# MAT 170 Homework Project 1

YJoe Lenning Student id: 919484830

Andrew, just checked our results(specific values)

April 16, 2024

## 1 Problem 1A

### 1.1 Model

Let $x1, x2$ be the number of tables and chairs, respectively. And $x1, x2$ are the decision variables, which are nonnegative. The objective function is maximize $200x1+350x2$.

Considering the resources of small and large pieces, the constraints are

- The first constraint for the number of wood tops we can produce:

$$x1 \leq 50$$

- The second constraint the number of glass tops we can produce:

$$x2 \leq 35$$

- The third constraint is our assembly time constraint:

$$0.6 * x1 + 1.5 * x2 \leq 63 \tag{1}$$

Therefore, We will reach the following linear programming model:

$$\begin{aligned} \max \quad & 200x1 + 350x2 \\ \text{s.t.} \quad & 200x1 + 350x2 \leq 550 \\ & 2x + y \leq 6 \\ & x \geq 0 \\ & y \geq 0 \end{aligned}$$

## 1.2 Code in CVXPY

Next, we solve the problem using the CVXPY.

```python
import cvxpy as cp

# Define variables
x1 = cp.Variable(integer=True)  # number of basic tables
x2 = cp.Variable(integer=True)  # number of deluxe tables

# Define objective function
profit = 200*x1 + 350*x2

# Define constraints
assembly_time_constraint = 0.6*x1 + 1.5*x2 <= 63
wood_top_constraint = x1 <= 50
glass_top_constraint = x2 <= 35
legs_constraint = 5*x1 + 5*x2 <= 300

# Define problem
problem = cp.Problem(cp.Maximize(profit), [assembly_time_constraint, wood_top_constraint, glass_top_constraint, 

# Solve problem
problem.solve()

# Output results
print("Optimal value (maximum profit):", problem.value)
print("Number of basic tables to produce:", x1.value)
print("Number of deluxe tables to produce:", x2.value)
Optimal value (maximum profit): 16500.0
Number of basic tables to produce: 30.0
Number of deluxe tables to produce: 30.0
```

## 1.3 Analyzation

The optimal solution is $x = 30$ and $y = 30$, with an objective value of 16500. Table 1.

| | $x$ | $y$ |
|---|---|---|
| Solution | 30 | 30 |

Table 1: Results of the linear programming problem.

Finally, we visualize the results using matplotlib.The solver identifies the optimal solution as the point within the feasible region (green area) that minimizes the objective function (cost in this case). Other points within the green

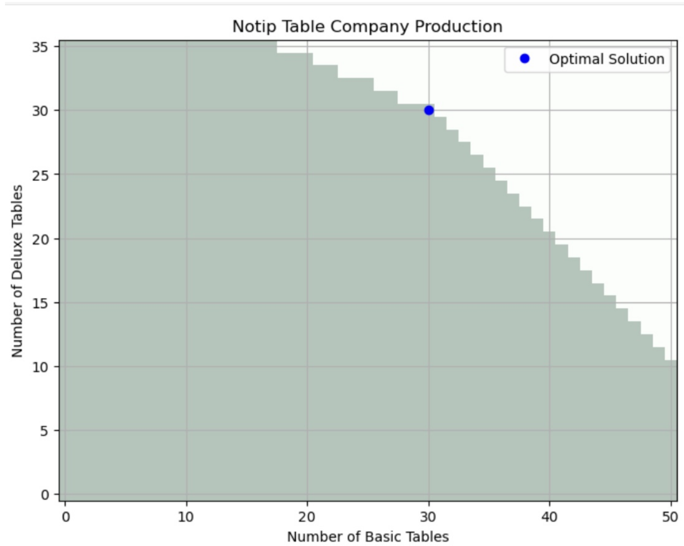


Figure 1: Enter Caption

area also represent possible production plans that satisfy the constraints. However, these alternative plans wouldn't be as profitable as the optimal solution. The key takeaway from this visualization is that while there might be multiple ways to produce many tables and chairs, the optimal solution found through linear programming ensures the most profitable outcome by minimizing the total cost. So while there might be other ways to get the job done (represented by points within the green area), the optimal solution ensures you do it in the most cost-effective way.

