MATH 441 Learning Portfolio

Derryl Sayo

Table of contents

1 Introduction

To the Reader,

The following set of articles documents my knowledge gained while researching various topics in mathematical optimization, specifically the design of the models and solution methods used in linear and integer programming and other forms of general convex optimization. Included are a set of artifacts that aim to provide insights into the process of researching and formulating various optimization problems to help others better understand the mathematical representations for a variety of optimization problems.

Thoughout creating these artifacts and this class, I think that the most important lesson I learned is how to break down complex problems into more digestible pieces and build up solutions from smaller atoms. In each of the following articles, I follow a specific format of introducing a problem in parts such as variables, assumptions, and constraints because simply trying to figure out the grand design of a problem can be too overwhelming and I have even caught myself almost giving up because some problems seem too complicated at first. Laying out the facts and building up solutions has proven to be extremely effective in both academia and in my work as a software developer.

I would like to propose a grade of 90/100 for this portfolio. I believe that I have demonstrated my learning by explaining a variety of applications of optimization clearly and concisely and extrapolated my knowledge by solving original examples to the end of each report. However, I do believe there is always room for improvement specifically in the depth of the topics covered.

Overall, I believe that taking this course was had a meaningful impact in how I work through problem solving and critical thinking tasks inside and outside of academics. I have always been interested in learning about how mathematics is utilized in the world around us and I am going to continue learning about the beautiful results that math presents by applying the techniques and lessons I have learned creating this portfolio.

Sincerely,

Derryl Sayo

2 Linear Programming: Exhaustive Search vs. Simplex Method

```
from random import randint
import numpy as np
import itertools
import scipy
import time
import matplotlib.pyplot as plt
```

2.1 What is Linear Programming?

Linear programming is defined by the standard form:

$$\begin{aligned} \text{Maximize:} \quad & \mathbf{c}^T\mathbf{x} \\ \text{Subject To:} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned}$$

Where $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{x} \in \mathbb{R}^n$, A is a $m \times n$ matrix, $\mathbf{b} \in \mathbb{R}^m$.

The x_i are refered to as **decision variables** whose values are to be decided and are restricted by the set of m **constraints** of the form:

$$a_{i,0}x_{i,0} + \cdots + a_{i,n-1}x_{i,n-1} \begin{Bmatrix} \geq \\ = \\ \leq \end{Bmatrix} b_i$$

The **objective function** is a linear combination of the n decision variables with weights c_i written as:

$$c_0x_0 + c_1x_1 + \cdots + c_{n-1}x_{n-1}$$

In other words, linear programming is a way to model optimization problems with a linear objective function restricted by linear equality and inequality constraints.

Solving an LP problem entails finding the set of decision variables that will lead to the maximum/minimum value of the objective function while satisfying all constraints.

2.2 Different Ways to Solve Linear Programming Problems

The Fundamental Theorem of Linear Programming (see Wikipedia contributors 2022a) states that the optimal value of a bounded linear programming problem will occur at the intersection of n constraint hyperplanes (ie. a vertex of the convex polytope enclosing the feasible set of solutions).

At first glance, a naive approach would be to enumerate every single vertex, check each vertex's feasibility, and then calculate the max objective value over all vertices. However, as we will see later, this approach will run into complications as the scale of the problems increase.

During the mid 1900s, linear programming became vital in the efforts of WW2 and ecomonics post war with many economists and mathematicians studying this class of problems independently (George B. Dantzig 1990). The **simplex method**, developed by George Bernard Dantzig and detailed in *Linear Programming and Extensions* (George Bernard Dantzig 1963), arose as an efficient method to solve linear programming problems by iterating through adjacent feasible vertices until the optimal solution is reached.

Other methods to solve linear programming problems include a family of interior-point methods which are not covered here.

Below we will detail the first two methods for solving linear programming problems and compare the runtimes of each.

2.2.1 Geometry of Linear Programming

Before we can iterate through the vertices, we first establish how we can programmatically find these vertices.

LP problems have linear constraints which can be represented as hyperplanes of dimension $n-1$ which make up a bounded polytope in $\mathbb{R}^n$. A vertex of a polytope in $\mathbb{R}^n$ is a point on its boundary at the intersection of n hyperplanes.

The optimal value of a bounded linear programming problem will be at one of these vertices. So, for each combination of n constraints from the total pool of $n+m$ constraints (ie. m constraints defined explicitly in A and n decision variable non-negativity constraints defined implicitly) we pick n rows from A and b and solve $Ax = b$ for the intersection. If we do this for

each $\binom{n+m}{n}$ constraint intersections, we will have found every single vertex possible (feasible and infeasible) which we can then check for feasibility and then optimality.

2.2.2 Exhaustive Search by Enumerating All Vertices

To be able to use linear programming solvers, we must create random LP problems. Below is a simple generator that returns the A, b, c matrices/vectors with randomly selected dimensions m and n:

```
def generate_random_LPP():
        """
        Generate a random linear programming problem with `n` decision
 ↪  variables and `m` constraints.

        Returns:
        - A matrix `A` of LP constraints
        - A vector `b` of LP constraint values
        - The number of decision variables `n`
        """

        # Number of constraints in the A matrix
        m = randint(1, 12)

        # Number of decision variables
        n = randint(1, 12)

        A = np.rint(np.random.rand(m, n) * 100)
        b = np.rint(np.random.rand(m)  * 100)
        c = np.rint(np.random.rand(n) * 100)

        return A, b, c
```

Then, we create a python function that will iterate through all feasible vertices and find the maximal objective value for the optimal solution:

```
def get_vertices(A, b):
        """
        Find all vertices for an LP problem defined by A and b.

        Params:
        - A: m x n matrix of constraints (LHS of constraint inequalities)
```

```
        - b: m x 1 matrix of constraint values (ie. RHS of constraint
↪  inequalities)

        Returns:
        A list of feasible vertices
        """

        m = np.shape(A)[0]
        n = np.shape(A)[1]

        # Use itertools to get all n-tuple combinations from the m + n
         ↪  constraints
        row_index_combinations = itertools.combinations(range(m + n), n)

        # Since LP in standard form implicitly defines the x >= 0 constraint
         ↪  which corresponds to the x_i = 0 hyperplane
        # So, we vstack the identity matrix onto A and hstack a vector of 0s
         ↪  to b to get
        # all m + n constraint hyperplanes
        A_stacked = np.vstack((A, np.eye(n)))
        b_stacked = np.hstack((b, np.zeros(n)))

        feasible_vertices = []

        for row_combo in row_index_combinations:
                # Select n constraints
                A_constraints = A_stacked[list(row_combo), :]
                b_constraints = b_stacked[list(row_combo)]

                x = None

                try:
                  # Solve for the intersection of n constraint hyperplanes.
                   ↪  If
                  # we cannot solve due to a singular matrix, catch the error
                  # and continue
                  x = np.linalg.solve(A_constraints[:, :n], b_constraints)

                  # Set too small values to 0 (account for floating point
                   ↪  errors)
                  x[(np.abs(x) < 1e-14)] = 0
```

```
                # Check that the vertex satisfies non-negative constraints
                 ↪  and all inequality constraints
                # within a set tolerance
                if np.all(x >= 0) and np.all((A @ x) <= (b + 1e-10)):
                  feasible_vertices.append(x)
              except:
                pass

        return feasible_vertices


def LP_exhaustive_search(A, b, c):
        """
        Solve the given linear programming problem by exhaustive search
 ↪  through all vertices.

        Params:
        - A: m x n matrix of constraints
        - b: m x 1 vector of constraint inequality values
        - c: 1 x n vector of objective function coefficients

        Returns:
        - Optimal objective value of min -c * x
        """
        feasible_vertices = get_vertices(A, b)

        objective_values = []

        # Check the objective value for each of the found vertices
        for vertex in feasible_vertices:
                vertex_obj_val = np.dot(-c, vertex)
                objective_values.append(vertex_obj_val)

        return np.min(objective_values)
```

2.2.3 Solving LP Problems by Simplex Method Using SciPy

Detailed below is a python function which solves the given linear programming problem using `scipy.optimize.linprog` (SciPy Developers 2024) using the simplex method:

```
def LP_simplex(A, b, c):
        """
        Wrapper to call scipy.optimize.linprog with method="simplex" for
 ↪  consistency
        """
        return scipy.optimize.linprog(-c, A_ub=A, b_ub=b, method="simplex")
```

2.3 Comparing Exhaustive Search vs. Simplex Method

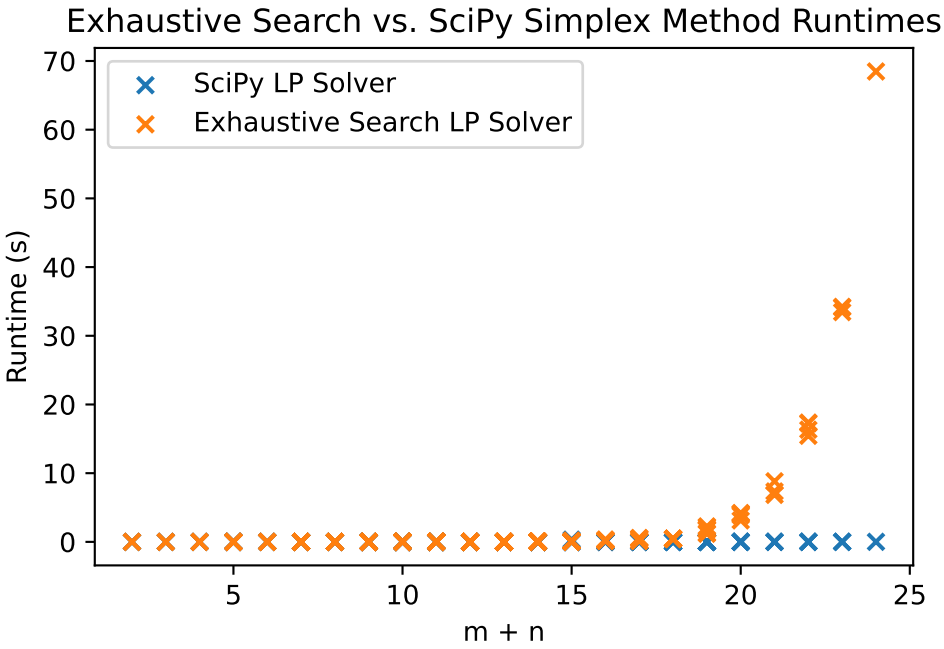
We can now run compare these two routines. Below we run 150 iterations of randomized LP problems of varying sizes and measure the runtimes of both the exhaustive search and the SciPy solver:

```
num_iterations = 150

scipy_test_results = []
exhaustive_test_results = []

for _ in range(1, num_iterations):
    A, b, c = generate_random_LPP()
    m, n = np.shape(A)

    t_scipy = time.time()
    scipy_res = LP_simplex(A, b, c)
    duration_scipy = time.time() - t_scipy
    scipy_test_results.append((m + n, duration_scipy))

    t_exhaustive = time.time()
    exhaustive_res = LP_exhaustive_search(A, b, c)
    duration_exhaustive = time.time() - t_exhaustive
    exhaustive_test_results.append((m + n, duration_exhaustive))

    # Check that the results are the same to 5 decimal places
    assert(np.round(scipy_res.fun, 5) == np.round(exhaustive_res, 5))
```

2.3.1 Visualizing Exhaustive Search vs. Simplex Method

Below is a figure comparing the runtimes between exhaustive search and the simplex method for problems of size $m + n$:

```
x1, y1 = zip(*scipy_test_results)
x2, y2 = zip(*exhaustive_test_results)

plt.scatter(x1, y1, label="SciPy LP Solver", marker="x")
plt.scatter(x2, y2, label="Exhaustive Search LP Solver", marker="x")

plt.xlabel("m + n")
plt.ylabel("Runtime (s)")
plt.title("Exhaustive Search vs. SciPy Simplex Method Runtimes")
plt.legend()

plt.show()
```



With the current implementation, the exhaustive search method's runtime grows exponentially as the problem size grows and already takes a non-trivial amount of time for small problems of size $n + m \approx 25$. Meanwhile, the simplex method remains efficient and doesn't grow rapidly.

2.4 Runtime Analysis

In order to compare the efficiency of these two algorithms we first need to establish some notion of quantifiable measures and distinguish between worst case and average case runtimes.

Oftentimes, we use some function on the number of operations required in a given routine.

In the case of linear programming solvers, we know that the optimal value exists at a vertex of the feasible region and that the basis of the simplex algorithm is iterating through adjacent vertices with increasing objection function value so a sensible measure would be to see how much many vertices we must check before arriving at the optimal solution.

2.4.1 Worst Case Time

In regards to the simplex method/other forms of vertex based solvers, the "worst case" runtime can be thought of as "if I take the longest route possible, how many vertices must I pass through"? For a problem with m constraints and n decision variables, each vertex exists as the intersection of n constraint hyperplanes. So, the number of vertices is $n + m$ choose n, or:

$$\binom{n+m}{n} = \frac{(n+m)!}{n!m!}$$

Written as the fraction of factorials, it becomes clear that this grows exponentially relative to problem size. For example, a problem with 12 constraints and 10 decision variables in the worst-case needs to check $\binom{22}{10} = 646646$ vertices. For non-trivial LP problems, this number quickly becomes too large to viably search through every vertex until optimality is reached.

