CS 5800: Algorithms SEC 05 Summer Full 2024 (Seattle)
Synthesis 3

Payal Chavan

08/09/2024

Chapter 1

Linear Programming

1.1 Introduction

Linear Programming is a mathematical concept utilized to determine the most optimal solution for a linear function. It is the process of determining an optimal solution to a problem in which the decision variables and the constraints have a linear relationship. This technique is an effective tool for generating data-driven decisions that can result in increased efficiency and cost savings. It has applications in a variety of sectors and professions, from production planning to logistics optimization.

Linear programming is a method for optimizing a specific situation. By leveraging linear programming, we can achieve the best possible outcome within the given constraints, effectively utilizing available resources.

Some of the common terminologies used in Linear Programming are:

1. **Decision Variables:** In linear programming (LP), decision variables represent the quantities we aim to optimize. For instance, if we're overseeing the production of two products (A and B), the total number of units produced for A (denoted as (X)) and B (denoted as (Y)) serves as our decision variables.

2. **Objective Function:** In linear programming (LP), the objective function defines our goal, whether it's maximizing or minimizing a certain quantity. For instance, in a production scenario aiming to maximize profit, the objective function could be expressed as: $[Z = 3X + 2Y]$ where (Z) represents total profit, and the coefficients 3 and 2 correspond to the profit per unit of products A and B, respectively.

3. **Constraints:** In linear programming (LP), constraints are limitations or restrictions that decision variables must adhere to. These constraints are typically expressed as linear inequalities. For example, if we're managing production with a limited budget, we might have the following constraints:
Budget constraint: $(2X + 3Y \leq 1000)$ Non-negativity constraint: $(X, Y \geq 0)$ These constraints ensure that we avoid overspending and that production quantities remain non-negative.

4. **Non-Negative restrictions:** For all linear programs, the decision variables should always take non-negative values. This means the values for decision variables should be greater than or equal to 0.

What is Linear Programming Formula?
General Linear Programming Formulas are,

1) Objective Function: $Z = ax + by$
2) Constraints: $px + qy \leq r$, $sx + ty \leq u$
3) Non-Negative Restrictions: $x \geq 0$, $y \geq 0$

1.2 Applications of Linear Programming

Linear programming (LP) is a versatile technique employed across numerous domains to enhance operations while adhering to given constraints. Let's delve into some primary areas of application:

1. **Agriculture:** Farmers and agricultural planners utilize linear programming to determine optimal crop selection, land utilization, and resource distribution, aiming to maximize yields and profits while conserving resources.
2. **Energy Management:** In the energy sector, linear programming is employed to optimize the combination of energy production methods. This involves balancing traditional energy sources with renewable ones to minimize costs and environmental impact while satisfying demand.
3. **Financial Planning:** Businesses and financial analysts leverage linear programming for portfolio optimization, risk management, and capital budgeting. This approach aids in making investment decisions that aim to maximize returns while minimizing risk.
4. **Supply Chain Optimization:** Linear programming aids companies in reducing costs and enhancing efficiency within their supply chains. It is utilized to identify the most cost-effective transportation routes, optimize warehouse operations, and develop effective inventory management strategies.
5. **Healthcare Logistics:** Linear programming is used in the healthcare industry to maximize the distribution of resources, including hospital beds, medical personnel, and equipment. It's essential for enhancing patient care, cutting down on wait times, and successfully controlling expenses.

1.3 Linear Programming Problems

Linear Programming Problems (LPP) focus on optimizing a linear function to determine its optimal value, which can be either a maximum or minimum. In LPP, these linear functions are known as objective functions. An objective function can include multiple variables that must meet specific conditions and adhere to linear constraints.

How to solve Linear Programming Problems?

Before solving linear programming problems, we must first formulate them according to standard parameters. The steps for solving linear programming problems are as follows:

1. **Step 1:** Identify the decision variables in the problem.
2. **Step 2:** Construct the objective function and determine whether it needs to be minimized or maximized.
3. **Step 3:** List all the constraints of the linear problem.
4. **Step 4:** Ensure that the decision variables adhere to non-negativity restrictions.
5. **Step 5:** Solve the linear programming problem using a suitable method, typically the simplex or graphical method.

1.4 Linear Programming in Standard Form

Linear Programming (LP) in standard form is a systematic representation of linear programming problems, designed to simplify their solution.

A linear program is said to be in standard form if it meets the following criteria:

1. **Objective Function:** In this form, the objective function is to be maximized or minimized, subject to a set of linear constraints.
2. **Equality Constraints:** All constraints are expressed as equalities.
3. **Non-negativity Constraints:** All variables are restricted to be non-negative.

Mathematical Representation

A standard form of linear program in matrix notation can be expressed as follows:
Maximize $z = c^T x$
subject to $Ax = b$ and $x \geq 0$ where:

(z) is the objective function value.
(c) is a vector of coefficients for the objective function.
(x) is a vector of decision variables.
(A) is a matrix of coefficients for the constraints.
(b) is a vector of constants on the right-hand side of the constraints.

Transforming to Standard Form

The following steps can be used to transform a linear programming problem into standard form:

1. **Objective Function:** When dealing with linear programming problems, it's essential to express the objective function in the correct form. If you're maximizing a quantity, keep the objective function as is. However, if it's a minimization problem, multiply the objective function by -1. This adjustment ensures that the solver works correctly and aligns with the problem's optimization direction.
2. **Equality Constraints:** When transforming inequality constraints in linear programming, follow these steps:
   (a) For a "$\leq$" constraint, introduce a slack variable (usually denoted as (s_i)) to convert it into an equality:
   Original constraint: $a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b_i$
   Converted equality: $a_1x_1 + a_2x_2 + \ldots + a_nx_n + s_i = b_i$
   (b) For a "$\geq$" constraint, introduce a surplus variable (usually denoted as (s_i)) to convert it into an equality:
   Original constraint: $a_1x_1 + a_2x_2 + \ldots + a_nx_n \geq b_i$
   Converted equality: $a_1x_1 + a_2x_2 + \ldots + a_nx_n - s_i = b_i$
   (c) If the right-hand side of the constraint is negative, multiply the entire constraint by -1 to make it positive:
   Original constraint: $a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq -b_i$
   Converted equality: $-a_1x_1 - a_2x_2 - \ldots - a_nx_n + s_i = b_i$
3. **Non-negativity Constraints:** When dealing with decision variables in linear programming, it's crucial to ensure they remain non-negative. If you encounter a variable that could potentially take negative values, consider expressing it as the difference between two non-negative variables. This approach maintains the non-negativity constraint while allowing for flexibility in representing the original variable.

Let's consider an example on linear programming problem and apply the transformation steps to bring it to standard form. Suppose we have the following linear program:

Maximize: $[z = 3x_1 + 2x_2]$
Subject to: $[2x_1 + x_2 \leq 10]$ $[x_1 + 3x_2 \geq 6]$ $[x_1, x_2 \geq 0]$

We'll follow the steps to convert the constraints into equalities and ensure non-negativity for the decision variables.

