

**M.Sc. (Five Year Integrated) in Computer Science
(Artificial Intelligence & Data Science)**

Fourth Semester

Laboratory Record

**23-813-0407: OPTIMIZATION TECHNIQUES
LAB**

*Submitted in partial fulfillment
of the requirements for the award of degree in
Master of Science (Five Year Integrated)
in Computer Science (Artificial Intelligence & Data Science) of
Cochin University of Science and Technology (CUSAT)
Kochi*



Submitted by

**MUHSINA BEEGUM
(81323016)**

**DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY (CUSAT)
KOCHI-682022**

APRIL 2025

DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY (CUSAT)
KOCHI, KERALA-682022



*This is to certify that the software laboratory record for **23-813-0407: OPTIMIZATION TECHNIQUES LAB** is a record of work carried out by **MUHSINA BEEGUM (81323016)**, in partial fulfillment of the requirements for the award of degree in **Master of Science (Five Year Integrated)** in **Computer Science (Artificial Intelligence & Data Science)** of Cochin University of Science and Technology (CUSAT), Kochi. The lab record has been approved as it satisfies the academic requirements in respect of the fourth semester laboratory prescribed for the Master of Science (Five Year Integrated) in Computer Science degree.*

Faculty Member in-charge

Dr. Remya.K.Sasi
Assistant Professor
Department of Computer Science
CUSAT

Dr. Madhu S Nair
Professor and Head
Department of Computer Science
CUSAT

CONTENTS

Sl.No	Title	Page no
1	Travelling salesman Problem	1
2	0/1 Knapsack Problem	3
3	Furniture Profit Maximization using lpp	4
4	Crop Profit Maximization using lpp	6
5	Cake Revenue Maximization using lpp	8
6	Linear Programming Without Package	10
7	Transportation Problem	12
8	Assignment Problem	15
9	EOQ	18

1. TRAVELLING SALESMAN PROBLEM

AIM

To implement the Travelling Salesman Problem (TSP) using an appropriate algorithm in order to determine the shortest possible route that visits each city exactly once and returns to the starting city.

PROGRAM

```
from itertools import permutations

def tsp_brute_force(graph):
    nodes=list(graph.keys())
    shortest_path=None
    min_cost=float('inf')

    for perm in permutations(nodes):
        cost=sum(graph[perm[i]][perm[i+1]]for i in range (len(perm)-1))
        cost+=graph[perm[-1]][perm[0]]

        if cost < min_cost:
            min_cost=cost
            shortest_path=perm

    return shortest_path,min_cost

def input_graph():
    graph={}
    n=int(input("Enter the number of nodes : "))
    print("enter the nodes(separated by space) :")
    nodes=input().split()

    for node in nodes:
        graph[node]={}
    print ("Enter the distance between nodes :")

    for i in range(n):
        for j in range (i+1 , n):
            print(f"Distance between nodes {nodes[i]} and {nodes[j]}:",end="")
            distance=int(input())
            graph[nodes[i]][nodes[j]] = distance
```

```
        graph[nodes[j]][nodes[i]] = distance
    return graph

# Main program
graph = input_graph()
path, cost = tsp_brute_force(graph)
print("Shortest Path:", path, "with Cost:", cost)
```

SAMPLE INPUT-OUTPUT

```
PS C:\Users\muhsi\OneDrive\Desktop\POTD-Series-\OT_labexam> python salesman.py
Enter the number of nodes : 4
enter the nodes(separated by space) :
A B C D
Enter the distance between nodes :
Distance between nodes A and B:3
Distance between nodes A and C:5
Distance between nodes A and D:6
Distance between nodes B and C:8
Distance between nodes B and D:2
Distance between nodes C and D:6
Shortest Path: ('A', 'B', 'D', 'C') with Cost: 16
```

2. 0/1 KNAPSACK PROBLEM

AIM

To implement the 0/1 Knapsack Problem using dynamic programming in order to determine the maximum total value that can be obtained by selecting items within a given weight capacity, where each item can be either included or excluded (0/1 choice).

