CS498: Algorithmic Engineering

Lecture 9: Large Scale Integer Linear Programs, Lazy Cuts, User Cuts.

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 05 – 02/17/2026

Outline

1. Recap: Branch and Bound & LP Relaxation Strength

2. Case Study (IP): Directed TSP, Lazy Constraints and User Cuts

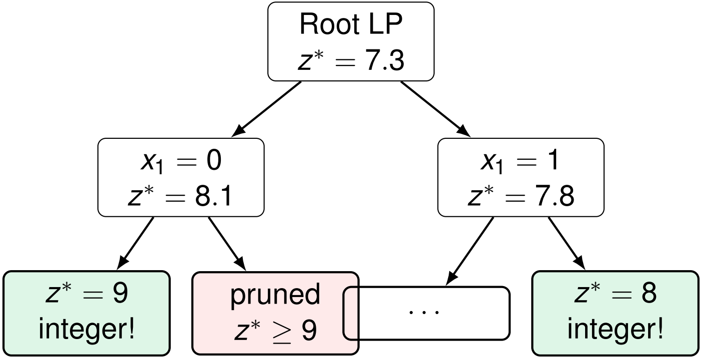
3. Practical Tips and Summary

Branch and Bound: The Big Picture

Recall from Lectures 5–7: Branch and Bound solves IPs by building a **search tree**.

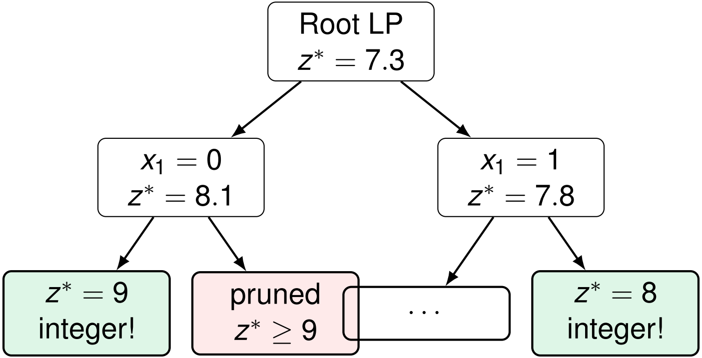
Branch and Bound: The Big Picture

Recall from Lectures 5–7: Branch and Bound solves IPs by building a **search tree**.



Branch and Bound: The Big Picture

Recall from Lectures 5–7: Branch and Bound solves IPs by building a **search tree**.



At **every node**: solve an LP relaxation to get a bound, then branch or prune.

LP Relaxation Quality Is Everything

For a minimization IP:

$$\text{LP}^* \leq \text{IP}^*$$

LP Relaxation Quality Is Everything

For a minimization IP:

$$\text{LP}^* \;\leq\; \text{IP}^*$$

The LP relaxation gives a **lower bound**. The closer it is to IP^*, the more nodes we can prune.

LP Relaxation Quality Is Everything

For a minimization IP:

$$\text{LP}^* \;\le\; \text{IP}^*$$

The LP relaxation gives a **lower bound**. The closer it is to IP*, the more nodes we can prune.

Strong Relaxation

- $\text{LP}^* \approx \text{IP}^*$

LP Relaxation Quality Is Everything

For a minimization IP:

$$\text{LP}^* \leq \text{IP}^*$$

The LP relaxation gives a **lower bound**. The closer it is to IP^*, the more nodes we can prune.

Strong Relaxation

- $\text{LP}^* \approx \text{IP}^*$
- Most branches pruned early

LP Relaxation Quality Is Everything

For a minimization IP:

$$\text{LP}^* \le \text{IP}^*$$

The LP relaxation gives a **lower bound**. The closer it is to IP^*, the more nodes we can prune.

Strong Relaxation

- $\text{LP}^* \approx \text{IP}^*$
- Most branches pruned early
- Small tree, fast solve

Weak Relaxation

- $\text{LP}^* \ll \text{IP}^*$

LP Relaxation Quality Is Everything

For a minimization IP:

$$\text{LP}^* \;\leq\; \text{IP}^*$$

The LP relaxation gives a **lower bound**. The closer it is to IP^*, the more nodes we can prune.

Strong Relaxation

- $\text{LP}^* \approx \text{IP}^*$
- Most branches pruned early
- Small tree, fast solve

Weak Relaxation

- $\text{LP}^* \ll \text{IP}^*$
- Can't prune anything

LP Relaxation Quality Is Everything

For a minimization IP:

$$\mathrm{LP}^* \leq \mathrm{IP}^*$$

The LP relaxation gives a **lower bound**. The closer it is to IP^*, the more nodes we can prune.

Strong Relaxation

- $\mathrm{LP}^* \approx \mathrm{IP}^*$
- Most branches pruned early
- Small tree, fast solve

Weak Relaxation

- $\mathrm{LP}^* \ll \mathrm{IP}^*$
- Can't prune anything
- Huge tree, slow solve

LP Relaxation Quality Is Everything

For a minimization IP:

$$\text{LP}^* \le \text{IP}^*$$

The LP relaxation gives a **lower bound**. The closer it is to IP^*, the more nodes we can prune.

Strong Relaxation

- $\text{LP}^* \approx \text{IP}^*$
- Most branches pruned early
- Small tree, fast solve

Weak Relaxation

- $\text{LP}^* \ll \text{IP}^*$
- Can't prune anything
- Huge tree, slow solve

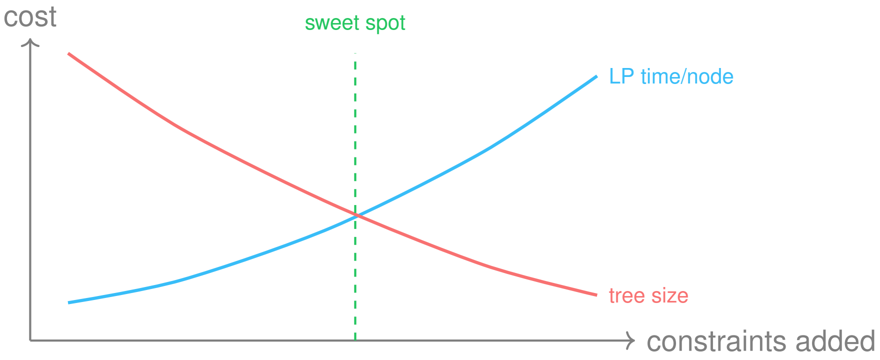
Takeaway: The quality of the LP relaxation largely determines how fast B&B converges.

The Formulation Strength Tradeoff

From Lecture 6 and Homeworks: we can **strengthen** the LP relaxation by adding more constraints (e.g., cover inequalities, lifted cuts, Sherali-Adams, etc).

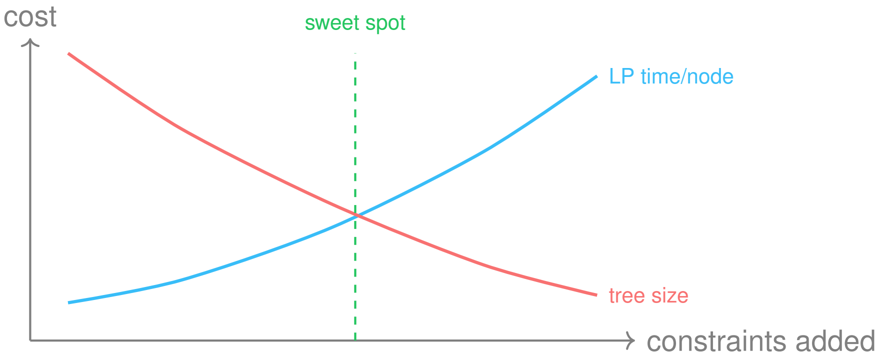
The Formulation Strength Tradeoff

From Lecture 6 and Homeworks: we can **strengthen** the LP relaxation by adding more constraints (e.g., cover inequalities, lifted cuts, Sherali-Adams, etc).



The Formulation Strength Tradeoff

From Lecture 6 and Homeworks: we can **strengthen** the LP relaxation by adding more constraints (e.g., cover inequalities, lifted cuts, Sherali-Adams, etc).



Tradeoff: More constraints $\Rightarrow$ tighter LP $\Rightarrow$ fewer nodes. But each LP solve takes longer.

The Bridge: Row Generation Inside Branch and Bound

In Lecture 8, we solved LPs with exponentially many constraints using **row generation**:

- Start with a small LP.

The Bridge: Row Generation Inside Branch and Bound

In Lecture 8, we solved LPs with exponentially many constraints using **row generation**:

- Start with a small LP.
- Find violated constraints via a **separation oracle**.

The Bridge: Row Generation Inside Branch and Bound

In Lecture 8, we solved LPs with exponentially many constraints using **row generation**:

- Start with a small LP.
- Find violated constraints via a **separation oracle**.
- Add only the violated ones.

The Bridge: Row Generation Inside Branch and Bound

In Lecture 8, we solved LPs with exponentially many constraints using **row generation**:

- Start with a small LP.
- Find violated constraints via a **separation oracle**.
- Add only the violated ones.

Key insight for today: Gurobi is already solving LP relaxations at every B&B node. We can inject constraints *during* the search.

The Bridge: Row Generation Inside Branch and Bound

In Lecture 8, we solved LPs with exponentially many constraints using **row generation**:

- Start with a small LP.
- Find violated constraints via a **separation oracle**.
- Add only the violated ones.