2.4.2 Average Case Time

As seen in the figure above, practical LP solvers' runtime does not grow exponentially even though the theoretical worst-case runtime does grow exponentially. Explanations may include that the exhaustive search implementation detailed above requires solving a linear system of equations for every $n+m$ choose n vertices without extra heurstics to trim vertices outside the feasible regions compared to the actual geometry of the problem allowing for very few pivots to reach optimality.

Further research has gone into why the simplex algorithm performs well under most circumstances. In "Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time" (Spielman and Teng 2003), Spielman and Teng have shown that under "smoothed analysis", the simplex method has polynomial-time complexity rather than exponential.

2.5 Conclusion

Above, we have detailed code to generate and solve linear programming problems using exhaustive search through all vertices and have seen that the runtime grows exponentially when compared to the more stable simplex algorithm. Althrough both algorithms theoretically have the same worst case, recent analysis has shown why the simplex algorithm usually only takes polynomial time.

When reading further into this topic, future topics of interest include a more in-depth look into the smoothed analysis paper on the practical efficiency of the simplex algorithm, interior point LP solvers, and the Klee-Minty cube for a problem designed to demonstrate worst-case performance of the simplex algorithm.

3 Optimal Assignment of the Keyboard Number Row

```
import math
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
```

```
(CVXPY) Apr 15 11:53:58 AM: Encountered unexpected exception importing solver
GLOP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
(CVXPY) Apr 15 11:53:58 AM: Encountered unexpected exception importing solver
PDLP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
```

3.1 Background

On a standard computer keyboard, the base-10 numbers are arranged in sequential order:

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

As context, I have designed and built ergonomic mechanical keyboards in hopes of reducing wrist, hand, and finger strain as a large portion of my work is done digitially through programming, note-taking, etc. My current design iteration reduces the number of keys from a standard 108 to 48 arranged in a 4×12 grid (ie. "ortholinear") fashion. As a result, I have had to move many of the characters to multiple layers accessed by holding buttons underneath my thumbs. In the small keyboards community, these layers are mostly left as an "exercise

for the reader" where empirical experimentation leads to the variety of keymappings shared around the internet.

As a result, my compromise/solution for typing the numbers is to access it via holding a layer button and assigning the home row to the sequential order as noted above with the left pink on 0 and the right on 9. However, this sequential order does not reflect the frequency these numbers appear in text. For example, in my experience 0 has been a widely used character in typing dates, prices, etc. yet it is placed underneath my second weakest finger the right pinky.

In pursuit of a solution, one idea would be to use integer programming as a canonical assignment problem to assign each number to keyboard location which we will explore here.

3.2 Problem Statement

On a keyboard with the base-10 numbers arranged in sequential order in a single row, we have $i = 0, \dots, 9$ positions with $i = 0$ representing the left pinky and $i = 9$ representing the right pinky and $j = 0, \dots 9$ numbers.

Assign each position to a number to minimize the cost of typing frequently used numbers.

3.2.1 Decision Variables

Let x_{ij} be assignment of position i with number j. $x_{ij} = 1$ if the assignment is true and 0 if not.

```
X = cp.Variable((10, 10), integer=True)
```

3.2.2 Costs

The cost of assigning a number to a location comes is the combination of two factors:

1. The relative comfort level of each finger pressing a key. This will be subjective and we will be able to parametrize it to suit each individual. Higher values imply less comfortable so should incur a higher cost.
2. The frequency ranking of the number in text. Higher values are assigned to more frequently used numbers as there is more repeated finger stress. Some ideas where this can be generated is by counting frequencies by feeding in a training dataset (eg. Linux kernel code, online shopping prices…) however as a proof of concept the rankings are assigned by personal experience.

We take the outer product of these two vectors to form a 10×10 matrix where each position c_{ij} represents a cost of assigning finger i to number j:

```
# Ranking the home row finger positions from 1-10
# NOTE: The index is the finger position.
comfort_levels = [10, 8, 3, 4, 6, 7, 1, 2, 5, 9]

# Approximate rankings for number frequency in
# NOTE: The index is the number on the keyboard.
frequency_ranking = [1/1, 1/2, 1/3, 1/4, 1/5, 1/7, 1/8, 1/9, 1/6, 1/10]

# Construct the cost matrix
C = np.outer(comfort_levels, frequency_ranking)

print(np.round(C, 3))
```

```
[[10.     5.     3.333  2.5    2.     1.429  1.25   1.111  1.667  1.   ]
 [ 8.     4.     2.667  2.     1.6    1.143  1.     0.889  1.333  0.8  ]
 [ 3.     1.5    1.     0.75   0.6    0.429  0.375  0.333  0.5    0.3  ]
 [ 4.     2.     1.333  1.     0.8    0.571  0.5    0.444  0.667  0.4  ]
 [ 6.     3.     2.     1.5    1.2    0.857  0.75   0.667  1.     0.6  ]
 [ 7.     3.5    2.333  1.75   1.4    1.     0.875  0.778  1.167  0.7  ]
 [ 1.     0.5    0.333  0.25   0.2    0.143  0.125  0.111  0.167  0.1  ]
 [ 2.     1.     0.667  0.5    0.4    0.286  0.25   0.222  0.333  0.2  ]
 [ 5.     2.5    1.667  1.25   1.     0.714  0.625  0.556  0.833  0.5  ]
 [ 9.     4.5    3.     2.25   1.8    1.286  1.125  1.     1.5    0.9  ]]
```

3.2.3 Objective Function

We try to find the best *overall* assignment by minimizing the total cost. This can be written as the objective function $\min \sum_{i,j} c_{ij} x_{ij}$

```
objective = cp.Minimize(cp.sum(cp.multiply(C,X)))
```

3.2.4 Constraints

Standard to the assignment problem we have the constraints that

1. Each position is assigned to a single number:

$$\sum_j x_{ij} = 1$$

2. Each number is assigned to a single position:

$$\sum_i x_{ij} = 1$$

```
constraint1 = [X >= 0]
constraint2 = [cp.sum(X[i,:]) == 1 for i in range(10)]
constraint3 = [cp.sum(X[:,j]) == 1 for j in range(10)]
constraints = constraint1 + constraint2 + constraint3
```

We can add other constraints to further improve the comfort of the assignment. Some ideas that we can explore is adding a limit/penalty to typing multiple frequently used numbers on the same hand, moving commonly used bigrams (ie. different numbers used in succession) to different hands.

3.2.5 Solution

We use CVXPY to solve this integer programming problem:

```
problem = cp.Problem(objective, constraints)
problem.solve()
X.value
```

```
array([[-0., -0., -0., -0., -0., -0., -0., -0., -0.,  1.],
       [-0., -0., -0., -0., -0., -0.,  1., -0., -0., -0.],
       [-0., -0.,  1., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0.,  1., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0.,  1., -0.],
       [-0., -0., -0., -0., -0.,  1., -0., -0., -0., -0.],
       [ 1., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0.,  1., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0.,  1., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0.,  1., -0., -0.]])
```

So, our optimal layout is:

$$\begin{bmatrix} 9 & 6 & 2 & 3 & 8 & 5 & 0 & 1 & 4 & 7 \end{bmatrix}$$

However, this solution is pretty trivial and close to how we would intuitively rearrange the keys if we knew the relative frequencies of each key.

Let's see if we can do better by modifying the costs or adding more constraints.

3.3 Costs of Re-Arranging Keys

We can add extra cost to the objective function to account for muscle memory retraining. One natural conclusion would be that moving the key further would incur more "cost" in finding the optimal layout. In addition, to prevent completely dominating the weights, we can potentially scale the distance cost as I have done below.

Let $C = [c_{ij}]$ be the cost of assigning number j to position i. Like before, this is computed by the outer product of the comfort and frequency ranking vectors with an added element-wise multiplication by the distance costs (scaled by some selected value as stated above).

```
dist_scale = 0.2

# The numbers are offset by 1 from their positional index
# (ie. index 0 is number 1, index 9 is number 0)
distances = np.zeros((10, 10))
for i in range(10):
        for j in range(1,11):
                distances[j - 1, i] = abs(j - i - 1)

print("Distance cost matrix: \n", distances)
```

```
Distance cost matrix:
 [[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
 [1. 0. 1. 2. 3. 4. 5. 6. 7. 8.]
 [2. 1. 0. 1. 2. 3. 4. 5. 6. 7.]
 [3. 2. 1. 0. 1. 2. 3. 4. 5. 6.]
 [4. 3. 2. 1. 0. 1. 2. 3. 4. 5.]
 [5. 4. 3. 2. 1. 0. 1. 2. 3. 4.]
 [6. 5. 4. 3. 2. 1. 0. 1. 2. 3.]
 [7. 6. 5. 4. 3. 2. 1. 0. 1. 2.]
 [8. 7. 6. 5. 4. 3. 2. 1. 0. 1.]
 [9. 8. 7. 6. 5. 4. 3. 2. 1. 0.]]
```

```
C = np.outer(comfort_levels, frequency_ranking) + (dist_scale * distances)

print("Cost matrix: \n", np.round(C, 3))
```

```
Cost matrix:
 [[10.     5.2    3.733  3.1    2.8    2.429  2.45   2.511  3.267  2.8  ]
 [ 8.2    4.     2.867  2.4    2.2    1.943  2.     2.089  2.733  2.4  ]
 [ 3.4    1.7    1.     0.95   1.     1.029  1.175  1.333  1.7    1.7  ]
 [ 4.6    2.4    1.533  1.     1.     0.971  1.1    1.244  1.667  1.6  ]
 [ 6.8    3.6    2.4    1.7    1.2    1.057  1.15   1.267  1.8    1.6  ]
 [ 8.     4.3    2.933  2.15   1.6    1.     1.075  1.178  1.767  1.5  ]
 [ 2.2    1.5    1.133  0.85   0.6    0.343  0.125  0.311  0.567  0.7  ]
 [ 3.4    2.2    1.667  1.3    1.     0.686  0.45   0.222  0.533  0.6  ]
 [ 6.6    3.9    2.867  2.25   1.8    1.314  1.025  0.756  0.833  0.7  ]
 [10.8    6.1    4.4    3.45   2.8    2.086  1.725  1.4    1.7    0.9  ]]
```

Now, we can re-formulate the integer programming problem using the new costs in the objective function and solve again using CVXPY:

```
objective = cp.Minimize(cp.sum(cp.multiply(C,X)))
problem = cp.Problem(objective, constraints)
problem.solve()
X.value
```

```
array([[-0., -0., -0., -0., -0., -0.,  1., -0., -0., -0.],
       [-0., -0.,  1., -0., -0., -0., -0., -0., -0., -0.],
       [-0.,  1., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0.,  1., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0.,  1., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0.,  1., -0., -0., -0., -0.],
       [ 1., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0.,  1., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0.,  1., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0., -0.,  1.]])
```

This time, we get our optimal layout as

$$\begin{bmatrix}6 & 2 & 1 & 3 & 4 & 5 & 0 & 7 & 8 & 9\end{bmatrix}$$

which only the 6 being moved the most and the most common keys under the most comfortable fingers.

3.4 Conclusion, Thoughts, and Future Improvements

As seen above, finding the optimal number row layout can be formulated and solved as an integer programming problem by minimizing the cost of key reassignment, number frequency,

and the relative comfort levels of each finger position. Additional costs and constraints can be added to further tailor the generated layout to a person's typing style and workload.

One aspect I would like to improve upon is to vary the number frequencies based on typing workload by finding some datasets that more accurately count the number of times each number is pressed. For example, manual price entry on an online shopping website might use 9 (ie. $19.99) more often compared to a computer programmer who might use 1s and 0s more.

4 Zero-Sum Games and Linear Programming

```
import numpy as np
import cvxpy as cp
```

```
(CVXPY) Apr 15 11:54:01 AM: Encountered unexpected exception importing solver
GLOP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
(CVXPY) Apr 15 11:54:01 AM: Encountered unexpected exception importing solver
PDLP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
```

4.1 Game Theory

In general, game theory is the study of mathematical models representing the interactions between agents (whether that be a person, animal, computer program, government, etc.) who aims to perform the most optimal action given the circumstances (see Wikipedia contributors 2024a).

The study of game theory can be applied to analyzing simple games such as Rock-Paper-Scissors to extending into the study of cooperation among living organisms (as seen in (Axelrod and Hamilton 1981)).

4.1.1 Zero-Sum Matrix Games

One specific form of game that is analyzed is the "zero-sum" game.

Consider the situation in which player 1 (P1) and player 2 (P2) each choose from a finite set of stretegies and play a game such that P1 wins. This results in in P1 gaining some amount c

and P2 losing that same amount $-c$, whether that be money, points, or some other notion of "utility". Then, for each intersection of each of the choices, the net gain/loss is equal to zero such that the entire game's sum of gains/losses is zero as well (hence "zero-sum").

In other words, zero-sum games are scenarios in which one player gains some amount and the other loses that same amount.

