching
- Reliability Branching

Which fractional variable do we branch on?

Classical options:

- Most Fractional
- Pseudo-Cost Branching
- Strong Branching
- Reliability Branching

Modern solvers: hybrids & extensions of these schemes also including other information.

Pseudo-Cost Branching

Pseudo-Cost Branching

Goal: estimate by how much the bound will tighten (min: increase) in each child.

Pseudo-Cost Branching

Goal: estimate by how much the bound will tighten (min: increase) in each child.

Idea: maintain two scores $\Psi_-(x_i)$, $\Psi_+(x_i)$ for each integer variable x_i .

Pseudo-Cost Branching

Goal: estimate by how much the bound will tighten (min: increase) in each child.

Idea: maintain two scores $\Psi_-(x_i)$, $\Psi_+(x_i)$ for each integer variable x_i .

$\Psi_-(x_i)$: estimates by how much the bound improves if branching reduces x_i by 1

Pseudo-Cost Branching

Goal: estimate by how much the bound will tighten (min: increase) in each child.

Idea: maintain two scores $\Psi_-(x_i)$, $\Psi_+(x_i)$ for each integer variable x_i .

$\Psi_-(x_i)$: estimates by how much the bound improves if branching reduces x_i by 1

$\Psi_+(x_i)$: estimates by how much the bound improves if branching increases x_i by 1

Pseudo-Cost Branching

Goal: estimate by how much the bound will tighten (min: increase) in each child.

Idea: maintain two scores $\Psi_-(x_i)$, $\Psi_+(x_i)$ for each integer variable x_i .

$\Psi_-(x_i)$: estimates by how much the bound improves if branching reduces x_i by 1

$\Psi_+(x_i)$: estimates by how much the bound improves if branching increases x_i by 1

Score usage: For each candidate variable, estimate the bounds of both children.

Pseudo-Cost Branching

Goal: estimate by how much the bound will tighten (min: increase) in each child.

Idea: maintain two scores $\Psi_-(x_i)$, $\Psi_+(x_i)$ for each integer variable x_i .

$\Psi_-(x_i)$: estimates by how much the bound improves if branching reduces x_i by 1

$\Psi_+(x_i)$: estimates by how much the bound improves if branching increases x_i by 1

Score usage: For each candidate variable, estimate the bounds of both children.

For min (parent bound p): $E_- = p + \Psi_-(x_i) \cdot (x_i - \lfloor x_i \rfloor)$, $E_+ = p + \Psi_+(x_i) \cdot (\lceil x_i \rceil - x_i)$

Pseudo-Cost Branching

Goal: estimate by how much the bound will tighten (min: increase) in each child.

Idea: maintain two scores $\Psi_-(x_i)$, $\Psi_+(x_i)$ for each integer variable x_i .

$\Psi_-(x_i)$: estimates by how much the bound improves if branching reduces x_i by 1

$\Psi_+(x_i)$: estimates by how much the bound improves if branching increases x_i by 1

Score usage: For each candidate variable, estimate the bounds of both children.

For min (parent bound p): $E_- = p + \Psi_-(x_i) \cdot (x_i - \lfloor x_i \rfloor)$, $E_+ = p + \Psi_+(x_i) \cdot (\lceil x_i \rceil - x_i)$

Make branching decision based on E_- , E_+ , e.g.: max min, max average, min max, ...

Pseudo-Cost Branching

Goal: estimate by how much the bound will tighten (min: increase) in each child.

Idea: maintain two scores $\Psi_-(x_i)$, $\Psi_+(x_i)$ for each integer variable x_i .

$\Psi_-(x_i)$: estimates by how much the bound improves if branching reduces x_i by 1

$\Psi_+(x_i)$: estimates by how much the bound improves if branching increases x_i by 1

Score usage: For each candidate variable, estimate the bounds of both children.

For min (parent bound p): $E_- = p + \Psi_-(x_i) \cdot (x_i - \lfloor x_i \rfloor)$, $E_+ = p + \Psi_+(x_i) \cdot (\lceil x_i \rceil - x_i)$

Make branching decision based on E_- , E_+ , e.g.: max min, max average, min max, ...

Score update: When a child's LP is solved with solution c , use $\Delta = c - p$ to update.

Pseudo-Cost Branching

Goal: estimate by how much the bound will tighten (min: increase) in each child.

Idea: maintain two scores $\Psi_-(x_i)$, $\Psi_+(x_i)$ for each integer variable x_i .

$\Psi_-(x_i)$: estimates by how much the bound improves if branching reduces x_i by 1

$\Psi_+(x_i)$: estimates by how much the bound improves if branching increases x_i by 1

Score usage: For each candidate variable, estimate the bounds of both children.

For min (parent bound p): $E_- = p + \Psi_-(x_i) \cdot (x_i - \lfloor x_i \rfloor)$, $E_+ = p + \Psi_+(x_i) \cdot (\lceil x_i \rceil - x_i)$

Make branching decision based on E_- , E_+ , e.g.: max min, max average, min max, ...

Score update: When a child's LP is solved with solution c , use $\Delta = c - p$ to update.

If c was a down branch on x_i , update $\Psi_-(x_i) = \lambda \Psi_-(x_i) + (1 - \lambda) \Delta / (x_{i,p} - \lfloor x_{i,p} \rfloor)$;

Pseudo-Cost Branching

Goal: estimate by how much the bound will tighten (min: increase) in each child.

Idea: maintain two scores $\Psi_-(x_i)$, $\Psi_+(x_i)$ for each integer variable x_i .