Key insight for today: Gurobi is already solving LP relaxations at every B&B node. We can inject constraints *during* the search.

Two moments where we can add rows:

The Bridge: Row Generation Inside Branch and Bound

In Lecture 8, we solved LPs with exponentially many constraints using **row generation**:

- Start with a small LP.
- Find violated constraints via a **separation oracle**.
- Add only the violated ones.

Key insight for today: Gurobi is already solving LP relaxations at every B&B node. We can inject constraints *during* the search.

Two moments where we can add rows:

(A) Integer incumbent found

Solution $x \in \{0, 1\}^n$.
Check: is it *truly feasible*?
If not $\Rightarrow$ add constraint via `cbLazy`.

The Bridge: Row Generation Inside Branch and Bound

In Lecture 8, we solved LPs with exponentially many constraints using **row generation**:

- Start with a small LP.
- Find violated constraints via a **separation oracle**.
- Add only the violated ones.

Key insight for today: Gurobi is already solving LP relaxations at every B&B node. We can inject constraints *during* the search.

Two moments where we can add rows:

(A) Integer incumbent found

Solution $x \in \{0, 1\}^n$.
Check: is it *truly feasible*?
If not $\Rightarrow$ add constraint via `cbLazy`.

(B) Fractional node LP solved

Solution $x \in [0, 1]^n$.
Check: any violated inequality?
If yes $\Rightarrow$ tighten via `cbCut`.

Directed TSP: Base Model (Degree Constraints)

Recall how we modelled the directed TSP problem.

Directed TSP: Base Model (Degree Constraints)

Recall how we modelled the directed TSP problem. Variables:

$$x_{ij} \in \{0, 1\} \qquad (i \neq j)$$

Directed TSP: Base Model (Degree Constraints)

Recall how we modelled the directed TSP problem. Variables:

$$x_{ij} \in \{0, 1\} \qquad (i \neq j)$$

Degree constraints:

$$\sum_{j \neq i} x_{ij} = 1, \quad \sum_{j \neq i} x_{ji} = 1 \quad \forall i \in V$$

Directed TSP: Base Model (Degree Constraints)

Recall how we modelled the directed TSP problem. Variables:

$$x_{ij} \in \{0, 1\} \qquad (i \neq j)$$

Degree constraints:

$$\sum_{j \neq i} x_{ij} = 1, \quad \sum_{j \neq i} x_{ji} = 1 \quad \forall i \in V$$

These enforce exactly 1-out / 1-in at every node.

Directed TSP: Base Model (Degree Constraints)

Recall how we modelled the directed TSP problem. Variables:

$$x_{ij} \in \{0, 1\} \qquad (i \neq j)$$

Degree constraints:

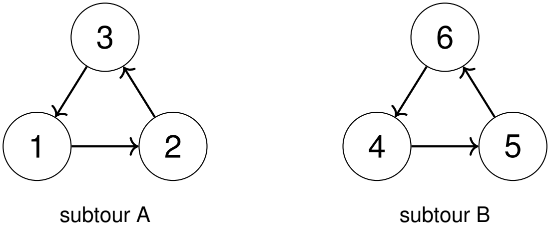
$$\sum_{j \neq i} x_{ij} = 1, \quad \sum_{j \neq i} x_{ji} = 1 \quad \forall i \in V$$

These enforce exactly 1-out / 1-in at every node.

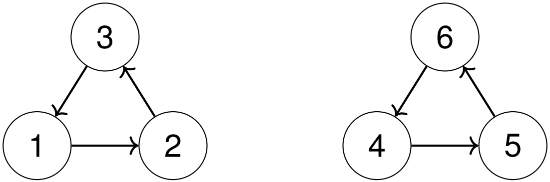
But

They do *not* enforce one single tour. They allow multiple disjoint cycles (subtours).

Picture: Degree Constraints Can Produce Multiple Cycles



Picture: Degree Constraints Can Produce Multiple Cycles



subtour A

subtour B

We need to **forbid subtours**.

Recall: MTZ Subtour Elimination (Lecture 7)

In Lecture 7, we eliminated subtours using **ordering variables**:

$$u_i \in [1, n], \qquad u_j \geq u_i + 1 - n(1 - x_{ij}) \quad \forall\, i \neq j,\ i, j \in \{2, \ldots, n\}$$

Recall: MTZ Subtour Elimination (Lecture 7)

In Lecture 7, we eliminated subtours using **ordering variables**:

$$u_i \in [1, n], \qquad u_j \geq u_i + 1 - n(1 - x_{ij}) \quad \forall\, i \neq j,\ i, j \in \{2, \ldots, n\}$$

Advantages:

- Only $O(n^2)$ constraints: polynomial, easy to write down.

Recall: MTZ Subtour Elimination (Lecture 7)

In Lecture 7, we eliminated subtours using **ordering variables**:

$$u_i \in [1, n], \qquad u_j \geq u_i + 1 - n(1 - x_{ij}) \quad \forall\, i \neq j,\ i, j \in \{2, \ldots, n\}$$

Advantages:

- Only $O(n^2)$ constraints: polynomial, easy to write down.
- Simple to implement in Gurobi (just write the constraints...).

Recall: MTZ Subtour Elimination (Lecture 7)

In Lecture 7, we eliminated subtours using **ordering variables**:

$$u_i \in [1, n], \qquad u_j \geq u_i + 1 - n(1 - x_{ij}) \quad \forall\, i \neq j,\; i, j \in \{2, \ldots, n\}$$

Advantages:

- Only $O(n^2)$ constraints: polynomial, easy to write down.
- Simple to implement in Gurobi (just write the constraints...).

So why not just use MTZ again?

The Problem with MTZ: Weak LP Relaxation

The LP relaxation of MTZ is **notoriously weak**.

The Problem with MTZ: Weak LP Relaxation

The LP relaxation of MTZ is **notoriously weak**.

MTZ Relaxation

- The $u_i \in [1, n]$ variables are continuous.

The Problem with MTZ: Weak LP Relaxation

The LP relaxation of MTZ is **notoriously weak**.

MTZ Relaxation

- The $u_i \in [1, n]$ variables are continuous.
- When x_{ij} is fractional, the Big-M constraint $u_j \geq u_i + 1 - n(1 - x_{ij})$ becomes very loose.

The Problem with MTZ: Weak LP Relaxation

The LP relaxation of MTZ is **notoriously weak**.

MTZ Relaxation

- The $u_i \in [1, n]$ variables are continuous.
- When x_{ij} is fractional, the Big-M constraint $u_j \geq u_i + 1 - n(1 - x_{ij})$ becomes very loose.
- LP bound is far below IP optimum.

The Problem with MTZ: Weak LP Relaxation

The LP relaxation of MTZ is **notoriously weak**.

MTZ Relaxation

- The $u_i \in [1, n]$ variables are continuous.
- When x_{ij} is fractional, the Big-M constraint $u_j \geq u_i + 1 - n(1 - x_{ij})$ becomes very loose.
- LP bound is far below IP optimum.

Consequence for B&B

- Weak lower bounds $\Rightarrow$ little to no pruning.

The Problem with MTZ: Weak LP Relaxation

The LP relaxation of MTZ is **notoriously weak**.

MTZ Relaxation

- The $u_i \in [1, n]$ variables are continuous.
- When x_{ij} is fractional, the Big-M constraint $u_j \geq u_i + 1 - n(1 - x_{ij})$ becomes very loose.
- LP bound is far below IP optimum.

Consequence for B&B

- Weak lower bounds $\Rightarrow$ little to no pruning.
- Solver explores a big tree.

The Problem with MTZ: Weak LP Relaxation

The LP relaxation of MTZ is **notoriously weak**.

MTZ Relaxation

- The $u_i \in [1, n]$ variables are continuous.
- When x_{ij} is fractional, the Big-M constraint $u_j \geq u_i + 1 - n(1 - x_{ij})$ becomes very loose.
- LP bound is far below IP optimum.

Consequence for B&B

- Weak lower bounds $\Rightarrow$ little to no pruning.
- Solver explores a big tree.
- Scales poorly on **hard instances**.

The Problem with MTZ: Weak LP Relaxation

The LP relaxation of MTZ is **notoriously weak**.

MTZ Relaxation

- The $u_i \in [1, n]$ variables are continuous.
- When x_{ij} is fractional, the Big-M constraint $u_j \geq u_i + 1 - n(1 - x_{ij})$ becomes very loose.
- LP bound is far below IP optimum.

Consequence for B&B

- Weak lower bounds $\Rightarrow$ little to no pruning.
- Solver explores a big tree.
- Scales poorly on **hard instances**.

We need a formulation whose LP relaxation is **much tighter**.

DFJ: Stronger but Exponential

The **Dantzig–Fulkerson–Johnson** (DFJ) formulation uses a different set of subtour elimination constraints:

DFJ: Stronger but Exponential

The **Dantzig–Fulkerson–Johnson** (DFJ) formulation uses a different set of subtour elimination constraints:

	MTZ	**DFJ**
Extra variables	u_i (continuous)	none
# constraints	$O(n^2)$	$2^n - 2$
LP relaxation	weak	**much tighter**
Implementation	direct	requires **row generation**

DFJ: Stronger but Exponential

The **Dantzig–Fulkerson–Johnson** (DFJ) formulation uses a different set of subtour elimination constraints:

	MTZ	**DFJ**
Extra variables	u_i (continuous)	none
# constraints	$O(n^2)$	$2^n - 2$
LP relaxation	weak	**much tighter**
Implementation	direct	requires **row generation**

DFJ gives a dramatically better LP relaxation, but we cannot write down all $O(2^n)$ constraints.