Zero-sum games are played in the following manner:

- P1 will choice some action available to them secretly and P2 will choice some action available to them independently secretly. Let's say that these actions are chosen at the exact same time and neither player can react to the other's choice until the game is played and resolved.
- Then, each choice is revealed and the points are awarded based on the pre-defined outcome of each player choosing their strategy.

We say that P1 will choose some row $i \in 1, \dots, m$ and P2 will choose some column $j \in 1, \dots, n$ and the payoff to P1 is the intersection a_{ij} in an $m \times n$ payoff matrix A.

So, a zero-sum game can be summarized as a single matrix:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}$$

(Note that there are representations in which the payouts for both players are written as (x, y) where x is the payoff to P1 and y is the payout to P2, but in zero-sum games it is implied that $y = -x$ so we could omit the P2 payoff as seen above.)

Another important aspect of zero-sum games is that the choice of strategy does not have to be picking a single row/col deterministically (ie. play with a "pure" strategy) and there exists the notion of playing with a "mixed" strategy where the choice is based on a probability distribution among the possibilities for each player.

Now, the "optimal" strategy for each player is to maximize their points but for every singular choice there may be a choice that the other player can make that could reduce your payout. So how should we decide which strategy, pure or mixed, to play to maximize the point gains for both players?

4.1.2 Nash Equilibrium, Minimax, and Solving Zero-Sum Games

If both players play optimally, intuitively there should be some point in which both players are satisfied with the gain/loss outcome.

Consider the game represented as a matrix from P1's perspective. Consider the scenario where P1 picks some strategy and then P2 counters. For any strategy, P1 wants to maximize their

payoff but P2 wants to respond by minmizing their own loss. To mitigate the worst case of gaining the least payoff, P1 wants to maximize their minimum gain over all strategies. Using such a strategy implies that they will always make at least this amount.

In the inverse scenario, P2 picks some strategy and P1 responds. P2 wants to minimize their loss but P1 wants to respond with the move that gains them the most. The worst case that P2 wants to mitigate is that P1 will pick the response to maximize P2's loss, so P2 wants to try to minimize the maximum loss. Finding the strategy that mins the max loss implies that using this strategy, P2 will always lose at most this amount.

Together, these two "safety strategies" to mitigate the worst case for each player eventually intersect into what is known as a "Nash Equilibrium". Consequently, we get the result that both strategies are optimal responses to each other and there is no incentive to switch since no more value can be gained/lost by switching.

In particular, this is point and value is defined and proven in John van Neumann's minimax theorem which states over the set of all (potentially mixed) strategies for P1 Δ^m (ie. a vector of length m that represents probability of each row which sums to 1) and all (potentially mixed) strategies for P2 Δ^n and a game A,

$$\max_{x \in \Delta^m} \min_{x \in \Delta^n} x^T A y = \min_{y \in \Delta^n} \max_{x \in \Delta^n} x^T A y$$

for a game value of $x^T A y$.

So for two player zero-sum games, if we find the pair of strategies that result in a Nash Equilibrium then we will know exactly how much is lost/gained worst case for each player and the game is considered "solved" since there will be no incentive for each player to switch strategies against the opponent who is playing optimally.

4.2 Matrix Games Formulated as a Linear Programming Problem

So, to solve game, we want to find the Nash Equilibrium between strategies x for P1 and y for P2 for a game represented by payoff matrix A. We set up the components of a linear programming problem:

4.2.1 Decision Variables

We consider P1's perspective. P1's decision is what strategy x to use where x is a vector of length m and x_i is the probability of choosing strategy i:

Conversely, P2's decision is what strategy y to use where y is a vector of length n and y_j is the probability of choosing strategy j:

4.2.2 Objective Function

The objective is to find the value of the zero-sum game v. For P1, that means to maximize this value so the objective for P1 is:

$$\max v$$

And for P2, this means to minimize this value:

$$\min v$$

4.2.3 Constraints

We have the following common constraints:

- Both x and y are probability vectors and must sum to 1:

$$\sum_i x_i = 1,\ \sum_j y_j = 1$$

- You cannot play a strategy with negative probability:

$$x_i \geq 0,\ y_j \geq 0$$

Then, we add one more constraint specific to each player:

- From above, the objective for P1 is to find some strategy that maximizes their minimum gain:

$$x^T A \geq v$$

- The objective for P2 is to find some strategy that minimizes their maximum loss:

$$Ay \leq v$$

4.2.4 LPP for P1

$$\begin{aligned} & \max \quad v \\ \text{subject to} \quad & x^T A \geq v \\ & \sum_i x_i = 1 \\ & x \geq 0 \end{aligned}$$

4.2.5 LPP for P2

$$\begin{aligned} & \min \quad v \\ \text{subject to} \quad & Ay \leq v \\ & \sum_j y_j = 1 \\ & y \geq 0 \end{aligned}$$

4.2.6 LP and Minimax

One last note is that these two LPPs are the dual for each other and thus by strong duality, their values v are equal and this is precisely the result of the minimax theorem above!

4.3 Solving Matrix Games using CVXPY

Using the above LPP formulation, we can create the following python function to solve an arbitrary zero-sum game using CVXPY:

```
def solve_P1(A):
  """
  Solve P1's optimal strategy for a game given by matrix A

  Parameters:
   - A: m x n payout matrix for the game

  Returns:
    - The value of the game and the optimal strategy x for P1
  """
  m, _ = np.shape(A)

  # X is the decision variables for the strategy
```

```
  X = cp.Variable(m)
  # V is the value of the game
  V = cp.Variable()
  objective = cp.Maximize(V)

  # P1 gets at least v for each choice of strategy
  constraint1 = [(X.T @ A) >= V]
  # Probabilities are >= 0
  constraint2 = [X.T >= 0]
  # X[i] must sum to 1
  constraint3 = [cp.sum(X.T) == 1]
  constraints = constraint1 + constraint2 + constraint3

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

  return problem.value, X.value

def solve_P2(A):
  """
  Solve P2's optimal strategy for a game given by matrix A

  Parameters:
   - A: m x n payout matrix for the game

  Returns:
    - The value of the game and the optimal strategy y for P2
  """
  _, n = np.shape(A)

  # Y is the decision variables for the strategy
  Y = cp.Variable(n)
  # V is the value of the game
  V = cp.Variable()
  objective = cp.Minimize(V)

  # P2 gets at most v for each choice of strategy
  constraint1 = [(A @ Y) <= V]
  # Probabilities are >= 0
  constraint2 = [Y >= 0]
  # Y[j] must sum to 1
  constraint3 = [cp.sum(Y) == 1]
```

```
    constraints = constraint1 + constraint2 + constraint3

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

    return problem.value, Y.value
```

4.3.1 Solving Rock-Paper-Scissors

The game of canonical Rock-Paper-Scissors is given by the matrix below where the player gains \$1 if they win or \$0 if they tie. Let $i, j = 0$ be Rock, $i, j = 1$ be Paper, and $i, j = 2$ be Scissors:

$$A = \begin{bmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{bmatrix}$$

```
A = np.array([[0, -1, 1], [1, 0, -1], [-1, 1, 0]])

value_x, X = solve_P1(A)
print("Value: " + str(value_x))
print("P1 Strategy: " + str(X) + "\n")

value_y, Y = solve_P2(A)
print("Value: " + str(value_y))
print("P2 Strategy: " + str(Y))
```

```
Value: 2.4450394122394545e-10
P1 Strategy: [0.33333333 0.33333333 0.33333333]

Value: -2.4450394122394545e-10
P2 Strategy: [0.33333333 0.33333333 0.33333333]
```

```
/home/derry/.local/share/virtualenvs/MATH441-vlJmoJsg/lib/python3.8/site-packages/cvxpy/redu
FutureWarning:
    Your problem is being solved with the ECOS solver by default. Starting in
    CVXPY 1.5.0, Clarabel will be used as the default solver instead. To
    continue
    using ECOS, specify the ECOS solver explicitly using the
    ``solver=cp.ECOS``
    argument to the ``problem.solve`` method.
```

```
  warnings.warn(ECOS_DEPRECATION_MSG, FutureWarning)
```

So, each player should pick uniformly $[1/3, 1/3, 1/3]$ as their strategy such that P1 would gain at most \$0 and P2 would lose at most \$0.

4.3.2 Tech Reads in Super Smash Bros.

In the fighting game Super Smash Bros (and other fighting games in general), there is a game state called "advantage" in which a player (let's say P1) is playing to take a stock/life of player 2 and the other is in "disadvantage" in which they are playing to live and reset to a "neutral" game state or gain an advantage. There is a mechanic in Smash in which a player can "tech" when hitting the ground while in the "hitstun" state after getting attacked to remain invulnerable while moving left, right, or stay in-place. A player can also mistime the input and "miss" their tech which leaves them vulnerable and have to perform a slow standing animation. A "tech-chase" situation is when the advantage state player attempts to "read" (ie. correctly guess) the defensive option chosen by P2 to maintain advantage state and potentially take a stock.

In a tech situation, P1 will attempt to read one of 4 options that P2 can make:

1. Tech in-place
2. Tech left
3. Tech right
4. No tech

Let the payout be $a_{i,j} \in [0, 1]$ which represents the probability that P1 takes the stock for each situation. Then, consider the following matrix where row/col index 0 is in-place, 1 is left, 2 is right, and 3 is no option.

$$A = \begin{bmatrix} 1 & 0.3 & 0.3 & 0.8 \\ 0.25 & 1 & 0 & 0.6 \\ 0.25 & 0 & 1 & 0.6 \\ 0.5 & 0.3 & 0.3 & 1 \end{bmatrix}$$

```
A = np.array([
  [1, 0.25, 0.25, 0.8],
  [0.25, 1, 0, 0.6],
  [0.25, 0, 1, 0.6],
  [1, 0.3, 0.3, 1]
])

value_x, X = solve_P1(A)
```

```
print("Value: " + str(value_x))
print("P1 Strategy: " + str(np.round(X, 4)) + "\n")

value_y, Y = solve_P2(A)
print("Value: " + str(value_y))
print("P2 Strategy: " + str(np.round(Y, 4)))
```

```
Value: 0.44736842104900976
P1 Strategy: [0.     0.3684 0.3684 0.2632]

Value: 0.44736842104415886
P2 Strategy: [0.2105 0.3947 0.3947 0.    ]
```

Under these conditions, it is best for P1 to predict in-place 26%, left or right 37%, and never no-tech for the payout of taking a stock at least 45% of the time. Conversely, P2's best options are to pick in-place 21%, left or right 39.5%, and never no-tech for the payout of losing a stock at most 45% of the time.

4.4 Conclusion

The structure and properties of zero-sum games allows it to be solved using linear programming to leverage strong duality in finding the optimal strategies for either player and finding the value of the game which represents the minimum gain/maximum loss for each player in the worst case.

5 Integer Programming and Chessboard Puzzles

```
import numpy as np
import cvxpy as cp
```

```
(CVXPY) Apr 15 11:54:03 AM: Encountered unexpected exception importing solver
GLOP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
(CVXPY) Apr 15 11:54:03 AM: Encountered unexpected exception importing solver
PDLP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
```

5.1 Chessboard Puzzles

The problem statement for a *chessboard puzzle* (or more specifically *chessboard non-attacking puzzles*) is as follows:

> Given a large number of non-pawn chess pieces (rook, bishop, knight, queen, king) of a single type and none of the others, how do we place the **maximum** number of pieces such that **no capturing can occur** if the pieces are played legally according to the standard rules of chess

This entry aims to summarize, comment, and implement some of the methods described in "An Application of Graph Theory and Integer Programming: Chessboard Non-Attacking Puzzles" (see Foulds and Johnston 1984) to better understand how we can mathematically model this family of problems to find exact solutions.

5.1.1 Why is this Problem "Hard"?

Although the premise is very simple once you understand the movement rules of each piece, a non-trivial question would be "how many **distinct ways** are there to place n-queens"? In particular, the space of possible placements is enormous. Even for the 8-queens problem on an 8×8 board, there are a total of

$$\binom{64}{8} = 4,426,165,368$$

positions with only **92** possible solutions. If you simply try all possible placements and then verify they work, it is too computationally expensive for a brute force search. There are some shortcuts to reduce the search space such as using the fact that we can only place a single queen in each column (reduces the 8-queens possibilities to $8^8 = 16,777,216$) but this is still exponentially growing and reaches the limits of a standard computer's computational power very quickly. For example, the search space under this assumption for the 13-queens problem is 13^{13} possibilities or

$$13^{13} = 3.0287511 \times 10^{14}$$

Although we won't be writing the code to return the total number of solutions, it is worth noting the difficulty of finding solutions for larger chess boards.

5.2 Solutions using Integer Programming

Our objective is to **place** (ie. **assign**) chess pieces to locations on a chess board such that they do not attack each other **constrained by the legal attacking moves of chess**.

This can be recognized as a canonical LP assignment problem with chess specific constraints.

Although the paper seeks to build up the problem from the basic principles of linear programming and matrix properties, I found that there is a gap of information missing in explaining how the problem is intuitively formulated mathematically if I wasn't already familiar with the canonical formulation of the problem.