$\Psi_-(x_i)$: estimates by how much the bound improves if branching reduces x_i by 1

$\Psi_+(x_i)$: estimates by how much the bound improves if branching increases x_i by 1

Score usage: For each candidate variable, estimate the bounds of both children.

For min (parent bound p): $E_- = p + \Psi_-(x_i) \cdot (x_i - \lfloor x_i \rfloor)$, $E_+ = p + \Psi_+(x_i) \cdot (\lceil x_i \rceil - x_i)$

Make branching decision based on E_- , E_+ , e.g.: max min, max average, min max, ...

Score update: When a child's LP is solved with solution c , use $\Delta = c - p$ to update.

If c was a down branch on x_i , update $\Psi_-(x_i) = \lambda \Psi_-(x_i) + (1 - \lambda) \Delta / (x_{i,p} - \lfloor x_{i,p} \rfloor)$;

typically, variants are used if the number of observed 'samples' is low.

Pseudo-Cost Branching: Main Issue

Score initialization: what are good initial scores? all 0?

Score initialization: what are good initial scores? all 0?

Problem: scores get better over time, but initial choices are among the most important!

Score initialization: what are good initial scores? all 0?

Problem: scores get better over time, but initial choices are among the most important!

Solution: track 'reliability' (i.e., sample size) of the scores; in doubt, use another strategy

Score initialization: what are good initial scores? all 0?

Problem: scores get better over time, but initial choices are among the most important!

Solution: track ‘reliability’ (i.e., sample size) of the scores; in doubt, use another strategy

Alternative: restarts (more prominent in SAT solvers, but used in MIP solvers too)

Strong Branching

Strong Branching

Strong Branching: try multiple candidate branching variables, solving both child LP's

Strong Branching

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

Strong Branching

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

- Which candidates do we explore? Often depending on multiple heuristics.

Strong Branching

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

- Which candidates do we explore? Often depending on multiple heuristics.
- How many candidates do we explore? Runtime/branching decision quality trade-off.

Strong Branching

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

- Which candidates do we explore? Often depending on multiple heuristics.
- How many candidates do we explore? Runtime/branching decision quality trade-off.
- Do we solve child LP's completely or run (dual) simplex with an iteration limit?

Strong Branching

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

- Which candidates do we explore? Often depending on multiple heuristics.
- How many candidates do we explore? Runtime/branching decision quality trade-off.
- Do we solve child LP's completely or run (dual) simplex with an iteration limit?
- Until when do we consider strong branching (tree depth limit, node limit, ...)?

Strong Branching

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

- Which candidates do we explore? Often depending on multiple heuristics.
- How many candidates do we explore? Runtime/branching decision quality trade-off.
- Do we solve child LP's completely or run (dual) simplex with an iteration limit?
- Until when do we consider strong branching (tree depth limit, node limit, ...)?

Benefits:

Strong Branching

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

- Which candidates do we explore? Often depending on multiple heuristics.
- How many candidates do we explore? Runtime/branching decision quality trade-off.
- Do we solve child LP's completely or run (dual) simplex with an iteration limit?
- Until when do we consider strong branching (tree depth limit, node limit, ...)?

Benefits:

- Can spend more time on branching close to the root (only few nodes)

Strong Branching

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

- Which candidates do we explore? Often depending on multiple heuristics.
- How many candidates do we explore? Runtime/branching decision quality trade-off.
- Do we solve child LP's completely or run (dual) simplex with an iteration limit?
- Until when do we consider strong branching (tree depth limit, node limit, ...)?

Benefits:

- Can spend more time on branching close to the root (only few nodes)
- These branches tend to have high impact

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

- Which candidates do we explore? Often depending on multiple heuristics.
- How many candidates do we explore? Runtime/branching decision quality trade-off.
- Do we solve child LP's completely or run (dual) simplex with an iteration limit?
- Until when do we consider strong branching (tree depth limit, node limit, ...)?

Benefits:

- Can spend more time on branching close to the root (only few nodes)
- These branches tend to have high impact
- Can easily 'initialize' our pseudo-costs

Strong Branching

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

- Which candidates do we explore? Often depending on multiple heuristics.
- How many candidates do we explore? Runtime/branching decision quality trade-off.
- Do we solve child LP's completely or run (dual) simplex with an iteration limit?
- Until when do we consider strong branching (tree depth limit, node limit, ...)?

Benefits:

- Can spend more time on branching close to the root (only few nodes)
- These branches tend to have high impact
- Can easily 'initialize' our pseudo-costs
- Can feed heuristics/scores on which parts of the search space seem promising

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

- Which candidates do we explore? Often depending on multiple heuristics.
- How many candidates do we explore? Runtime/branching decision quality trade-off.
- Do we solve child LP's completely or run (dual) simplex with an iteration limit?
- Until when do we consider strong branching (tree depth limit, node limit, ...)?

Benefits:

- Can spend more time on branching close to the root (only few nodes)
- These branches tend to have high impact
- Can easily 'initialize' our pseudo-costs
- Can feed heuristics/scores on which parts of the search space seem promising
- Can sometimes discover fixed variables or cuts

Strong Branching

Strong Branching: try multiple candidate branching variables, solving both child LP's

Multiple open points/trade-offs:

- Which candidates do we explore? Often depending on multiple heuristics.
- How many candidates do we explore? Runtime/branching decision quality trade-off.
- Do we solve child LP's completely or run (dual) simplex with an iteration limit?
- Until when do we consider strong branching (tree depth limit, node limit, ...)?