DFJ: Stronger but Exponential

The **Dantzig–Fulkerson–Johnson** (DFJ) formulation uses a different set of subtour elimination constraints:

	MTZ	**DFJ**
Extra variables	u_i (continuous)	none
# constraints	$O(n^2)$	$2^n - 2$
LP relaxation	weak	**much tighter**
Implementation	direct	requires **row generation**

DFJ gives a dramatically better LP relaxation, but we cannot write down all $O(2^n)$ constraints.

The deal: we trade a compact-but-weak LP formulation for a tight-but-huge one, and use **row generation** to make it tractable.

DFJ Subtour Elimination (Directed)

For every nonempty proper subset $S \subset V$:

$$\sum_{i \in S, j \notin S} x_{ij} \geq 1$$

DFJ Subtour Elimination (Directed)

For every nonempty proper subset $S \subset V$:

$$\sum_{i \in S,\, j \notin S} x_{ij} \geq 1$$

Interpretation: every subset S must have **at least one arc leaving**. This kills subtours.

DFJ Subtour Elimination (Directed)

For every nonempty proper subset $S \subset V$:

$$\sum_{i \in S, j \notin S} x_{ij} \geq 1$$

Interpretation: every subset S must have **at least one arc leaving**. This kills subtours.

The problem

There are $2^n - 2$ such subsets. We cannot write them all down.

DFJ Subtour Elimination (Directed)

For every nonempty proper subset $S \subset V$:

$$\sum_{i \in S, j \notin S} x_{ij} \geq 1$$

Interpretation: every subset S must have **at least one arc leaving**. This kills subtours.

The problem

There are $2^n - 2$ such subsets. We cannot write them all down.

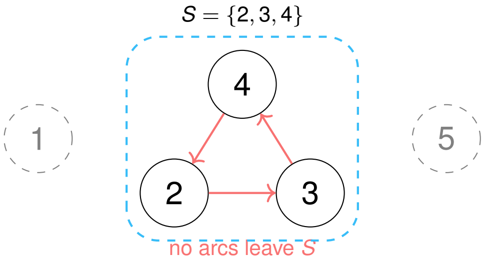
So we will **generate them on demand**. But how?

Why DFJ Kills Subtours

Suppose a solution contains a subtour on nodes $S = \{2, 3, 4\}$:

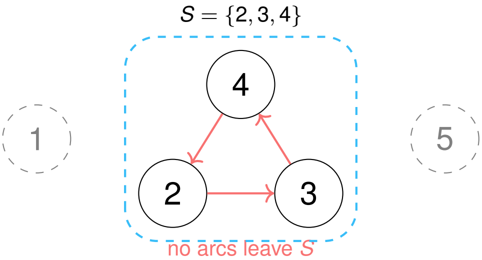
Why DFJ Kills Subtours

Suppose a solution contains a subtour on nodes $S = \{2, 3, 4\}$:



Why DFJ Kills Subtours

Suppose a solution contains a subtour on nodes $S = \{2, 3, 4\}$:

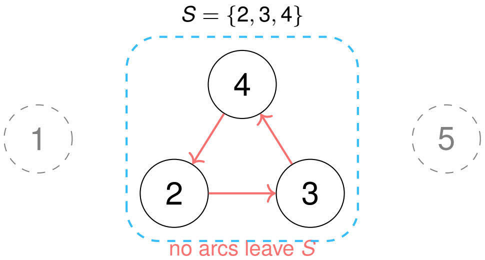


The DFJ constraint for this S requires:

$$\sum_{i \in S, j \notin S} x_{ij} \geq 1$$

Why DFJ Kills Subtours

Suppose a solution contains a subtour on nodes $S = \{2, 3, 4\}$:



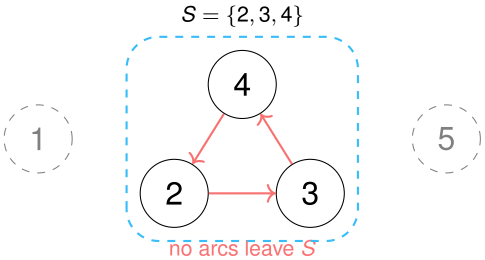
The DFJ constraint for this S requires:

$$\sum_{i \in S, j \notin S} x_{ij} \geq 1$$

But a subtour uses **zero** arcs leaving S, so the LHS $= 0 < 1$. Violated!

Why DFJ Kills Subtours

Suppose a solution contains a subtour on nodes $S = \{2, 3, 4\}$:



The DFJ constraint for this S requires:

$$\sum_{i \in S, j \notin S} x_{ij} \geq 1$$

But a subtour uses **zero** arcs leaving S, so the LHS $= 0 < 1$. Violated!

A single Hamiltonian tour always has at least one arc leaving every proper subset $\Rightarrow$ satisfied.

The Full IP with DFJ (In Theory)

If we could write it all down, the IP would be:

$$
\begin{aligned}
\min \quad & \sum_{i \neq j} c_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{j \neq i} x_{ij} = 1 && \forall i \in V \\
& \sum_{j \neq i} x_{ji} = 1 && \forall i \in V \\
& \sum_{i \in S, j \notin S} x_{ij} \geq 1 && \forall \emptyset \subset S \subset V \\
& x_{ij} \in \{0, 1\}
\end{aligned}
$$

The Full IP with DFJ (In Theory)

If we could write it all down, the IP would be:

$$
\begin{aligned}
\min \quad & \sum_{i \neq j} c_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{j \neq i} x_{ij} = 1 && \forall i \in V \\
& \sum_{j \neq i} x_{ji} = 1 && \forall i \in V \\
& \sum_{i \in S, j \notin S} x_{ij} \geq 1 && \forall \emptyset \subset S \subset V \\
& x_{ij} \in \{0, 1\}
\end{aligned}
$$

This is a **correct and tight** formulation. But with $2^n - 2$ subtour constraints.

The Full IP with DFJ (In Theory)

If we could write it all down, the IP would be:

$$
\begin{aligned}
\min \quad & \sum_{i \neq j} c_{ij} x_{ij} \\
\text{s.t.} \quad & \sum_{j \neq i} x_{ij} = 1 && \forall i \in V \\
& \sum_{j \neq i} x_{ji} = 1 && \forall i \in V \\
& \sum_{i \in S, j \notin S} x_{ij} \geq 1 && \forall \emptyset \subset S \subset V \\
& x_{ij} \in \{0, 1\}
\end{aligned}
$$

This is a **correct and tight** formulation. But with $2^n - 2$ subtour constraints.

The Lecture 8 approach: row generation, start small, add violated constraints iteratively. But now we are inside an IP, not just an LP...

From LP Row Generation to IP Row Generation

In Lecture 8 (min s–t cut), we had:

- An LP with exponentially many constraints.

From LP Row Generation to IP Row Generation

In Lecture 8 (min s–t cut), we had:

- An LP with exponentially many constraints.
- We solved a restricted LP, found violated constraints, added them, and repeated.

From LP Row Generation to IP Row Generation

In Lecture 8 (min s–t cut), we had:

- An LP with exponentially many constraints.
- We solved a restricted LP, found violated constraints, added them, and repeated.

Now the situation is different:

- We have an **IP** with exponentially many constraints.

From LP Row Generation to IP Row Generation

In Lecture 8 (min s–t cut), we had:

- An LP with exponentially many constraints.
- We solved a restricted LP, found violated constraints, added them, and repeated.

Now the situation is different:

- We have an **IP** with exponentially many constraints.
- Gurobi is running **Branch and Bound**, solving LP relaxations at every node.

From LP Row Generation to IP Row Generation

In Lecture 8 (min s–t cut), we had:

- An LP with exponentially many constraints.
- We solved a restricted LP, found violated constraints, added them, and repeated.

Now the situation is different:

- We have an **IP** with exponentially many constraints.
- Gurobi is running **Branch and Bound**, solving LP relaxations at every node.
- We cannot just do an outer loop, we need to inject constraints **inside** the B&B process.

From LP Row Generation to IP Row Generation

In Lecture 8 (min s–t cut), we had:

- An LP with exponentially many constraints.
- We solved a restricted LP, found violated constraints, added them, and repeated.

Now the situation is different:

- We have an **IP** with exponentially many constraints.
- Gurobi is running **Branch and Bound**, solving LP relaxations at every node.
- We cannot just do an outer loop, we need to inject constraints **inside** the B&B process.

Gurobi's mechanism: callbacks, or functions we write that Gurobi calls at specific moments during the solve.

Two Strategies for On-Demand Generation

From Lecture 8, we know how to do row generation for LPs.

Two Strategies for On-Demand Generation

From Lecture 8, we know how to do row generation for LPs.

Inside a MIP solver, we get **two** opportunities:

Two Strategies for On-Demand Generation

From Lecture 8, we know how to do row generation for LPs.

Inside a MIP solver, we get **two** opportunities:

Strategy 1 (This section): Only check **integer** solutions.

- Sufficient for **correctness**.
- Uses `cbLazy`.

