CS498 Homework 4

February 17, 2026

1 Problem 1: Multi-Period Airline Yield Management (Auto-graded)

1.1 Problem Statement

An airline sells tickets over multiple booking periods for a **single departure flight** (think ORD to SFO). You must build a Mixed Integer Program to maximize expected profit.

1.2 The Business Problem

1.2.1 Setup

- The airline sells tickets over **T booking periods** (e.g., Period 0 = 3 weeks before departure, Period 1 = 2 weeks before, Period 2 = 1 week before)
- **All tickets are for ONE departure flight** that happens after all booking periods end
- There are **C cabin classes** (e.g., First, Business, Economy)
- In each booking period, for each class, the airline must choose **exactly one** of **K price options** to offer
- **The airline must decide how many planes to operate for this flight** (between 0 and `max_planes` identical planes)
- There are **S scenarios**, each with a known probability representing demand uncertainty
- Each plane has a fixed number of seats per class and costs `plane_cost` to operate

1.2.2 Timeline Clarification

```
Week -3 (Period 0): Set prices → Customers book tickets
Week -2 (Period 1): Adjust prices → More customers book
Week -1 (Period 2): Final prices → Last customers book
Week  0: DEPARTURE → The planes fly with all passengers from all periods.
```

Key points: - You make **one capacity decision**: how many planes to operate for the departure flight - You make **multiple pricing decisions**: you can change prices between booking periods - Sales accumulate: a customer who books in Period 0 occupies a seat that's no longer available in Period 1

1.2.3 Key Business Rules

1. **Price commitment per period:** In each booking period and class, you must pick exactly one price option (you can't offer multiple prices simultaneously for the same class in the same booking period). You CAN change prices between periods.

2. **Demand is price-dependent and period-dependent:** For each combination of (booking period, scenario, class, price option), there's a forecast of how many NEW customers want to purchase seats during that booking period at that price.
3. **Sales cannot exceed demand:** You cannot sell more seats than the demand forecast for the chosen price in that booking period and scenario.
4. **Cumulative capacity constraint:** Total seats sold across ALL booking periods in a class cannot exceed the total available capacity (seats per plane × number of planes).
5. **Scenario-based optimization:** You don't know which scenario will occur, so you optimize **expected value** across scenarios. Your pricing and capacity decisions must be made before knowing which scenario realizes.

1.2.4 Note: Most airlines face this exact problem:

- They must commit to capacity (number of planes/aircraft size) well in advance
- They adjust ticket prices over time as departure approaches (typically prices rise closer to departure)
- They must balance: high prices (more revenue per seat) vs. low prices (more seats filled)
- Early capacity commitment is expensive, but underestimating demand means lost revenue

1.3 What You Must Implement

Submit a file named `hw4_p1_yield.py` containing:

```
import gurobipy as gp
from gurobipy import GRB
import numpy as np

def solve_yield_management(prices, demands, scenario_probs, seats_per_plane,
                           plane_cost, max_planes):
    """
    Solve the multi-period airline yield management problem.

    Parameters
    ----------
    prices : dict
        prices[t, c, k] = ticket price for booking period t, class c, option k
        where t in {0,...,T-1}, c in {0,...,C-1}, k in {0,...,K-1}

    demands : dict
        demands[t, s, c, k] = NEW demand in booking period t, scenario s,
                              class c, price option k
        where s in {0,...,S-1}
        This represents customers wanting to buy in period t (not cumulative).

    scenario_probs : list of length S
        scenario_probs[s] = probability of scenario s
        Sum equals 1.0
```

```
    seats_per_plane : dict
        seats_per_plane[c] = number of seats of class c on each plane

    plane_cost : float
        Cost to operate one plane for the departure flight

    max_planes : int
        Maximum number of planes available for this flight

    Returns
    -------
    result : dict with keys:
        {
            'price_selection': dict mapping (t,c) -> k
                (the chosen price option for booking period t and class c),
            'planes': int
                (number of planes to operate - single decision),
            'sales': dict mapping (t,s,c,k) -> quantity sold
                (seats sold in period t, under scenario s, for class c, at price option k),
            'objective': float
                (optimal expected **profit**)
        }

    All keys must be present. Sales should include all (t,s,c,k) combinations,
    even if value is 0.
    """
    # TODO: Your model here

    raise NotImplementedError
```

