CS498 Homework 2

January 28, 2026

1 Problem 1: Sherali–Adams & Maximum Independent Set

This is a **coding-only** problem. It will be autograded on Gradescope.

We will:

1. Implement the **basic LP relaxation** for Maximum Independent Set (MIS) on an arbitrary graph.
2. Understand why this basic LP can be **too weak** (gives an over-optimistic objective).
3. Strengthen the LP by adding extra **odd-cycle inequalities** that make the relaxation much tighter.
   - In the literature, these come from a "lift-and-project" procedure called the **Sherali–Adams hierarchy**, but for this problem, you only need to understand them as **extra valid inequalities** that improve and strengthens the LP.

1.1 Graph Input: Adjacency Matrix

The autograder will pass a graph to your function as a **NumPy adjacency matrix**:

- `adj` is a `numpy.ndarray` of shape `(n, n)`.
- `adj[i, j] = 1` if there is an edge between vertices `i` and `j`, otherwise `0`.
- The graph is:
  - **Undirected**: `adj[i, j] == adj[j, i]`
  - **Simple**: `adj[i, i] = 0` (no self-loops)

Example helper (you don't need to write this; the autograder will):

```
import numpy as np

def make_cycle5():
    n = 5
    adj = np.zeros((n, n), dtype=int)
    edges = [(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
    for u, v in edges:
        adj[u, v] = 1
        adj[v, u] = 1
    return adj
```

1.2 Decision Variables

For a graph with vertices $V = \{0, 1, \dots, n-1\}$, we introduce one variable per vertex:

- $x_i \in [0, 1]$ for each vertex $i \in V$.

Interpretation:

- $x_i = 1$: we choose vertex i (room i is in the independent set).
- $x_i = 0$: we don't choose vertex i.
- Fractional values (like $x_i = 0.4$) are allowed in the LP relaxation.

1.3 Part A – The Basic MIS LP Relaxation (for any graph)

The **Maximum Independent Set** problem asks for the largest set of vertices with no edges inside the set.

We model a **fractional relaxation** with this LP:

$$\max \quad \sum_i x_i \text{ s.t.} \qquad x_i + x_j \le 1 \quad \text{for each edge } (i, j),\ 0 \le x_i \le 1 \quad \text{for all } i.$$

Why is this reasonable?

- The constraint $x_i + x_j \le 1$ prevents us from simultaneously setting $x_i = x_j = 1$ on an edge, which would violate independence.
- The bounds $0 \le x_i \le 1$ ensure the variables are between 0 and 1.

However, this is only a **relaxation**: it allows fractional solutions that are not valid independent sets. On some graphs, the LP optimum can be strictly **larger** than the true maximum independent set size.

1.4 Part B – Why the Basic LP Can Be Too Weak: The 5-Cycle Example

Consider the **5-cycle** C_5: vertices $0, 1, 2, 3, 4$ in a ring.

- The true maximum independent set size on C_5 is **2**.
- But the basic LP has an optimal solution with all $x_i = 0.5$:
  - Each edge constraint: $x_i + x_j = 0.5 + 0.5 = 1$, so it is feasible.
  - Objective value: $\sum_i x_i = 5 \cdot 0.5 = 2.5$.

So the LP says "2.5 rooms". The relaxation is **too optimistic**.

1.5 Part C – Strengthening the LP by Adding More Inequalities

One standard way to improve an LP relaxation is to add more **valid inequalities**:

- An inequality is *valid* if **every** true independent set (with $x_i \in \{0, 1\}$) satisfies it.
- If an inequality is valid but **violated** by some fractional LP solution, then adding it will **cut off** that bad fractional solution and make the relaxation tighter.

1.5.1 Odd-Cycle Inequality

On the 5-cycle, it is impossible to choose **3** vertices without picking two adjacent ones, so no independent set can have three vertices in the cycle.