Two Strategies for On-Demand Generation

From Lecture 8, we know how to do row generation for LPs.

Inside a MIP solver, we get **two** opportunities:

Strategy 1 (This section): Only check **integer** solutions.

- Sufficient for **correctness**.
- Uses `cbLazy`.

Strategy 2 (Next section): Also check **fractional** LP solutions.

- Improves **performance** by tightening the relaxation.
- Uses `cbCut`.

Two Strategies for On-Demand Generation

From Lecture 8, we know how to do row generation for LPs.

Inside a MIP solver, we get **two** opportunities:

Strategy 1 (This section): Only check **integer** solutions.

- Sufficient for **correctness**.
- Uses `cbLazy`.

Strategy 2 (Next section): Also check **fractional** LP solutions.

- Improves **performance** by tightening the relaxation.
- Uses `cbCut`.

Let's start with Strategy 1: getting a **correct** solution first.

The Deal: Being Lazy with Gurobi

Imagine walking up to Gurobi and saying:

The Deal: Being Lazy with Gurobi

Imagine walking up to Gurobi and saying:

> *"Hey Gurobi, I have this TSP model. The full formulation has* $O(2^n)$ *constraints, but I'm too lazy to write them all. Here's the deal:*

The Deal: Being Lazy with Gurobi

Imagine walking up to Gurobi and saying:

> *"Hey Gurobi, I have this TSP model. The full formulation has $O(2^n)$ constraints, but I'm too lazy to write them all. Here's the deal:*
>
> *I'll give you just the degree constraints. You go ahead and run Branch and Bound.*

The Deal: Being Lazy with Gurobi

Imagine walking up to Gurobi and saying:

> *"Hey Gurobi, I have this TSP model. The full formulation has $O(2^n)$ constraints, but I'm too lazy to write them all. Here's the deal:*
>
> *I'll give you just the degree constraints. You go ahead and run Branch and Bound.*
>
> *Every time you think you've found an integer solution, show it to me first. If it's a valid tour, great, keep it. If it has subtours, I'll tell you which constraint you're violating, and you go back to work."*

The Deal: Being Lazy with Gurobi

Imagine walking up to Gurobi and saying:

> *"Hey Gurobi, I have this TSP model. The full formulation has $O(2^n)$ constraints, but I'm too lazy to write them all. Here's the deal:*
>
> *I'll give you just the degree constraints. You go ahead and run Branch and Bound.*
>
> *Every time you think you've found an integer solution, show it to me first. If it's a valid tour, great, keep it. If it has subtours, I'll tell you which constraint you're violating, and you go back to work."*

This is **exactly** what `cbLazy` does. You are literally being "lazy" about the constraints, only providing them when Gurobi asks.

The Deal: Being Lazy with Gurobi

Imagine walking up to Gurobi and saying:

> *"Hey Gurobi, I have this TSP model. The full formulation has $O(2^n)$ constraints, but I'm too lazy to write them all. Here's the deal:*
>
> *I'll give you just the degree constraints. You go ahead and run Branch and Bound.*
>
> *Every time you think you've found an integer solution, show it to me first. If it's a valid tour, great, keep it. If it has subtours, I'll tell you which constraint you're violating, and you go back to work."*

This is **exactly** what `cbLazy` does. You are literally being "lazy" about the constraints, only providing them when Gurobi asks.

And Gurobi is surprisingly okay with this arrangement. It just needs you to set one flag: `LazyConstraints = 1`.

The Idea: Check Integer Solutions for Subtours

During Branch and Bound, Gurobi periodically finds **integer** candidate solutions (incumbents).

The Idea: Check Integer Solutions for Subtours

During Branch and Bound, Gurobi periodically finds **integer** candidate solutions (incumbents).

Normally, Gurobi would accept these and update the best known solution.

The Idea: Check Integer Solutions for Subtours

During Branch and Bound, Gurobi periodically finds **integer** candidate solutions (incumbents).

Normally, Gurobi would accept these and update the best known solution.

Our intervention: Before Gurobi accepts an incumbent, we check:

1. Does this integer solution form a **single tour**?

The Idea: Check Integer Solutions for Subtours

During Branch and Bound, Gurobi periodically finds **integer** candidate solutions (incumbents).

Normally, Gurobi would accept these and update the best known solution.

Our intervention: Before Gurobi accepts an incumbent, we check:

1. Does this integer solution form a **single tour**? $\Rightarrow$ Accept it.

The Idea: Check Integer Solutions for Subtours

During Branch and Bound, Gurobi periodically finds **integer** candidate solutions (incumbents).

Normally, Gurobi would accept these and update the best known solution.

Our intervention: Before Gurobi accepts an incumbent, we check:

1. Does this integer solution form a **single tour**? $\Rightarrow$ Accept it.
2. Does it contain **subtours**?

The Idea: Check Integer Solutions for Subtours

During Branch and Bound, Gurobi periodically finds **integer** candidate solutions (incumbents).

Normally, Gurobi would accept these and update the best known solution.

Our intervention: Before Gurobi accepts an incumbent, we check:

1. Does this integer solution form a **single tour**? $\Rightarrow$ Accept it.
2. Does it contain **subtours**? $\Rightarrow$ Reject it by adding the violated DFJ constraint.

The Idea: Check Integer Solutions for Subtours

During Branch and Bound, Gurobi periodically finds **integer** candidate solutions (incumbents).

Normally, Gurobi would accept these and update the best known solution.

Our intervention: Before Gurobi accepts an incumbent, we check:

1. Does this integer solution form a **single tour**? $\Rightarrow$ Accept it.
2. Does it contain **subtours**? $\Rightarrow$ Reject it by adding the violated DFJ constraint.

This is exactly the `cbLazy` mechanism.

Separation Oracle for Integer Solutions (Easy!)

Given an integer solution $x^* \in \{0, 1\}^{|E|}$, how do we detect if there is a subtour or not?

Separation Oracle for Integer Solutions (Easy!)

Given an integer solution $x^* \in \{0, 1\}^{|E|}$, how do we detect if there is a subtour or not?

1. Build a graph using only edges where $x^*_{ij} = 1$.

Separation Oracle for Integer Solutions (Easy!)

Given an integer solution $x^* \in \{0, 1\}^{|E|}$, how do we detect if there is a subtour or not?

1. Build a graph using only edges where $x^*_{ij} = 1$.
2. Find **strongly connected components** (DFS).

Separation Oracle for Integer Solutions (Easy!)

Given an integer solution $x^* \in \{0, 1\}^{|E|}$, how do we detect if there is a subtour or not?

1. Build a graph using only edges where $x^*_{ij} = 1$.
2. Find **strongly connected components** (DFS).
3. If there is more than one component $\Rightarrow$ subtours exist.

Separation Oracle for Integer Solutions (Easy!)

Given an integer solution $x^* \in \{0, 1\}^{|E|}$, how do we detect if there is a subtour or not?

1. Build a graph using only edges where $x^*_{ij} = 1$.
2. Find **strongly connected components** (DFS).
3. If there is more than one component $\Rightarrow$ subtours exist.

For each subtour component S with $|S| < n$:

$$\sum_{i \in S, j \notin S} x_{ij} \geq 1$$

is violated (the LHS is 0, since no arc leaves S).

Separation Oracle for Integer Solutions (Easy!)

Given an integer solution $x^* \in \{0,1\}^{|E|}$, how do we detect if there is a subtour or not?

1. Build a graph using only edges where $x^*_{ij} = 1$.
2. Find **strongly connected components** (DFS).
3. If there is more than one component $\Rightarrow$ subtours exist.

For each subtour component S with $|S| < n$:

$$\sum_{i \in S, j \notin S} x_{ij} \geq 1$$

is violated (the LHS is 0, since no arc leaves S).

Complexity: $O(n+m)$: linear time. Very fast.

Gurobi Callbacks: How They Work

A **callback** is a Python function you write that Gurobi calls at specific moments during the solve.

Gurobi Callbacks: How They Work

A **callback** is a Python function you write that Gurobi calls at specific moments during the solve.

```
def callback(model, where):
    # 'where' tells you WHAT just happened inside the solver
    if where == GRB.Callback.MIPSOL:
        # An integer solution was just found!
        ...
    elif where == GRB.Callback.MIPNODE:
        # A node LP relaxation was just solved!
        ...

m.optimize(callback)   # pass your function to optimize()
```

Gurobi Callbacks: How They Work

A **callback** is a Python function you write that Gurobi calls at specific moments during the solve.

```python
def callback(model, where):
    # 'where' tells you WHAT just happened inside the solver
    if where == GRB.Callback.MIPSOL:
        # An integer solution was just found!
        ...
    elif where == GRB.Callback.MIPNODE:
        # A node LP relaxation was just solved!
        ...

m.optimize(callback)   # pass your function to optimize()
```

Key idea: `where` is an event type. You react only to the events you care about.