1.4 Modeling Guidance Hints

You need to decide:

1. **What are your decision variables?** Think about:
   - How do you represent "which price option is chosen" for each (period, class) pair?
   - How do you represent "how many planes to operate" (one decision for the entire flight)?
   - How do you represent "how many seats to sell" for each (period, scenario, class, price option)?
2. **What constraints enforce the business rules?**
   - How do you ensure exactly one price is picked per (period, class)?
   - How do you ensure you can't sell seats at a price you didn't choose?
   - How do you ensure sales in each period don't exceed that period's demand?
   - How do you ensure TOTAL sales across all periods don't exceed plane capacity?
3. **What's the objective?**
   - Maximize expected profit = expected revenue - cost
   - **Expected revenue**: For each scenario s with probability p[s], sum (price[t,c,k] $\times$

sales[t,s,c,k]) over all t,c,k, then multiply by p[s], then sum over all scenarios
- **Cost**: plane_cost × number_of_planes (single cost, not per period)
- Example: If scenario 1 (prob 0.3) gives revenue \$150K and scenario 2 (prob 0.7) gives revenue \$200K, and you operate 2 planes at \$30K each, then expected profit = 0.3×150K + 0.7×200K - 2×30K = \$125K

Some ideas for modelling: - Binary variables with constraints, OR - Gurobi's `addSOS(GRB.SOS_TYPE1, ...)` for price selection - Single integer variable for plane count - Whatever works!

1.5 Example Test Case

```
# Small instance: 2 booking periods, 2 scenarios, 2 classes, 2 price options each

# (period, class, option) -> price in dollars
prices = {
    (0, 0, 0): 1000, (0, 0, 1): 800,   # Period 0, Class 0 (First)
    (0, 1, 0): 500,  (0, 1, 1): 400,   # Period 0, Class 1 (Economy)
    (1, 0, 0): 1200, (1, 0, 1): 900,   # Period 1, Class 0 (prices can be different!)
    (1, 1, 0): 600,  (1, 1, 1): 450,   # Period 1, Class 1
}

# (period t, scenario s, class c, price option k) -> NEW demand in that period
demands = {
    (0, 0, 0, 0): 10, (0, 0, 0, 1): 15,  # Period 0, Scenario 0, Class 0
    (0, 0, 1, 0): 30, (0, 0, 1, 1): 40,  # Period 0, Scenario 0, Class 1
    (0, 1, 0, 0): 20, (0, 1, 0, 1): 25,  # Period 0, Scenario 1, Class 0
    (0, 1, 1, 0): 50, (0, 1, 1, 1): 60,  # Period 0, Scenario 1, Class 1
    (1, 0, 0, 0): 15, (1, 0, 0, 1): 20,  # Period 1, Scenario 0, Class 0
    (1, 0, 1, 0): 35, (1, 0, 1, 1): 45,  # Period 1, Scenario 0, Class 1
    (1, 1, 0, 0): 25, (1, 1, 0, 1): 30,  # Period 1, Scenario 1, Class 0
    (1, 1, 1, 0): 55, (1, 1, 1, 1): 65,  # Period 1, Scenario 1, Class 1
}

scenario_probs = [0.3, 0.7]  # Probabilities for scenarios 0 and 1
seats_per_plane = {0: 20, 1: 50}  # 20 First class, 50 Economy per plane
plane_cost = 30000.0  # Cost to operate one plane
max_planes = 3  # Can operate 0, 1, 2, or 3 planes

result = solve_yield_management(prices, demands, scenario_probs,
                                seats_per_plane, plane_cost, max_planes)

print(f"Optimal profit: ${result['objective']:,.2f}")
print(f"Price selections: {result['price_selection']}")
print(f"Number of planes: {result['planes']}")  # Single number now!
```