1. **Objective Function in Standard Form:** Already in maximization form.
2. **Convert Inequality Constraints to Equalities:**
   For the constraint $(2x_1 + x_2 \leq 10)$, introduce a slack variable (s_1): $[2x_1 + x_2 + s_1 = 10]$
   For the constraint $(x_1 + 3x_2 \geq 6)$, introduce a surplus variable (s_2): $[x_1 + 3x_2 - s_2 = 6]$
3. **Ensure Non-Negativity for Decision Variables:** Since both (x_1) and (x_2) are decision variables, we already have the non-negativity constraint: $[x_1, x_2 \geq 0]$

1.5 Linear Programming Methods

We utilize several kinds of methods to address linear programming problems. The two most widely applied methods are:
1) Simplex Method
2) Graphical Method

Now, let's understand these two methods in detail.

1) **Simplex Method:** The simplex method is an iterative algorithm designed to find the optimal solution for linear programming problems. It operates by transitioning from one feasible solution to another along the edges of the feasible region until the optimal solution is identified.
The simplex method is highly effective for managing large-scale linear programming problems and is extensively utilized in practice.

Given below are the key steps involved in using Simplex Method:

1. **Formulate the Problem:** Define the linear programming problem based on the given constraints.
2. **Convert Inequalities to Equations:** Transform all inequalities into equalities by adding slack variables where necessary.
3. **Construct the Initial Simplex Table:** Represent each constraint equation in a row and place the objective function in the bottom row. This forms the initial Simplex table.
4. **Identify the Pivot Column:** Locate the greatest negative entry in the bottom row. The column containing this entry is the pivot column.
5. **Determine the Pivot Row:** Divide the entries in the right-most column by the corresponding entries in the pivot column (excluding the bottom row). The row with the smallest quotient is the pivot row. The pivot element is at the intersection of the pivot row and pivot column.
6. **Perform Row Operations:** Use matrix operations to make all entries in the pivot column zero, except for the pivot element.
7. **Check for Optimality:** Examine the bottom row for negative entries. If there are no negative entries, the process is complete. If negative entries remain, repeat from step 4.
8. **Obtain the Solution:** The final Simplex table provides the solution to the problem.

2) **Graphical Method:** The graphical method is ideal for solving linear programming problems that involve two decision variables, making it perfect for two-dimensional scenarios.
The graphical method is restricted to problems with just two variables because it relies on visual representation.

Given below are the key steps involved in using Graphical Method:

1. **Plot the Constraints:** Draw the constraints as lines or curves on a coordinate plane.
2. **Feasible Region:** Determine the feasible region, which is the area where all constraint regions overlap.
3. **Objective Function Line:** Plot the objective function line, typically a straight line representing the objective function.
4. **Optimal Solution:** The optimal solution is found at the vertex of the feasible region where the objective function line touches.

1.6 Example

Let's consider an example to understand the linear programming problem.

Imagine a company that manufactures two distinct products: Product A and Product B. With limited resources in both time and money, the objective is to allocate these resources in the most efficient way to maximize profit. Here are the specifics:

1. **Resources:**
   Product A: Requires 2 hours of time and $\$100$ of investment per unit.
   Product B: Requires 3 hours of time and $\$150$ of investment per unit.
2. **Selling Prices:**
   Product A: Sells for $\$300$ per unit.
   Product B: Sells for $\$400$ per unit.
3. **Constraints:**
   Time: The company has a total of 100 hours available for production.
   Budget: The production budget is $\$10,000$.
4. **Objective:**
   Maximize Profit: The goal is to maximize total profit, which is the revenue minus costs.

Now, based on the above given specifics we will formulate the linear programming problem.

1. **Decision Variables:**
   Let x_1 be the number of units of Product A produced.
   Let x_2 be the number of units of Product B produced.
2. **Objective Function:**
   The total profit can be expressed as:Maximize $Z = 300x_1 + 400x_2$
3. **Constraints:**
   (a) **Time Constraint:** $2x_1 + 3x_2 \leq 100$
   (b) **Budget Constraint:** $100x_1 + 150x_2 \leq 10,000$
   (c) **Non-negativity Constraints:** $x_1 \geq 0, \quad x_2 \geq 0$

1.7 Exercise Problem

Question: We want to minimize the cost of a diet while ensuring it meets specific nutritional requirements. We have two food items, A and B, each providing different amounts of protein and carbohydrates per unit. Our goal is to find the optimal combination of these foods to achieve at least 60 units of protein and 70 units of carbohydrates. For protein content, Food A provides 3 units, while Food B offers 4 units. When it comes to carbohydrates, Food A has 4 units, whereas Food B contains 2 units. The cost per unit of Food A is $2 and Food B is $3. Formulate the linear programming problem to minimize the cost.

Solution:

1. **Step 1: Identify the Decision Variables.**
   (x): The number of units of Food A.
   (y): The number of units of Food B.
2. **Step 2: Define the Constraints.**
   (a) **Protein Requirement:** The diet must contain at least 60 units of protein. $[3x + 4y \geq 60]$

(b) **Carbohydrate Requirement:** The diet must contain at least 70 units of carbohydrates. $[4x + 2y \geq 70]$

(c) **Non-negativity:** The number of units of Food A and Food B cannot be negative. $[x \geq 0,\ y \geq 0]$

3. **Define the Objective Function:** We want to minimize the cost, which is the total cost of the diet. The cost per unit of Food A is \$2, and for Food B, it's \$3. Therefore, the objective function is:
Minimize $[Z = 2x + 3y]$

Now, let's use graphical representation to plot the feasible region defined by these constraints. We'll find the intersection points and evaluate the objective function at those points.

1)**Convert Inequalities to Equalities to Find Intersection Points:**
We need to find the intersection points of the lines:
$3x + 4y = 60$
$4x + 2y = 70$

2)**Find the Intersection Points:**
We solve the system of equations:
$3x + 4y = 60$
$4x + 2y = 70$

3)**Find the Intersection with Axes:**
For $3x + 4y = 60$:
When $x = 0, 4y = 60 \implies y = 15$
When $y = 0, 3x = 60 \implies x = 20$

For $4x + 2y = 70$:
When $x = 0, 2y = 70 \implies y = 35$
When $y = 0, 4x = 70 \implies x = 17.5$

4)**Next, we solve the intersection of the lines:**
$3x + 4y = 60$
$4x + 2y = 70$

Let's solve these equations simultaneously.

The intersection point of the lines $3x + 4y = 60$ and $4x + 2y = 70$ is $(16, 3)$.

5)**Vertices of the Feasible Region:**
Intersection with axes for $3x + 4y = 60$:
$(20, 0)$
$(0, 15)$

Intersection with axes for $4x + 2y = 70$:
$(17.5, 0)$
$(0, 35)$

Intersection of the two lines:
$(16, 3)$

According to our objective function $2x + 3y = 2(20) + 3(0) \implies 40$. So the total minimum cost for the diet is \$40. Given below is the graph showing feasible region and it's optimal solution.