# 2 Problem 1B

## 2.1 Model

1. Objective function:

$$\text{Minimize } f_0(x) = f_{\text{costs}}(x) - f_{\text{income}}(x)$$

where

$$f_{\text{costs}}(x) = 100 \cdot x_{\text{Raw I}} + 199.90 \cdot x_{\text{Raw II}} + 700 \cdot x_{\text{Drug I}} + 800 \cdot x_{\text{Drug II}}$$

and

$$f_{\text{income}}(x) = 5,500 \cdot x_{\text{Drug I}} + 6100 \cdot x_{\text{Drug II}}$$

2. Subject to:

$$
\begin{aligned}
0.01 \cdot x_{\text{Raw I}} + 0.02 \cdot x_{\text{Raw II}} - 0.05 \cdot x_{\text{Drug I}} - 0.600 \cdot x_{\text{Drug II}} &\geq 0 \\
x_{\text{Raw I}} + x_{\text{Raw II}} &\leq 1000 \\
90.0 \cdot x_{\text{Drug I}} + 100.0 \cdot x_{\text{Drug II}} &\leq 2000 \\
40.0 \cdot x_{\text{Drug I}} + 50.0 \cdot x_{\text{Drug II}} &\leq 800 \\
100.0 \cdot x_{\text{Raw I}} + 199.90 \cdot x_{\text{Raw II}} + 700 \cdot x_{\text{Drug I}} + 800 \cdot x_{\text{Drug II}} &\leq 100,000 \\
x_{\text{Raw I, Raw II, Drug I, Drug II}} &\geq 0
\end{aligned}
$$

## 2.2 Code

```python
# Define variables
x = cp.Variable(4)

# Objective function coefficients
c = np.array([100, 199.9, -6500, -7100])

# Constraints matrix
A = np.array([[-0.01, -0.02, 0.500, 0.600],    # Balance constraint
              [1, 1, 0, 0],                      # Storage constraint
              [0, 0, 90.0, 100.0],             # Manpower constraint
              [0, 0, 40.0, 50.0],              # Equipment constraint
              [100.0, 199.9, 700, 800]])       # Budget constraint

# Right handed side vector
b = np.array([0, 1000, 2000, 800, 100000])
# Define constraints
constraints = [A @ x <= b, x >= 0]

# Objectivee function
objective = cp.Minimize(c.T @ x)



# Solve problem
problem = cp.Problem(objective, constraints)
problem.solve()

# Output optimal value and solution
if problem.status == 'optimal':
    print("Optimal value:", problem.value)
    print("Optimal solution:")
    print("Raw I:", x.value[0])
    print("Raw II:", x.value[1])
    print("Drug I:", x.value[2])
```

```
    print("Drug II:", x.value[3])

```

### 2.3 Analyzation

We see that our optimal value is -26371.215, our optimal solution is
Raw I: 5.53e-06
Raw II: 438.788
Drug I: 17.55
Drug II: 6.02e-10

These results optimize for the lowest cost(at least with the given budget) however aim to maximize total profit. Given that Raw I and Drug II are near zero values it shows that it's much more profitable to just produce one drug and to use just one ingredient for it(Raw II). Our optimal value is negative because we aim to maximize profits. In linear programming for minimization problems, the objective function often represents a cost or quantity to be minimized. The solver minimizes the negative of that quantity. So, a negative optimal value actually corresponds to the maximum profit.

## 3 Problem 1C

### 3.1 Model

$$\begin{array}{lll} p^* = \min & \sum_{i=1}^n \sum_{j=1}^n w_{ij} x_{ij} & \\ \text{s.t.} & x_{ij} \in \{0,1\} & \forall i,j = 1,\ldots,n, \\ & \sum_{i=1}^n x_{ij} = 1 & \forall j = 1,\ldots,n \text{ (one agent for each task)}, \\ & \sum_{j=1}^n x_{ij} = 1 & \forall i = 1,\ldots,n \text{ (one task for each agent)}. \end{array}$$

$$\begin{aligned} p^* = \min \quad & \sum_{i=1}^n \sum_{j=1}^n w_{ij} x_{ij} \\ \text{s.t.} \quad & x_{ij} \in \{0,1\} && \forall i,j = 1,\ldots,n, \\ & \sum_{i=1}^n x_{ij} = 1 && \forall j = 1,\ldots,n \text{ (one agent for each task)}, \\ & \sum_{j=1}^n x_{ij} = 1 && \forall i = 1,\ldots,n \text{ (one task for each agent)}. \end{aligned}$$