PROGRAM

```
def knapsack_01(values, weights, capacity):
    n = len(values)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], values[i - 1] +
                               dp[i - 1][w - weights[i - 1]])
            else:
                dp[i][w] = dp[i - 1][w]

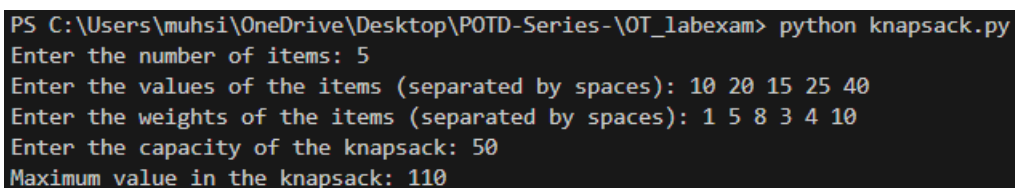
    return dp[n][capacity]

def input_knapsack():
    n = int(input("Enter the number of items: "))
    values = list(map(int, input("Enter the values of the items
(separated by spaces): ").split()))
    weights = list(map(int, input("Enter the weights of the items
(separated by spaces): ").split()))
    capacity = int(input("Enter the capacity of the knapsack: "))

    return values, weights, capacity

values, weights, capacity = input_knapsack()
max_value = knapsack_01(values, weights, capacity)
print(f"Maximum value in the knapsack: {max_value}")
```

SAMPLE INPUT-OUTPUT



```
PS C:\Users\muhsi\OneDrive\Desktop\POTD-Series-\OT_labexam> python knapsack.py
Enter the number of items: 5
Enter the values of the items (separated by spaces): 10 20 15 25 40
Enter the weights of the items (separated by spaces): 1 5 8 3 4
Enter the capacity of the knapsack: 50
Maximum value in the knapsack: 110
```

3. FURNITURE PROFIT MAXIMIZATION

AIM

To implement a linear programming model to solve the problem where a furniture company produces chairs and tables from two resources, wood and metal. The company has 125 units of wood and 80 units of metal available. Each chair requires 1 unit of wood and 3 units of metal, while each table requires 5 units of wood and 1 unit of metal. The profit from selling a chair is 20 dollars, and the profit from selling a table is 30 dollars. The objective is to determine how many chairs and tables the company should produce to maximize its profit and to find the maximum profit using a linear programming package.

PROGRAM

```
from scipy.optimize import linprog
# to maximize 20x (chairs) + 30y (tables) [since linprog does minimization by default]
c = [-20, -30]

#coefficients of constraints x + 5y <= 125 && 3x + y <= 80
A = [
    [1, 5],
    [3, 1]
]

#solutions of constraints
b = [125, 80]

#bounds for x >= 0 and y >=0
x_bounds = (0, None)
y_bounds = (0, None)

result = linprog(c, A_ub = A, b_ub = b, bounds =[x_bounds, y_bounds], method = 'highs')

if result.success:
    chairs = result.x[0]
    tables = result.x[1]

    max_profit = -result.fun #converting to positive profit from minimization

    print(f"Optimal number of chairs: {chairs}")
    print(f"Optimal number of tables: {tables}")
    print(f"Maximum amount of profit: ${max_profit}")
```

```
else:
    print("No solution found.")
\begin{figure}
    \centering
    \includegraphics[width=0.5\linewidth]{furniture.png}
    \caption{Enter Caption}
    \label{fig:enter-label}
\end{figure}
```

SAMPLE INPUT-OUTPUT

```
PS C:\Users\muhsi\OneDrive\Desktop\POTD-Series-\OT_labexam> python furniture.py
Optimal number of chairs: 19.64285714285714
Optimal number of tables: 21.071428571428573
Maximum amount of profit: $1025.0
```


4. CROP PROFIT MAXIMIZATION

AIM

To implement a linear programming model to solve the problem where a farmer has a field of 60 acres to plant two crops, wheat and barley. The farmer must plant at least 20 acres of wheat and at least 10 acres of barley. He has 1200 pounds of fertilizer and 600 pounds of insecticide available. Each acre of wheat requires 20 pounds of fertilizer and 10 pounds of insecticide, while each acre of barley requires 10 pounds of fertilizer and 15 pounds of insecticide. The profit from an acre of wheat is 200 dollars, and the profit from an acre of barley is 150 dollars. The objective is to determine how many acres of each crop the farmer should plant to maximize his profit and to find the maximum profit using a linear programming package.