Below, we will walk through the intuition on formulating any problem which involves placing things into a finite space under certain restrictions as an assignment IP problem.

5.2.1 Decision Variables

The core to the assignment problem is to find a way to mathematically represent a way to say "I want to place some item at some location uniquely".

If we think about how we would organize objects in real life, there is really only one decision to be made for each object in question: Should it be placed here or not?

Before even considering how the matrix will look, we know that a chess board is a perfect square and can be uniquely indexed by a row i and a column j. For each of these coordinates (i, j), we want to place some value (let's say 1) if a piece is placed there and another value (let's say 0) if a piece is not placed there.

Let x_{ij} be the assignment of a piece at location (i, j). Then,

$$x_{ij} = \begin{cases} 1 & \text{the piece is placed at location } (i,j) \\ 0 & \text{the piece is not placed at location } (i,j) \end{cases}$$

This binary decision variable formulation is quite powerful. Routing problem solutions use this idea as well by asking "should I take this road or not?" and the knapsack problem makes decisions based on "should I take this item or not?"

5.2.2 Objective Function

Our goal to solve the chessboard puzzle is to fit **as many** of the specified piece on an $n \times n$ chess board. In mathematical terms we want to **maximize** the total number of assigned pieces:

$$\max \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} x_{ij}$$

Although in this case we are unconstraining the number of pieces and letting the algorithm decide, some problems wish to simply find a feasible solution and can be represented by the objective function:

$$\max 1$$

5.2.3 Constraints

We will formulate the constraints for the n-queens problem. To find a feasible solution, we must satisfy the following:

1. Decision variables x_{ij} cannot be assigned a negative value (ie. must be assigned 0 or 1)
2. Only a single queen can exist in each row, column, and diagonal

The first constraint is typical for linear/integer programming. If we flatten the matrix of decision variables into a large vectors, we want each

$$x_{ij} \geq 0$$

for each element of the aggregated X vector.

The flavor of the second constraint is often found in assignment problems and follows from the design of the decision variables.

The rows and columns constraint is quite intuitive since a chess grid can be visualized easily. The idea is as follows for the rows: Consider *only* a single row i. Then, we want to look at every column in that row and count how many assignment occurences are found. Since we know that the variables can only take two values, the assignment will be *unique* if it ends up summing to 1. Thus:

$$\sum_{i=0}^{n-1} x_{ij} = 1,\ i = 1, \dots, n$$

A similar idea is used for the columns

$$\sum_{j=0}^{n-1} x_{ij} = 1,\ j = 0, \dots, n-1$$

The diagonals are a little more complicated. We use the idea that each diagonal lies on some "level" where the coordinates summed/subtracted equal some number.

For example, take this 4×4 chess board. For the second anti-diagonal (bottom-left to top-right lines), we have the (0-indexed) grid coordinates $[(2,0),(1,1),(0,2)]$. Summing each $i+j$, we see that this diagonal has "level" $k = 2$. The largest "level" will include the coordinate $(n-1, n-2)$ for level $k = 2n-3$. If we continue aggregate this pattern, we get that the anti-diagonal levels are $i + j = k,\ k = 1, \dots, 2n-3$.

$$\begin{bmatrix} * & * & (0,2) & * \\ * & (1,1) & * & * \\ (2,0) & * & * & * \\ * & * & * & * \end{bmatrix}$$

Similarly, we can use the same idea for the diagonals (top-left to bottom-right lines) but we subtract the coordinates. For the second diagonal, we have grid coordinates $[(0,1),(1,2),(2,3)]$ such that each $i-j=-1$. If we continue this for each subtraction, we get that the levels are $i-j=l,\ l=-(n-2),\ldots,n-2$.

$$\begin{bmatrix} * & (0,1) & * & * \\ * & * & (1,2) & * \\ * & * & * & (2,3) \\ * & * & * & * \end{bmatrix}$$

For the anti-diagonals, we get the following constraints where we can have at most 1 queen on each anti-diagonal:

$$\sum_{i+j=k}^{n} x_{ij} \leq 1,\ k=1,\ldots,2n-3$$

And the diagonals:

$$\sum_{i-j=l}^{n} x_{ij} \leq 1,\ l=-(n-2),\ldots,n-2$$

5.3 Integer Programming Solutions using Python

We will use CVXPY with the above formulation to solve the n-queens problem.

```
def ip_nqueens(n):
        # Define CVXPY variables
        X = cp.Variable((n, n), integer=True)

        # Define objective function
        objective = cp.Maximize(cp.sum(X))

        # Define constraints
        constraints = []
```

```
    # Assignment problem decision variables are non-positive 0/1
    constraints = constraints + [X >= 0]

    # Rook constraints
    # All rows have at most 1 queen
    constraints = constraints + [cp.sum(X[i,:]) <= 1 for i in range(n)]

    # All cols have at most 1 queen
    constraints = constraints + [cp.sum(X[:,j]) <= 1 for j in range(n)]

    # "Bishop constraints" - there is a single queen in each diagonal
    # Anti-diagonals (ie. level lines go from bottom-left to top-right)
     ↪  have
    # at most 1 queen
    for k in range(1, 2 * n - 3):
            diagonal = []
            for i in range(n):
                    for j in range(n):
                            if (i + j == k):
                                    diagonal.append(X[i,j])

            constraints.append(cp.sum(diagonal) <= 1)

    # Diagonal constraints (ie. level lines go from top-right to
     ↪  bottom-left) have
    # at most 1 queen
    for l in range(-1 * (n - 2), n - 2):
            diagonal = []
            for i in range(n):
                    for j in range(n):
                            if (i - j == l):
                                    diagonal.append(X[i,j])

            constraints.append(cp.sum(diagonal) <= 1)

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

    return np.round(X.value, 3)
```

```
ip_nqueens(4)
```

```
array([[-0., -0.,  1., -0.],
       [ 1., -0., -0., -0.],
       [-0., -0., -0.,  1.],
       [-0.,  1., -0., -0.]])
```

```
ip_nqueens(8)
```

```
array([[-0., -0.,  1., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0.,  1., -0.],
       [-0.,  1., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0.,  1.],
       [-0., -0., -0., -0.,  1., -0., -0., -0.],
       [ 1., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0.,  1., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0.,  1., -0., -0.]])
```

```
ip_nqueens(13)
```

```
array([[-0., -0.,  1., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0., -0., -0.,  1., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0.,  1.],
       [-0., -0., -0.,  1., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0.,  1., -0., -0., -0., -0.],
       [-0., -0., -0., -0.,  1., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0.,  1., -0., -0., -0., -0., -0.],
       [-0.,  1., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0.,  1., -0.],
       [-0., -0., -0., -0., -0., -0.,  1., -0., -0., -0., -0., -0., -0.],
       [ 1., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0., -0.],
       [-0., -0., -0., -0., -0., -0., -0., -0., -0.,  1., -0., -0., -0.],
       [-0., -0., -0., -0., -0.,  1., -0., -0., -0., -0., -0., -0., -0.]])
```

5.4 Conclusion

Above we have detailed a way of solving chessboard non-attacking puzzles with integer programming whose core ideas behind its formulation is able to be expanded into a whole class of combinatorial optimization problems.

Understanding how the n-queens problem relates to the assignment problem is formulated is essential to formulating a variety of other combinatorial optimization problems as an integer programming problems such as vehicle routing and the knapsack packing.

6 Graph Coloring using Integer Programming

```
import numpy as np
import cvxpy as cp
import networkx as nx
import seaborn as sns
import matplotlib as mpl
import matplotlib.pyplot as plt
```

```
(CVXPY) Apr 15 11:54:07 AM: Encountered unexpected exception importing solver
GLOP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
(CVXPY) Apr 15 11:54:07 AM: Encountered unexpected exception importing solver
PDLP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
```

6.1 Graph Coloring - Problem Statement

Graph coloring is the classifcation of problems where we assign color labels to elements of a graph under certain constraints (Wikipedia contributors 2024b).

A common variation is **vertex coloring** in which we assign colors to each vertex in the graph with no adjacent vertices being the same color.

6.1.1 Some Graph Terminology

- Let V be the set of vertices/nodes.

- Let $E \subset \{(x, y) \mid x, y \in V, x \neq y\}$ be the set of edges. An edge exists between two distinct vertices and the set of edges can include any number of the possible pairs of vertices. We can represent these as tuples of vertices $(x, y) \in E$.
- Then, let $G = (V, E)$ be a graph. A graph can be represented (ie. drawn) using only the set of edges/vertices.
- $|A|$ is the cardinality of set A which is the number of elements in the set.

6.2 Representation as an Integer Programming Problem

We formulate the vertex coloring problem with the following problem statement

> How can we color all vertices of a given graph using the minimum number of colors such that no two adjacent vertices are the same color?

6.2.1 Decision Variables

Consider a set of i graph nodes/vertices and a set of j colors.

Similar to other assignment problems using binary integer programming, let

$$x_{ij} = \begin{cases} 1 & \text{vertex } i \text{ is assigned to color } j \\ 0 & \text{otherwise} \end{cases}$$

However, this alone is not enough to represent all choices made in the optimization problem statement. We introduce a second variable

$$w_j = \begin{cases} 1 & \text{color } j \text{ is used in the graph coloring} \\ 0 & \text{otherwise} \end{cases}$$

to keep track of how many colors are used in the solution.

On the selection of the upper bound of j:

The allowed indices that the vertices can take is $i = 0, \ldots, |V| - 1$ but how large should we make j? If we want to see if a certain coloring is feasible, we could limit j to some target value but for the purposes of this demonstration and a reasonable limit is to set let $j = 0, \ldots, |V| - 1$ as well. In short, we are stating that the worst-case that the algorithm will be able to spit out is if each vertex is colored a separate color. This is a reasonable assumption when you consider graph coloring solutions to the assignment problem (eg. exam assignment); the easiest way to make all exams have no conflict with each other is to simply schedule each on seaparate days.

6.2.2 Objective Function

Our goal is to minimize the number of colors used in the graph coloring:

$$\min \sum_j w_j$$

Since $w_j = 1$ if and only if color j is used, this will find the minimum possible number of colors to satisfy all constraints.

6.2.3 Constraints

For the vertex coloring problem, we want to satisfy the following constraints:

- Every vertex has exactly 1 color assigned:

$$\sum_j x_{ij} = 1,\ i = 0, \dots, |V| - 1$$

- For every set of edges, **at most one of the vertices has the color** j. We look at each vertex in an edge and make sure that for each j, the count is ≤ 1:

$$x_{uj} + x_{vj} \leq 1,\ \forall (u, v) \in E,\ j = 0, \dots, |V| - 1$$

- We need a way to increment the w_j count when a vertex is colored with color j (ie. $x_{ij} = 1$). The intuition is as follows. Let's say that we have assigned vertex x_i with a color j already. Then by definition, $x_{ij} = 1$. However, since color j is used then we would need to set $w_j = 1$ as well. So, for every node and "color counter" w:

$$x_{ij} \leq w_j,\ i = 0, \dots, |V| - 1,\ j = 0, \dots, |V| - 1$$

- Decision variables cannot be negative:

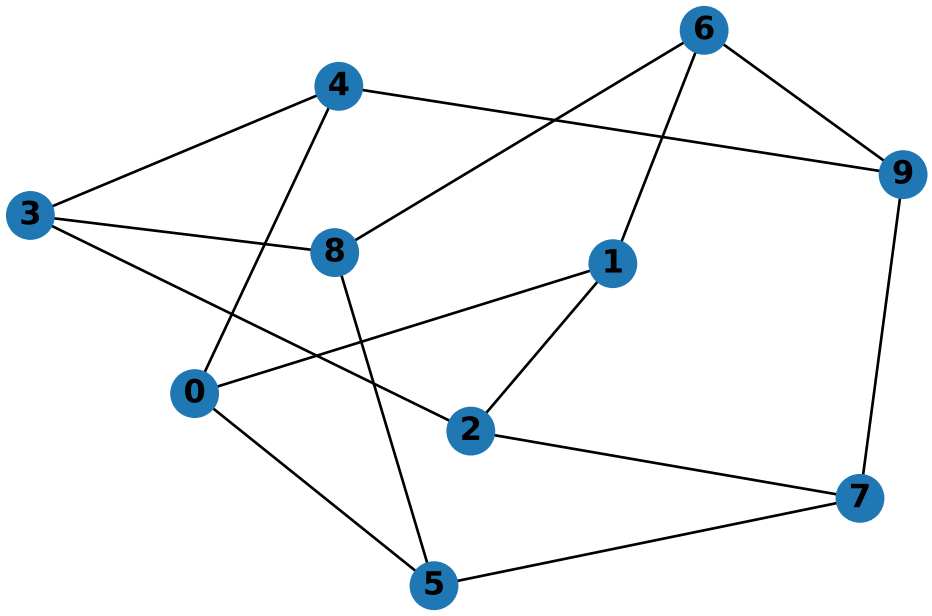
$$x_{ij} \geq 0,\ w_j \geq 0,\ \forall i, j$$

6.2.4 Solutions using Python

Let's implement these constraints using Python. We will use NetworkX (Hagberg, Swart, and Schult 2008) to represent and visualize graphs and CVXPY (Diamond and Boyd 2016).