The `MIPSOL` Event: An Integer Solution Was Found

When `where == GRB.Callback.MIPSOL`:

The `MIPSOL` Event: An Integer Solution Was Found

When `where == GRB.Callback.MIPSOL`:

What you can read:

```
# The integer solution values
xsol = model.cbGetSolution(model._x)

# Objective value of this incumbent
obj = model.cbGet(GRB.Callback.MIPSOL_OBJ)

# Which B&B node found it
node = model.cbGet(GRB.Callback.MIPSOL_NODCNT)

# Number of solutions found so far
solcnt = model.cbGet(GRB.Callback.MIPSOL_SOLCNT)
```

The `MIPSOL` Event: An Integer Solution Was Found

When `where == GRB.Callback.MIPSOL`:

What you can read:

```
# The integer solution values
xsol = model.cbGetSolution(model._x)

# Objective value of this incumbent
obj = model.cbGet(GRB.Callback.MIPSOL_OBJ)

# Which B&B node found it
node = model.cbGet(GRB.Callback.MIPSOL_NODCNT)

# Number of solutions found so far
solcnt = model.cbGet(GRB.Callback.MIPSOL_SOLCNT)
```

What you can do:

```
# Reject this solution by adding a constraint
model.cbLazy( <linear expression> >= <rhs> )
```

Subtour Detection Code

```python
import networkx as nx

def find_subtours(n, xsol):
    """Given integer solution (all 0.0 or 1.0), return list of subtour node sets."""
    G = nx.DiGraph()
    G.add_nodes_from(range(n))

    for i in range(n):
        for j in range(n):
            if i != j and int(round(xsol[i, j]))==1:  # round 1.0 to 1 and 0.0 to 0
                G.add_edge(i, j)

    components = list(nx.strongly_connected_components(G))

    # A subtour is any component smaller than n
    return [S for S in components if len(S) < n]
```

Subtour Detection Code

```python
import networkx as nx

def find_subtours(n, xsol):
    """Given integer solution (all 0.0 or 1.0), return list of subtour node sets."""
    G = nx.DiGraph()
    G.add_nodes_from(range(n))

    for i in range(n):
        for j in range(n):
            if i != j and int(round(xsol[i, j]))==1:  # round 1.0 to 1 and 0.0 to 0
                G.add_edge(i, j)

    components = list(nx.strongly_connected_components(G))

    # A subtour is any component smaller than n
    return [S for S in components if len(S) < n]
```

If `find_subtours` returns an empty list ⇒ the solution is a valid tour.

The cbLazy Callback

```python
def callback(model, where):
    if where == GRB.Callback.MIPSOL:
        x = model._x
        n = model._n

        # 1. Read the integer solution
        xsol = model.cbGetSolution(x)

        # 2. Find subtours
        subtours = find_subtours(n, xsol)

        # 3. For each subtour, add a lazy constraint
        for S in subtours:
            complement = set(range(n)) - S
            model.cbLazy(
                gp.quicksum(x[i,j] for i in S for j in complement) >= 1
            )
```

The cbLazy Callback

```python
def callback(model, where):
    if where == GRB.Callback.MIPSOL:
        x = model._x
        n = model._n

        # 1. Read the integer solution
        xsol = model.cbGetSolution(x)

        # 2. Find subtours
        subtours = find_subtours(n, xsol)

        # 3. For each subtour, add a lazy constraint
        for S in subtours:
            complement = set(range(n)) - S
            model.cbLazy(
                gp.quicksum(x[i,j] for i in S for j in complement) >= 1
            )
```

Each `cbLazy` call adds one DFJ inequality. Gurobi rejects the incumbent and continues searching.

Building the Model (cbLazy Only)

```
import gurobipy as gp
from gurobipy import GRB

m = gp.Model("TSP_lazy_only")
x = m.addVars(V, V, vtype=GRB.BINARY, name="x")
for i in V:
    x[i,i].ub = 0   # no self-loops

m.setObjective(
    gp.quicksum(c[i,j]*x[i,j] for i in V for j in V if i != j),
    GRB.MINIMIZE)

for i in V:
    m.addConstr(gp.quicksum(x[i,j] for j in V if j != i) == 1)
    m.addConstr(gp.quicksum(x[j,i] for j in V if j != i) == 1)

m.Params.LazyConstraints = 1   # REQUIRED: tells Gurobi to expect cbLazy

m._x = x    # attach variables so callback can access them
m._n = n
m.optimize(callback)
```

Important: `LazyConstraints = 1`

If you forget this parameter

```
# m.Params.LazyConstraints = 1   # OOPS, commented out!
m.optimize(callback)
```

Gurobi will **silently ignore** all `cbLazy` calls. Your model will return solutions with subtours.

Important: `LazyConstraints = 1`

If you forget this parameter

```
# m.Params.LazyConstraints = 1   # OOPS, commented out!
m.optimize(callback)
```

Gurobi will **silently ignore** all `cbLazy` calls. Your model will return solutions with subtours.

Why does this parameter exist?

Important: `LazyConstraints = 1`

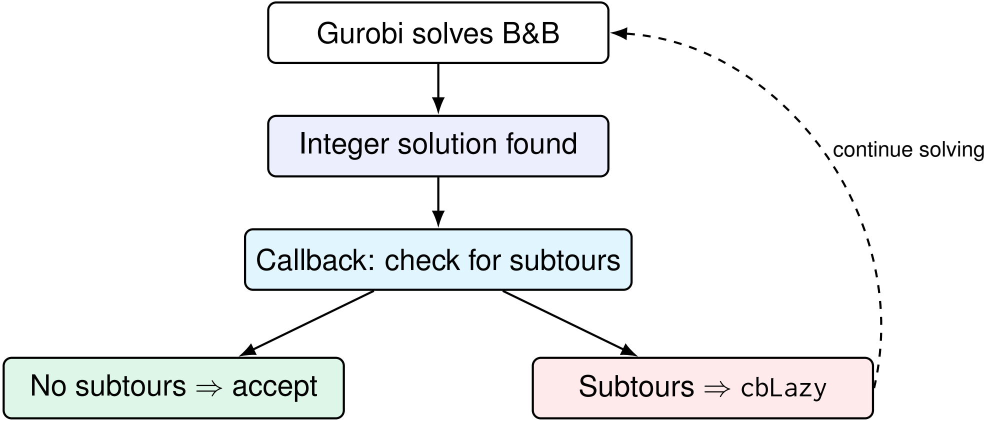
If you forget this parameter

```
# m.Params.LazyConstraints = 1   # OOPS, commented out!
m.optimize(callback)
```

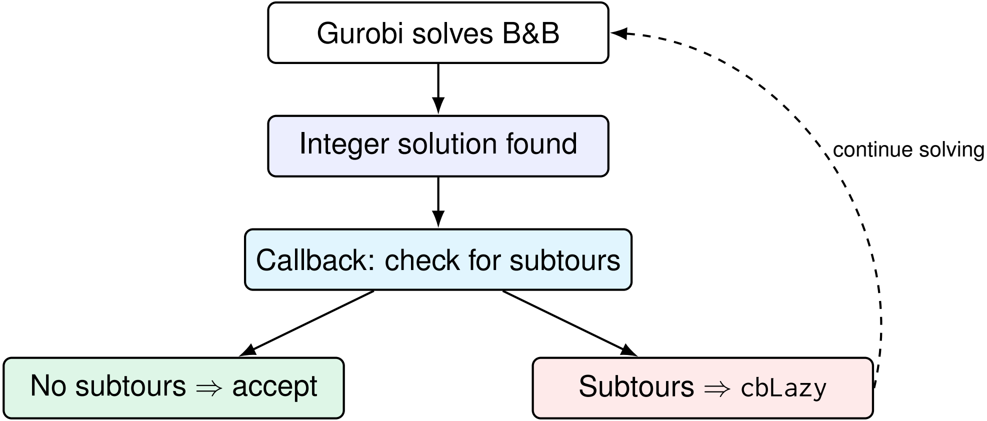
Gurobi will **silently ignore** all `cbLazy` calls. Your model will return solutions with subtours.

Why does this parameter exist?
When lazy constraints are enabled, Gurobi cannot use certain presolve reductions and heuristics that assume the model is complete. So it needs to know *in advance* that you plan to add constraints during the solve.

cbLazy: What Happens at Runtime

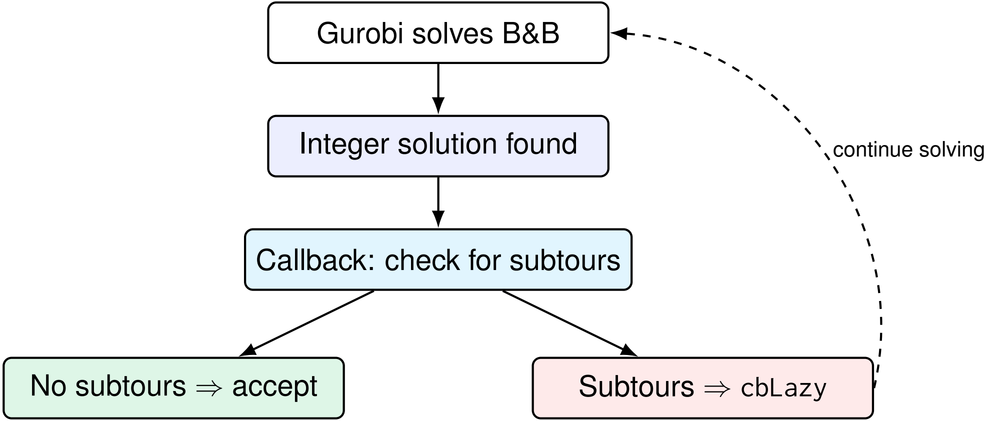


cbLazy: What Happens at Runtime



This is sufficient for **correctness**: Gurobi will never return a solution with subtours.

cbLazy: What Happens at Runtime



This is sufficient for **correctness**: Gurobi will never return a solution with subtours.