PROGRAM

```
from scipy.optimize import linprog
c = [-200, -150]

A = [
    [1, 1],
    [20,10],
    [10,15]
]

#solutions of constraints
b = [60,1200, 600]

#bounds for x >= 0 and y >=0
x_bounds = (20, None)
y_bounds = (10, None)

result = linprog(c, A_ub = A, b_ub = b, bounds =[x_bounds, y_bounds], method = 'highs')

if result.success:
    wheat = result.x[0]
    barley = result.x[1]

    max_profit = -result.fun #converting to positive profit from minimization

    print(f"Optimal number of wheat: {wheat}")
```

```
    print(f"Optimal number of barley: {barley}")
    print(f"Maximum amount of profit: ${max_profit}")
else:
    print("No solution found.")
```

SAMPLE INPUT-OUTPUT

```
PS C:\Users\muhsi\OneDrive\Desktop\POTD-Series-\OT_labexam> python farmers.py
Optimal number of wheat: 45.0
Optimal number of barley: 10.0
Maximum amount of profit: $10500.0
```

5. CAKE REVENUE MAXIMIZATION

AIM

To implement a linear programming model to solve the problem where a bakery sells two types of cakes, chocolate and vanilla. The bakery has a daily budget of 500 dollars and can produce at most 400 cakes per day. Each chocolate cake costs 2 dollars to make and sells for 5 dollars, while each vanilla cake costs 1 dollar to make and sells for 3 dollars. The bakery also needs to meet the demand of at least 100 chocolate cakes and at least 50 vanilla cakes per day. The objective is to determine how many cakes of each type the bakery should produce to maximize its revenue and to find the maximum revenue using a linear programming package.

PROGRAM

```
from scipy.optimize import linprog
c = [-5, -3]

A = [
    [2, 1],
    [1,1]
]

#solutions of constraints
b = [500,1200]

#bounds for x >= 0 and y >=0
x_bounds = (100, None)
y_bounds = (50, None)

result = linprog(c, A_ub = A, b_ub = b, bounds =[x_bounds, y_bounds], method = 'highs')

if result.success:
    chocolatecakes = result.x[0]
    vanillacakes = result.x[1]

    max_profit = -result.fun #converting to positive profit from minimization

    print(f"Optimal number of chocolate cakes: {chocolatecakes}")
    print(f"Optimal number of vanilla cakes: {vanillacakes}")
    print(f"Maximum amount of profit: ${max_profit}")
else:
    print("No solution found.")
```

SAMPLE INPUT-OUTPUT

```
PS C:\Users\muhsi\OneDrive\Desktop\POTD-Series-\OT_labexam> python bakery.py
Optimal number of chocolate cakes: 100.0
Optimal number of vanilla cakes: 300.0
Maximum amount of profit: $1400.0
```

6. LINEAR PROGRAMMING WITHOUT PACKAGE

AIM

To solve a linear programming problem manually without using any linear programming package, where the objective is to maximize $p = 2u + 3u + u$ subject to the constraints:

$$u + u + u \leq 4$$

$$u + 2u - u \leq 2$$

$$u, u, u \geq 0$$

The goal is to determine the values of u , u , and u that maximize the objective function while satisfying all constraints.

PROGRAM

```
import numpy as np
c = np.array([2,3,1])

A = np.array([[1,1,1],
              [-1,-2,1]])

b = np.array([54, -2])

bounds = [(0, None), (0, None), (0, None)]

best_p = -np.inf
best_solution = None

u1_range = np.linspace(0,54,100)
u2_range = np.linspace(0,54,100)
u3_range = np.linspace(0,54,100)

for u1 in u1_range:
    for u2 in u2_range:
        for u3 in u3_range:
            if(u1 + u2 + u3 <= 54) and (-u1 - 2*u2 + u3 <= -2)
            and (u1>= 0) and (u2>= 0) and (u3>=0):
                p = 2*u1 + 3*u2 + u3

                if p > best_p:
                    best_p = p
                    best_solution = (u1,u2,u3)
```

```
print("Maximum p: ", best_p)
print("Best solution (u1,u2,u3): ", tuple(map(int,best_solution)))
```

SAMPLE INPUT-OUTPUT

```
PS C:\Users\muhsi\OneDrive\Desktop\POTD-Series-\OT_labexam> python lpp.py
Maximum p: 162.0
Best solution (u1,u2,u3): (0, 54, 0)
```

7. TRANSPORTATION PROBLEM

AIM

Develop a program to find the optimal transportation plan for a given set of supply and demand points, along with the associated transportation costs. The program should minimize the total transportation cost.