This can be expressed as the inequality:

$$x_0 + x_1 + x_2 + x_3 + x_4 \leq 2.$$

- Every independent set satisfies this (maximum of 2 vertices in a 5-cycle).
- The fractional solution with all $x_i = 0.5$ **violates** it (sum = 2.5).
- If we **add** this inequality to the LP, the optimum drops from 2.5 down to 2, which matches the true integral answer.

This is exactly what we mean by **strengthening** the LP: adding more inequalities that are true for all 0–1 solutions, but remove some fractional ones.

1.5.2 General Odd-Cycle Inequalities

More generally, for any **odd cycle** C of length $|C|$ in a graph, we can add:

$$\sum_{i \in C} x_i \leq \frac{|C| - 1}{2}.$$

Reason:

- In any independent set on an odd cycle of length $|C|$, you can pick at most $\frac{|C|-1}{2}$ vertices.
- So this inequality is valid for all **0–1** solutions.
- But it often cuts off fractional solutions of the basic LP.

These inequalities can be derived systematically using a "lift-and-project" method called the **Sherali–Adams hierarchy**. For this problem however, you do **not** need to know the full theory, just think of them as **extra constraints** that make the LP closer to the true combinatorial problem.

1.6 Part D – Using NetworkX to Find Odd Cycles

We'd like to automatically find all cycles in the graph and add these odd-cycle inequalities.

We'll use **NetworkX**, a Python graph library.

1.6.1 Installing NetworkX (on your machine)

On your own machine, you can install NetworkX with:

```
pip install networkx
```

On Gradescope, `networkx` will already be installed in the autograder environment.

1.6.2 Building a Graph from the Adjacency Matrix

Inside your solution file:

```
import numpy as np
import networkx as nx

def build_graph_from_adj(adj):
    # adj is a numpy array of shape (n, n)
    G = nx.from_numpy_array(adj)  # undirected simple graph
    return G
```

`nx.from_numpy_array(adj)`:

- Creates an undirected NetworkX graph.
- Nodes are labeled `0, 1, ..., n-1`.
- An edge `(i, j)` is added whenever `adj[i, j] != 0`.

1.6.3 Listing Simple Cycles

NetworkX can produce a **cycle basis** of the graph. A basis for cycles of a network is a minimal collection of cycles such that any cycle in the network can be written as a sum of cycles in the basis.

```
G = build_graph_from_adj(adj)
cycles = nx.cycle_basis(G)
```

- `cycles` is a list of simple cycles.
- Each cycle is a list of vertices (no repeated start/end vertex).
  - Example for a 5-cycle: `[0, 1, 2, 3, 4]`.

For each cycle `C` in `cycles`:

- If `len(C)` is **odd**, we add the linear inequality $\sum_{i\in C} x_i \le (|C|-1)/2$.

These are the "strengthening constraints" we want.

1.7 What You Must Implement

Create a file named **`hw2_p1_sherali_adams.py`** that defines **exactly** this function:

```
import numpy as np
import gurobipy as gp
import networkx as nx

def solve_mis_hierarchy(adj):
    """
    Compare the basic MIS LP relaxation and a strengthened LP with
    odd-cycle inequalities on an arbitrary graph.

    Inputs
    ------
    adj : numpy.ndarray, shape (n, n)
        Adjacency matrix of an undirected simple graph.
        - adj[i, j] = adj[j, i] in {0, 1}
        - adj[i, i] = 0

    Outputs
    -------
    x_basic : np.ndarray, shape (n,)
```

```
        Optimal solution of the basic LP relaxation:
            max sum_i x_i
            s.t. x_i + x_j <= 1 for each edge (i, j),
                 0 <= x_i <= 1.

    obj_basic : float
        Optimal objective value of the basic LP relaxation.

    x_sa : np.ndarray, shape (n,)
        Optimal solution of the strengthened LP with additional
        odd-cycle inequalities:
            For every odd cycle C found by networkx.cycle_basis(G):
                sum_{i in C} x_i <= (|C| - 1) / 2.

    obj_sa : float
        Optimal objective value of the strengthened LP.
    """
    # TODO: implement
    raise NotImplementedError
```

1.8 Implementation Requirements

You must:

1.8.1 1. Basic LP

Build and solve the **basic LP**:

- Variables:
  - `x[i]` for `i in {0, ..., n-1}`, continuous with `0 <= x[i] <= 1`.
- Objective:
  - Maximize $\sum_i x_i$.
- Edge constraints:
  - For each edge `(i, j)` where `adj[i, j] == 1`, add the constraint `x[i] + x[j] <= 1`.