First, let's create a simple graph. NetworkX has a built-in function to create a simple Petersen graph (see Wikipedia contributors 2024c) which will serve as a good starting point.

```
G = nx.petersen_graph()
nx.draw(G, with_labels=True, font_weight='bold')
```

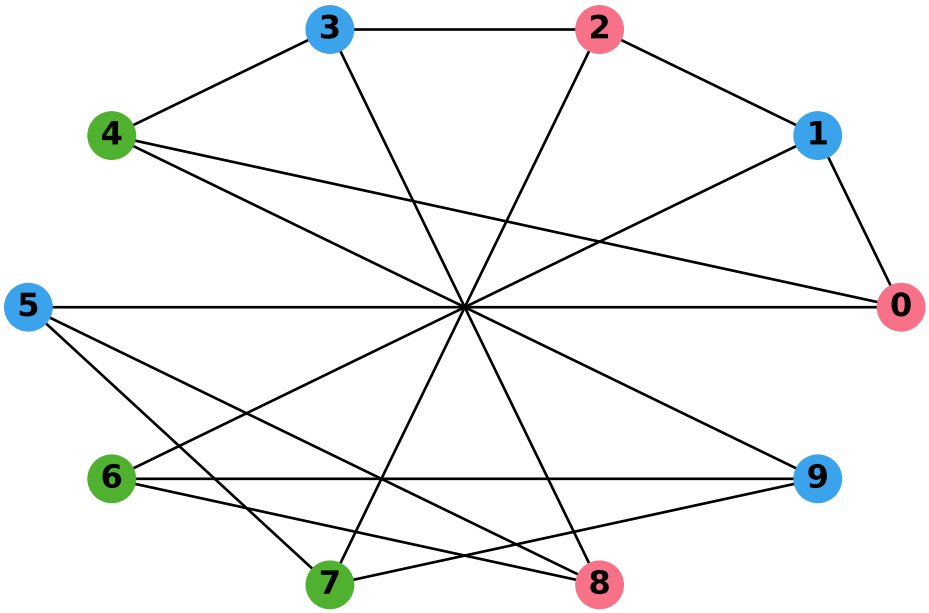


Now we can implement the problem defined above using CVXPY:

```
def draw_graph(G, color_assignment):
  """
  Draw the graph coloring for input graph G
  """
  # Filter out all columns of all-zeros
  color_assignment = color_assignment[:, color_assignment.any(0)]

  # Get the number of colors
  num_colors = np.shape(color_assignment)[1]
```

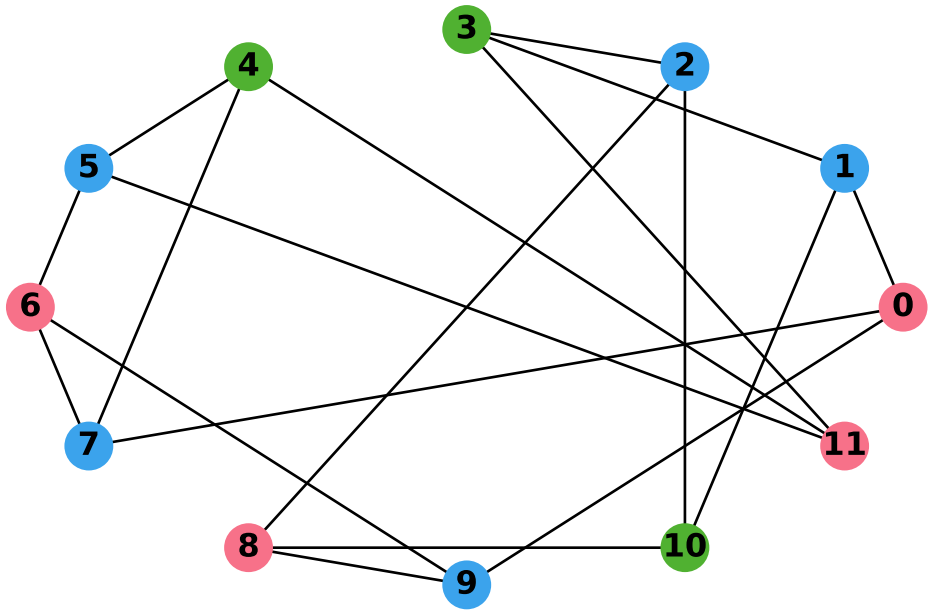
```
  # Generate a color palette using seaborn
  palette = sns.color_palette("husl", num_colors)
  color_map = []

  # `node` is the index of the node by default
  for node in G:
    color_idx = np.argmax(color_assignment[node,:])
    color_map.append(palette[color_idx])


  nx.draw_circular(G, node_color=color_map, with_labels=True,
↪  font_weight='bold')

def graph_coloring(G):
  """
  Solve the vertex coloring graph for an input graph G.

  Params:
    - G: NetworkX graph instance

  Returns:
    -
  """
  # Extract graph data from G
  vertices = G.nodes
  edges = G.edges
  num_vertices = len(vertices)
  num_colors = len(vertices)

  # Define decision variables
  X = cp.Variable((num_vertices, num_colors), integer=True)
  W = cp.Variable(num_colors, integer=True)

  # Define objective function
  objective = cp.Minimize(cp.sum(W))

  # Define constraints
  constraints = []

  # Non-negativity of decision variables
  constraints = constraints + [X >= 0]
```

```
  constraints = constraints + [W >= 0]

  # Every vertex must be colored
  constraints = constraints + [cp.sum(X[i,:]) == 1 for i in
↪   range(num_vertices)]

  # For every edge, at most 1 vertex has color j
  for edge in edges:
    vertex_1 = edge[0]
    vertex_2 = edge[1]

    for j in range(num_colors):
      constraints = constraints + [X[vertex_1, j] + X[vertex_2, j] <= 1]

  # If a color is taken, increment w_j
  constraints = constraints + [X[i,j] <= W[j] for i in range(num_vertices)
↪   for j in range(num_colors)]

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

  # color_assignment = X.value

  return X.value

color_assignment = graph_coloring(G)
draw_graph(G, color_assignment)
```

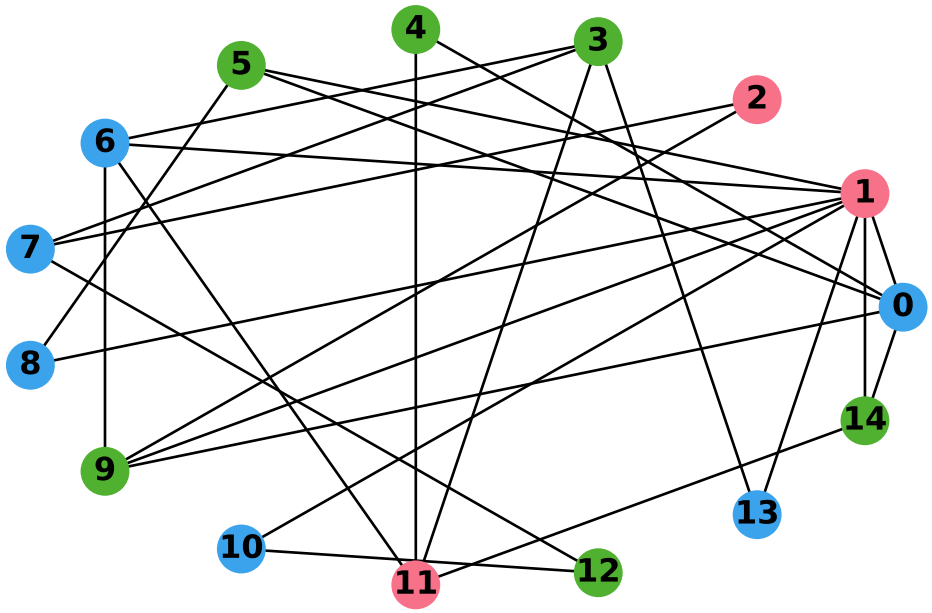


We can try it on some other graphs too:

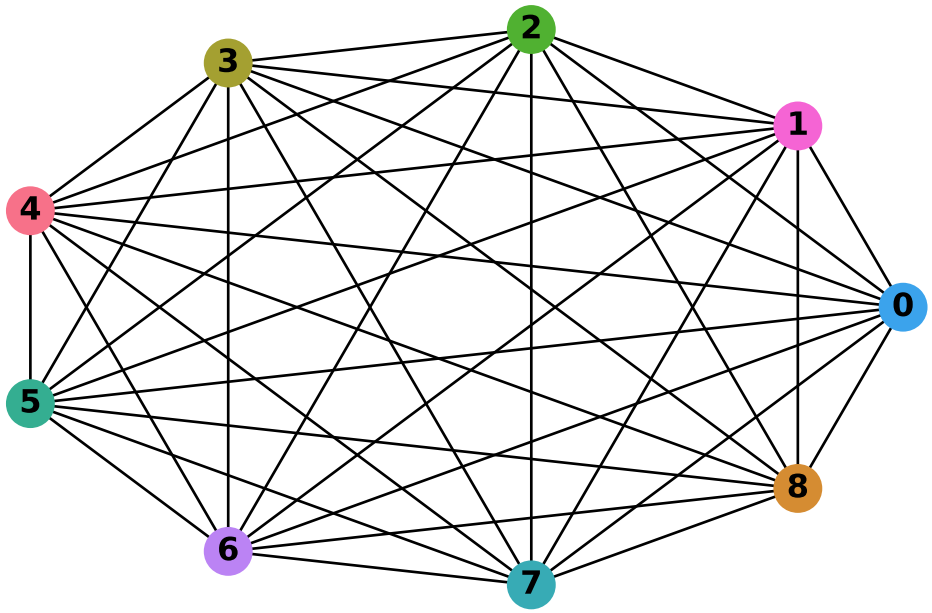
```
G2 = nx.random_regular_graph(3, 12)
G2 = nx.convert_node_labels_to_integers(G2, first_label=0)

color_assignment = graph_coloring(G2)
draw_graph(G2, color_assignment)
```




```
G3 = nx.gnm_random_graph(15, 25)
G3 = nx.convert_node_labels_to_integers(G3, first_label=0)

color_assignment = graph_coloring(G3)
draw_graph(G3, color_assignment)
```




```
G4 = nx.complete_graph(9)
G4 = nx.convert_node_labels_to_integers(G4, first_label=0)

color_assignment = graph_coloring(G4)
draw_graph(G4, color_assignment)
```



6.3 Applications

Vertex graph coloring provides a convenient way to represent and solve many problems involving the assignment of resources under conflict constraints.

As a result, this formulation of the graph coloring problem can be used for various resource scheduling/assignment problems in real life such as radio frequency allocation, job allocation, and team building problems (Thadani, Bagora, and Sharma 2022).

6.3.1 Optimizing Cooking Steps

I love cooking and I was inspired by Alex's video on how fast you can feasibly make eggs benedict. Although the challenge is long over by now, such a scenario can be represented and "solved" using a graph coloring approach.

Let G be our graph and let each vertex $v \in V$ represent some task required in cooking. Then, let each edge $(x, y) \in E$ represent some scheduling conflict between tasks x and y. This can be because they require the same cooking utensil, they are blocked until the completion of the other is complete, etc.

We will apply these methods to a steak frites dinner with a bruschetta appetizer below.

Bruschetta

0. Chop tomatoes
1. Chop basil
2. Mix tomatoes, basil, olive oil, garlic, balsamic vinegar
3. Slice bread
4. Top bread with bruschetta and serve

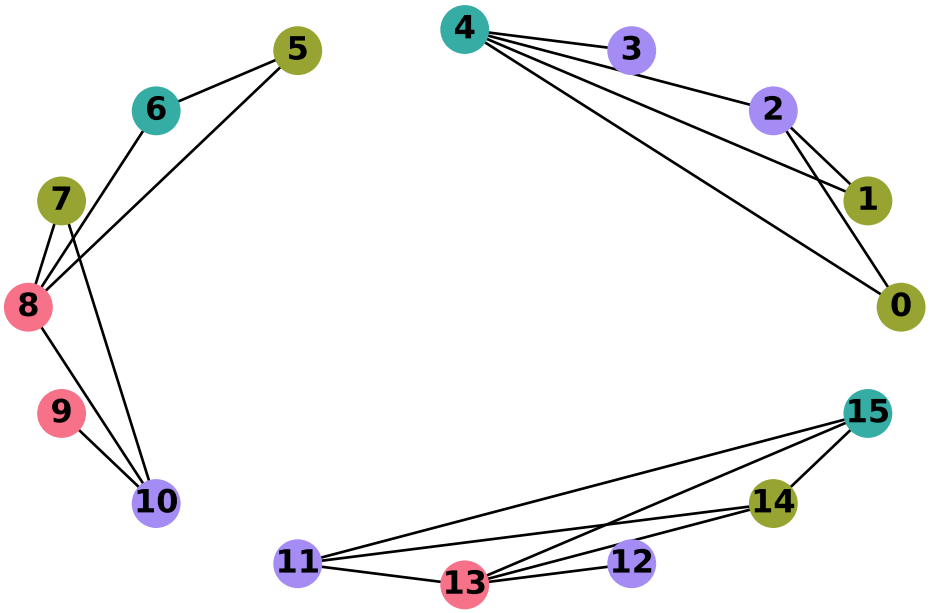
Grilled Ribeyes and Corn

5. Salt and dry brine steak
6. Prepare steak with fresh pepper and pat dry
7. Preheat grill
8. Grill steak
9. Prepare corn
10. Grill corn

Baked Fries

11. Cut potatoes into thick cut fries
12. Boil water with baking soda and salt
13. Boil fries until tender
14. Let fries cool and toss in spices and olive oil
15. Bake in oven, turning occasionally until crispy

```
# Graph representation of tasks list
G = nx.Graph()
G.add_nodes_from(range(11))
G.add_edges_from([
    # Bruschetta
    (0, 2), (1, 2), (0, 4), (1, 4), (2, 4), (3, 4),

    # Steak and corn
    (5, 6), (5, 8), (6, 8), (7, 8), (7, 10), (8, 10), (9, 10),

    # Baked fries
    (11, 13), (12, 13), (11, 14), (13, 14), (11, 15), (13, 15), (14, 15)
])

color_assignment = graph_coloring(G)
draw_graph(G, color_assignment)
```



The graphical interpretation can be done as follows. Consider each subgraph separately and read the steps in order. All steps with the same color can be done at the same time before moving onto the next color/steps.