Benefits:

- Can spend more time on branching close to the root (only few nodes)
- These branches tend to have high impact
- Can easily 'initialize' our pseudo-costs
- Can feed heuristics/scores on which parts of the search space seem promising
- Can sometimes discover fixed variables or cuts

Reliability Branching: Strong Branching + Pseudo Costs (once reliable)

- **Efficient LP solver**
 - Root relaxation
 - Later on: warm starting
- **Branch & Bound**
 - Exploration order
 - Branch variable selection
 - Propagation & conflict analysis
- **Cutting planes**
 - Cutting plane templates & heuristics
 - Cut management (Selection, Scoring, Forgetting, ...)
- **Presolve**
 - Problem size reduction (eliminating variables/constraints, ...)
 - Tightening (strengthening bounds, symmetry breaking, ...)
 - Numerical preprocessing (scaling, ...)
 - Exposing structure (helps cutting planes & heuristics, alternative solution strategies)
- **Primal heuristics**

Cutting Plane Management

- **Cutting plane templates**

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*
 - Examples: Clique constraints (see exercise), Cover cuts (e.g., in Knapsack)

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*
 - Examples: Clique constraints (see exercise), Cover cuts (e.g., in Knapsack)
 - LP/rounding-based: Gomory cuts, Mixed-Integer Rounding (sometimes in MMA)

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*
 - Examples: Clique constraints (see exercise), Cover cuts (e.g., in Knapsack)
 - LP/rounding-based: Gomory cuts, Mixed-Integer Rounding (sometimes in MMA)
 - Way more (and more general) templates

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*
 - Examples: Clique constraints (see exercise), Cover cuts (e.g., in Knapsack)
 - LP/rounding-based: Gomory cuts, Mixed-Integer Rounding (sometimes in MMA)
 - Way more (and more general) templates
 - Counts are shown in CPLEX/Gurobi logs; no further info available

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*
 - Examples: Clique constraints (see exercise), Cover cuts (e.g., in Knapsack)
 - LP/rounding-based: Gomory cuts, Mixed-Integer Rounding (sometimes in MMA)
 - Way more (and more general) templates
 - Counts are shown in CPLEX/Gurobi logs; no further info available
- **Cut Management**

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*
 - Examples: Clique constraints (see exercise), Cover cuts (e.g., in Knapsack)
 - LP/rounding-based: Gomory cuts, Mixed-Integer Rounding (sometimes in MMA)
 - Way more (and more general) templates
 - Counts are shown in CPLEX/Gurobi logs; no further info available
- **Cut Management**
 - Having too many cuts in an LP slows it down

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*
 - Examples: Clique constraints (see exercise), Cover cuts (e.g., in Knapsack)
 - LP/rounding-based: Gomory cuts, Mixed-Integer Rounding (sometimes in MMA)
 - Way more (and more general) templates
 - Counts are shown in CPLEX/Gurobi logs; no further info available
- **Cut Management**
 - Having too many cuts in an LP slows it down
 - Cuts can improve bounds drastically, but can also have negligible impact

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*
 - Examples: Clique constraints (see exercise), Cover cuts (e.g., in Knapsack)
 - LP/rounding-based: Gomory cuts, Mixed-Integer Rounding (sometimes in MMA)
 - Way more (and more general) templates
 - Counts are shown in CPLEX/Gurobi logs; no further info available
- **Cut Management**
 - Having too many cuts in an LP slows it down
 - Cuts can improve bounds drastically, but can also have negligible impact
 - Excessive cuts can make the solution much less sparse

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*
 - Examples: Clique constraints (see exercise), Cover cuts (e.g., in Knapsack)
 - LP/rounding-based: Gomory cuts, Mixed-Integer Rounding (sometimes in MMA)
 - Way more (and more general) templates
 - Counts are shown in CPLEX/Gurobi logs; no further info available
- **Cut Management**
 - Having too many cuts in an LP slows it down
 - Cuts can improve bounds drastically, but can also have negligible impact
 - Excessive cuts can make the solution much less sparse
 - (Temporarily) deleting/forgetting cuts can be necessary → cut pool

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*
 - Examples: Clique constraints (see exercise), Cover cuts (e.g., in Knapsack)
 - LP/rounding-based: Gomory cuts, Mixed-Integer Rounding (sometimes in MMA)
 - Way more (and more general) templates
 - Counts are shown in CPLEX/Gurobi logs; no further info available
- **Cut Management**
 - Having too many cuts in an LP slows it down
 - Cuts can improve bounds drastically, but can also have negligible impact
 - Excessive cuts can make the solution much less sparse
 - (Temporarily) deleting/forgetting cuts can be necessary → cut pool
 - If multiple cuts are available, which ones should be added?

- **Cutting plane templates**
 - General-purpose solvers have heuristics for finding cutting planes
 - These correspond to cutting plane *templates*
 - Examples: Clique constraints (see exercise), Cover cuts (e.g., in Knapsack)
 - LP/rounding-based: Gomory cuts, Mixed-Integer Rounding (sometimes in MMA)
 - Way more (and more general) templates
 - Counts are shown in CPLEX/Gurobi logs; no further info available
- **Cut Management**
 - Having too many cuts in an LP slows it down
 - Cuts can improve bounds drastically, but can also have negligible impact
 - Excessive cuts can make the solution much less sparse
 - (Temporarily) deleting/forgetting cuts can be necessary → cut pool
 - If multiple cuts are available, which ones should be added?
 - Which cutting plane heuristic should run?