1.6 Hints for Debugging

- Start simple: test with 1 period, 1 scenario, 1 class first

- Check that your "exactly one price" constraint works by inspecting `price_selection` output
- Verify that sales are zero for non-selected price options
- The capacity constraint should sum sales across ALL periods: `sum over t of sales[t,s,c,k] <= seats_per_plane[c] × num_planes`
- Make sure your objective correctly weights scenarios by their probabilities
- There's only ONE plane cost (not per period): `plane_cost × num_planes`

1.7 Grading (100 points)

Your solution will be tested on multiple instances: - Small instances (2 periods, 2 scenarios, 2 classes) - Medium instances (3 periods, 3 scenarios, 3 classes, 3 options) - Large instances (5+ periods, 5+ scenarios)

Points awarded for: - Correct **optimal** objective value (± 0.01 tolerance) - Feasible solutions (all constraints satisfied) even if suboptimal/incorrect objective value.

2 Problem 2: Solving Survivable Network Design LP Relaxation Using Row Generation (Autograded)

2.1 Problem Statement

You are designing a telecommunications backbone network that must survive link failures. Given a graph and costs for each edge, you must select edges to build so that the network is **k-edge-connected** (remains connected even if k-1 edges fail), while minimizing total cost.

Challenge: There are exponentially many connectivity constraints. You'll solve the LP relaxation using a separation oracle.

2.2 The Network Design Problem

2.2.1 Setup

- **Graph**: Undirected $G = (V, E)$
- **Edge costs**: $c_e \geq 0$ for each edge $e \in E$.
- **Survivability requirement**: Network must be **k-edge-connected**

2.2.2 What is k-edge-connected?

A graph is **k-edge-connected** if it remains connected after removing any (k-1) edges.

Equivalently: For every partition of nodes into two sets S and $V - S$, there must be **at least k edges** crossing between them.

2.2.3 Examples

- **k=1**: Spanning tree (minimally connected)
- **k=2**: Survives any single edge failure (common for telecom)
- **k=3**: Survives any two simultaneous edge failures

2.3 Integer Programming Formulation

2.3.1 Decision Variables

For each edge $e \in E$: $x_e \in \{0, 1\}$: equals 1 if edge e is selected to be built.

2.3.2 Objective

Minimize total cost:

$$\min \sum_{e \in E} c_e x_e$$

2.3.3 Constraints

For every nonempty proper subset $S \subset V$ (where $S \neq \emptyset, S \neq V$):

$$\sum_{e \in \delta(S)} x_e \geq k$$

where $\delta(S)$ = edges with exactly one endpoint in S (the "cut"). We cannot enumerate them all.

Note that for $k = 1$, this problem is equivalent to the minimum spanning tree. For $k \geq 2$, this problem is NP-Hard.

If we solve this using Integer Programming in Gurobi, we saw that Gurobi needs to solve the LP relaxation (i.e. $0 \leq x_e \leq 1$) during the branch and bound algorithm to solve the integer program.

2.4 LP Relaxation

The LP relaxation replaces $x_e \in \{0, 1\}$ with $0 \leq x_e \leq 1$:

$$\min \sum_{e \in E} c_e x_e$$

Such that:

- $\sum_{e \in \delta(S)} x_e \geq k$ for all cuts $S \subset V, S \neq \emptyset, S \neq V$.
- $0 \leq x_e \leq 1$.

Your Task: Solve this LP using row generation with a separation oracle.

2.5 The Separation Oracle

Given a current solution x^* (where $0 \leq x_e^* \leq 1$), you need to find a violated cut constraint, or prove none exists.