For example, we see that we should chop the tomatoes and basil at the same time then mix the bruschett and slice the bread together before serving. Then, we should dry brine the steak and preheat the grill at the same time, then prepare the steak, then grill and prepare corn, then grill the corn. Finally, we should cut the potatoes and start boiling the water before par boiling the potatoes, seasoning, and then baking.

6.4 Conclusion

Graph coloring can be used to represent a variety of assignment/scheduling problems with a simple integer programming formulation. As a result, many problems involving some assignment under conflicting relationships can be solved for the minimum non-conflicting solution by a very intuitive visualization using graph networks.

7 Network Flow with Congestion

```
import numpy as np
import cvxpy as cp
import networkx as nx
```

```
(CVXPY) Apr 15 11:54:12 AM: Encountered unexpected exception importing solver
GLOP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
(CVXPY) Apr 15 11:54:12 AM: Encountered unexpected exception importing solver
PDLP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
```

7.1 Network Min-Cost Flow

The Network Flow problem involves finding the **cheapest** way to send some set of supply sources through a collection of paths with set costs to reach some set of demand sinks.

This problem has many applications in fields such as distribution systems in operations planning to internet routing systems.

7.1.1 Adding Congestion Costs

In real-life scenarios, however, we are limited by physics, city zoning, etc. which can impact the amount we can transport over a single wire/road/path. The question we will answer and analyze below is how do we represent this mathematically and how different are the solutions when we add congestion built into our mathematical network flow model?

The core formulation will explain and iterate upon the network flow problem statement in (Vanderbei 2020).

Aside: Some Graph Terminology

Like in Graph Coloring, Network Flow is solved over a graph.

- Let V be the set of vertices/nodes.
- Let $E \subset \{(x, y) \mid x, y \in V, x \neq y\}$ be the set of edges. An edge exists between two distinct vertices and the set of edges can include any number of the possible pairs of vertices. We can represent these as tuples of vertices $(x, y) \in E$.
- Then, let $G = (V, E)$ be a **graph**. A graph can be represented (ie. drawn) using only the set of edges/vertices.
- $|A|$ is the cardinality of set A which is the number of elements in the set.

7.2 Representation as a Linear Programming Problem

Consider a network that is represented by a set of vertices V and edges $E \subseteq \{(x, y) \mid x, y \in V, x \neq y\}$ which make a graph $G = (V, E)$.

Then, we can assign a cost to each edge $c_{i,j},\ (i, j) \in E$ which represents the cost of transportation along edge $(i, j) \in E$. Furthermore, we assign a "contribution" value $b_i,\ i = 1, \ldots, |V|$ to each node in the network. If $b_i < 0$, we consider this node to be a "supplier"/source where the flow will originate from. If $b_i > 0$ these are "consumers"/sinks where the flow should end. If $b_i = 0$, then these are neutral nodes and could represent something that simply passes along the flow.

7.2.1 Decision Variables

We need to decide how much supply to send over a link to reach a demand destination. So, as is typical of assignment type decision variables we assign each (i, j) into an edge matrix and use the variable x_{ij} to be how much flow we are sending through edge (i, j) in the final solution.

7.2.2 Objective Functions

Network Flow

The typical congestion flow objective is to minimize the total cost of sending x_{ij} over edge (i, j) with edge cost c_{ij}:

$$\min \sum_{(i,j) \in E} c_{ij} x_{ij}$$

Network Flow with Congestion

Congestion occurs when when we try to transport too much over a single link (eg. if we all try to drive on the same highway, we create traffic). Thus, we can modify the objective function by multiplying the cost above again by how much we are sending over (i, j). This will penalize the total cost if we try to allocate too much over a single link since now we have the quadratic objective.

$$\min \sum_{(i,j)\in E} c_{ij} x_{ij}^2$$

7.2.3 Constraints

Further Assumptions

To simplify the model, we will add the following assumption for the design of the problem itself.

In order to make sure that the solution is feasible and all supply is accounted for, we must have a fully contained system where the supply and demand are equal (otherwise, we may non-deterministically add/remove cost from the system).

$$\sum_{i=1}^{|V|} b_i = 0$$

Then, the constraints are common between the two objective functions and are formulated as follows.

Non-negativity

The flow must be non-negative (we obviously can't send negative delivery trucks down a highway):

$$x_{ij} \geq 0, \forall (i, j) \in E$$

Flow Balance

The flow into (ie. $\sum_i x_{ik}$) and out of a node k (ie. $\sum_j x_{kj}$) must be equal to the demand of that node (ie. $-b_k$) (called the **flow balance constraints**). This is combined with assumption 1 above to make sure that the end solution is unique; if total $supply+demand = 0$, then the flows are balanced such that $supply = -demand$ at each node. No cost will not be unaccounted for and therefore we will be able to uniquely find a solution.

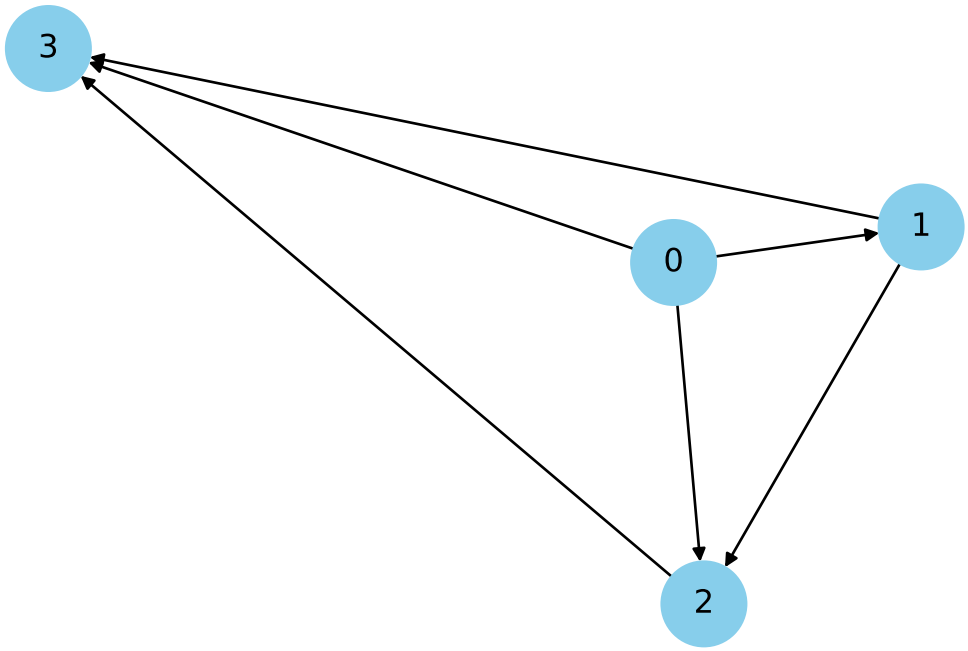
$$\sum_i x_{ik} - \sum_j x_{kj} = b_k, \ \forall (i,k), (k,j) \in E, \ \forall k \in V$$

The flow balance constraint seems pretty abstract, but its representation in matrix form will help explain it more clearly.

Since LP problem constraints are traditionally represented as matrices, the flow balance constraints form a convenient way to fully encode the edges in a incidence matrix (see Wikipedia contributors 2022b).

For example, we create the incidence matrix for the following graph (which we will use as an example below)

```
G = nx.DiGraph()
G.add_nodes_from(
  [
    (0, {"demand": -5}),
    (1, {"demand": 0}),
    (2, {"demand": 0}),
    (3, {"demand": 5}),
   ]
)

G.add_weighted_edges_from(
  [
    (0, 1, 3),
    (0, 2, 6),
    (0, 3, 1),
    (1, 3, 1),
    (1, 2, 3),
    (2, 3, 2),
  ]
)

nx.draw(G, with_labels=True, node_color="skyblue", node_size=1000,
 ↪  font_size=12)
```



$$A = \begin{bmatrix} -1 & -1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & -1 & 0 \\ 0 & 1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

Each row is the node i and each column represents some edge with index j where

$$a_{ij} = \begin{cases} -1 & \text{edge } j \text{ leaves node } i \\ 1 & \text{edge } j \text{ enters node } i \end{cases}$$

For example the $j = 3$ column represents the edge $(1, 3)$.

Going though each column, we have that the index of the -1 is the head node and the index of the 1 is the tail node. Then, the flow balance constraints can be interpreted by going through each vertex (ie. the rows) and making sure that all flow determined by going in and out is b_i.

Edge Capacity

If needed, upper bound capacity constraints. If present, we add new constants u_{ij}, $\forall (i, j) \in E$ where we set some limit on how much flow we choose for x_{ij}. If we do not define any, we implicitly set the limit to infinity.

$$x_{ij} \leq u_{ij}, \ \forall (i,j) \in E$$

7.3 Solutions using Python

We write the min-cost network flow problem solution using CVXPY in python:

```
def min_cost_network_flow(G, with_congestion=False):
  """
  Calculates the min-cost network flow for a given graph G.

  Params:
    - G: NetworkX graph
    - congestion: Flag to solve network flow with congestion

  Returns:
    - Min-cost network flow solution dictionary of form {(i,j): flow}
  """

  vertices = G.nodes
  edges = G.edges
  num_vertices = len(vertices)
  num_edges = len(edges)

  X = cp.Variable(num_edges)
  C = np.array([edge[2]["weight"] for edge in G.edges(data=True)])
  b = np.array(list(nx.get_node_attributes(G, "demand").values()))

  # Assertion that supply/demand is balanced
  assert np.sum(b) == 0

  A = nx.incidence_matrix(G, oriented=True).toarray()

  objective = cp.Minimize(cp.sum(C @ X))

  if with_congestion:
    objective = cp.Minimize(cp.sum(C @ cp.square(X)))

  constraints = []

  # Non-negative flow
```

```
  constraints = constraints + [X >= 0]

  # Flow balance - we define the incidence matrix using NetworkX
  # and multiply by X to determine the flow in/out of a node
  constraints = constraints + [A @ X <= b]

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

  flow_dict = None

  if problem.status not in ["infeasible", "unbounded"]:
    flow_dict = {edge: value for edge, value in zip(G.edges,
↪   np.round(X.value, 4))}

  # Get linear objective value - needed since congestion will
  # inflate the objective value rather than return the actual
  # cost
  cost = C @ X.value

  return flow_dict, cost


def show_flows(G, flow_dict):
  """
  Display min-cost network flow solution for a graph G using flows in
↪   flow_dict

  Params:
    - G: NetworkX graph
    - flow_dict: Dictionary of edges and flow value, from
↪   `min_cost_network_flow()`
  """
  pos = nx.shell_layout(G)
  nx.draw(G, pos, with_labels=False, node_color="skyblue", node_size=1000,
↪   font_size=12)

  demand_dict = nx.get_node_attributes(G, "demand")
  node_labels = {node_idx: (node_idx, demand) for node_idx, demand in
↪   demand_dict.items()}
  nx.draw_networkx_labels(G, pos, labels=node_labels)
  nx.draw_networkx_edge_labels(G, pos, edge_labels=flow_dict)
```