- **Efficient LP solver**
 - Root relaxation
 - Later on: warm starting
- **Branch & Bound**
 - Exploration order
 - Branch variable selection
 - Propagation & conflict analysis
- **Cutting planes**
 - Cutting plane templates & heuristics
 - Cut management (Selection, Scoring, Forgetting, ...)
- **Presolve**
 - Problem size reduction (eliminating variables/constraints, ...)
 - Tightening (strengthening bounds, symmetry breaking, ...)
 - Numerical preprocessing (scaling, ...)
 - Exposing structure (helps cutting planes & heuristics, alternative solution strategies)
- **Primal heuristics**

- **Efficient LP solver**
 - Root relaxation
 - Later on: warm starting
- **Branch & Bound**
 - Exploration order
 - Branch variable selection
 - Propagation & conflict analysis
- **Cutting planes**
 - Cutting plane templates & heuristics
 - Cut management (Selection, Scoring, Forgetting, ...)
- **Presolve**
 - Problem size reduction (eliminating variables/constraints, ...)
 - Tightening (strengthening bounds, symmetry breaking, ...)
 - Numerical preprocessing (scaling, ...)
 - Exposing structure (helps cutting planes & heuristics, alternative solution strategies)
- **Primal heuristics**

Primal Heuristics

Construction Heuristics

Construction Heuristics

- Zero Heuristic (just set everything to 0), other simple greedy heuristics

Construction Heuristics

- Zero Heuristic (just set everything to 0), other simple greedy heuristics
- Zero Objective Heuristic (remove the objective, see if we easily find a solution)

Construction Heuristics

- Zero Heuristic (just set everything to 0), other simple greedy heuristics
- Zero Objective Heuristic (remove the objective, see if we easily find a solution)
- LP rounding (rounding to integers; differing degrees of ‘cleverness’/propagation)

Construction Heuristics

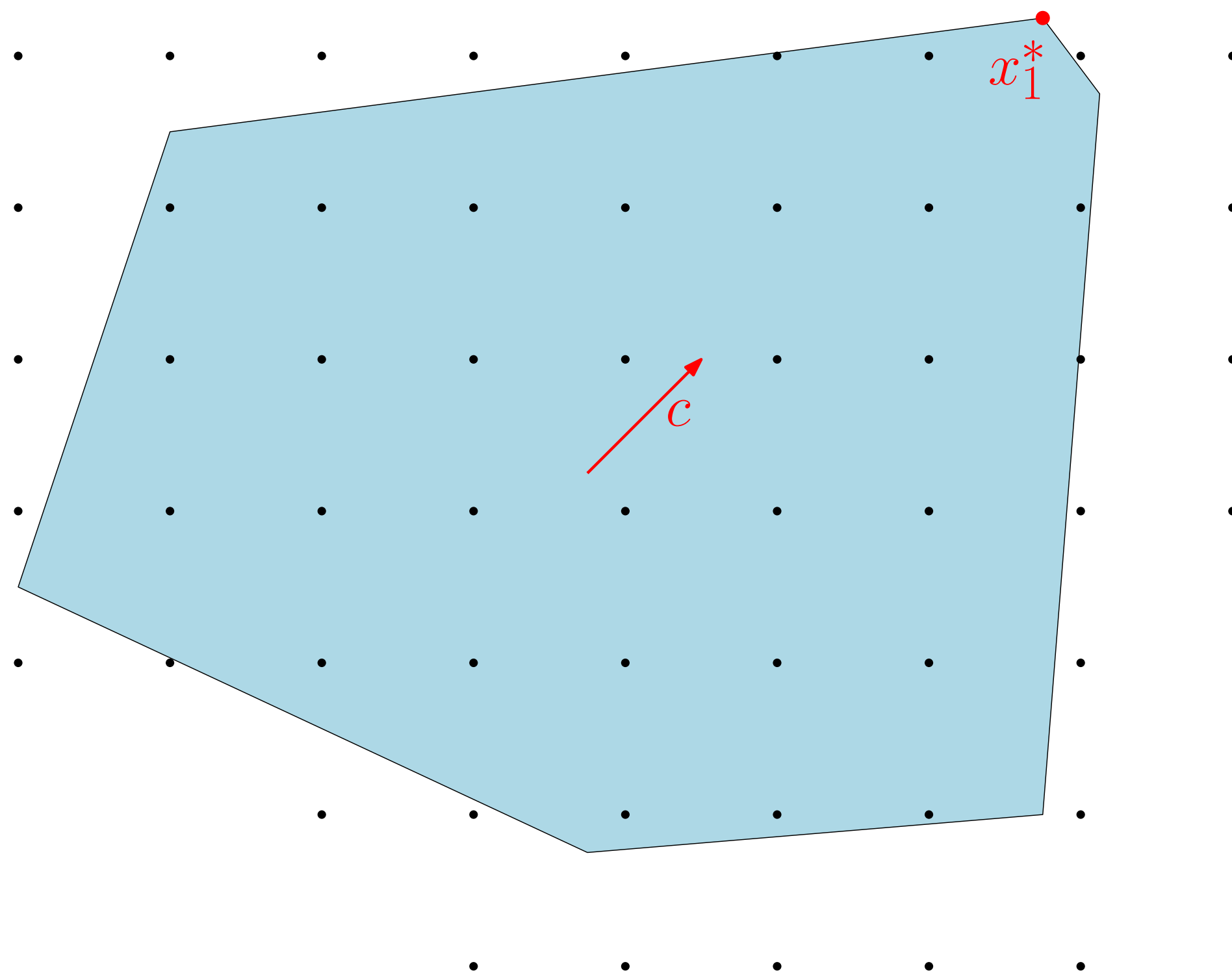
- Zero Heuristic (just set everything to 0), other simple greedy heuristics
- Zero Objective Heuristic (remove the objective, see if we easily find a solution)
- LP rounding (rounding to integers; differing degrees of ‘cleverness’/propagation)
- Diving (rounding + resolving LP, potentially limited backtracking)