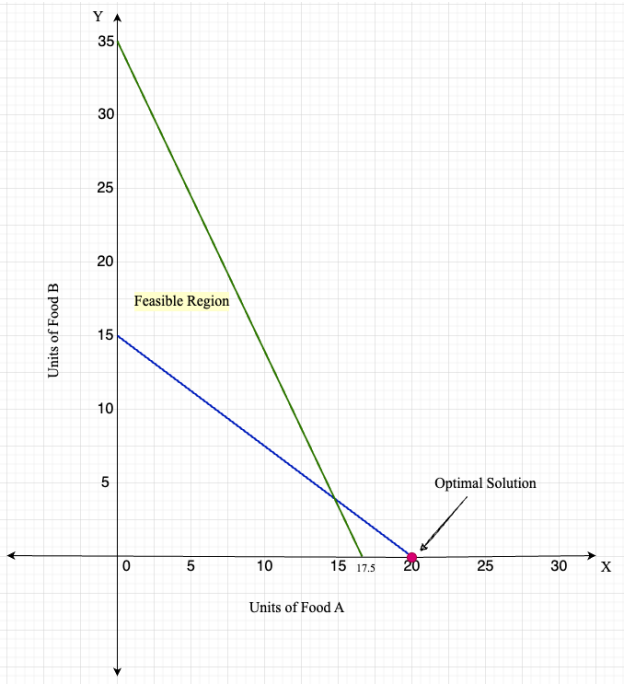

Figure 1: Graph showing Feasible Region and Optimal Solution.

Hence, 20 units of Food A and 0 units of Food B is required to minimize the total cost of diet.

1.8 Coding Problem

Question:

As the head of the building department, you oversee the final inspections of two types of new commercial buildings: gas stations and restaurants. These inspections are conducted by three different inspectors: a plumbing inspector, an electrical inspector, and a building inspector.
The plumbing inspector needs 4 hours to inspect a gas station and 2 hours to inspect a restaurant.
The electrical inspector requires 2 hours to inspect a gas station and 6 hours to inspect a restaurant.
The building inspector takes 4 hours to inspect a gas station and 6 hours to inspect a restaurant.
Given their other responsibilities, the plumbing inspector has 28 hours available per week for inspections, the electrical inspector has 30 hours, and the building inspector has 36 hours. Your goal is to maximize the number of commercial buildings inspected each week. Set up this problem as a linear programming problem.

Solution:

Let's setup the linear programming problem.

1. **Define the Decision Variables:**
   Let (x_1) be the number of gas stations inspected per week.
   Let (x_2) be the number of restaurants inspected per week.
2. **Define the Objective Function:**
   Maximize the total number of inspections: Maximize $(Z = x_1 + x_2)$.
3. **Write the Constraints:**
   (a) Plumbing Inspector Constraint: $[4x_1 + 2x_2 \leq 28]$
   (b) Electrical Inspector Constraint: $[2x_1 + 6x_2 \leq 30]$

(c) Building Inspector Constraint: $[4x_1 + 6x_2 \leq 36]$

(d) Non-negativity Constraints: $[x_1 \geq 0, \quad x_2 \geq 0]$

Below are the outputs for the Linear Programming solution.

```
Optimal number of gas stations to inspect: 8.0
x1 - Optimal number of gas stations to inspect: 6.0
x2 - Optimal number of restaurants to inspect:  2.0
```

Figure 2: Output of the Coding solution

Given below is the feasible region and optimal solution for the given linear programming problem.

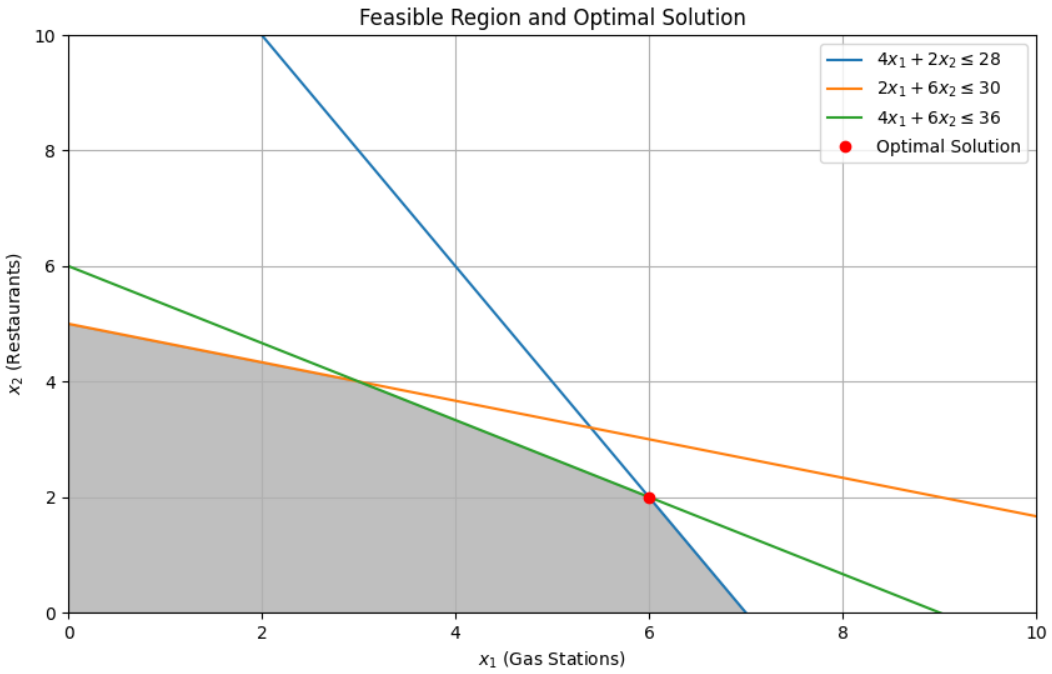

Figure 3: Feasible Region and Optimal Solution Plot

The shaded region represents the feasible solutions, indicating the possible combinations of gas stations and restaurants that can be inspected each week to maximize the total number of commercial building inspections. The 'blue' line represents the Plumbing Inspector's available inspection hours for gas stations and restaurants. The 'orange' line shows the Electrical Inspector's available inspection hours for gas stations and restaurants. The 'green' line depicts the Building Inspector's available inspection hours for gas stations and restaurants. The optimal solution (indicated by a red dot) is found at the point $(6, 2)$, meaning 6 gas stations and 2 restaurants can be inspected each week.

Please refer to the Python file 'LinearProg.py' for complete solution.

Chapter 2
Definitions of P and NP

2.1 Introduction

P vs. NP is an unusual discovery in the field of theoretical computer science. Since the issue hasn't been resolved, it has gained popularity. It essentially poses the following question: Is it simple to solve a problem if it is simple to verify that a solution is correct?

Let's now first define P-type and NP-type problems.

P-type problems: P-type problems are those for which solutions can be computed by computers in polynomial time. Additionally, the correctness of these solutions can also be verified by computers. In other words, both finding the solution and checking its validity can be efficiently performed by algorithms in polynomial time.

NP-type problems: NP-type problems, which stands for non-deterministic polynomial type problems, are challenging for computers to solve. Unlike P-type problems, their solutions are not available in polynomial time. However, the good news is that their solutions can still be verified efficiently by computers. Non-deterministic algorithms exist for solving these problems, although they operate in non-deterministic polynomial time.

In this case, we are aware that P-type problems are an NP-type problem's subset. Since a computer can check a problem with the same ease that it can solve it, all P problems are also NP problems.

The central question in the P vs NP problem is whether every problem that can be efficiently verified (in NP) can also be solved efficiently (in P). In simpler terms, if we can quickly check a solution, can we also find that solution quickly?
For example, consider Sudoku puzzles: verifying a filled grid's correctness is easy (an NP problem), but can we rapidly solve any arbitrary Sudoku puzzle? That's the essence of the P problem.