But is the solver **efficient**?

The Problem with cbLazy Alone

With only degree constraints in the model, the **LP relaxation is very weak**.

The Problem with cbLazy Alone

With only degree constraints in the model, the **LP relaxation is very weak**.

Consequence:

- LP bounds are weak $\Rightarrow$ almost no pruning.

The Problem with cbLazy Alone

With only degree constraints in the model, the **LP relaxation is very weak**.

Consequence:

- LP bounds are weak $\Rightarrow$ almost no pruning.
- Gurobi explores a **massive** B&B tree.

The Problem with cbLazy Alone

With only degree constraints in the model, the **LP relaxation is very weak**.

Consequence:

- LP bounds are weak $\Rightarrow$ almost no pruning.
- Gurobi explores a **massive** B&B tree.
- The solver finds many integer incumbents with subtours, rejects them via `cbLazy`, but makes slow progress.

Can We Do Better?

Observation: The DFJ constraints are valid for *all* feasible tours, not just integer ones.

Can We Do Better?

Observation: The DFJ constraints are valid for *all* feasible tours, not just integer ones.

So if a **fractional** LP solution at a B&B node violates a DFJ constraint, we can add it right there to tighten the relaxation.

Can We Do Better?

Observation: The DFJ constraints are valid for *all* feasible tours, not just integer ones.

So if a **fractional** LP solution at a B&B node violates a DFJ constraint, we can add it right there to tighten the relaxation.

Why this helps

- Tighter LP relaxation $\Rightarrow$ better bounds at every node.

Can We Do Better?

Observation: The DFJ constraints are valid for *all* feasible tours, not just integer ones.

So if a **fractional** LP solution at a B&B node violates a DFJ constraint, we can add it right there to tighten the relaxation.

Why this helps

- Tighter LP relaxation $\Rightarrow$ better bounds at every node.
- Better bounds $\Rightarrow$ more pruning $\Rightarrow$ smaller tree.

Can We Do Better?

Observation: The DFJ constraints are valid for *all* feasible tours, not just integer ones.

So if a **fractional** LP solution at a B&B node violates a DFJ constraint, we can add it right there to tighten the relaxation.

Why this helps

- Tighter LP relaxation $\Rightarrow$ better bounds at every node.
- Better bounds $\Rightarrow$ more pruning $\Rightarrow$ smaller tree.
- We are strengthening the relaxation **during** the solve.

Can We Do Better?

Observation: The DFJ constraints are valid for *all* feasible tours, not just integer ones.

So if a **fractional** LP solution at a B&B node violates a DFJ constraint, we can add it right there to tighten the relaxation.

Why this helps

- Tighter LP relaxation $\Rightarrow$ better bounds at every node.
- Better bounds $\Rightarrow$ more pruning $\Rightarrow$ smaller tree.
- We are strengthening the relaxation **during** the solve.

This is the `cbCut` mechanism: adding **user cuts** at fractional node LP solutions.

cbLazy vs cbCut: Two Different Jobs

`cbLazy` (Correctness)

- Triggered at **integer** solutions (`MIPSOL` event)

cbLazy vs cbCut: Two Different Jobs

`cbLazy` (Correctness)

- Triggered at **integer** solutions (`MIPSOL` event)
- Purpose: reject invalid incumbents

cbLazy vs cbCut: Two Different Jobs

`cbLazy` (Correctness)

- Triggered at **integer** solutions (`MIPSOL` event)
- Purpose: reject invalid incumbents
- Required for a correct answer

cbLazy vs cbCut: Two Different Jobs

`cbLazy` (Correctness)

- Triggered at **integer** solutions (`MIPSOL` event)
- Purpose: reject invalid incumbents
- Required for a correct answer
- Separation: connected components (easy, $O(n+m)$)

`cbCut` (Performance)

- Triggered at **fractional** node LPs (`MIPNODE` event)

cbLazy vs cbCut: Two Different Jobs

`cbLazy` (Correctness)

- Triggered at **integer** solutions (`MIPSOL` event)
- Purpose: reject invalid incumbents
- Required for a correct answer
- Separation: connected components (easy, $O(n+m)$)

`cbCut` (Performance)

- Triggered at **fractional** node LPs (`MIPNODE` event)
- Purpose: tighten LP relaxation

cbLazy vs cbCut: Two Different Jobs

`cbLazy` (Correctness)

- Triggered at **integer** solutions (`MIPSOL` event)
- Purpose: reject invalid incumbents
- Required for a correct answer
- Separation: connected components (easy, $O(n+m)$)

`cbCut` (Performance)

- Triggered at **fractional** node LPs (`MIPNODE` event)
- Purpose: tighten LP relaxation
- Optional, but can speed up solving dramatically

cbLazy vs cbCut: Two Different Jobs

`cbLazy` (Correctness)

- Triggered at **integer** solutions (`MIPSOL` event)
- Purpose: reject invalid incumbents
- Required for a correct answer
- Separation: connected components (easy, $O(n+m)$)

`cbCut` (Performance)

- Triggered at **fractional** node LPs (`MIPNODE` event)
- Purpose: tighten LP relaxation
- Optional, but can speed up solving dramatically
- Separation: minimum cut (harder, but polynomial)

cbLazy vs cbCut: Two Different Jobs

`cbLazy` (Correctness)

- Triggered at **integer** solutions (`MIPSOL` event)
- Purpose: reject invalid incumbents
- Required for a correct answer
- Separation: connected components (easy, $O(n+m)$)

`cbCut` (Performance)

- Triggered at **fractional** node LPs (`MIPNODE` event)
- Purpose: tighten LP relaxation
- Optional, but can speed up solving dramatically
- Separation: minimum cut (harder, but polynomial)

Analogy:

- `cbLazy` Building inspector: “This violates code. Tear it down.” (correctness).

cbLazy vs cbCut: Two Different Jobs

`cbLazy` (Correctness)

- Triggered at **integer** solutions (`MIPSOL` event)
- Purpose: reject invalid incumbents
- Required for a correct answer
- Separation: connected components (easy, $O(n+m)$)

`cbCut` (Performance)

- Triggered at **fractional** node LPs (`MIPNODE` event)
- Purpose: tighten LP relaxation
- Optional, but can speed up solving dramatically
- Separation: minimum cut (harder, but polynomial)

Analogy:

- `cbLazy` Building inspector: “This violates code. Tear it down.” (correctness).
- `cbCut` = better architectural blueprints that prevent bad designs from ever being built. Fewer disasters to inspect later. (efficiency).

Separation Oracle for Fractional Solutions

Given a fractional solution $x^* \in [0, 1]^{|E|}$, we want to find a subset $S \subset V, S \neq \emptyset, S \neq V$ such that:

$$\sum_{i \in S, j \notin S} x^*_{ij} < 1$$

Separation Oracle for Fractional Solutions

Given a fractional solution $x^* \in [0,1]^{|E|}$, we want to find a subset $S \subset V, S \neq \emptyset, S \neq V$ such that:

$$\sum_{i \in S, j \notin S} x^*_{ij} < 1$$

Key observation: Define a directed graph with arc capacities $c_{ij} := x^*_{ij}$.

Separation Oracle for Fractional Solutions

Given a fractional solution $x^* \in [0,1]^{|E|}$, we want to find a subset $S \subset V, S \neq \emptyset, S \neq V$ such that:

$$\sum_{i \in S, j \notin S} x^*_{ij} < 1$$

Key observation: Define a directed graph with arc capacities $c_{ij} := x^*_{ij}$. Then for any subset S:

$$\sum_{i \in S, j \notin S} x^*_{ij} = \text{capacity of directed cut leaving } S.$$

Separation Oracle for Fractional Solutions

Given a fractional solution $x^* \in [0,1]^{|E|}$, we want to find a subset $S \subset V, S \neq \emptyset, S \neq V$ such that:

$$\sum_{i \in S, j \notin S} x^*_{ij} < 1$$

Key observation: Define a directed graph with arc capacities $c_{ij} := x^*_{ij}$.
Then for any subset S:

$$\sum_{i \in S, j \notin S} x^*_{ij} = \text{capacity of directed cut leaving } S.$$

So finding the **most violated** DFJ constraint = finding the **global minimum directed cut**.

Separation Oracle for Fractional Solutions

Given a fractional solution $x^* \in [0,1]^{|E|}$, we want to find a subset $S \subset V, S \neq \emptyset, S \neq V$ such that:

$$\sum_{i \in S, j \notin S} x^*_{ij} < 1$$

Key observation: Define a directed graph with arc capacities $c_{ij} := x^*_{ij}$. Then for any subset S:

$$\sum_{i \in S, j \notin S} x^*_{ij} = \text{capacity of directed cut leaving } S.$$

So finding the **most violated** DFJ constraint = finding the **global minimum directed cut**.

If the minimum cut value < 1: that cut set S gives a violated DFJ inequality.
If the minimum cut value ≥ 1: all DFJ constraints are satisfied.