Write a program to solve a transportation problem using the Vogel's Approximation Method (VAM). The program should take the supply and demand for each location, as well as the transportation costs between them, as input and return the optimal transportation plan (minimizing total cost). Implement a function to calculate the initial feasible solution based on VAM

PROGRAM

```
import numpy as np

def calculate_penalties(costs, supply, demand):
    row_penalties = []
    column_penalties = []

    for i in range(len(costs)):
        if supply[i] > 0:
            sorted_costs = sorted([costs[i][j]
                                   for j in range(len(costs[i])) if demand[j] > 0])
            row_penalties.append(sorted_costs[1] -
                                 sorted_costs[0] if len(sorted_costs) > 1 else 0)
        else:
            row_penalties.append(-1)

    for j in range(len(costs[0])):
        if demand[j] > 0:
            col_costs = [costs[i][j] for i in range(len(costs)) if supply[i] > 0]
            sorted_costs = sorted(col_costs)
            column_penalties.append(sorted_costs[1] - sorted_costs[0]
                                    if len(sorted_costs) > 1 else 0)
        else:
            column_penalties.append(-1)

    return row_penalties, column_penalties
```

```
def VAM(supply, demand, costs):
    supply = supply.copy()
    demand = demand.copy()
    rows, cols = len(costs), len(costs[0])
    transportation_plan = np.zeros((rows, cols))

    while np.sum(supply) > 0 and np.sum(demand) > 0:
        row_penalties, column_penalties = calculate_penalties(costs, supply, demand)

        max_row_penalty = max(row_penalties)
        max_col_penalty = max(column_penalties)

        if max_row_penalty >= max_col_penalty:
            row_index = row_penalties.index(max_row_penalty)

            min_col_index = np.argmin([costs[row_index][j] if demand[j] > 0
                                       else float('inf') for j in range(cols)])

            allocation = min(supply[row_index], demand[min_col_index])
            transportation_plan[row_index][min_col_index] = allocation
            supply[row_index] -= allocation
            demand[min_col_index] -= allocation
        else:
            col_index = column_penalties.index(max_col_penalty)

            min_row_index = np.argmin([costs[i][col_index] if supply[i] > 0
                                       else float('inf') for i in range(rows)])

            allocation = min(supply[min_row_index], demand[col_index])
            transportation_plan[min_row_index][col_index] = allocation
            supply[min_row_index] -= allocation
            demand[col_index] -= allocation
    return transportation_plan

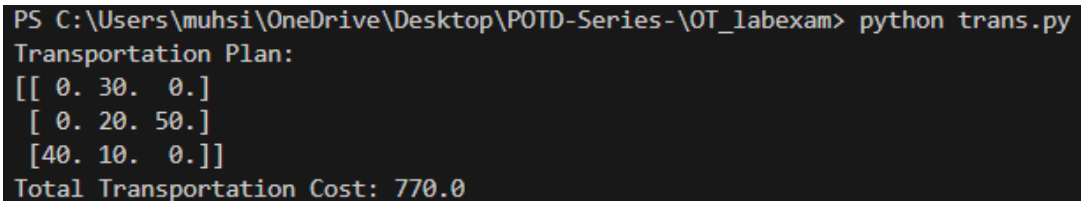
def calculate_total_cost(transportation_plan, costs):
    total_cost = np.sum(transportation_plan * np.array(costs))
    return total_cost

supply = [30, 70, 50]
demand = [40, 60, 50]
```



```
costs = [  
    [8, 6, 10],  
    [9, 5, 7],  
    [3, 2, 4]  
]  
  
transportation_plan = VAM(supply, demand, costs)  
  
total_cost = calculate_total_cost(transportation_plan, costs)  
  
print("Transportation Plan:")  
print(transportation_plan)  
print(f"Total Transportation Cost: {total_cost}")
```

SAMPLE INPUT-OUTPUT



```
PS C:\Users\muhsi\OneDrive\Desktop\POTD-Series-\OT_labexam> python trans.py  
Transportation Plan:  
[[ 0. 30.  0.]  
 [ 0. 20. 50.]  
 [40. 10.  0.]  
Total Transportation Cost: 770.0
```

8. ASSIGNMENT PROBLEM

AIM

Create a program to solve an assignment problem where a set of tasks needs to be assigned to a set of workers, with associated costs. The program should minimize the total assignment cost while ensuring each task is assigned to only one worker and each worker receives only one task.