Separation Problem: Does there exist $S \subset V$ such that $\sum_{e \in \delta(S)} x_e^* < k$?

Separation Oracle Algorithm: Read on Wikipedia/any other source on the Stoer-Wagner global minimum cut algorithm. Look at `networkx.stoer_wagner(G)`. Build a graph H where for each edge e with $x_e^* > 0$, we add the edge with weight x_e^* (note $x_e^* \approx 0$ means the edge shouldn't be added). If the graph H has more than one connected component $(C_1, ..., C_k)$, then the set $S = C_1$ has $\sum_{e \in \delta(S)} x_e^* = 0$, and S is a violated cut constraint. Otherwise, if the graph is one connected component, we check if the global min cut value $< k$, at which case we add this cut constraint. And if global minimum cut value is $\geq k$, then we stop!

2.5.1 Note: Stoer-Wagner is not the fastest global-min-cut algorithm for undirected graphs theoretically. For example, see the Hao-Orlin algorithm, Karger's algorithm, or more recent faster algorithms. We're just using it because networkx neatly implements it with one API call :).

2.6 What You Must Implement

Submit a file named `hw4_p2_survivable.py` containing:

```
import networkx as nx
import gurobipy as gp
from gurobipy import GRB

def solve_survivable_network_lp(nodes, edges, costs, k, max_iterations=1000):
    """
```

```
    Solve the LP relaxation of k-edge-connected network design using cutting planes.

    Parameters
    ----------
    nodes : list
        List of node identifiers, e.g., [0, 1, 2, 3, 4]

    edges : list of tuples
        List of undirected edges as (u, v) pairs.
        Example: [(0,1), (1,2), (2,3)]
        Each edge appears once; (u,v) and (v,u) are the same edge.
        It is guranteed that the graph with ALL the edges is k-connected.

    costs : dict
        costs[(u,v)] = cost to build edge (u,v) [Undirected]

    k : int
        Required edge connectivity (1 <= k <= 5)

    max_iterations : int, optional
        Maximum number of row generation iterations (default: 1000)

    Returns
    -------
    result : dict with keys:
        {
            'edges': dict mapping edge tuple -> value in [0,1]
                     e.g., {(0,1): 1.0, (1,2): 0.5, ...}
                     Include all edges.

            'total_cost': float
                         Total cost of the LP solution

            'is_optimal': bool
                         True if row generation algorithm converged after <= max_iterations (n
                         False if we hit max_iterations without convergence.
        }

    Notes
    -----
    - Start with LP having no cut constraints, only 0<=x_e<=1 and objective being Minimize \su
    - Iteratively add violated cut constraints using Stoer-Wagner
    - Stop when no violated cuts exist or max_iterations reached
    - Return the LP optimal solution (may be fractional for k 2)
    """
    # TODO: Implement cutting-plane/row-generation method with Stoer-Wagner separation oracle
    raise NotImplementedError
```

2.7 Implementation Requirements

1. **Start with an LP** that has:
   - Variables: $0 \leq x_e \leq 1$ for each edge
   - Objective: minimize total cost
   - Constraints: NONE initially.
2. **Cutting-plane, or row-generation, loop**:
   - Solve the current LP
   - If there is more than one connected component, then set S to be any of the components.
   - Otherwise, use Stoer-Wagner to find a violated cut
   - If found, add the cut constraint and repeat
   - If not found, you're done!
3. **Return the LP solution** (not rounded):
   - All edge values from the final LP
   - Total cost
   - Whether it converged

2.8 Example Test Cases

2.8.1 Example 1: Triangle (k=2)

```
nodes = [0, 1, 2]
edges = [(0,1), (1,2), (0,2)]
costs = {(0,1): 1.0, (1,2): 1.0, (0,2): 1.0}
k = 2

result = solve_survivable_network_lp(nodes, edges, costs, k)

print(f"Total cost: {result['total_cost']}")
print(f"Edge values: {result['edges']}")
print(f"Optimal: {result['is_optimal']}")

# Expected: All three edges at 1.0, total cost = 3.0
```