The `MIPNODE` Event: A Node LP Was Solved

When `where == GRB.Callback.MIPNODE`:

The `MIPNODE` Event: A Node LP Was Solved

When `where == GRB.Callback.MIPNODE`:

What you can read:

```
# Status of the node LP (must be OPTIMAL to read solution)
status = model.cbGet(GRB.Callback.MIPNODE_STATUS)

# The fractional LP solution (only if status == GRB.OPTIMAL)
xstar = model.cbGetNodeRel(model._x)

# Current node count
nodecnt = model.cbGet(GRB.Callback.MIPNODE_NODCNT)

# Current best objective bound
objbnd = model.cbGet(GRB.Callback.MIPNODE_OBJBND)
```

The `MIPNODE` Event: A Node LP Was Solved

When `where == GRB.Callback.MIPNODE`:

What you can read:

```
# Status of the node LP (must be OPTIMAL to read solution)
status = model.cbGet(GRB.Callback.MIPNODE_STATUS)

# The fractional LP solution (only if status == GRB.OPTIMAL)
xstar = model.cbGetNodeRel(model._x)

# Current node count
nodecnt = model.cbGet(GRB.Callback.MIPNODE_NODCNT)

# Current best objective bound
objbnd = model.cbGet(GRB.Callback.MIPNODE_OBJBND)
```

What you can do:

```
# Add a globally valid inequality to tighten the relaxation
model.cbCut( <linear expression> >= <rhs> )
```

Important: Check `MIPNODE_STATUS`

Not every `MIPNODE` event has an optimal LP solution.

Important: Check `MIPNODE_STATUS`

Not every `MIPNODE` event has an optimal LP solution.

The node LP might be:

- **Optimal**: we can read the solution and separate.

Important: Check `MIPNODE_STATUS`

Not every `MIPNODE` event has an optimal LP solution.

The node LP might be:

- **Optimal**: we can read the solution and separate.
- **Infeasible**: this node is already pruned. Nothing to do.

Important: Check `MIPNODE_STATUS`

Not every `MIPNODE` event has an optimal LP solution.

The node LP might be:

- **Optimal**: we can read the solution and separate.
- **Infeasible**: this node is already pruned. Nothing to do.
- **Cutoff**: bound exceeds incumbent. Already pruned.

Important: Check MIPNODE_STATUS

Not every MIPNODE event has an optimal LP solution.

The node LP might be:

- **Optimal**: we can read the solution and separate.
- **Infeasible**: this node is already pruned. Nothing to do.
- **Cutoff**: bound exceeds incumbent. Already pruned.

Always guard your cbCut code

```
if where == GRB.Callback.MIPNODE:
    status = model.cbGet(GRB.Callback.MIPNODE_STATUS)
    if status != GRB.OPTIMAL:
        return    # nothing to separate, skip!
    xstar = model.cbGetNodeRel(model._x)
    # ... now safe to work with xstar ...
```

Practical Tip: Limit cbCut to Early Nodes

Adding user cuts at **every** node can slow the solver down (oracle overhead).

Practical Tip: Limit cbCut to Early Nodes

Adding user cuts at **every** node can slow the solver down (oracle overhead).

A common trick: only add cuts at the **root node** or early in the tree.

Practical Tip: Limit cbCut to Early Nodes

Adding user cuts at **every** node can slow the solver down (oracle overhead).

A common trick: only add cuts at the **root node** or early in the tree.

```python
if where == GRB.Callback.MIPNODE:
    status = model.cbGet(GRB.Callback.MIPNODE_STATUS)
    if status != GRB.OPTIMAL:
        return

    nodecnt = model.cbGet(GRB.Callback.MIPNODE_NODCNT)
    if nodecnt > 100:       # stop adding cuts deep in the tree
        return

    xstar = model.cbGetNodeRel(model._x)
    # ... separate and add cbCut ...
```

Practical Tip: Limit cbCut to Early Nodes

Adding user cuts at **every** node can slow the solver down (oracle overhead).

A common trick: only add cuts at the **root node** or early in the tree.

```
if where == GRB.Callback.MIPNODE:
    status = model.cbGet(GRB.Callback.MIPNODE_STATUS)
    if status != GRB.OPTIMAL:
        return

    nodecnt = model.cbGet(GRB.Callback.MIPNODE_NODCNT)
    if nodecnt > 100:       # stop adding cuts deep in the tree
        return

    xstar = model.cbGetNodeRel(model._x)
    # ... separate and add cbCut ...
```

Why? The root node LP benefits the most from tightening (all subtree bounds improve). Deep nodes have small subproblems where the overhead is not worth it.

Fractional Separation: Global Min-Cut with igraph

```python
import igraph as ig  # networkx does not have global directed min-cut!

def mincut_separate_dfj(n, xstar):
    """
    Find the most violated DFJ constraint in a fractional solution.
    Returns (S, cut_value) if violated, else (None, None).
    """
    edges = [(i, j) for i in range(n) for j in range(n) if i != j]
    caps  = [float(xstar[i, j]) for (i, j) in edges]

    g = ig.Graph(n=n, edges=edges, directed=True)
    g.es["cap"] = caps

    cut = g.mincut(capacity="cap")   # global directed min-cut
    val = float(cut.value)

    if val < 1 - 1e-6:   # tolerance for numerical safety
        A, B = cut.partition
        S = set(A) if 0 < len(A) < n else set(B)
        if 0 < len(S) < n:
            return S, val
    return None, None
```

The Unified Callback

```python
def callback(model, where):
    x, n = model._x, model._n

    # (A) CORRECTNESS: reject integer solutions with subtours
    if where == GRB.Callback.MIPSOL:
        xsol = model.cbGetSolution(x)
        subtours = find_subtours(n, xsol)
        for S in subtours:
            complement = set(range(n)) - S
            model.cbLazy(
                gp.quicksum(x[i,j] for i in S for j in complement) >= 1)

    # (B) PERFORMANCE: tighten LP relaxation with DFJ cuts
    if where == GRB.Callback.MIPNODE:
        if model.cbGet(GRB.Callback.MIPNODE_STATUS) != GRB.OPTIMAL: return
        if model.cbGet(GRB.Callback.MIPNODE_NODCNT) != 0: return
        xstar = model.cbGetNodeRel(x)
        S, val = mincut_separate_dfj(n, xstar)
        if S is not None:
            comp = set(range(n)) - S
            model.cbCut(
                gp.quicksum(x[i,j] for i in S for j in comp) >= 1)
```

Putting It All Together

```python
m.Params.LazyConstraints = 1   # required for cbLazy
m._x = x
m._n = n
m.optimize(callback)
```

Putting It All Together

```
m.Params.LazyConstraints = 1   # required for cbLazy
m._x = x
m._n = n
m.optimize(callback)
```

What happens during the solve:

- At fractional node LPs: `cbCut` tightens the relaxation $\Rightarrow$ better bounds, more pruning.

Putting It All Together

```
m.Params.LazyConstraints = 1   # required for cbLazy
m._x = x
m._n = n
m.optimize(callback)
```

What happens during the solve:

- At fractional node LPs: `cbCut` tightens the relaxation ⇒ better bounds, more pruning.
- At integer incumbents: `cbLazy` rejects solutions with subtours ⇒ correctness.

Putting It All Together

```
m.Params.LazyConstraints = 1   # required for cbLazy
m._x = x
m._n = n
m.optimize(callback)
```

What happens during the solve:

- At fractional node LPs: `cbCut` tightens the relaxation $\Rightarrow$ better bounds, more pruning.
- At integer incumbents: `cbLazy` rejects solutions with subtours $\Rightarrow$ correctness.
- Result: correct answer, found faster.

Why Both? Can't We Use Only cbCut?

Short answer: No. `cbCut` alone does not guarantee correctness.

Why Both? Can't We Use Only cbCut?

Short answer: No. `cbCut` alone does not guarantee correctness.

Why not?

- Gurobi treats user cuts (`cbCut`) as **optional hints**. It may discard them during presolve, heuristics, or when rebuilding the LP.

Why Both? Can't We Use Only cbCut?

Short answer: No. `cbCut` alone does not guarantee correctness.

Why not?

- Gurobi treats user cuts (`cbCut`) as **optional hints**. It may discard them during presolve, heuristics, or when rebuilding the LP.
- Lazy constraints (`cbLazy`) are treated as **part of the true model**. Gurobi guarantees every returned solution satisfies them.

Why Both? Can't We Use Only cbCut?

Short answer: No. `cbCut` alone does not guarantee correctness.

Why not?

- Gurobi treats user cuts (`cbCut`) as **optional hints**. It may discard them during presolve, heuristics, or when rebuilding the LP.
- Lazy constraints (`cbLazy`) are treated as **part of the true model**. Gurobi guarantees every returned solution satisfies them.

Rule

- Constraints required for **correctness** $\Rightarrow$ `cbLazy`.
- Constraints added only for **speed** $\Rightarrow$ `cbCut`.
- For TSP: we need `cbLazy` at minimum. `cbCut` is a performance bonus (and we should measure to see if it is actually helpful to the solver).

When to Use What

You have a MIP with many constraints. Three options:

When to Use What

You have a MIP with many constraints. Three options:

Option 1: Add all constraints upfront.

- If the number of constraints is polynomial and manageable (e.g., MTZ for TSP).
- Simple. No callbacks needed.
- But: may give a weaker relaxation or a huge model.

When to Use What

You have a MIP with many constraints. Three options:

Option 1: Add all constraints upfront.

- If the number of constraints is polynomial and manageable (e.g., MTZ for TSP).
- Simple. No callbacks needed.
- But: may give a weaker relaxation or a huge model.