Write a program to solve an assignment problem using the Hungarian Algorithm. The program should take the cost matrix representing the assignment costs between different tasks and workers as input and return the optimal assignment (minimizing total cost) with no conflicts.

PROGRAM

```
import numpy as np

def hungarian_algorithm(cost_matrix):
    # Step 1: Subtract the row minima
    cost_matrix = cost_matrix - cost_matrix.min(axis=1)[:, None]

    # Step 2: Subtract the column minima
    cost_matrix = cost_matrix - cost_matrix.min(axis=0)

    # Step 3: Initialize labels for rows and columns
    n = len(cost_matrix)
    u = np.zeros(n)
    v = np.zeros(n)

    # Step 4: Create matching arrays
    ind = np.full(n, -1)

    # Step 5: Start the assignment process
    for i in range(n):
        links = np.full(n, -1)
        mins = np.full(n, np.inf)
        visited = np.zeros(n, dtype=bool)
        marked_i = i
        marked_j = -1
        j = -1
        while True:
            j = -1
```

```
        for j_ in range(n):
            if visited[j_]:
                continue
            cur = cost_matrix[marked_i, j_] - u[marked_i] - v[j_]
            if cur < mins[j_]:
                mins[j_] = cur
                links[j_] = marked_j
            if mins[j_] < mins[j] or j == -1:
                j = j_
        delta = mins[j]
        for j_ in range(n):
            if visited[j_]:
                u[ind[j_]] += delta
                v[j_] -= delta
            else:
                mins[j_] -= delta
        visited[j] = True
        marked_j = j
        marked_i = ind[marked_j]
        if marked_i == -1:
            break
    while True:
        if links[j] != -1:
            ind[j] = ind[links[j]]
            j = links[j]
        else:
            ind[j] = marked_i
            break

    return ind

# Function to compute the total cost of the optimal assignment
def calculate_total_cost(cost_matrix, assignment):
    total_cost = sum(cost_matrix[i, assignment[i]] for i in range(len(assignment)))
    return total_cost

# Example usage:
cost_matrix = np.array([
    [4, 2, 8, 5],
    [2, 3, 7, 6],
    [5, 8, 1, 4],
```

```
        [6, 4, 3, 2]
    ])

# Solve the assignment problem using the Hungarian Algorithm
assignment = hungarian_algorithm(cost_matrix)

# Calculate the total cost of the optimal assignment
total_cost = calculate_total_cost(cost_matrix, assignment)

print(f"Optimal Assignment: {assignment}")
print(f"Total Cost: {total_cost}")
```

SAMPLE INPUT-OUTPUT

```
PS C:\Users\muhsi\OneDrive\Desktop\POTD-Series-\OT_labexam> python ass.py
Optimal Assignment: [-1 -1 -1 -1]
Total Cost: 17
```

9. EOQ

AIM

Write a Python function that calculates the Economic Order Quantity (EOQ) given the annual demand, ordering cost, and holding cost per unit. The function should take these three parameters as input and return the calculated EOQ.

A company manufactures 10,000 units of a product annually. The ordering cost per order is 200, and the holding cost per unit per year is 5. Write Python code to calculate the EOQ using the function you defined above. Print the calculated EOQ to the console

PROGRAM

```
import math

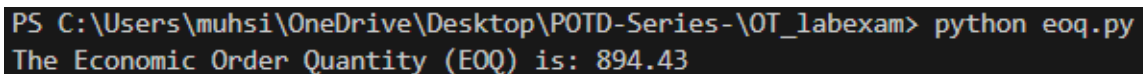
def calculate_eoq(annual_demand, ordering_cost, holding_cost):
    eoq = math.sqrt((2 * annual_demand * ordering_cost) / holding_cost)
    return eoq

# Given company data
annual_demand = 10000
ordering_cost = 200
holding_cost = 5

# Calculate EOQ
eoq = calculate_eoq(annual_demand, ordering_cost, holding_cost)

# Print the result
print(f"The Economic Order Quantity (EOQ) is: {eoq:.2f}")
```

SAMPLE INPUT-OUTPUT



```
PS C:\Users\muhsi\OneDrive\Desktop\POTD-Series-\OT_labexam> python eoq.py
The Economic Order Quantity (EOQ) is: 894.43
```