Construction Heuristics

- Zero Heuristic (just set everything to 0), other simple greedy heuristics
- Zero Objective Heuristic (remove the objective, see if we easily find a solution)
- LP rounding (rounding to integers; differing degrees of ‘cleverness’/propagation)
- Diving (rounding + resolving LP, potentially limited backtracking)
- Feasibility Pump (variant)

Construction Heuristics

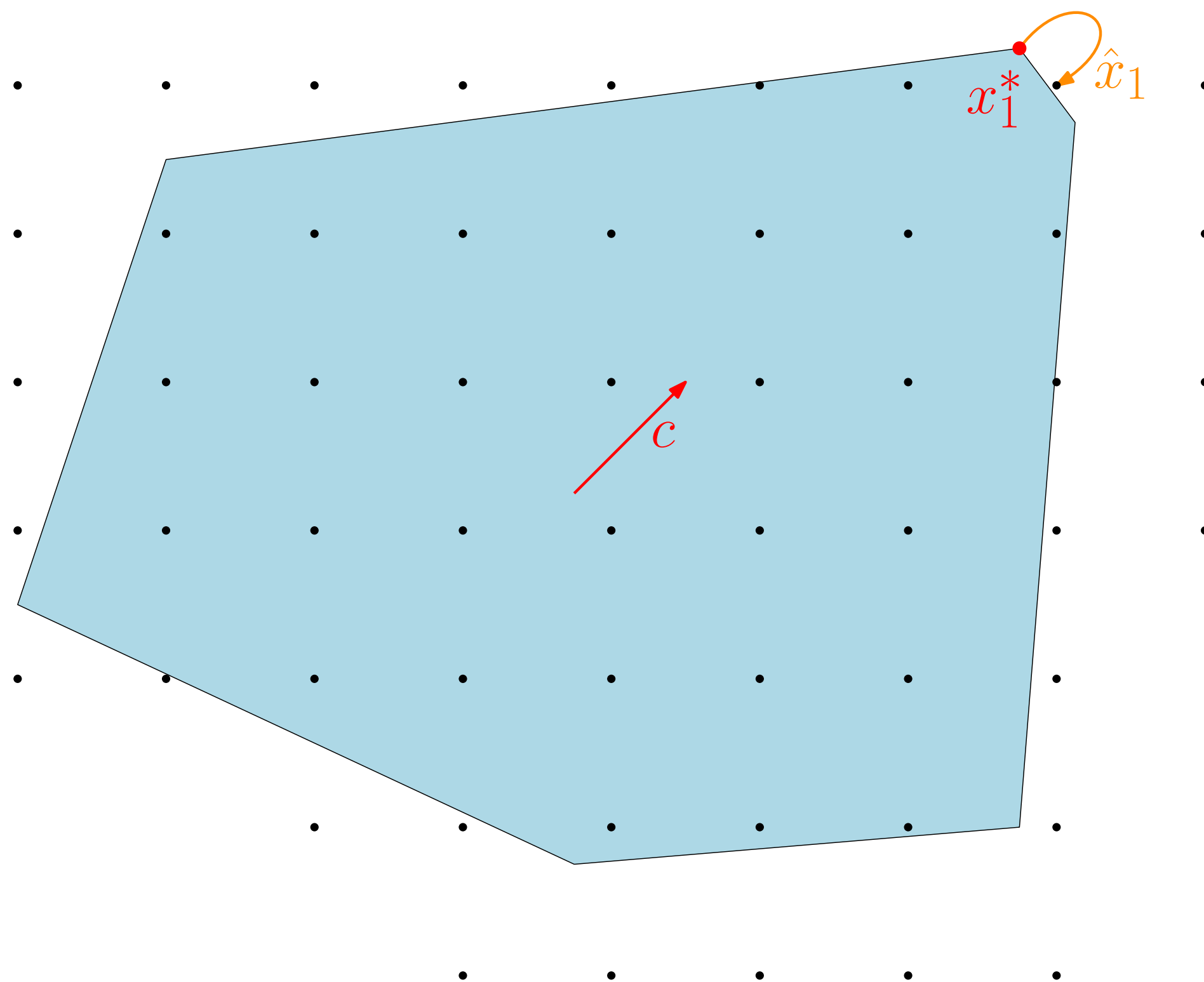
- Zero Heuristic (just set everything to 0), other simple greedy heuristics
- Zero Objective Heuristic (remove the objective, see if we easily find a solution)
- LP rounding (rounding to integers; differing degrees of 'cleverness'/propagation)
- Diving (rounding + resolving LP, potentially limited backtracking)
- Feasibility Pump (variant)



```
x* = solve_relaxation()
while not terminate():
    if is_integral(x*):
        return x*
    x_hat = round(x*)
    if detect_cycle(x_hat):
        perturb(x_hat)
    x* = min_distance(x_hat)
```