1.8.2 2. Strengthened LP with Odd-Cycle Inequalities

Build and solve a **second LP** that is identical to the basic LP but with additional cycle constraints.

Return:

```
return x_basic_vals, obj_basic, x_sa_vals, obj_sa
```

1.9 What You Should Observe

Run your code (or check Gradescope feedback) on different graphs:

- **Triangle** K_3:

  - Basic LP optimum   1.5 (using $x_i = 0.5$).
  - Odd-cycle inequality for the triangle $\sum x_i \leq 1$ forces the optimum down to 1.
- **5-cycle** C_5:
  - Basic LP optimum   2.5 (all $x_i = 0.5$).
  - Odd-cycle inequality for the 5-cycle forces the optimum down to 2, the true MIS size.
- **4-cycle** C_4:
  - There are **no odd cycles**, so there are no extra inequalities to add.
  - The strengthened LP equals the basic LP; the objective values should match.

In all of these cases, you're seeing the same phenomenon: by **strengthening the LP with valid inequalities** (here, odd-cycle inequalities), you push the LP solution closer to the true combinatorial answer.

This is the core idea behind hierarchies like **Sherali–Adams**, they systematically generate families of such inequalities to **tighten** relaxations. Can you think of other inequalities that would tighten MIS for the clique for example? (not for submission)

---

2 Problem 2 — Modelling: The Gridlock Gambit (Coding + Written)

> *You've joined **TransiLogic AI**, a startup using optimization to reduce urban gridlock.* The city's road network has limited capacity, congestion costs money, and traffic demand must be met exactly. Your job is to **model traffic routing as a linear program**, solve it at scale, and understand **which roads are actually valuable**.

This problem has **two parts**:

1. **(Coding)** Formulate and solve a **minimum-cost flow LP** using Gurobi.
2. **(Conceptual)** Use **dual values (shadow prices)** to analyze *what-if* scenarios on a small network.

2.1 Background: What Is a Min-Cost Flow Problem?

A **min-cost flow problem** models the movement of a commodity (here: vehicles) through a directed network.

- **Nodes** represent intersections or hubs.
- **Directed arcs** represent roads.
- Each arc has:
  - a **capacity**: maximum vehicles per unit time,
  - a **unit cost**: fuel, tolls, congestion, or time per vehicle.

Some nodes **supply** vehicles, some **demand** vehicles, and others simply pass vehicles through.

The goal is to **route all required vehicles at minimum total cost**, subject to: - road capacities, - flow conservation at every node.

This problem has a clean and powerful **linear programming formulation**.

2.2 Network Model

We are given a directed graph $G = (V, E)$.

2.2.1 Nodes

Each node $n \in V$ has a supply value b_n:

- $b_n > 0$: node supplies flow (source),
- $b_n < 0$: node demands flow (sink),
- $b_n = 0$: transshipment node.

Supplies balance demands:

$$\sum_{n \in V} b_n = 0.$$

2.2.2 Arcs

Each directed arc $(u, v) \in E$ has:

- capacity $c_{uv} \geq 0$,
- unit cost $k_{uv} \geq 0$.