2.8.2 Example 2: Line graph (k=1)

```
nodes = [0, 1, 2, 3]
edges = [(0,1), (1,2), (2,3)]
costs = {(0,1): 1.0, (1,2): 1.0, (2,3): 1.0}
k = 1

result = solve_survivable_network_lp(nodes, edges, costs, k)

# Expected: All three edges at 1.0, total cost = 3.0 (spanning tree)
```

2.9 Grading (100 points)

Your solution will be tested on multiple instances:

2.9.1 Small graphs (30 points)

- 5-10 nodes, 10-20 edges
- k = 1, 2, 3
- Tests basic correctness

2.9.2 Medium graphs (40 points)

- 50-60 nodes, 100-150 edges
- k = 1, 2, 3
- Tests algorithmic correctness

2.9.3 Large graphs (30 points)

- 80-100 nodes, 250-350 edges
- k = 2, 3, 4
- Tests efficiency (must complete in reasonable time)
- Might take ~3 minutes on the autograder.

Points awarded for: - Correct optimal cost (±0.01-0.1 tolerance depending on size) - Feasible LP solution (all cut constraints satisfied) - Convergence within max_iterations

3 Problem 3: Semidefinite Programming with Eigenvalue Oracle (Autograded)

3.1 Problem Statement

You will solve a **Semidefinite Program (SDP)** using a row generation method with an eigenvalue-based separation oracle. This problem shows how linear algebra (eigenvalues/eigenvectors) connects to optimization with **infinitely** many constraints!!

3.2 The Optimization Problem

We want to optimize over a symmetric **matrix variable** X:

$$\max\langle C, X\rangle$$

Subject to:

- $\langle A_i, X\rangle \leq b_i$ for $i = 1, ..., m$.
- $X \succeq 0$ (X is positive semidefinite).

Where: - X is an $n \times n$ symmetric matrix (decision variable) - C, A_i are $n \times n$ symmetric matrices (given) - b_i are scalars (given) - $\langle A, B\rangle = \text{trace}(A^T B) = \sum_{ij} A_{ij} B_{ij}$ (Frobenius inner product)

Without the constraint $X \succeq 0$, this is a linear program with variables $X_{0,0}, ..., X_{n-1,n-1}$! (check).

3.2.1 What does $X \succeq 0$ mean?

X is **positive semidefinite (PSD)** if for all vectors $v \in \mathbb{R}^n$:

$$v^T X v \geq 0$$

This is **infinitely many constraints** (one for each real vector v)!

Equivalently: $X \succeq 0$ if and only if all eigenvalues of X are non-negative.

The PSD constraint $X \succeq 0$ represents infinitely many linear inequalities. We cannot enumerate them all!

Solution: Use an **eigenvalue oracle** to generate violated constraints on-the-fly.

3.3 The Eigenvalue Separation Oracle

Given a candidate matrix X^* (from solving a relaxed LP), check if $X^* \succeq 0$:

3.3.1 Algorithm:

1. **Compute eigenvalues** of X*
   - Let $\lambda_{\min}$ = smallest eigenvalue
   - Let $v_{\min}$ = corresponding eigenvector
2. **Check feasibility**:
   - If $\lambda_{\min} \geq 0$ (for small $\epsilon > 0$): X^* is PSD.

- If $\lambda_{\min} < 0$: X^* violates PSD constraint.

3. **Generate cut** (if violated):
   - We have: $v_{\min}^T X^* v_{\min} = \lambda_{\min} < 0$.
   - Add the linear constraint:

$$v_{\min}^T X^* v_{\min} \geq 0$$

   - In matrix form:

$$\langle v_{\min} v_{\min}^T, X \rangle \geq 0.$$

   - This cuts off the current infeasible point X^*.

Assume that the SDP is feasible and bounded.