Construction Heuristics

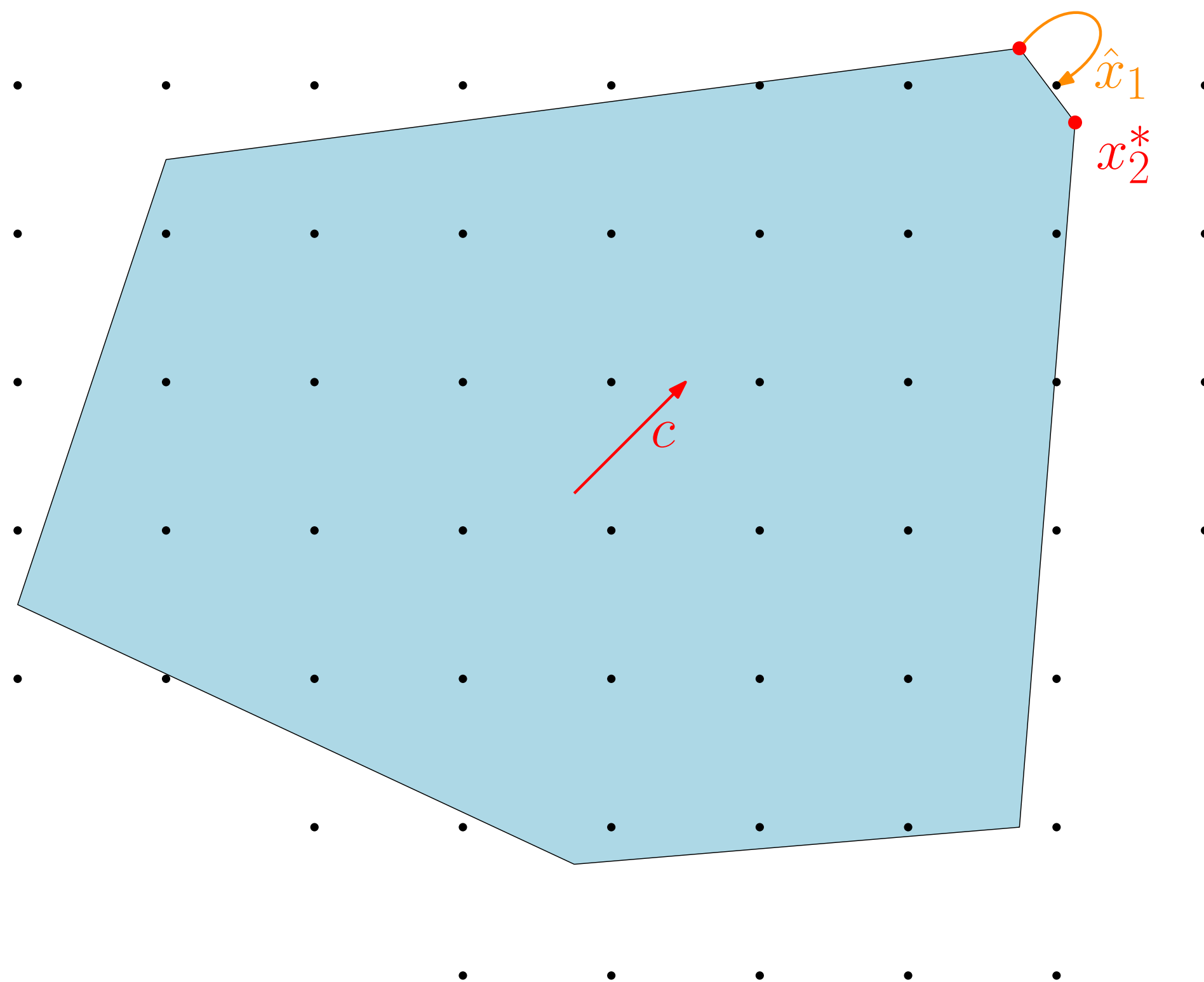
- Zero Heuristic (just set everything to 0), other simple greedy heuristics
- Zero Objective Heuristic (remove the objective, see if we easily find a solution)
- LP rounding (rounding to integers; differing degrees of 'cleverness'/propagation)
- Diving (rounding + resolving LP, potentially limited backtracking)
- Feasibility Pump (variant)



```
x* = solve_relaxation()
while not terminate():
    if is_integral(x*):
        return x*
    x-hat = round(x*)
    if detect_cycle(x-hat):
        perturb(x-hat)
    x* = min_distance(x-hat)
```