Exploring the Relationship Between P and NP

The central question in the P vs NP problem revolves around the relationship between two classes of problems: P (problems solvable in polynomial time) and NP (problems whose solutions can be efficiently verified). Specifically, we seek to determine whether P equals NP. In simpler terms, can every problem with a quickly verifiable solution also be solved quickly? Despite extensive research, this question remains unanswered. If P equals NP, verifying correctness would be equivalent to finding the solution method. If not, some problems lack quick solutions, even though their correctness can still be verified efficiently.

Classification of P and NP

Let's delve into the classification of problems using an analogy to real-life difficulty levels. Just as we categorize tasks based on their complexity, we do the same with computational problems. The P class represents problems that are solvable efficiently (in polynomial time), akin to easy tasks. On the other hand, NP encompasses problems whose solutions can be verified efficiently, even if finding the solution itself is challenging—think of them as moderately difficult.

$P \rightarrow Easy$
$NP \rightarrow Medium$
$NP - Complete \rightarrow Hard$
$NP - Hard \rightarrow Hardest$

Now, let's visualize this relationship through a diagram.

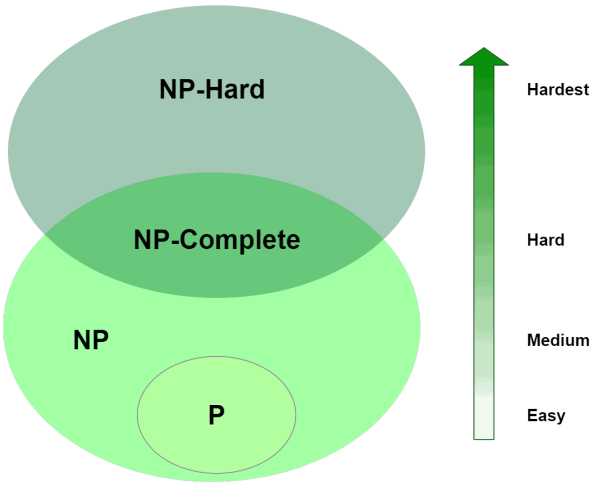

Figure 4: Relationship Between P and NP.

Given the diagram, we make the assumption that P and NP are distinct sets, or more precisely, $P \neq NP$. This assumption remains unproven but appears to be true. Additionally, there is an intriguing overlap between NP and NP-Hard. When a problem belongs to both of these sets, we refer to it as NP-Complete.

NP-Hard: An NP-Hard problem is one for which an algorithm that solves it can be transformed into an algorithm that solves any NP problem (non-deterministic polynomial time problem). In other words, NP-hard problems are "at least as hard as any NP problem," although they may actually be even more challenging.

NP-Complete: An NP-complete problem meets the following criteria: 1) It belongs to the class NP, meaning that solutions can be verified in polynomial time. 2) It is NP-hard, implying that it is at least as challenging as the hardest problems in NP.

2.2 Applications of P and NP

What if we could now demonstrate that $P = NP$? This would imply that there are efficient solutions for problems involving efficient verification. The following are some practical uses that would be affected:

1. **Prime Factorization Algorithms:** This would make it simpler to create and calculate cryptographic algorithms, such as RSA. Even with smaller key sizes, deciphering an RSA encryption would be easier.
2. **Vehicle Routing (Travelling Salesman Problem):** By determining the shortest route between cities, transportation efficiency can be greatly increased and costs can be reduced.
3. **Facility Location:** By optimizing supply chains and cutting costs, the best location for a factory relocation could be identified.
4. **Circuit Designing:** Relying less on hardware, solving big boolean circuits via approximations would become more efficient.
5. **Scheduling Algorithms:** Various sectors would benefit from effective scheduling for work, appointments, or resources.

2.3 Examples of P and NP

Example of P-type

One classic example of a problem in P is Binary Search.

Binary Search Algorithm: Finding a target value's location within a sorted array is possible with the help of the binary search algorithm. The way it operates is that until the target value is located or the search space is empty, the portion of the array that might contain the target value is continually divided in half.

Let's break down the steps and discuss its time complexity.

1. Start with the entire array.
2. Determine the middle element of the current search interval.
3. If the middle element is equal to the target value, return its position.
4. If the target value is less than the middle element, repeat the search on the left half of the array.
5. If the search space is empty (i.e., the target value is not in the array), terminate the search.

The key insight is that Binary Search divides the search space in half at each step, leading to a logarithmic time complexity. Specifically, it runs in $\mathcal{O}(\log n)$ time, where 'n' represents the size of the array. Since this time complexity is polynomial (logarithmic), Binary Search belongs to the class P (polynomial time problems).

Example of NP-type

One classic example of an NP problem is the Traveling Salesman Problem (TSP).

Traveling Salesman Problem (TSP): The TSP is a classic optimization problem where a salesperson needs to visit a set of cities exactly once and return to the starting city while minimizing the total distance traveled. Given a list of cities and the distances between each pair of cities, the goal is to find the shortest possible route that satisfies these conditions.

Decision Version of TSP: In the decision version of TSP, we're given a set of cities, the distances between them, and a threshold value 'k.' The task is to determine whether there exists a tour (a sequence of cities) such that the total distance traveled is less than or equal to 'k.' This decision problem is in NP because verifying a given solution (i.e., checking if a tour meets the distance constraint) can be done in polynomial time.

Time Complexity: Finding the optimal tour (i.e., solving TSP) itself is computationally challenging and is believed to require more than polynomial time (it's an NP-hard problem). However, verifying a given tour's distance (i.e., checking if it satisfies the constraint) can be done in polynomial time by summing the distances between consecutive cities in the tour. Therefore, TSP falls within the NP complexity class.

2.4 Exercise Problem

Question: Given a set of n cities and a distance matrix D where $D[i][j]$ denotes the distance between city i and city j, the goal is to determine the shortest possible route that visits each city exactly once and returns to the origin city.

Input:
1) A list of cities $[C_1, C_2, C_3, \ldots, C_n]$
2) A distance matrix D of size $n \times n$, where $D[i][j]$ is the distance between city C_i and city C_j

Output:
1) The shortest possible route that visits each city exactly once and returns to the origin city.
2) The total distance of this shortest route.

Consider four cities with the following distance matrix:

	C_1	C_2	C_3	C_4
C_1	0	10	15	20
C_2	10	0	35	25
C_3	15	35	0	30
C_4	20	25	30	0

Tasks:
1) Solve the problem using a brute-force approach, which is an NP-type algorithm.
2) Discuss why this problem is classified as NP-hard.
3) Explain why finding an efficient algorithm that runs in polynomial time (if one exists) would imply that P=NP.

Solution:

Task 1:
Let's first solve this Travelling Salesman Problem (TSP) using a brute-force approach.

We will follow the below steps to determine the solution.

Step 1: Generate all permutations of cities (excluding the starting city, as it remains fixed).
Step 2: Calculate the total distance for each permutation.
Step 3: Choose the permutation with the minimum total distance.