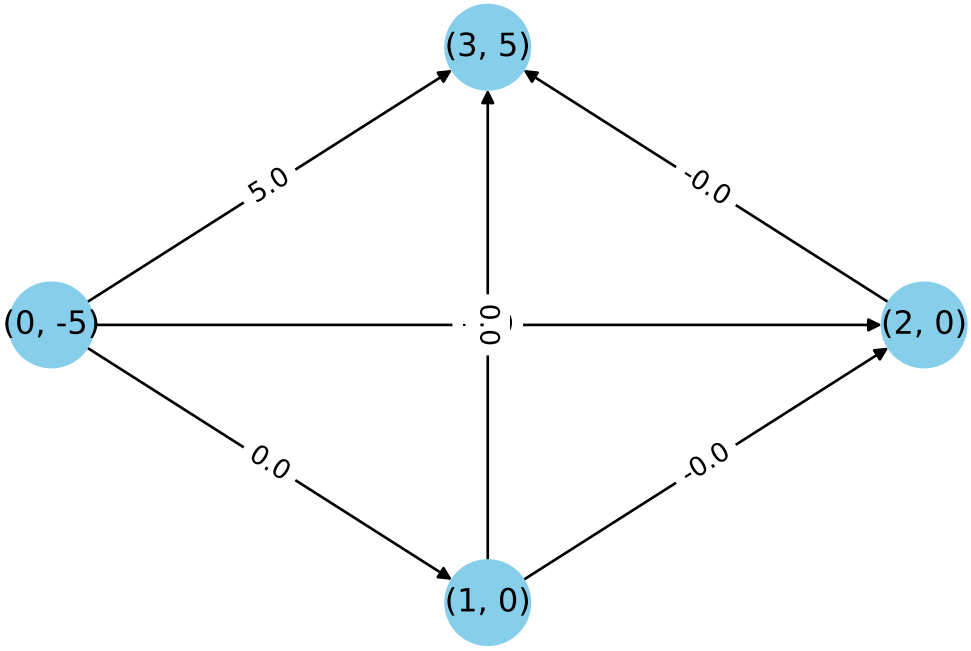
We can now compare our solution against NetworkX's `min_cost_flow()` function with a simple graph:

```
G = nx.DiGraph()
G.add_nodes_from(
  [
    (0, {"demand": -5}),
    (1, {"demand": 0}),
    (2, {"demand": 0}),
    (3, {"demand": 5}),
   ]
)

G.add_weighted_edges_from(
  [
    (0, 1, 3),
    (0, 2, 6),
    (0, 3, 1),
    (1, 3, 1),
    (1, 2, 3),
    (2, 3, 2),
  ]
)


min_cost_no_congestion, objective_value = min_cost_network_flow(G,
 ↪  with_congestion=False)
print(min_cost_no_congestion)
print(objective_value)

nx_flowDict = nx.min_cost_flow(G)
print(nx_flowDict)
print(nx.cost_of_flow(G, nx_flowDict))

show_flows(G, min_cost_no_congestion)
```

```
/home/derry/.local/share/virtualenvs/MATH441-vlJmoJsg/lib/python3.8/site-packages/cvxpy/redu
FutureWarning:
    Your problem is being solved with the ECOS solver by default. Starting in
    CVXPY 1.5.0, Clarabel will be used as the default solver instead. To
    continue
    using ECOS, specify the ECOS solver explicitly using the
    ``solver=cp.ECOS``
```

```
    argument to the ``problem.solve`` method.

  warnings.warn(ECOS_DEPRECATION_MSG, FutureWarning)
```

```
{(0, 1): 0.0, (0, 2): -0.0, (0, 3): 5.0, (1, 3): 0.0, (1, 2): -0.0, (2, 3):
-0.0}
4.999999999897586
{0: {1: 0, 2: 0, 3: 5}, 1: {3: 0, 2: 0}, 2: {3: 0}, 3: {}}
5
```



7.3.1 Comparing Solutions With and Without Congestion

We compare the min-cost network flow solutions for the graph in Vanderbei example 14.1 (see Vanderbei 2020, 230–31)

```
G = nx.DiGraph()
G.add_nodes_from(
  [
    (0, {"demand": 0}),
    (1, {"demand": 0}),
    (2, {"demand": 6}),
```

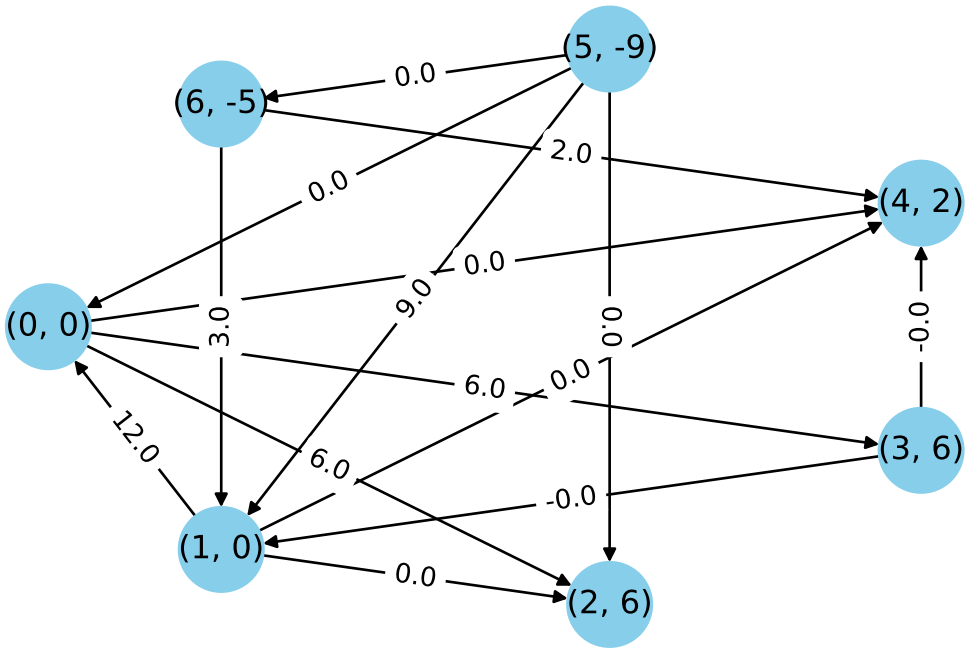
```
    (3, {"demand": 6}),
    (4, {"demand": 2}),
    (5, {"demand": -9}),
    (6, {"demand": -5}),
   ]
)

G.add_weighted_edges_from(
  [
    (0, 2, 48), (0, 3, 28), (0, 4, 10),
    (1, 0, 7), (1, 2, 65), (1, 4, 7),
    (3, 1, 38), (3, 4, 15),
    (5, 0, 56), (5, 1, 48), (5, 2, 108), (5, 6, 24),
    (6, 1, 33), (6, 4, 19),
  ]
)
```

Min-cost network flow without congestion

```
min_cost_no_congestion, objective_value = min_cost_network_flow(G,
 ↪  with_congestion=False)
print(min_cost_no_congestion)
print(objective_value)
show_flows(G, min_cost_no_congestion)

# NX solution for extra validation
nx_flowDict = nx.min_cost_flow(G)
print(nx_flowDict)
print(nx.cost_of_flow(G, nx_flowDict))
```

```
/home/derry/.local/share/virtualenvs/MATH441-vlJmoJsg/lib/python3.8/site-packages/cvxpy/redu
FutureWarning:
    Your problem is being solved with the ECOS solver by default. Starting in
    CVXPY 1.5.0, Clarabel will be used as the default solver instead. To
    continue
    using ECOS, specify the ECOS solver explicitly using the
    ``solver=cp.ECOS``
    argument to the ``problem.solve`` method.

  warnings.warn(ECOS_DEPRECATION_MSG, FutureWarning)
```

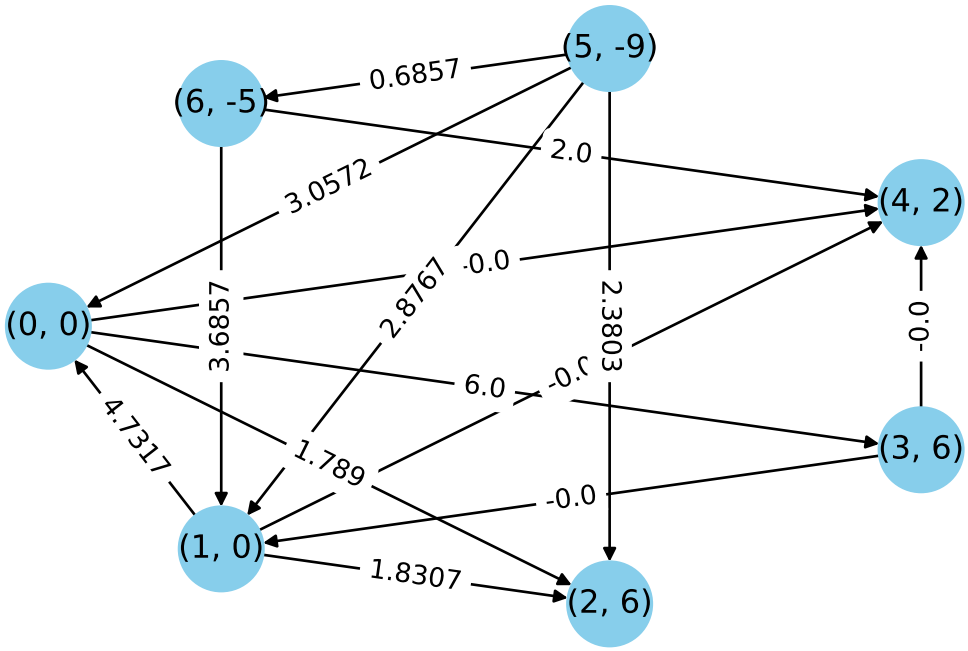
```
{(0, 2): 6.0, (0, 3): 6.0, (0, 4): 0.0, (1, 0): 12.0, (1, 2): 0.0, (1, 4):
0.0, (3, 1): -0.0, (3, 4): -0.0, (5, 0): 0.0, (5, 1): 9.0, (5, 2): 0.0, (5,
6): 0.0, (6, 1): 3.0, (6, 4): 2.0}
1109.0000000447496
{0: {2: 6, 3: 6, 4: 0}, 1: {0: 12, 2: 0, 4: 0}, 2: {}, 3: {1: 0, 4: 0}, 4:
{}, 5: {0: 0, 1: 9, 2: 0, 6: 0}, 6: {1: 3, 4: 2}}
1109
```



Min-cost network flow with congestion

```
min_cost_with_congestion, objective_value = min_cost_network_flow(G,
 ↪  with_congestion=True)
print(objective_value)
show_flows(G, min_cost_with_congestion)
```

```
1148.4367330171174
```



7.4 Conclusion and Analysis of Results

As seen in the above, adding the congestion term works in spreading out the flow load around a larger subset of the edges (also depending on the network transportation costs). There could be many uses for adding congestion costs in order to spread out the flow along more edges. For example, packet delivery in an internet network is sensitive to congestion enough to add specific protocols to add mechanisms to control the entry of packets into the network. This could help give a more realistic analysis on packet flow as to not overwhelm the network.

Another interesting lesson this can teach us is about the **design** of optimization problems. This congestion term adds a more "soft constraint" similar to regularization terms (eg. $\max f(x) - \lambda g(x)$ where $\lambda g(x)$ is a penalty) to make sure that the problem is able to consider "warnings" rather than restricting it using explicit constraints. For example, there is nothing restricting the feasibility of sending a lot of data through a single link, but there might be other concerns with slowing down the network if it does happen.

In summary, there are a variety of ways to improve the mathematical model for an optimization problem to better fit the intricacies of a generic problem statement which can include the technique detailed above of restricting flow by quadratically increasing the cost as more flow is sent through a single edge.

8 Convex Optimization, DCP, and the Smallest Enclosing Ball Problem

---

```
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt
```

```
(CVXPY) Apr 15 11:54:16 AM: Encountered unexpected exception importing solver
GLOP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
(CVXPY) Apr 15 11:54:16 AM: Encountered unexpected exception importing solver
PDLP:
RuntimeError('Unrecognized new version of ortools (9.9.3963). Expected <
9.8.0. Please open a feature request on cvxpy to enable support for this
version.')
```

8.1 Convex Optimization

In previous entries, we have been primarily using CVXPY (see Diamond and Boyd 2016), a *convex optimization* package for Python, to solve a variety of optimization problems.

The keyword in CVXPY is *convex optimization* and attempting to solve an unsupported problem will throw an error:

```
try:
  x = cp.Variable(1)
  objective = cp.Minimize(x**4 - 3*x**2 + 2)
  problem = cp.Problem(objective)
  problem.solve()
except Exception as err:
  print(err)
```

```
Problem does not follow DCP rules. Specifically:
The objective is not DCP. Its following subexpressions are not:
power(var1, 4.0) + -3.0 @ power(var1, 2.0) + 2.0
```

In this entry, we will take a look into the definitions of convex optimization problems, Disciplined Convex Programming (DCP) rules, and see how we can verify that a problem's components are solvable by CVXPY by hand to better understand its capabilities.

8.1.1 What is Convex Optimization?

A **convex optimization** problem is a problem consisting on minimizing a *convex function* over a *convex set* ("Chapter 8: Convex Optimization," n.d.). Convex problems take the standard form:

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t} \quad & g_i(x) \leq 0, \quad i = 1, 2, \ldots, m \\ & h_j(x) = 0, \quad j = 1, 2, \ldots, p \end{aligned}$$

where $f, g_i : \mathbb{R}^n \to \mathbb{R}$ are convex functions representing the objective and inequality constraints and $h_i : \mathbb{R}^n \to \mathbb{R}$ are affine functions representing any equality constraints.