Construction Heuristics

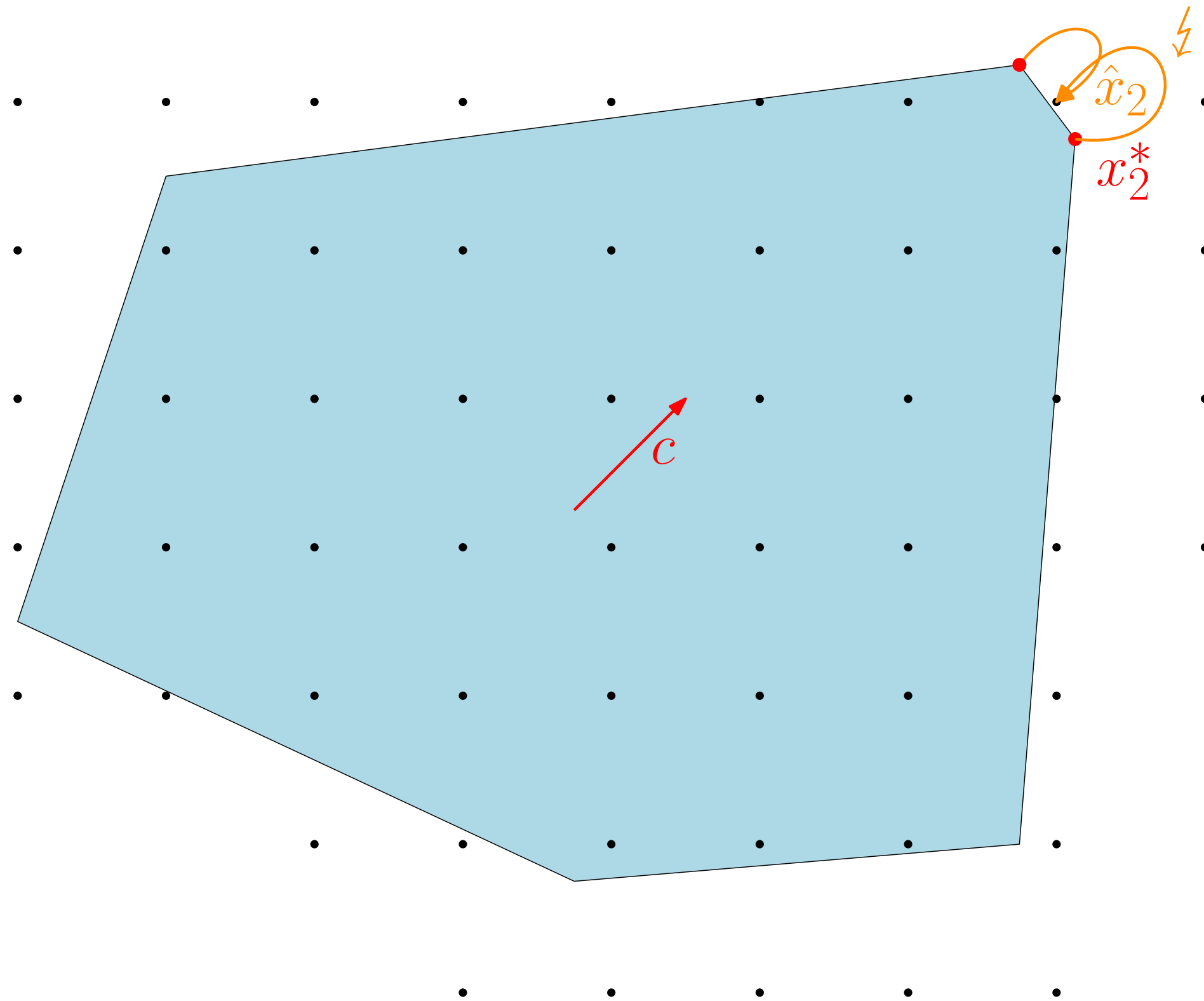
- Zero Heuristic (just set everything to 0), other simple greedy heuristics
- Zero Objective Heuristic (remove the objective, see if we easily find a solution)
- LP rounding (rounding to integers; differing degrees of 'cleverness'/propagation)
- Diving (rounding + resolving LP, potentially limited backtracking)
- Feasibility Pump (variant)



```
x* = solve_relaxation()
while not terminate():
    if is_integral(x*):
        return x*
    x_hat = round(x*)
    if detect_cycle(x_hat):
        perturb(x_hat)
    x* = min_distance(x_hat)
```

Construction Heuristics

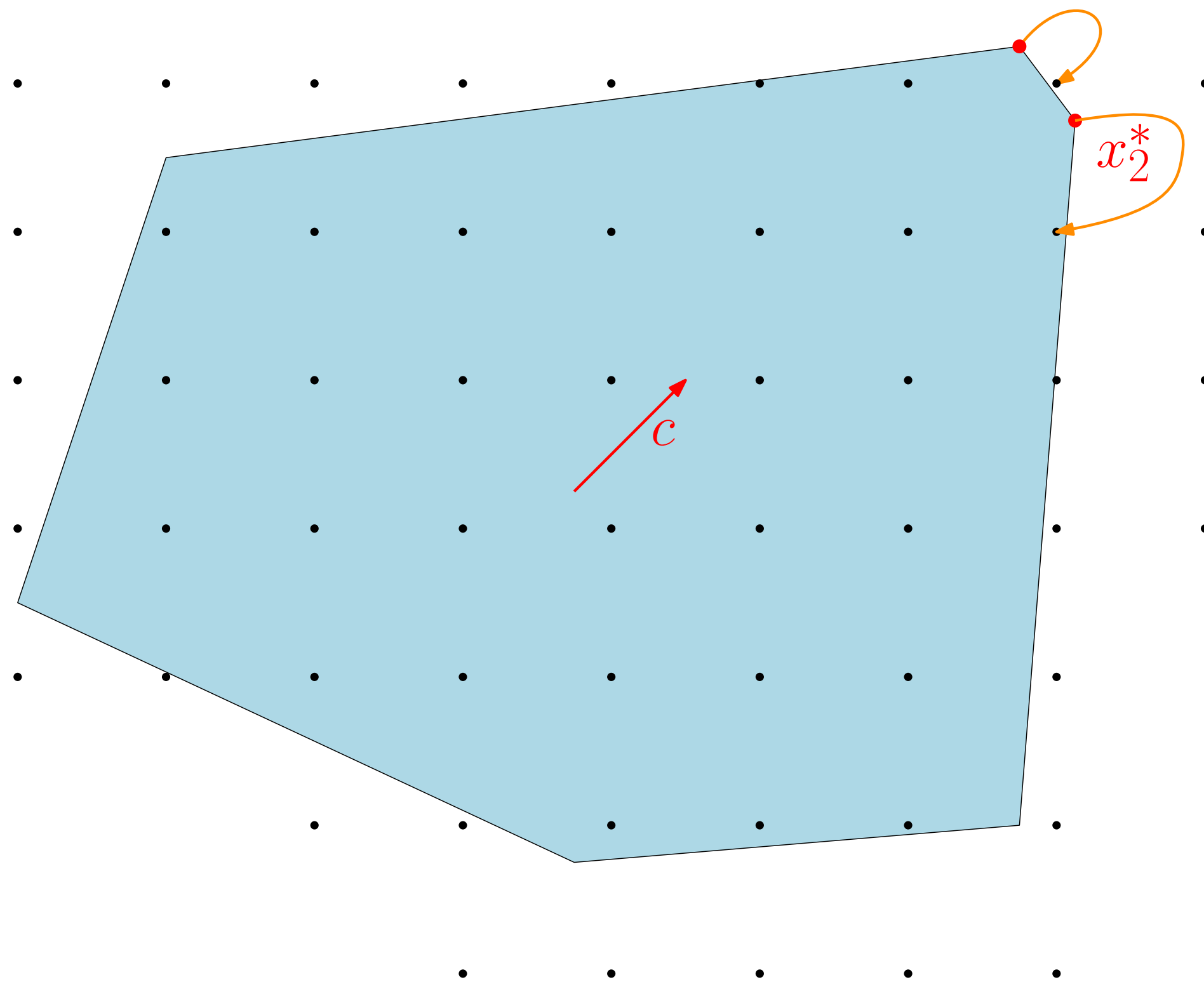
- Zero Heuristic (just set everything to 0), other simple greedy heuristics
- Zero Objective Heuristic (remove the objective, see if we easily find a solution)
- LP rounding (rounding to integers; differing degrees of ‘cleverness’/propagation)
- Diving (rounding + resolving LP, potentially limited backtracking)
- Feasibility Pump (variant)



```
x* = solve_relaxation()
while not terminate():
    if is_integral(x*):
        return x*
    x_hat = round(x*)
    if detect_cycle(x_hat):
        perturb(x_hat)
    x* = min_distance(x_hat)
```