Let's now compute the distances:

Route 1: $C1 \rightarrow C2 \rightarrow C3 \rightarrow C4 \rightarrow C1$
Total distance: $10 + 35 + 30 + 20 = 95$

Route 2: $C1 \rightarrow C2 \rightarrow C4 \rightarrow C3 \rightarrow C1$
Total distance: $10 + 25 + 30 + 15 = 80$

Route 3: $C1 \rightarrow C3 \rightarrow C2 \rightarrow C4 \rightarrow C1$
Total distance: $15 + 35 + 25 + 20 = 95$

Route 4: $C1 \rightarrow C3 \rightarrow C4 \rightarrow C2 \rightarrow C1$
Total distance: $15 + 30 + 25 + 10 = 80$

Route 5: $C1 \rightarrow C4 \rightarrow C2 \rightarrow C3 \rightarrow C1$
Total distance: $20 + 25 + 35 + 15 = 95$

Route 6: $C1 \rightarrow C4 \rightarrow C3 \rightarrow C2 \rightarrow C1$
Total distance: $20 + 30 + 35 + 10 = 95$

Hence, the shortest distance is Route 2 and Route 4 with a minimum total distance of 80.

Taking into account each possible combination of cities and figuring out the total distance for each combination is an easy method to solve the TSP.
However, larger instances makes this brute-force method impractical due to its $\mathcal{O}(n!)$ of exponential time complexity.

Task 2:
The Traveling Salesman Problem (TSP) is classified as NP-hard because it is at least as difficult as the most challenging problems in the NP class (problems that can be verified in polynomial time). Specifically, TSP is NP-hard because it can be reduced to the Hamiltonian Cycle Problem, which involves finding a cycle that visits each vertex exactly once. If we can solve TSP in polynomial time, we can also solve the Hamiltonian Cycle Problem in polynomial time, thereby confirming

that TSP is NP-hard.

Task 3:
If an efficient algorithm that runs in polynomial time were found for the Traveling Salesman Problem (TSP), it would imply that P (the class of problems solvable in polynomial time) is equal to NP (the class of problems verifiable in polynomial time). This discovery would have profound implications for fields such as computer science, cryptography, and optimization, enabling many real-world problems to be solved efficiently. However, proving that P=NP remains an open question, and no polynomial-time algorithm for TSP has been discovered to date.

2.5 Coding Problem

Question: Demonstrate the above exercise problem using a python code.

Solution:
Please refer to the Python file 'TSP.py' for complete solution.

Chapter 3
Hash Tables

3.1 Introduction

Hashing

Before diving into hash tables, it's important to grasp the concept of hashing. In essence, hashing involves taking an input of variable length and converting it into a fixed-length output, known as a hash code or simply a hash.
A hashing function is tasked with converting a variable-length input into a hash. Since there isn't a universal hash function, we can design hash functions based on the typical characteristics of our data inputs.

Here are some everyday examples of hashing:

1) In universities, each student is given a unique roll number, which can be used to access their information.
2) In libraries, each book is assigned a unique identifier, which helps determine details about the book, such as its exact location in the library or the users it has been issued to.

In both cases, students and books are effectively hashed to unique numbers.

Imagine you have an object and you want to assign it a key to simplify searching. You could use a basic array-like data structure where keys (integers) serve directly as indices to store values. However, when the keys are large and can't be used directly as indices, hashing becomes necessary.

In hashing, large keys are transformed into smaller keys using hash functions. These values are then stored in a data structure known as a hash table. The goal of hashing is to evenly distribute entries (key/value pairs) across an array. Each element is given a key (converted key), which allows for $\mathcal{O}(1)$ time access. The hash function computes an index based on the key, indicating where an entry can be located or inserted.

Hashing involves two main steps:
1) An element is transformed into an integer using a hash function. This integer serves as an index to store the original element in the hash table.
2) The element is then stored in the hash table, allowing for quick retrieval using the hashed key.

```
hash_value = hashfunc(key)
index = hash_value % array_size
```

In this approach, the hash value is independent of the array size and is reduced to an index (a number between 0 and $array\_size - 1$) using the modulo operator (%).

Hash Function

A hash function is any function that maps a data set of arbitrary size to a data set of fixed size, which is then placed into a hash table. The values produced by a hash function are known as hash values/ hash codes/ hash sums/ hashes.

To ensure an effective hashing mechanism, a good hash function should meet the following basic requirements:
1) Easy to compute: It should be straightforward to compute and not overly complex.
2) Uniform distribution: It should distribute entries uniformly across the hash table to avoid clustering.
3) Minimize collisions: Collisions, where different elements are assigned the same hash value, should be minimized.

Note: Regardless of how well-designed a hash function is, collisions are inevitable. Therefore, it is crucial to manage collisions using various collision resolution techniques to maintain the performance of a hash table.

Now that we've thoroughly explored the fundamentals of hashing, hash functions, and the criteria for selecting a good hash function, it's time to shift our focus to our main topic: Hash Tables.

Hash Tables

A hash table is a data structure designed for the efficient insertion, lookup, and removal of key-value pairs. It relies on the concept of hashing, where each key is converted by a hash function into a unique index within an array. This index serves as the storage location for the corresponding value. Simply put, a hash table maps keys to their associated values.

For example, we can use a hash table to link people's names to their personal information. In this scenario, the names serve as our raw keys. A hash function processes these keys to determine their corresponding indexes in the hash table, allowing for direct access to the personal information.

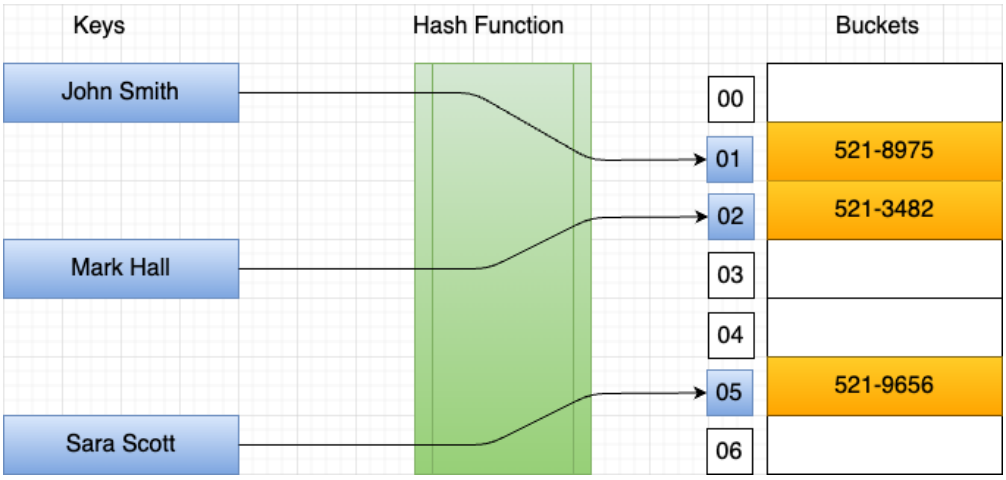

Figure 5: Hash Table Diagram

Hash tables exemplify the tradeoff between time and space. If we had unlimited time, we could link all keys to the same index and perform a binary search to retrieve specific data.
Conversely, with unlimited space, we could use the entire key as the index, creating as many individual memory buckets as needed to store the data associated with each key.

However, in reality, we have neither infinite time nor space. Therefore, we must handle hash collisions and index sharing, as discussed below.

Collisions in Hash Tables

Collisions occur when two or more keys map to the same array index. Techniques for resolving collisions include chaining, open addressing, and double hashing.

1. **Open Addressing:** Collisions are resolved by searching for the next available slot in the table. If the initial slot is occupied, the hash function is applied to subsequent slots until an empty one is found. Variants of this approach include double hashing, linear probing, and quadratic probing.
2. **Separate Chaining:** Each slot in the hash table contains a linked list of objects that hash to that slot. If multiple keys hash to the same slot, they are added to the linked list. This method is straightforward and can handle multiple collisions effectively.