Convex optimization problems are a superset of problems involving linear programming, integer programming, and quadratic programming which is why these can all be solved by CVXPY. Note that all of these problems follow the above standard form with specific objective and constraints; they all aim to min/max some objective f subject to some set of constraints which form a convex set.

One very important and handy property of convex problems is that **all local optimum are also *global* optimum** which we have experienced empirically through LP and QP problems. There is less of a worry to verify the global optimality of convex optimization solutions as any solution is the best.

We will now define some more of the intricacies of the definition above, namely what are *convex functions* and *convex sets*.

Convex Sets

A set $C \subseteq \mathbb{R}^n$ is called *convex* if

$$\forall x, y \in C, \ \lambda \in [0, 1], \ \lambda x + (1 - \lambda) y \in C$$

In other words, we can take any two points inside some set and the line drawn between the two points are also contained in the set ("Chapter 6: Convex Sets," n.d.).

Some examples of convex sets include circles, ellipses, squares and other polyhedra, and even infinite sets like $\mathbb{R}, \mathbb{R}^+$ while examples of sets that are not convex are stars and bananas since you can find a pair of points where the line between them are not inside the set.

An important feature of convex sets are that all other points are visible from any location inside the set. This follows from the line intuition detailed above since all lines of sight are contained in the set which also gives an intuitive representation that any locally optimal point is globally optimal; it's like seeing all other feasible solutions and being able to verify that there are no hidden points that could have a better value than the current optimum.

Convex Functions

A function $f : C \to \mathbb{R}$ defined on a convex set $C \subseteq \mathbb{R}^n$ is *convex* if

$$\forall x, y \in C,\ \lambda \in [0, 1],\ f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

This can be seen as drawing a line between two points on the set (the RHS of the inequality) and if all function values between the two points are underneath that line ("Chapter 7: Convex Functions," n.d.).

One very special property of convex functions is that there are a variety of operations that preserve convexity including but not limited to multiplication by non-negative scalars (ie. αf), summation of convex functions (ie. $f_1 + f_2 + \cdots + f_n$), and composition with a nondecreasing convex function (ie. $g(f(x))$). This will come in handy later.

Examples of convex functions include lines and quadratics which allows the above definition for convex optimization to hold for LP and QP objective functions and constraints.

8.1.2 Disciplined Convex Programming (DCP)

Before CVXPY can employ methods to solve the input problem, it must first verify that it is convex. In CVXPY, the problem is constructed from an objective function and a list of constraints (ie. follows the semantic definition for a convex problem above).

The objective and constraint functions will be broken into sub-components which we can use **composition rules for convex functions as stated above** to determine the *curvature* of each component. DCP labels each component as one of constant, affine, convex, concave, or unknown.

Afterward, the entire problem can be labelled as convex if it follows the following rules:

- The problem objective must be either convex (minimize) or concave (maximize)
- The constraints are one of `affine == affine`, `convex <= concave`, or `concave >= convex`

If the problem has components that satisfy these rules, then it can be **guaranteed** solvable by CVXPY (even if it takes an exponential amount of time).

More information behind the theory of DCP in general can be found in (Grant, Boyd, and Ye 2006).

8.2 Smallest Enclosing Ball/Chebyshev Center Problem

We will illustrate some convex analysis with a simple problem:

> Given a set of m points $P = \{p_1, \dots, p_m\},\ p_i \in \mathbb{R}^n$ in some n-dimensional space $\mathbb{R}^n$, how can we enclose all points in a n-dimensional ball?

We can describe this problem in the standard form

$$\begin{aligned} \min_{r} \quad & r \\ \text{s.t} \quad & ||x - p_i|| \leq r,\ i = 1, 2, \dots, m \end{aligned}$$

where we try to find both a center point $x \in \mathbb{R}^n$ and the radius $r \in \mathbb{R}$.

Below we will go through the problem to determine if it follows DCP rules and then solve it using Python.

8.2.1 Objective Function

In order to follow DCP rules, the objective $f(x) = r$ must be convex.

Proof. We show that $f(x) = r$ is convex. Let $x, y \in \mathbb{R}^n$ and $\lambda \in [0, 1]$. Then,

$$f(\lambda x + (1-\lambda)y) = r \leq r = \lambda r + (1-\lambda)r = \lambda f(x) + (1-\lambda)f(y)$$

□

In fact, a constant function is affine and affine functions are both convex and convex (ie. $r = r \iff r \geq r \wedge r \leq r$). Hence, the objective function for the smallest enclosing ball problem passes DCP rules.

8.2.2 Constraints

In order to follow DCP rules, we show that the constraint LHS $||x - p_i||$ is convex (the RHS is concave since it is a constant).

Proof. Like in how DCP is actually applied in CVXPY, we break down $g(x) = ||x - p_i||$ into smaller components. The interior $x - p_i$ is the summation of affine functions which is both concave and convex.

So, we show that the 2-norm $h(z) = ||z||$ is convex. Let $x, y \in \mathbb{R}^n$ and $\lambda \in [0, 1]$. Then,

$$\begin{aligned} h(\lambda x + (1-\lambda)y) = ||\lambda x + (1-\lambda)y|| &\leq ||\lambda x|| + ||(1-\lambda)y|| \\ &= \lambda||x|| + (1-\lambda)||y|| \\ &= \lambda h(x) + (1-\lambda)h(y) \end{aligned}$$

by the triangle inequality.

Hence, norms are convex and the constraint $||x - p_i|| \leq r$ is convex $\leq$ concave so this constraint follows DCP rules. □

Since now we have mathematically shown that all components of this problem follow DCP rules, we can confidently use CVXPY below to guarantee a solution is able to be obtained.

8.2.3 Solutions using Python

We write and solve the above problem using CVXPY:

```
def chebyshev_center(P):
  """
  Find the smallest enclosing ball for a set of points P

  Params:
    - P: numpy matrix of points

  Returns:
    A tuple (x, r) representing the center and radius of the ball
  """
  m = np.shape(P)[0]
  n = np.shape(P)[1]

  r = cp.Variable(1)
  X = cp.Variable(n)
```

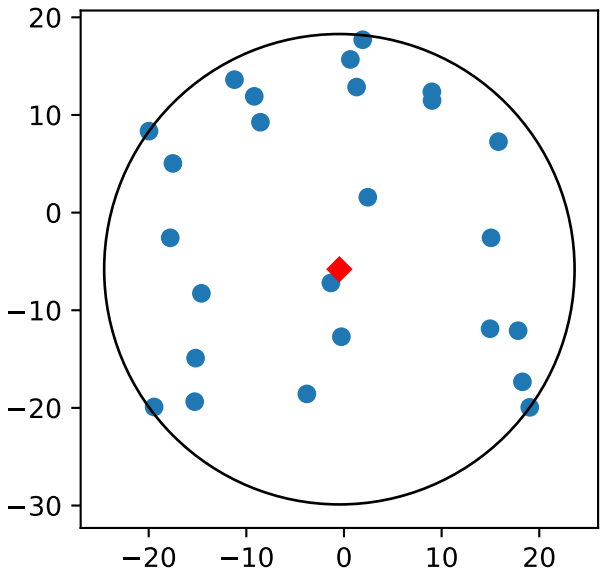
```
  objective = cp.Minimize(r)
  constraints = [cp.norm(X - P[i]) <= r for i in range(m)]

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

  return objective.value, X.value

def generate_points(num_points, coord_range):
  """
  Generate num_points in R^2

  Params:
    - num_points: number of points to generate (m)
    - coord_range: tuple (min, max) to generate points within

  Returns:
    num_points x 2 matrix of points
  """
  range_min, range_max = coord_range

  x_coords = np.random.uniform(range_min, range_max, num_points)
  y_coords = np.random.uniform(range_min, range_max, num_points)

  points = np.column_stack((x_coords, y_coords))

  return points

def display_circle(r, c, points):
  """
  Display the smallest enclosing circle solution using plt (2D case only)

  Params:
    - r: radius of circle
    - c: center of circle [x_1, x_2]
    - points: m x 2 matrix of points
  """
  fig, ax = plt.subplots()
  ax.set_aspect("equal")

  # Plot points
```

```
  ax.scatter(points[:,0], points[:,1])

  # Plot circle and center
  ax.plot(c[0], c[1], "D", color="r")
  cir = plt.Circle((c[0], c[1]), r, fill=False)
  ax.add_patch(cir)

  plt.show()
```

We generate points and solve the smallest enclosing ball problem:

```
points = generate_points(25, (-20, 20))
radius, center = chebyshev_center(points)
display_circle(radius, center, points)
```

```
/home/derry/.local/share/virtualenvs/MATH441-vlJmoJsg/lib/python3.8/site-packages/cvxpy/redu
FutureWarning:
    Your problem is being solved with the ECOS solver by default. Starting in
    CVXPY 1.5.0, Clarabel will be used as the default solver instead. To
    continue
    using ECOS, specify the ECOS solver explicitly using the
    ``solver=cp.ECOS``
    argument to the ``problem.solve`` method.

  warnings.warn(ECOS_DEPRECATION_MSG, FutureWarning)
```



8.3 Conclusion

Convex optimization is the set of problems that requires convex objective and contraint functions. In order to verify the feasibility of the problem, convex optimization packages such as CVXPY leverage the properties of convex functions to determine whether it follows a set of rules. Although there is much more going on in the backend to verify function convexity (eg. all of the CVXPY built in functions are documented with curvature and signs), it follows the structured nature derived from the mathematical definitions of convexity.

As a result, if we can determine that we are optimizing over convex functions then we can solve a greater variety of problems including geometry using norms as constraints and all subsets of convex optimization like linear and integer programming.

Axelrod, Robert, and William D. Hamilton. 1981. "The Evolution of Cooperation." *Science* 211 (4489): 1390–96. http://www.jstor.org/stable/1685895.

"Chapter 6: Convex Sets." n.d. In *Introduction to Nonlinear Optimization: Theory, Algorithms, and Applications with Python and MATLAB, Second Edition*, 113–34. https://doi.org/10.1137/1.9781611977622.ch6.

"Chapter 7: Convex Functions." n.d. In *Introduction to Nonlinear Optimization: Theory, Algorithms, and Applications with Python and MATLAB, Second Edition*, 135–69. https://doi.org/10.1137/1.9781611977622.ch7.

"Chapter 8: Convex Optimization." n.d. In *Introduction to Nonlinear Optimization: Theory, Algorithms, and Applications with Python and MATLAB, Second Edition*, 171–206. https://doi.org/10.1137/1.9781611977622.ch8.

Dantzig, George B. 1990. "Origins of the Simplex Method." In *A History of Scientific Computing*, 141–51. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/87252.88081.

Dantzig, George Bernard. 1963. *Linear Programming and Extensions*. Santa Monica, CA: RAND Corporation. https://doi.org/10.7249/R366.

Diamond, Steven, and Stephen Boyd. 2016. "CVXPY: A Python-Embedded Modeling Language for Convex Optimization." *Journal of Machine Learning Research* 17 (83): 1–5.

Foulds, L. R., and D. G. Johnston. 1984. "An Application of Graph Theory and Integer Programming: Chessboard Non-Attacking Puzzles." *Mathematics Magazine* 57 (2): 95–104. http://www.jstor.org/stable/2689591.

Grant, Michael, Stephen Boyd, and Yinyu Ye. 2006. "Disciplined Convex Programming." In *Global Optimization: From Theory to Implementation*, edited by Leo Liberti and Nelson Maculan, 155–210. Boston, MA: Springer US. https://doi.org/10.1007/0-387-30528-9_7.

Hagberg, Aric, Pieter J. Swart, and Daniel A. Schult. 2008. "Exploring Network Structure, Dynamics, and Function Using NetworkX," January. https://www.osti.gov/biblio/960616.

SciPy Developers. 2024. "SciPy Optimize Module - Linprog." https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linprog.html.

Spielman, Daniel A., and Shang-Hua Teng. 2003. "Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time." https://arxiv.org/abs/cs/0111050.

Thadani, Satish, Seema Bagora, and Anand Sharma. 2022. "Applications of Graph Coloring in Various Fields." *Materials Today: Proceedings* 66: 3498–3501. https://doi.org/https://doi.org/10.1016/j.matpr.2022.06.392.

Vanderbei, Robert J. 2020. "Network Flow Problems." In *Linear Programming: Foundations and Extensions*, 229–56. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-39415-8_14.

Wikipedia contributors. 2022a. "Fundamental Theorem of Linear Programming — Wikipedia, the Free Encyclopedia." 2022. https://en.wikipedia.org/w/index.php?title=Fundamental_theorem_of_linear_programming&oldid=1113019829.

———. 2022b. "Incidence Matrix — Wikipedia, the Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Incidence_matrix&oldid=1109920869.

———. 2024a. "Game Theory — Wikipedia, the Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Game_theory&oldid=1209544242.

———. 2024b. "Graph Coloring — Wikipedia, the Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Graph_coloring&oldid=1195834521.

———. 2024c. "Petersen Graph — Wikipedia, the Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Petersen_graph&oldid=1209188117.