2.3 LP Formulation

2.3.1 Decision variables

For each arc $(u, v) \in E$:

$$f_{uv} \geq 0 \quad \text{flow on arc } (u, v).$$

2.3.2 Objective: Minimize total cost

$$\min \sum_{(u,v)\in E} k_{uv} f_{uv}.$$

2.3.3 Constraints

1. Capacity constraints (Can't direct flow to a road more than its capacity)

$$0 \leq f_{uv} \leq c_{uv} \qquad \forall (u, v) \in E.$$

2. Flow conservation (Cars are neither created nor destroyed) For each node $n \in V$:

$$\sum_{v:(n,v)\in E} f_{nv} - \sum_{u:(u,n)\in E} f_{un} = b_n.$$

Interpretation:
"What flows into a node must flow out, except at sources and sinks."

2.4 PART A — Coding: Large-Scale Min-Cost Flow

You will implement this LP in **Gurobi** and solve it for **large networks**.

The autograder will include graphs with **up to 100,000+ nodes and edges**.
You are expected to rely on the LP solver, **not** to write custom graph algorithms.

2.4.1 Data Passed to Your Function

Nodes and supplies

- `nodes`: list of node names (strings),
- `supply`: dictionary mapping node $\to b_n$.

Example:

```
nodes  = ["S", "A", "B", "C", "T"]
supply = {"S": 10, "A": 0, "B": 0, "C": 0, "T": -10}
```

Arcs A pandas DataFrame `arcs` with columns:

- `"from"`: tail node u,
- `"to"`: head node v,
- `"capacity"`: c_{uv},
- `"cost"`: k_{uv}.

2.4.2 What You Must Implement

Create a file named **`hw2_p2_flow.py`** that defines **exactly** this function:

```
import pandas as pd
import gurobipy as gp

def solve_gridlock(nodes, arcs, supply):
    """
    Solve a minimum-cost flow LP.

    Inputs
    ------
    nodes : list[str]
        Node names.

    arcs : pd.DataFrame
        Columns: 'from', 'to', 'capacity', 'cost'.

    supply : dict[str, float]
        Node supplies b_n (positive) and demands (negative).
        Must satisfy sum(b_n) = 0.

    Returns
    -------
    flow : dict[(str, str), float]
        flow[(u,v)] is the optimal flow on arc (u,v).

    total_cost : float
        Minimum total transportation cost.
    """
    # TODO
    raise NotImplementedError
```

2.4.3 Requirements

- Use **Gurobi** (`gurobipy`).
- All decision variables must be **continuous** and nonnegative.
- Do **not** hard-code:
  - number of nodes,
  - number of arcs,
  - node names,
  - capacities or costs.
- Your implementation must scale to **very large graphs**.
- Return **plain Python objects**, not Gurobi objects.

2.4.4 Important Observation (Not Directly Graded)

Even though you solve a **continuous LP**, it turns out that min-cost flow problems with integer data (i.e. integer cost and capacity) always admit **integer optimal flows**, and that is returned by Gurobi! You may notice this empirically in your outputs.

2.5 PART B — Interpretation: Shadow Prices & What-If Analysis

This part is **about understanding dual values**.

2.5.1 Toy Network:

Nodes:

$$\{S, A, B, C, T\}$$

From	To	Capacity	Cost
S	A	10	2.0
S	B	8	3.0
A	C	5	1.0
B	C	10	1.5
A	T	4	2.0
C	T	12	1.0

Supply: $b_S = +10$, Demand: $b_T = -10$.

2.5.2 Tasks

1. **Solve the baseline LP**
   - Report optimal flows using your solution from part A.
   - Report total cost.
   - Identify which capacity constraints are binding.
2. **Dual values (shadow prices)**
   - Report the dual value on each capacity constraint.
   - Interpret: *If you could add one unit of capacity to that road, how much would total cost change?*
3. **Scenario analysis: road closure**
   - Close arc (A $\to$C) by setting its capacity to zero.
   - Re-solve the LP.
   - How much does total cost increase?
   - Does this match the shadow-price prediction?
4. **Interpretation**
   - Which roads are bottlenecks?
   - Which capacities are most valuable?
   - How does the network reroute when a cheap link fails?
   - What economic story do the dual values tell about congestion and redundancy?