3. **Robin Hood Hashing:** To minimize chain length, Robin Hood hashing resolves collisions by swapping keys. If a new key hashes to an already-occupied slot, the algorithm compares the distances of the two keys from their ideal slots. If the new key is closer to its ideal slot, it swaps places with the existing key, bringing the existing key closer to its ideal slot. This method tends to reduce collisions and average chain length.

4. **Dynamic Resizing:** This feature allows the hash table to grow or shrink based on the number of elements it contains, maintaining an optimal load factor and ensuring quick lookup times.

3.2 Pseudocode Algorithm for Hash Tables

```
hash_function(key, table_size):
// Compute the hash value for the given key
hash_value = key MOD table_size
RETURN hash_value

add_to_hash_table(hash_table, key, value, table_size):
// Compute the hash value for the key using the hash function
hash_value = hash_function(key, table_size)
// Add the key-value pair to the hash table
hash_table[hash_value] = (key, value)
```

1) The hash_function computes the hash value for a given key by taking the modulo of the key with the table size.
2) The add_to_hash_table function then uses this hash value to determine the index where the corresponding key-value pair should be stored in the hash table.

3.3 Time and Space Complexity of Hash Tables

1. In the best-case scenario for a hash table, the time complexity for searching, inserting, and deleting data remains constant—$\mathcal{O}(1)$. This means that regardless of the operation, a single hash table lookup is sufficient to find the desired memory bucket.

2. In a hash table, the average time complexity for searching, inserting, and deleting data is constant, denoted as $\mathcal{O}(1)$. This means that, on average, a single hash table lookup is enough to find the desired memory bucket, regardless of the specific operation.

3. In a hash table, the worst-case time complexity for searching, inserting, and deleting data is typically $\mathcal{O}(n)$, which means linear time. This scenario arises when all the data in the hash table has keys that map to the same index.

4. For all the 3 scenarios - best case, average case, and worst case, the space complexity grows linearly with the number of key-value pairs stored in the hash map. Each key-value pair occupies constant space, contributing to the overall complexity. Hence, we have $\mathcal{O}(n)$, where n is the number of items inserted.

3.4 Applications of Hash Tables

Hashing tables, sometimes referred to as hash maps, are useful in a variety of real-world scenarios.

1. **Spell Checkers and Autocorrect:** Using hash tables facilitates the implementation of autocorrect and spell checkers. They swiftly check to see if a word is spelled correctly and keep a glossary of correct words.

2. **File Systems:** Hashed tables are used by file systems to handle file metadata, such as timestamps, permissions, and file names. A hash table is used to store each file's characteristics for quick and easy access.

3. **Network Routing Tables:** Hashing tables are used in networking for routing. To find the router or next hop for packet forwarding, IP addresses are hashed.

4. **Hash-Based Algorithms:** Digital signatures, password hashing, and safe data storage all use cryptographic hash algorithms (e.g., SHA-256).

3.5 Exercise Problem

Question: Given the following input keys: 4322, 1334, 1471, 9679, 1989, 6171, 6173, and 4199, and the hash function $(h(x) = x \mod 10)$, which of the following statements are true?

$i)$ 9679, 1989, and 4199 hash to the same value.
$ii)$ 1471 and 6171 hash to the same value.
$iii)$ All elements hash to the same value.
$iv)$ Each element hashes to a different value.

Options:
(A) i only
(B) ii only
(C) i and ii only
(D) iii or iv

Solution:

Let's compute the hash values using the given hash function.

$(h(9679) = 9679 \mod 10 = 9)$
$(h(1989) = 1989 \mod 10 = 9)$
$(h(4199) = 4199 \mod 10 = 9)$
$(h(1471) = 1471 \mod 10 = 1)$
$(h(6171) = 6171 \mod 10 = 1)$

Statements (i): 9679, 1989, and 4199 hash to the same value 9 is true.
Statements (ii) 1471 and 6171 hash to the same value 1 is also true.

Both statements (i) and (ii) are correct. Therefore, option (C) is the accurate option.

3.6 Coding Problem

Question:

You are provided with a HashTable class implemented in Python. This class allows insertion, retrieval, deletion, and updating of key-value pairs, as well as the ability to fetch all key-value pairs stored in the hash table. The hash table uses separate chaining to handle collisions.

Your tasks are as follows:

$1)$ Insertion: Add the following key-value pairs to an instance of the HashTable:
("grape", 4)
("melon", 6)
("kiwi", 7)

$2)$ Retrieval: Retrieve the value associated with the key "melon".

$3)$ Update: Update the value of the key "kiwi" to 15.

4) Deletion: Delete the key-value pair with the key "grape".

5) Fetch all key-value pairs: After performing the above operations, fetch all the key-value pairs in the hash table.

Solution:

Please refer to the Python file 'hash.py' for complete solution.

Chapter 4
Shortest path on a DAG using dynamic programming

4.1 Introduction

Before diving deep into the topic, let's first understand some of the fundamental concepts in the context of graph algorithms and optimization techniques.

Directed Acyclic Graph (DAG)

Directed Graph:
A graph in which each edge has a specific direction, connecting one vertex (node) to another is called as Directed Graph.

Acyclic Graph:
A graph that contains no cycles, ensuring that it's impossible to begin at one vertex and traverse a series of edges that ultimately return to the starting vertex is called as acyclic.

In essence, a Directed Acyclic Graph (DAG) is a graph characterized by directed edges and the absence of cycles. In a DAG, you can traverse from one vertex to another following the direction of the edges, but you will never be able to return to the starting vertex by following these edges. In essence, there is no possibility of looping back to the original point.

Example:

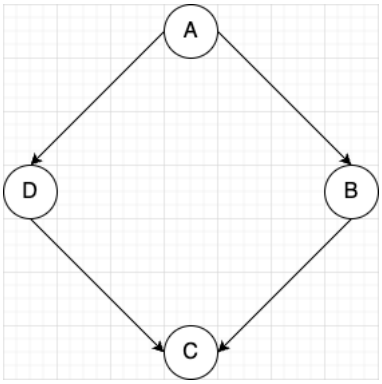

Figure 6: DAG Example

Shortest Path Problem in a DAG:
Given a Directed Acyclic Graph (DAG) with weighted edges (where each edge has a non-negative weight), we aim to determine the shortest path from a source vertex to all other vertices. Each edge has an associated numerical value, representing the "cost" to traverse that edge.

Dynamic Programming:
A technique for solving problems by decomposing them into smaller subproblems and storing the results of these subproblems to prevent redundant calculations is known as Dynamic Programming.

Relationship Between Dynamic Programming (DP) and Directed Acyclic Graphs (DAG):

1. **Subproblems and DAG Nodes:** Dynamic Programming breaks a problem into smaller, similar subproblems, solving each independently and using their results to address the original problem. In a Directed Acyclic Graph (DAG), nodes represent these subproblems or states.
2. **Transitions and DAG Edges:** Just as nodes in a DAG represent subproblems, the edges represent transitions. An edge indicates the transition from one subproblem to another.
3. **Cycles and Redundancy:** The graph is acyclic because cycles would cause some states to lead back to themselves after a series of transitions, resulting in redundancy.