Option 2: Use `cbLazy`.

- Constraints are exponential or too many to enumerate.
- You have a fast separation oracle for **integer** solutions.
- Required for **correctness** when constraints are omitted from the model.

When to Use What

You have a MIP with many constraints. Three options:

Option 1: Add all constraints upfront.

- If the number of constraints is polynomial and manageable (e.g., MTZ for TSP).
- Simple. No callbacks needed.
- But: may give a weaker relaxation or a huge model.

Option 2: Use `cbLazy`.

- Constraints are exponential or too many to enumerate.
- You have a fast separation oracle for **integer** solutions.
- Required for **correctness** when constraints are omitted from the model.

Option 3: Add `cbCut` on top.

- You also have a separation oracle for **fractional** solutions.
- Purely a performance optimization, tightens the LP relaxation.

Hot Start for MIPs: Not as Magical as for LPs

In Lecture 8, we saw that LP row generation benefits enormously from **hot starts**:

- Add one constraint $\Rightarrow$ dual simplex fixes the basis in a few pivots.
- Almost free re-optimization.

Hot Start for MIPs: Not as Magical as for LPs

In Lecture 8, we saw that LP row generation benefits enormously from **hot starts**:

- Add one constraint $\Rightarrow$ dual simplex fixes the basis in a few pivots.
- Almost free re-optimization.

For MIPs, the story is different.

Hot Start for MIPs: Not as Magical as for LPs

In Lecture 8, we saw that LP row generation benefits enormously from **hot starts**:

- Add one constraint $\Rightarrow$ dual simplex fixes the basis in a few pivots.
- Almost free re-optimization.

For MIPs, the story is different.

When Gurobi solves a MIP, it builds up a lot of internal state:

- The entire B&B tree (node queue, branching history).

Hot Start for MIPs: Not as Magical as for LPs

In Lecture 8, we saw that LP row generation benefits enormously from **hot starts**:

- Add one constraint $\Rightarrow$ dual simplex fixes the basis in a few pivots.
- Almost free re-optimization.

For MIPs, the story is different.

When Gurobi solves a MIP, it builds up a lot of internal state:

- The entire B&B tree (node queue, branching history).
- Cutting planes it discovered automatically (Gomory, MIR, etc.).

Hot Start for MIPs: Not as Magical as for LPs

In Lecture 8, we saw that LP row generation benefits enormously from **hot starts**:

- Add one constraint $\Rightarrow$ dual simplex fixes the basis in a few pivots.
- Almost free re-optimization.

For MIPs, the story is different.

When Gurobi solves a MIP, it builds up a lot of internal state:

- The entire B&B tree (node queue, branching history).
- Cutting planes it discovered automatically (Gomory, MIR, etc.).
- Incumbent solutions and their bounds.

Hot Start for MIPs: Not as Magical as for LPs

In Lecture 8, we saw that LP row generation benefits enormously from **hot starts**:

- Add one constraint $\Rightarrow$ dual simplex fixes the basis in a few pivots.
- Almost free re-optimization.

For MIPs, the story is different.

When Gurobi solves a MIP, it builds up a lot of internal state:

- The entire B&B tree (node queue, branching history).
- Cutting planes it discovered automatically (Gomory, MIR, etc.).
- Incumbent solutions and their bounds.

If you add a constraint and re-solve from scratch, **all of this is lost**. There is no "dual simplex trick" that cheaply repairs a B&B tree.

Why Callbacks Beat an Outer Loop for MIPs

Naive approach (outer loop):

1. Solve the full MIP.
2. Check solution. If violated, add constraint.
3. Solve the full MIP again from scratch.

Why Callbacks Beat an Outer Loop for MIPs

Naive approach (outer loop):

1. Solve the full MIP.
2. Check solution. If violated, add constraint.
3. Solve the full MIP again from scratch.

Each iteration restarts B&B. All tree information, cuts, and heuristic solutions are discarded. Very expensive.

Why Callbacks Beat an Outer Loop for MIPs

Naive approach (outer loop):

1. Solve the full MIP.
2. Check solution. If violated, add constraint.
3. Solve the full MIP again from scratch.

Each iteration restarts B&B. All tree information, cuts, and heuristic solutions are discarded. Very expensive.

Callback approach (`cbLazy`/`cbCut`):

1. Gurobi runs B&B **once**.
2. At specific moments, your callback injects constraints.
3. The solver continues from where it was. Tree, cuts, and incumbents are preserved.

Why Callbacks Beat an Outer Loop for MIPs

Naive approach (outer loop):

1. Solve the full MIP.
2. Check solution. If violated, add constraint.
3. Solve the full MIP again from scratch.

Each iteration restarts B&B. All tree information, cuts, and heuristic solutions are discarded. Very expensive.

Callback approach (`cbLazy/cbCut`):

1. Gurobi runs B&B **once**.
2. At specific moments, your callback injects constraints.
3. The solver continues from where it was. Tree, cuts, and incumbents are preserved.

Key insight

For LPs, an outer loop with hot starts works great (Lecture 8).

Common Pitfalls

1. Forgetting `LazyConstraints = 1`

```
# m.Params.LazyConstraints = 1   # forgot this!
m.optimize(callback)  # cbLazy calls are silently ignored!
```

Common Pitfalls

1. Forgetting `LazyConstraints = 1`

```
# m.Params.LazyConstraints = 1   # forgot this!
m.optimize(callback)  # cbLazy calls are silently ignored!
```

2. Not checking `MIPNODE_STATUS` before reading fractional solution

```
if where == GRB.Callback.MIPNODE:
    # BAD: node LP might be infeasible or cutoff
    xstar = model.cbGetNodeRel(x)  # can crash or give garbage
```

Common Pitfalls

1. Forgetting `LazyConstraints = 1`

```
# m.Params.LazyConstraints = 1   # forgot this!
m.optimize(callback)  # cbLazy calls are silently ignored!
```

2. Not checking `MIPNODE_STATUS` before reading fractional solution

```
if where == GRB.Callback.MIPNODE:
    # BAD: node LP might be infeasible or cutoff
    xstar = model.cbGetNodeRel(x)  # can crash or give garbage
```

3. Numerical tolerances

```
# BAD: exact comparison with floating point
if cut_value < 1:  ...

# GOOD: use a small tolerance
if cut_value < 1 - 1e-6:  ...
```

Common Pitfalls (continued)

4. Doing too much work in `cbCut`

Common Pitfalls (continued)

4. Doing too much work in `cbCut`

The callback is invoked at **every** node. If your separation oracle is expensive, this can make the solve slower, not faster.

Common Pitfalls (continued)

4. Doing too much work in `cbCut`

The callback is invoked at **every** node. If your separation oracle is expensive, this can make the solve slower, not faster.

```
# Practical pattern: limit cuts to early nodes
if where == GRB.Callback.MIPNODE:
    nodecnt = model.cbGet(GRB.Callback.MIPNODE_NODCNT)
    if nodecnt > 500:
        return   # stop adding cuts deep in the tree
```

Common Pitfalls (continued)

4. Doing too much work in `cbCut`
The callback is invoked at **every** node. If your separation oracle is expensive, this can make the solve slower, not faster.

```
# Practical pattern: limit cuts to early nodes
if where == GRB.Callback.MIPNODE:
    nodecnt = model.cbGet(GRB.Callback.MIPNODE_NODCNT)
    if nodecnt > 500:
        return   # stop adding cuts deep in the tree
```

5. Using `cbCut` alone for correctness

Common Pitfalls (continued)

4. Doing too much work in `cbCut`
The callback is invoked at **every** node. If your separation oracle is expensive, this can make the solve slower, not faster.

```
# Practical pattern: limit cuts to early nodes
if where == GRB.Callback.MIPNODE:
    nodecnt = model.cbGet(GRB.Callback.MIPNODE_NODCNT)
    if nodecnt > 500:
        return   # stop adding cuts deep in the tree
```

5. Using `cbCut` alone for correctness
Gurobi treats user cuts as **hints**. It may discard them during presolve or node processing. Never rely on `cbCut` for constraints that are required for a correct answer, use `cbLazy` for those.

Useful Callback Attributes

Inside a callback, you can query the solver's progress:

Useful Callback Attributes

Inside a callback, you can query the solver's progress:

Event	Attribute	What it gives you
MIPSOL	MIPSOL_OBJ	objective of the incumbent
MIPSOL	MIPSOL_NODCNT	node where incumbent was found
MIPSOL	MIPSOL_SOLCNT	total solutions found so far
MIPNODE	MIPNODE_STATUS	node LP status (check = OPTIMAL!)
MIPNODE	MIPNODE_OBJBND	current best bound
MIPNODE	MIPNODE_NODCNT	current node count

Useful Callback Attributes

Inside a callback, you can query the solver's progress:

Event	Attribute	What it gives you
MIPSOL	MIPSOL_OBJ	objective of the incumbent
MIPSOL	MIPSOL_NODCNT	node where incumbent was found
MIPSOL	MIPSOL_SOLCNT	total solutions found so far
MIPNODE	MIPNODE_STATUS	node LP status (check = OPTIMAL!)
MIPNODE	MIPNODE_OBJBND	current best bound
MIPNODE	MIPNODE_NODCNT	current node count

These are useful for logging, early termination, or deciding when to stop adding cuts.