2.5.3 Deliverable for Part B

A short, clear write-up including:

- Baseline flows and cost,
- Shadow prices and interpretation,
- Cost comparison after the closure,
- A brief investment recommendation (which road should be expanded and why).

2.6 Final Note

This problem illustrates an important lesson on dual values: **Dual variables are not abstract math, they are prices.** They tell you which constraints matter, how fragile a system is, and where investment has real value.

3 Problem 3 — Robust Line Fitting (Coding)

You're helping a lab calibrate a noisy sensor. You suspect a roughly linear relationship between input (x) and measured output (y), but the data has outliers — so instead of least squares, you'll fit the line using ***linear programming****.*

This is a **coding-only** problem. It will be autograded on Gradescope.

3.1 Background

Ordinary regression finds a, b minimizing

$$\sum_i (y_i - (a + bx_i))^2.$$

That's a **least-squares** problem, nonlinear and very sensitive to outliers (because large residuals get squared).

If we instead minimize **absolute deviations** or **the largest deviation**, the problem becomes a **linear program**.

You'll implement two such models for the line $y = a + bx$.

3.2 Inputs

Your functions will take two 1D NumPy arrays:

```
x_data : np.ndarray of shape (n,)
y_data : np.ndarray of shape (n,)
```

They contain the x_i and y_i values of the observed points.

Your code must work for **any** numeric data (do not hard-code sizes or numbers).

Example for testing:

```
import numpy as np
x_data = np.array([0, 1, 2, 3, 4, 5], float)
y_data = np.array([1.0, 1.9, 2.2, 3.1, 4.5, 5.3], float)
```

3.3 Part A [50 points] L_1 (Sum of Absolute Deviations)

We want the line $y = a + bx$ minimizing the **total absolute error**:

$$\min_{a,b} \sum_i |y_i - (a + bx_i)|.$$

The absolute value makes this nonlinear, but we can linearize it by introducing **nonnegative variables** $e_i \geq 0$ representing the magnitude of each residual:

$ r_i = y_i - (a + b x_i). $

We require

$$\left\{ r_i \leq e_i, \ -r_i \leq e_i. \right.$$

These two inequalities together mean $|r_i| \le e_i$.

Then minimize the total deviation $\sum_i e_i$.

3.3.1 LP Summary

Variable	Meaning
a, b	intercept and slope
e_i	absolute deviation of point i (≥ 0

Objective:

$$\min \sum_i e_i$$

Constraints:

$$y_i - (a + bx_i) \le e_i,\ -(y_i - (a + bx_i)) \quad \le e_i.$$

3.4 Part B: L_∞ (Minimax Deviation)

Now we minimize the **largest absolute deviation**:

$$\min_{a,b} \max_i |y_i - (a + bx_i)|.$$

Introduce a single variable

$$t \ge 0$$

representing the maximum deviation.

Then think of a set of constraints that would enforce

$$t \ge \max_i |y_i - (a + bx_i)|$$

Hint: Use part A first to get rid of the absolute value. The objective is simply $\min t$.

3.4.1 LP Summary

Variable	Meaning
a, b	intercept and slope
t	maximum absolute deviation

Objective:

$$\min t$$

Constraints: ???

3.5 What You Must Implement

Create a file **`hw2_p3_calibration.py`** defining:

```
import numpy as np
import gurobipy as gp

def l1_line_fit(x_data, y_data):
    """
    Fit y = a + b x minimizing sum of absolute deviations (L1 norm).

    Returns
    -------
    a : float
        Intercept
    b : float
        Slope
    obj_value : float
        Optimal sum of absolute deviations.
    """
    # TODO: implement
    raise NotImplementedError


def linf_line_fit(x_data, y_data):
    """
    Fit y = a + b x minimizing the maximum absolute deviation (L-infinity norm).

    Returns
    -------
    a : float
        Intercept
    b : float
        Slope
    t_value : float
        Minimum possible max absolute deviation.
    """
    # TODO: implement
    raise NotImplementedError
```

3.6 Implementation Requirements

- Use **Gurobi** (`gurobipy`).
- Your code must work for any number of points.
- Both problems are **pure LPs** with continuous variables.
- In both:
  - Create variables (`a`, `b`, `e[i]` or `t`).
  - Add linear constraints as above.
  - Set and solve the LP with `model.optimize()`.