Shortest Path on a DAG Using Dynamic Programming:

Now let's quickly understand the terms related to DAG for finding the shortest path.

Topological Sorting: Topological sorting of a Directed Acyclic Graph (DAG) is a linear sequence of vertices where, for every directed edge $u \rightarrow v$, vertex u precedes vertex v in the sequence.
This can be performed using Depth-First Search (DFS). This technique guarantees that when you process a vertex, all preceding vertices have already been processed.

Note: Topological sorting is not feasible for graphs that are not DAGs.

Example:

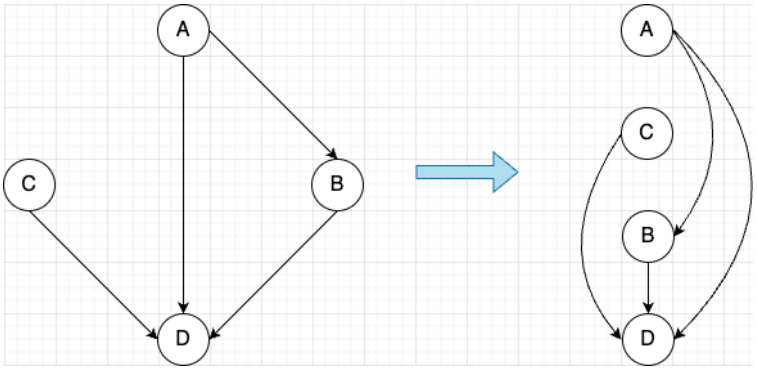

Figure 7: DAG and it's Topological Sorting

The topological sorting for the above DAG is: A, C, B, D.

Relaxation: The process of updating the shortest path estimate for a vertex is called as relaxation.
It involves the following steps: For each vertex (u) in topologically sorted order, update the shortest path estimates for all adjacent vertices (v) using the edge $(u \rightarrow v)$.

Using dynamic programming to find the shortest path in a DAG takes advantage of the topological order of vertices, ensuring efficient and accurate computation of shortest path distances. This approach is especially effective because the acyclic nature of the graph ensures that each vertex is processed in an optimal sequence, eliminating the need to handle cycles.

Steps to Find the Shortest Path on a DAG:

1. **Topologically Sort the DAG:** This provides a linear sequence of vertices.
2. **Initialize Distances:**
   Source Vertex: Set the distance to the source vertex to 0.
   Other Vertices: Initially set the distance to all other vertices to infinity (or a very large number).

3. **Relax Edges According to Topological Order:** For each vertex in the topologically sorted sequence, update the distances of its adjacent vertices.
4. **Retrieve the Shortest Path Distance:** After processing all vertices, obtain the shortest path distances from the source to each vertex.

4.2 Applications of DAG using Dynamic Programming

Using dynamic programming to find the shortest path in a Directed Acyclic Graph (DAG) has practical applications across various domains. Let's examine a few examples:

1. **Network Routing:** In computer networks, routers utilize shortest path algorithms to find the most efficient routes for data packets. A DAG-based approach ensures that no cycles are present, thereby preventing loops in routing decisions.
2. **Task Scheduling:** DAGs represent dependencies between tasks in project scheduling or parallel computing. Identifying the shortest path helps optimize task execution time by reducing dependencies and enhancing parallelism.
3. **Genetic Sequencing:** When aligning DNA sequences, using dynamic programming on a DAG efficiently identifies the most similar subsequences. This approach minimizes the edit distance or alignment score.
4. **Natural Language Processing (NLP):** In Natural Language Processing (NLP), parsing sentences involves building a dependency tree, which is a type of DAG. Shortest path algorithms assist in identifying grammatical relationships between words.

4.3 Pseudocode Algorithm for DAG using DP

shortestPathDAG(graph, source):
Initialize an array 'dist' with infinity values for all vertices
Set $dist[source] = 0$

Topological sort to ensure correct order of processing
$topologicalOrder = topologicalSort(graph)$

for each vertex u in topologicalOrder:
for each adjacent vertex v of u:
# Relaxation step
if $dist[u] + weight(u, v) < dist[v]$:
$dist[v] = dist[u] + weight(u, v)$

return $dist$

Helper function for topological sort
function $topologicalSort(graph)$:
Initialize an empty stack 'stack'
Initialize a boolean array 'visited' for all vertices (initially set to false)

for each vertex u in graph:
if not $visited[u]$:
$topologicalSortUtil(u, visited, stack)$

return $stack$

Recursive utility function for topological sort
function $topologicalSortUtil(u, visited, stack)$:
Mark u as visited
for each adjacent vertex v of u:
if not $visited[v]$:
$topologicalSortUtil(v, visited, stack)$
Push u onto the stack

Let's break down the pseudocode algorithm for finding the shortest path in a Directed Acyclic Graph (DAG) using dynamic programming:

1. **Initialization:** We begin by initializing an array named 'dist' to keep track of the shortest distances from the source vertex to all other vertices. Initially, all distances are set to infinity, except for the source vertex, which is assigned a distance of 0.

2. **Topological Sort:** The 'topologicalSort' function produces a topological order (a linear sequence of vertices) for the DAG. This order guarantees that vertices are processed in a manner that respects their dependencies, ensuring no edge points backward. A stack is used to maintain this order.

3. **Dynamic Programming Step:** For each vertex u in the topological order:
   1) Iterate over all adjacent vertices v of u.
   2) If the distance from the source to v through u (i.e., dist$[u]$ + weight(u, v)) is less than the current distance stored in dist$[v]$, update dist$[v]$.
   This step ensures that we determine the shortest path to each vertex by considering the optimal paths to its predecessors.

4. **Result:** Once all vertices have been processed, the 'dist' array holds the shortest distances from the source vertex to every other vertex.

4.4 Time and Space Complexity of DAG

1. The best-case scenario arises when the graph has a straightforward structure (e.g., few edges) and the topological sort is efficient. In this situation, the time complexity is mainly dictated by the topological sorting, which takes $\mathcal{O}(V + E)$ time.
2. The worst-case scenario occurs when the graph is densely connected (with many edges), making the topological sort more time-consuming. In this case, the time complexity is still dominated by the topological sorting, resulting in $\mathcal{O}(V + E)$ time.
3. The average-case time complexity is influenced by the distribution of graphs encountered. In practice, most real-world DAGs are neither extremely dense nor extremely sparse. Consequently, the average-case time complexity is also approximately $\mathcal{O}(V + E)$.
4. We need to store the distances for all vertices, which requires $\mathcal{O}(V)$ space. Additionally, we need space for the topological sort. Therefore, the space complexity is $\mathcal{O}(V)$.

4.5 Exercise Problem

Question: Given a Directed Acyclic Graph (DAG) with 6 vertices and the following edges with weights:

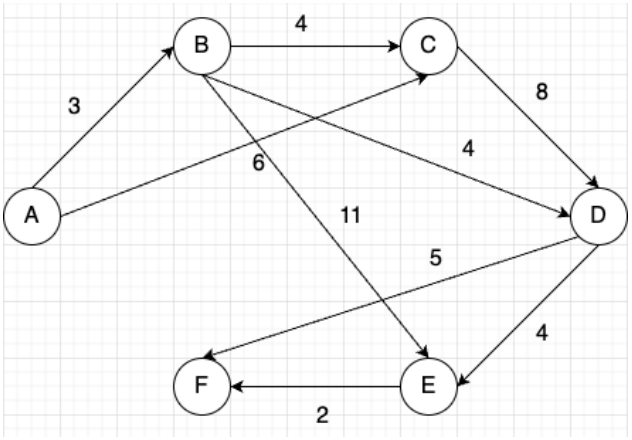

Figure 8: DAG

Find the shortest path from vertex A to vertex F using dynamic programming.