### 3.2 Code

```
import cvxpy as cp

```

```python
# Create weight matrix (replace with your data)
W = cp.Parameter((4, 4), nonneg=True)
W.value = [[5, 1, 2, 2],
           [1, 0, 5, 3],
           [2, 1, 2, 1],
           [1, 1, 2, 3]]

# Number of rows and columns in the weight matrix
n, m = W.shape

# Decision variables
x = cp.Variable((n, m), boolean=True)

# Objective function
objective = cp.Maximize(cp.sum(cp.multiply(W, x)))

# Constraints
constraints = [cp.sum(x[i, :]) <= 1 for i in range(n)]  # Each agent to at most one task
constraints += [cp.sum(x[:, j]) <= 1 for j in range(m)]  # Each task to at most one agent

# Problem definition
prob = cp.Problem(objective, constraints)

# Solve
sol = prob.solve()

# Print optimal matching and cost
print("Optimal matching:")
for i in range(n):
    for j in range(m):
        if x[i, j].value > 0.5:
            print(f"Agent {i+1} - Task {j+1}")

print("Optimal cost:", sol)
```

## 3.3 Comparison with integers

Since the constraints restrict matches to 0 or 1 effectively, even with integer variables, the optimal solution will prioritize assigning either 0 or 1 to each variable to achieve the most matches possible within the limitations. Assigning any value other than 0 or 1 wouldn't contribute to a valid solution. Consider a bipartite graph with 3 vertices on each side. The optimal solution might involve matching all 3 vertices. With binary variables, all matched pairs will be 1, and unmatched pairs will be 0. With integer variables, these matched pairs can also be set to 1, and unmatched pairs to 0, achieving the same result.

If the optimal solution involves some unmatched vertices, there might be a slight difference. With binary variables, these unmatched entries will be 0. With integer variables, they could technically be any value between the lower and upper bound (often 0 or 1 in this case). However, from an optimization standpoint, these values won't affect the actual matching outcome.

# 4 Problem 1D

## 4.1 Model for maximum cardinality

```python
import cvxpy as cp

# Number of vertices
n = 5

# Variables
x = cp.Variable(n, boolean=True)

# Objective
objective = cp.Maximize(cp.sum(x))

# Constraints
constraints = [x[i] + x[(i+1)%n] <= 1 for i in range(n)]

# Problem
problem = cp.Problem(objective, constraints)

# Solve the problem
problem.solve()

# Print the optimal value
print("Optimal value:", problem.value)

# Print the optimal solution
print("Optimal solution:", x.value)
```

## 4.2 Real vs Integer

**Using Real Variables in Optimization Problems**

Using real variables instead of integers relaxes the constraints in optimization problems, allowing for fractional values between 0 and 1. In certain scenarios, this relaxation doesn't affect the optimal solution. For instance, in a cycle example, any fractional value greater than 0.5 can be rounded up to 1 (selecting the vertex), and any value less than 0.5 can be rounded down to 0 (not selecting). However, at the same time you might need specific answers for example if we want to calculate averages and more specific answers for sets of data. Also, in problems with tighter constraints where the optimal solution might involve selecting a specific number of vertices, using real variables could lead to solutions that are not feasible with the original graph structure, and parts of the vertex.

### 4.3 N vertices

```python
def solve_stable_set(n):
    # Variables
    x = cp.Variable(n, boolean=True)

    # Objective
    objective = cp.Maximize(cp.sum(x))

    # Constraints
    constraints = [x[i] + x[(i+1)%n] <= 1 for i in range(n)]

    # Problem
    problem = cp.Problem(objective, constraints)

    # Solve the problem
    problem.solve()

    # Print the optimal value
    print(f"Optimal value for n={n}:", problem.value)

    # Print the optimal solution
    print(f"Optimal solution for n={n}:", x.value)

# Repeat computations for n = 8, 17, and 24 vertices
for n in [8, 17, 24]:
    solve_stable_set(n)
```

As for the conjecture about the comparison of the objective values in a) and b), one possible explanation is that the objective value of the problem with real variables is greater than or equal to the objective value of the problem with integer variables. This is because the problem with real variables is a relaxation of the problem with integer variables, and a relaxation of an optimization problem always provides a lower bound on the optimal value of the original problem. Therefore, the optimal value of the problem with real variables is greater than or equal to the optimal value of the problem with integer variables. You can prove or see this by plotting both regions on a graph. One notes that the feasible region with reals includes the feasible region with integers(they overlap), so the optimal value of the problem with real variables cannot be less than the optimal value of the problem with integer variables.