3.4 What You Must Implement

Submit a file named `hw4_p3_psd.py` containing:

```
import numpy as np
import gurobipy as gp
from gurobipy import GRB

def solve_sdp_with_cuts(C, A_list, b_list, n, max_iterations=100):
    """
    Solve SDP using cutting planes with eigenvalue oracle.

    Parameters
    ----------
    C : np.ndarray, shape (n, n)
        Objective matrix (symmetric)

    A_list : list of np.ndarray
        List of constraint matrices [A_1, A_2, ..., A_m]
        Each A_i is shape (n, n) and symmetric

    b_list : list of float
        Right-hand side values [b_1, b_2, ..., b_m]

    n : int
        Matrix dimension

    max_iterations : int, optional
        Maximum cutting plane iterations (default: 100)

    Returns
    -------
    result : dict with keys:
        {
            'X': np.ndarray, shape (n, n)
```

```
            Optimal matrix solution

        'objective': float
                    Optimal objective value  C, X

        'is_optimal': bool
                     True if converged (X is PSD within tolerance)
                     False if hit max_iterations
        'num_cuts': int
                   Number of eigenvalue cuts added
    }

Notes
-----
- Start with LP having only the linear constraints  A_i, X  <= b_i and maximizing  C, X
- No PSD constraints initially
- Iteratively:
  * Solve LP to get X*
  * Compute smallest eigenvalue  _min and eigenvector v_min of X*
  * If  _min < 0: add cut  v_min v_min^T, X   0 and repeat
  * If  _min   0: done (PSD satisfied)
- Return final matrix X as 2D numpy array
"""
# TODO: Implement cutting-plane method with eigenvalue oracle
raise NotImplementedError
```

3.5 Implementation Guide

3.5.1 Step 1: Set up the LP

You can use Gurobi addMVar, or just flatten X since it is an $n \times n$ matrix, so store as n^2 variables $X[i, j]$

```
# Create variables for matrix entries
X_vars = m.addVars(n, n, lb=-GRB.INFINITY, name="X")
```

3.5.2 Step 2: Add linear constraints

For each i, add: $\langle A_i, X \rangle \leq b_i$ as linear inequalities.

3.5.3 Step 3: Set objective

3.5.4 Step 4: Cutting plane loop

Consider researching `numpy.linalg.eigh`.

3.5.5 Step 5: Return result

```
return {
    'X': X_current,
```

```
    'objective': m.ObjVal,
    'is_optimal': converged,
    'num_cuts': num_cuts
}
```

3.6 Example Test Case

```
import numpy as np

# Simple 2×2 example
n = 2

# Maximize X[0,0] + X[1,1] (trace)
C = np.array([[1.0, 0.0],
              [0.0, 1.0]])

# Subject to: X[0,0] + X[1,1]   2
A_list = [np.array([[1.0, 0.0],
                    [0.0, 1.0]])]
b_list = [2.0]

result = solve_sdp_with_cuts(C, A_list, b_list, n)

print(f"Optimal objective: {result['objective']:.4f}")
print(f"Optimal X:\n{result['X']}")
print(f"Number of cuts added: {result['num_cuts']}")
print(f"Is PSD: {result['is_optimal']}")
```

3.7 Grading (100 points)

Your solution will be tested on:

3.7.1 Small instances (40 points)

- 2×2 and 3×3 matrices
- Simple constraints
- Tests basic correctness

3.7.2 Medium instances (40 points)

- 4×4 and 5×5 matrices
- Multiple linear constraints
- Tests algorithmic correctness

3.7.3 Large instances (20 points)

- 8×8 and 10×10 matrices
- Tests efficiency

Points awarded for: - Correct optimal objective (± 0.01 tolerance) - X is PSD (all eigenvalues -1e-6) - Satisfies all linear constraints - Convergence within max_iterations

4 Problem 4: Integer Survivable Network Design with Gurobi Callbacks (Autograded)

4.1 Problem Statement

In Problem 2, you solved the **LP relaxation** of the survivable network design problem using cutting planes. Now you'll solve the **integer version** using Gurobi's callback system to add constraints dynamically during branch-and-bound.