Construction Heuristics

- Zero Heuristic (just set everything to 0), other simple greedy heuristics
- Zero Objective Heuristic (remove the objective, see if we easily find a solution)
- LP rounding (rounding to integers; differing degrees of 'cleverness'/propagation)
- Diving (rounding + resolving LP, potentially limited backtracking)
- Feasibility Pump (variant)



```
x* = solve_relaxation()
while not terminate():
    if is_integral(x*):
        return x*
    x_hat = round(x*)
    if detect_cycle(x_hat):
        perturb(x_hat)
    x* = min_distance(x_hat)
```

Improvement Heuristics

Sometimes also called **polishing**

Sometimes also called **polishing**

Relaxation-Induced Neighborhood Search (RINS)

Sometimes also called **polishing**

Relaxation-Induced Neighborhood Search (RINS)

- Take a solution (or solutions) and a relaxation; fix variables they agree on & propagate

Sometimes also called **polishing**

Relaxation-Induced Neighborhood Search (RINS)

- Take a solution (or solutions) and a relaxation; fix variables they agree on & propagate
- Solve (with a time budget) a sub-MIP on the remaining variables

Sometimes also called **polishing**

Relaxation-Induced Neighborhood Search (RINS)

- Take a solution (or solutions) and a relaxation; fix variables they agree on & propagate
- Solve (with a time budget) a sub-MIP on the remaining variables
- Without a solution: RENS (Relaxation Enforced Neighborhood Search, fix integers)

Sometimes also called **polishing**

Relaxation-Induced Neighborhood Search (RINS)

- Take a solution (or solutions) and a relaxation; fix variables they agree on & propagate
- Solve (with a time budget) a sub-MIP on the remaining variables
- Without a solution: RENS (Relaxation Enforced Neighborhood Search, fix integers)

Local Branching

Sometimes also called **polishing**

Relaxation-Induced Neighborhood Search (RINS)

- Take a solution (or solutions) and a relaxation; fix variables they agree on & propagate
- Solve (with a time budget) a sub-MIP on the remaining variables
- Without a solution: RENS (Relaxation Enforced Neighborhood Search, fix integers)

Local Branching

- Take a solution

Sometimes also called **polishing**

Relaxation-Induced Neighborhood Search (RINS)

- Take a solution (or solutions) and a relaxation; fix variables they agree on & propagate
- Solve (with a time budget) a sub-MIP on the remaining variables
- Without a solution: RENS (Relaxation Enforced Neighborhood Search, fix integers)

Local Branching

- Take a solution
- Add a constraint that at most k variables can change (for some suitably small k)

Sometimes also called **polishing**

Relaxation-Induced Neighborhood Search (RINS)

- Take a solution (or solutions) and a relaxation; fix variables they agree on & propagate
- Solve (with a time budget) a sub-MIP on the remaining variables
- Without a solution: RENS (Relaxation Enforced Neighborhood Search, fix integers)

Local Branching

- Take a solution
- Add a constraint that at most k variables can change (for some suitably small k)
- Solve as a sub-MIP (with a time budget)

Recap

- **Efficient LP solver**
 - Root relaxation
 - Later on: warm starting
- **Branch & Bound**
 - Exploration order
 - Branch variable selection
 - Propagation & conflict analysis
- **Cutting planes**
 - Cutting plane templates & heuristics
 - Cut management (Selection, Scoring, Forgetting, ...)
- **Presolve**
 - Problem size reduction (eliminating variables/constraints, ...)
 - Tightening (strengthening bounds, symmetry breaking, ...)
 - Numerical preprocessing (scaling, ...)
 - Exposing structure (helps cutting planes & heuristics, alternative solution strategies)
- **Primal heuristics**

What can you do to help

What can you do to help

- **Primal heuristics:** often the easiest addition (MIP start + callback)

What can you do to help

- **Primal heuristics:** often the easiest addition (MIP start + callback)
- **Cutting planes:**

What can you do to help

- **Primal heuristics:** often the easiest addition (MIP start + callback)
- **Cutting planes:**
 - Parameter tuning

What can you do to help

- **Primal heuristics:** often the easiest addition (MIP start + callback)
- **Cutting planes:**
 - Parameter tuning
 - Custom cutting planes at the root

What can you do to help

- **Primal heuristics:** often the easiest addition (MIP start + callback)
- **Cutting planes:**
 - Parameter tuning
 - Custom cutting planes at the root
 - Custom cutting planes in callback — has consequences for presolve