Solution:

We will use the following steps to approach this problem:

1. **Topological Sort:** Perform a topological sort on the DAG.
   A valid topological order for the given graph is: A, B, C, D, E, F.
2. **Initialize Distances:** Initialize the distances from the source vertex (A) to all other vertices as infinity, except for the source vertex itself, which should be 0.
   $dist = A : 0, B : inf, C : inf, D : inf, E : inf, F : inf$

3. **Relax Edges According to Topological Order:** Iterate through the vertices in topological order and update the distances to the adjacent vertices if a shorter path is found.

   (a) For vertex A:
   Update distance to B: $dist[B] = min(inf, 0 + 3) = 3$
   Update distance to C: $dist[C] = min(inf, 0 + 6) = 6$

   (b) For vertex B:
   Update distance to C: $dist[C] = min(6, 3 + 4) = 6$
   Update distance to D: $dist[D] = min(inf, 3 + 4) = 7$
   Update distance to E: $dist[E] = min(inf, 3 + 11) = 14$

   (c) For vertex C:
   Update distance to D: $dist[D] = min(7, 6 + 8) = 7$

   (d) For vertex D:
   Update distance to E: $dist[E] = min(14, 7 + 4) = 11$
   Update distance to F: $dist[F] = min(inf, 7 + 5) = 12$

   (e) For vertex E:
   Update distance to F: $dist[F] = min(12, 11 + 2) = 12$

   (f) The final distances are:
   $dist = A : 0, B : 3, C : 6, D : 7, E : 11, F : 12$

4. **Retrieve the Shortest Path:**

   (a) Backtrack from the destination vertex (F) to the source vertex (A) to retrieve the path and distance.

   (b) Start from F, the predecessor is D (since the edge $D \to F$ with weight 5 was used).

   (c) From D, the predecessor is B (since the edge $B \to D$ with weight 4 was used).

   (d) From B, the predecessor is A (since the edge $A \to B$ with weight 3 was used).

   (e) The shortest path is: $A \to B \to D \to F$ with a total distance of 12.

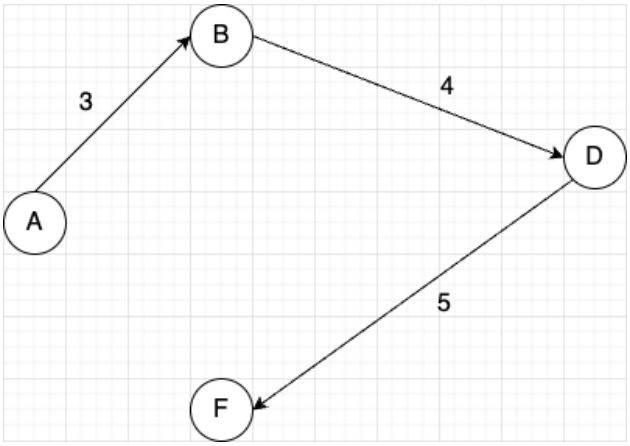

Figure 9: DAG Shortest Path

4.6 Coding Problem

Question: You are given a list of courses and their prerequisites, where each course is represented as a vertex in a Directed Acyclic Graph (DAG). Each prerequisite relationship between two courses is represented as a triplet (course1,

course2, difficulty), indicating that to take course2, you must first complete course1, and the weight represents the difficulty or time required to complete course1.

Your task is to find the shortest path from a given starting course S (0) to a target course T (5), taking into account the difficulty or time required to complete each prerequisite course.

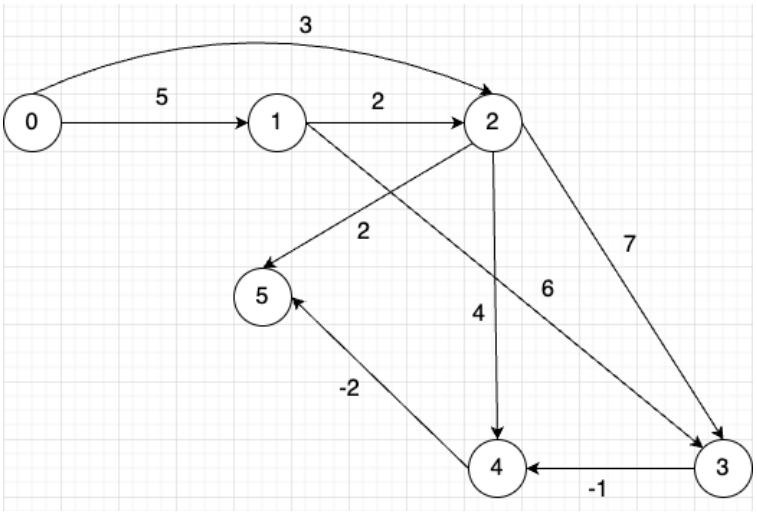

Figure 10: DAG Representing Courses.

Solution:

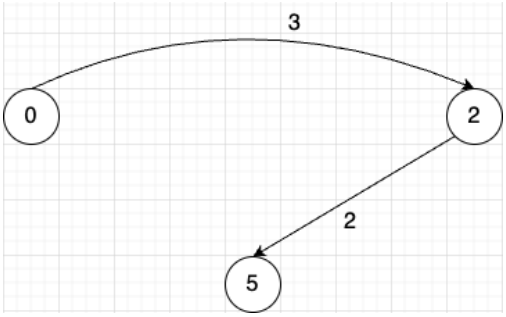

Figure 11: DAG Shortest Path

Please refer to the Python file 'dag.py' for complete solution.

Reflection:

In this Synthesis 3, I gained a comprehensive understanding of several key topics. I delved into Linear Programming, learning about the standard form and how to transform problems into this form. I also acquired valuable knowledge about P and NP-type problems, exploring their complexity classes and significance. Additionally, I explored the concept and applications of hash tables. Furthermore, I learned how to find the shortest path in a Directed Acyclic Graph (DAG) using dynamic programming techniques. These topics have deepened my understanding of these fundamental concepts and their practical applications.

Acknowledgements:

I would like to express my sincere gratitude to the following individuals and resources for their invaluable contributions to this assignment:
1) Prof. Bruce Maxwell: Thank you, Professor Bruce Maxwell for your guidance in this assignment. Your expertise in algorithms greatly influenced my work.
2) Classmates and TA's: I appreciate the discussions of my classmates, and TA's who clarified my doubts.
3) DPV Algorithms Textbook: This textbook was a useful resource for me to understand the basics of algorithms.
4) https://domino.ai/data-science-dictionary/hash-table: This resource helped me with the understanding of the basics of Hash Tables.
5) https://www.baeldung.com/cs/p-np-np-complete-np-hard: This website helped me in understanding the P and NP.
6) https://www.analyticsvidhya.com/blog/2017/02/lintroductory-guide-on-linear-programming-explained-in-simple-english/: This website helped me clear Linear Programming concepts.
7) https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/: This website is a great resource to understand how to find shortest path in DAGs.

Used Time Travel Day: $2nd$