This problem demonstrates the difference between:

- **User cuts (`cbCut`)**: Strengthen LP relaxations at fractional nodes to improve bounds during B&B.
- **Lazy constraints (`cbLazy`)**: Verify integer solutions are feasible.

4.2 The Integer Problem

Recall the k-edge-connected network design problem:

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e \, x_e \\ \text{s.t.} \quad & \sum_{e \in \delta(S)} x_e \geq k \qquad \forall \, S \subset V \\ & x_e \in \{0,1\} \qquad \forall \, e \in E. \end{aligned}$$

Where:

- $x_e = 1$ if edge e is selected, 0 otherwise
- $\delta(S)$ = edges with exactly one endpoint in S
- There are $2^{|V|} - 2$ cut constraints (exponentially many!) This makes them difficult to deal with them whether we are dealing with integer solution, or even in the LP relaxation during Branch and Bound.

4.3 Branch-and-Bound with Callbacks

Gurobi's branch-and-bound algorithm:

1. Solves LP relaxations at each node
2. Branches on fractional variables
3. Finds integer solutions
4. Prunes suboptimal branches

You'll use **callbacks** to inject cut constraints on-the-fly instead of enumerating them all upfront.

4.4 Two Types of Callbacks

4.4.1 1. User Cuts (`where == GRB.Callback.MIPNODE`)

When called: At fractional LP nodes during branch-and-bound

Purpose: Strengthen the LP relaxation by adding violated cuts

Your task:

- Get current fractional solution x^*
- Use Stoer–Wagner to find violated cuts
- Add them via `model.cbCut()`

Why useful: Tighter LP bounds ⇒ less branching ⇒ faster solve

Example:

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

    # Get fractional solution
    x_vals = model.cbGetNodeRel(x_vars)

    # Find violated cut using Stoer-Wagner
    # If found, add via:
    model.cbCut(lhs >= k)
```

There should be strong overlap between the code here and the solution of your Problem 2.

4.4.2 2. Lazy Constraints (`where == GRB.Callback.MIPSOL`)

When called: When Gurobi finds an integer solution

Purpose: Verify the *integer* solution satisfies ALL constraints (including exponentially many cuts)

Your task:

- Get current integer solution x^*
- Use Stoer–Wagner to check if the solution is k-edge-connected
- If violated, add cut via `model.cbLazy()`

Why needed: Gurobi doesn't know about our exponential constraints. Without lazy constraints, it might return an infeasible integer solution.

Example:

```
if where == GRB.Callback.MIPSOL:
    # Get integer solution
    x_vals = model.cbGetSolution(x_vars)

    # Check if k-connected using Stoer-Wagner
    # If violated, add via:
    model.cbLazy(lhs >= k)
```

4.5 Key Differences

	User Cuts (`cbCut`)	Lazy Constraints (`cbLazy`)
When	Fractional LP nodes	Integer solutions
Solution type	$x^* \in [0, 1]$	$x^* \in \{0, 1\}$
Purpose	Strengthen LP	Ensure feasibility
Required?	No (optimization)	Yes (correctness!)

	User Cuts (`cbCut`)	Lazy Constraints (`cbLazy`)
Gurobi callback	`GRB.Callback.MIPNODE`	`GRB.Callback.MIPSOL`
Add via	`model.cbCut()`	`model.cbLazy()`

4.6 What You Must Implement

Submit a file named `hw4_p4_callbacks.py` containing:

```
import networkx as nx
import gurobipy as gp
from gurobipy import GRB

def solve_survivable_integer(nodes, edges, costs, k):
    """
    Solve integer k-edge-connected network design using callbacks.