– Return the fitted coefficients and the optimal objective value.

3.7 Example (for local testing)

```
x = np.array([0, 1, 2, 3, 4, 5])
y = np.array([0.5, 1.2, 2.3, 2.8, 4.6, 5.4])

a1, b1, obj1 = l1_line_fit(x, y)
a_inf, b_inf, t_inf = linf_line_fit(x, y)

print("L1 fit:    y =", b1, "x +", a1, "   total |error| =", obj1)
print("L∞ fit:    y =", b_inf, "x +", a_inf, "   max |error| =", t_inf)
```

4 Problem 4 — Equivalent Forms of Linear Programming *(Written)*

Linear programs (LPs) can be written in many algebraically different but mathematically equivalent ways. In this problem, you'll show that these forms are all interchangeable, that is, any LP can be rewritten in any other "standard" form by adding only a small number of extra variables or constraints.

4.1 Background

Consider three common ways of expressing an LP:

1. **General form (mixed inequalities):**

$$\min_x \quad c^\top x \text{ s.t.} \qquad A_1x \le b_1,\, A_2x = b_2, \quad A_3x \ge b_3,\, x_j \text{ free for some } j.$$

2. **Standard (inequality) form:**

$$\min_x ; c^\top x \quad \text{s.t. } Ax \le b, ; x \ge 0.$$

3. **Canonical (equality) form:**

$$\min_x ; c^\top x \quad \text{s.t. } Ax = b, ; x \ge 0.$$

4.2 Part A — From general to standard form *(20 pts)*

Explain how to transform the **general** LP into **standard inequality form** using the following operations:

- Replace any variable that is unrestricted in sign ("free") by the difference of two *nonnegatives* $x_j = x_j^+ - x_j^-$.
- Convert all "($\ge$)" constraints into "($\le$)" constraints by multiplying by (-1).

Prove that these transformations do **not** change the optimal objective value, although they may introduce extra variables.

(Hint: each "free" variable introduces one new variable; each " " constraint can be flipped in place.)

4.3 Part B — From inequality to equality (slack form) *(20 pts)*

Show that any LP of the form

$$\min_x c^\top x \quad \text{s.t. } Ax \le b, ; x \ge 0$$

can be written equivalently as

$$\min_{x,s} c^\top x \quad \text{s.t. } Ax + s = b, ; x \ge 0, ; s \ge 0.$$

Explain why introducing one *slack variable* s_i per inequality constraint preserves the feasible region in one-to-one correspondence.

4.4 Part C — From equality to inequality form *(20 pts)*

Show how to reverse Part B: any equality constraint $a_i^\top x = b_i$ can be replaced by two inequalities. What happens to the number of constraints? Why does this again preserve the same optimal objective value?

4.5 Part D — Summary of Equivalence *(20 pts)*

Summarize all the transformations in one table, indicating:

- what gets added (new variables or new constraints),
- the typical number added per operation, and
- why none of these affect optimality (only the *representation*).

Example structure:

From → To	Additions	Rationale
General → Standard	free vars → 2 nonnegatives; flip “ ”	preserves feasible set
Standard → Canonical	add one slack per inequality	equality form, same opt
Canonical → Standard	replace each “=” by two “ ”	same feasible set

4.6 Part E — Conceptual Reflection *(20 pts)*

In your own words (150 words), explain why these forms are considered *equivalent* in optimization theory. What makes them interchangeable in proofs, algorithms (like simplex), or solver input formats?

(Your answer should emphasize that equivalence means: same feasible region in a higher-dimensional space and same optimal value, even if variable count changes slightly.)