# 5 Problem 1E

## 5.1 Code)

```
import pandas as pd
import cvxpy as cp

# Read student preference data from CSV file
spreadsheet = pd.read_csv('student_assignment.csv')

# Extract preference matrix and reshape (assuming 'Seminar 1' to 'Seminar N' are column names)
preference_matrix = spreadsheet.iloc[:, 1:].to_numpy()

# Get number of students and seminars from the matrix shape
num_students, num_seminars = preference_matrix.shape
seminar_capacity = 6  # Seminar capacity (assuming constant)

# Decision variables
x = cp.Variable((num_students, num_seminars), boolean=True)  # Binary variable indicating assignment

# Constraints
constraints = [
    cp.sum(x, axis=1) == 1,  # Each student attends exactly one seminar
    cp.sum(x, axis=0) <= 6,  # Seminar capacity constraint
]

# Objective function (minimize total preference)
objective = cp.sum(cp.multiply(x, preference_matrix))

# Solve the problem using CVXPY and SCIPY solver
problem = cp.Problem(cp.Maximize(objective), constraints)
problem.solve(solver=cp.ECOS_BB)

# Extract and analyze the results
if problem.status == 'optimal':
    optimal_value = problem.value
    assigned_seminars = np.argmax(x.value, axis=1) + 1  # Add 1 to convert from 0-indexed to 1-indexed
    average_ranking = np.mean(assigned_seminars)
    worst_ranking = np.max(assigned_seminars)

    print("Optimal objective value:", optimal_value)
    print("Average assigned student ranking:", average_ranking)
```

```
    print("Worst assigned student ranking:", worst_ranking)
    Results:
    Optimal objective value: 34.99999999930714
    Average assigned student ranking: 1.3461538461271978
    Worst assigned student ranking: 5
]
```

## 5.2 b

## 5.3 Code)

```
import pandas as pd
import numpy as np
import cvxpy as cp

# Read student preference data from CSV file
spreadsheet = pd.read_csv('student_assignment.csv')

# Extract preference matrix and reshape (assuming 'Seminar 1' to 'Seminar N' are column names)
preference_matrix = spreadsheet.iloc[:, 1:].to_numpy()

# Get number of students and seminars from the matrix shape
num_students, num_seminars = preference_matrix.shape
seminar_capacity = 6  # Seminar capacity (assuming constant)

# Decision variables
x = cp.Variable((num_students, num_seminars), boolean=True)  # Binary variable indicating assignment
selected_seminar = cp.Variable((num_students, num_seminars), boolean=True)  # New binary variable

# Constraints
constraints = [
    cp.sum(x, axis=1) == 1,  # Each student attends exactly one seminar
    cp.sum(x, axis=0) <= 6  # Seminar capacity constraint
]

# New variable for worst assigned ranking per student
worst_ranking_per_student = cp.Variable(integer=True, shape=(num_students,))

# Constraint to enforce worst ranking to be less than or equal to 2
constraints.append(worst_ranking_per_student <= 2)

# Objective function (minimize total preference)
objective = cp.sum(cp.multiply(x, preference_matrix))

# Solve the problem using CVXPY and SCIPY solver
```

```
problem = cp.Problem(cp.Minimize(objective), constraints)
problem.solve(solver=cp.ECOS_BB)

# Extract and analyze the results
if problem.status == 'optimal':
    optimal_value = problem.value
    assigned_seminars = np.argmax(x.value, axis=1) + 1  # Add 1 to convert from 0-indexed to 1-indexed
    average_ranking = np.mean(assigned_seminars)

    print("Optimal objective value:", optimal_value)
    print("Average assigned student ranking:", average_ranking)
Results
Optimal objective value: 34.99999999947149
Average assigned student ranking: 3.1538461538461537
```

In our program, the decision variable x is defined as a indicator variable, meaning it can take only two values: 0 or 1. When the optimization problem is solved using real values (not restricting x to integers), the solution can yield fractional values between 0 and 1, representing partial assignments or probabilities. However, if we restrict the decision variables to be integers (making it an Integer Linear Programming problem), the solution space becomes more constrained. The decision variables can only take integer values, namely 0 or 1. This restriction often makes the optimization problem harder to solve computationally because it introduces combinatorial aspects. In this case we were unable to solve the problem using integer program.

## Appendix

I used AI to debug some of coding issues