    Parameters
    ----------
    nodes : list
        List of node identifiers

    edges : list of tuples
        List of undirected edges as (u, v) pairs

    costs : dict
        costs[(u,v)] = cost to build edge (u,v)

    k : int
        Required edge connectivity (1 <= k <= 5)

    Returns
    -------
    result : dict with keys:
        {
            'edges': dict mapping edge tuple -> 0 or 1
                     Includes ALL edges (even those with value 0)

            'total_cost': float
                         Total cost of integer solution

            'num_user_cuts': int
                            Number of user cuts added

            'num_lazy_cuts': int
                            Number of lazy constraints added
        }
```

```
    Notes
    -----
    - Create integer variables: x_e in {0, 1}
    - Set LazyConstraints=1 parameter
    - Implement callback function that:
      * Adds user cuts at MIPNODE when LP is fractional
      * Adds lazy constraints at MIPSOL when integer solution found
    - Use Stoer-Wagner to find violated cuts in both cases
    """
    # TODO: Implement with callbacks
    raise NotImplementedError
```

4.7 Implementation Guide

4.7.1 Step 1: Create the Model

```
# Normalize edges
edges_norm = list({(min(u, v), max(u, v)) for u, v in edges})

m = gp.Model()
m.Params.OutputFlag = 0
m.Params.LazyConstraints = 1  # REQUIRED for lazy constraints!

# Integer variables
x = m.addVars(edges_norm, vtype=GRB.BINARY, name="x")

# Objective...
```

4.7.2 Step 2: Define Callback Function

```
# Track cuts added
stats = {"user": 0, "lazy": 0}

def callback(model, where):
    if where == GRB.Callback.MIPNODE:
        # Check if LP solution is available
        if model.cbGet(GRB.Callback.MIPNODE_STATUS) == GRB.OPTIMAL:
            # Get fractional solution
            x_vals = {e: model.cbGetNodeRel(x[e]) for e in edges_norm}


            # Find violated cut using Stoer-Wagner
            violated_cut = find_violated_cut(nodes, edges_norm, x_vals, k)

            if violated_cut is not None:
                # Add user cut
                stats["user"] += 1

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

```
        # Get integer solution
        x_vals = {e: model.cbGetSolution(x[e]) for e in edges_norm}


        # Check if k-connected
        violated_cut = find_violated_cut(nodes, edges_norm, x_vals, k)

        if violated_cut is not None:
            # Add lazy constraint
            stats["lazy"] += 1
```

4.7.3 Step 3: Solve with Callback

```
m.optimize(callback)
```

4.7.4 Step 4: Extract Solution

Return **all** edges with 0/1 values.

```
edge_vals = {e: int(round(x[e].X)) for e in edges_norm}

return {
    'edges': edge_vals,              # ALL edges, not just selected ones
    'total_cost': float(m.ObjVal),
    'num_user_cuts': stats["user"],
    'num_lazy_cuts': stats["lazy"],
}
```

4.7.5 Helper: Find Violated Cut

You can reuse some logic from Problem 2 (Hint: Hints ;).

4.8 Example Test Case

```
nodes = [0, 1, 2, 3]
edges = [(0,1), (1,2), (2,3), (3,0), (0,2), (1,3)]
costs = {e: 1.0 for e in edges}
k = 2

result = solve_survivable_integer(nodes, edges, costs, k)

print(f"Total cost: {result['total_cost']}")
print(f"Selected edges: {result['edges']}")
print(f"User cuts added: {result['num_user_cuts']}")
print(f"Lazy cuts added: {result['num_lazy_cuts']}")

# Expected: 4 edges forming 2-connected graph, cost = 4.0
```

4.9 Important Notes

1. **Must set `model.Params.LazyConstraints = 1`**: Otherwise Gurobi ignores lazy constraints!
2. **Performance**: Benchmark User cuts. If they slow down solving, feel free to only restrict them to root node.
3. **Correctness**: Lazy constraints ensure the integer solution is actually k-edge-connected.

[ ]: