RAJALAKSHMI ENGINEERING COLLEGE

RAJALAKSHMI NAGAR, THANDALAM – 602 105

RAJALAKSHMI
ENGINEERING COLLEGE

MA23434
Optimization Techniques for AI

Laboratory Observation Note Book

Name : GANESH S

Year / Branch / Section : 2/AIML/FA

Register No. : 2116-231501046

Semester : 4

Academic Year : 2024-2025

RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)

RAJALAKSHMI NAGAR, THANDALAM – 602 105

BONAFIDE CERTIFICATE

NAME ___GANESH S___ REGISTER NO. ___2116231501046___

ACADEMIC YEAR 2024-25 **SEMESTER**- IV **BRANCH**: AIML-B.Tech

This Certification is the Bonafide record of work done by the above student

in the **MA23434- Optimization Techniques for AI** Laboratory during

the year 2024 – 2025.

Signature of Faculty -in – Charge

Submitted for the Practical Examination held on ______________

Internal Examiner External Examiner

INDEX

Name: GANESH S **Branch:** AIML **Sec:** FA **Roll no:** 231501046

Ex. No: 1 Date:	TRANSPORTATION PROBLEM

AIM:

To solve the Transportation Problem for minimizing the cost of transporting goods from multiple sources to multiple destinations using Python.

PROCEDURE:

1. **Install Required Libraries:**
   - Make sure you have numpy and scipy installed. You can install them using pip if not already installed.
     !pip install numpy scipy
2. **Define the Problem:**
   - Identify the cost matrix, supply array, and demand array.
3. **Set Up the Problem in Python:**
   - Use the scipy.optimize.linprog function to set up and solve the Transportation Problem.
4. **Run the Code:**
   - Execute the Python code to get the optimized result.
5. **Analyze the Output:**
   - Interpret the results and validate the solution against the problem constraints.

MAIN CODE:

```
import numpy as np
from scipy.optimize import linprog

def transportation_problem(supply, demand, costs):
  if sum(supply) != sum(demand):
    if sum(supply)<sum(demand):
      diff = sum(demand) -  sum(supply)
      supply.append(diff)
      costs = np.hstack((costs,np.zeros((costs.shape[0],1))))
    else:
      diff=sum(supply)-sum(demand)
      demand.append(diff)
      costs = np.vstack((costs,np.zeros((1,costs.shape[1]))))

  num_sources=len(supply)
  num_destinations=len(demand)
  A_eq= np.zeros((num_sources+num_destinations,num_sources*num_destinations))

  for i in range(num_sources):
    A_eq[i,i*num_destinations:(i+1)*num_destinations]=1
```

```
 for j in range(num_sources,num_sources+num_destinations):
   A_eq[j,j-num_sources::num_destinations]=1

b_eq=np.concatenate((supply,demand))
c=costs.flatten()

result=linprog(c,A_eq=A_eq,b_eq=b_eq,method='simplex')
allocation=result.x.reshape(num_sources,num_destinations)
total_cost=result.fun

return allocation,total_cost
```

PROBLEM 1:

SOLVE THE IFBS FOR THE PROBLEM:

	D1	D2	D3	SUPPLY
	4	8	8	50
	2	7	6	40
	3	4	2	60
DEMAND	30	70	50	

CODE:

```
supply = [50,40,60]
demand = [30,70,50]
costs = np.array([[4,2,3], [8,7,4], [8,6,2]])

allocation, total_cost = transportation_problem(supply, demand, costs)

print("Optimal Allocation:")
print(allocation)
print("Total Cost:", total_cost)
```

OUTPUT:

```
Optimal Allocation:
[[ 0. 50. 0.]
[30. 10. 0.]
[ 0. 10. 50.]]
Total Cost: 570.0
```

PROBLEM 2:

SOLVE THE IFBS FOR THE PROBLEM:

	D1	D2	D3	D4	SUPPLY
S1	19	30	50	10	7
S2	70	30	40	60	9
S3	40	8	70	20	10
DEMAND	5	8	7	14	

CODE:

```
supply = [7,9,10]
demand = [5,8,7,14]
costs = np.array([[19,70,40], [30,30,8],[50,40,70],[10,60,20]])

allocation, total_cost = transportation_problem(supply, demand, costs)

print("Optimal Allocation:")
print(allocation)
print("Total Cost:", total_cost)
```

OUTPUT:

```
Optimal Allocation:
[[ 0. 0. 0.7.]
[0. 2. 7. 0.]
[0. 6. 0. 4.]
[ 5. 0. 0. 3.]]

Total Cost: 406.0
```

PROBLEM 3:

Consider the following transportation problem involving 3 sources and 3 destinations.

	D1	D2	D3	SUPPLY
S1	20	10	15	200
S2	10	12	9	300
S3	25	30	18	500
DEMAND	200	400	400	

CODE:

```
supply = [200,300,500]
demand = [200,400,400]
costs = np.array([[20,10,25], [10,12,30],[15,9,18]])

allocation, total_cost = transportation_problem(supply, demand, costs)

print("Optimal Allocation:")
print(allocation)
print("Total Cost:", total_cost)
```

OUTPUT:

```
Optimal Allocation:
[[  0. 200.0.]
[200. 100.0.]
[  0. 100. 400.]]
Total Cost: 13300.0
```

PROBLEM 4:

Consider the following transportation problem involving 3 sources and 4 destinations.

	D1	D2	D3	D4	SUPPLY
S1	3	1	7	4	300
S2	2	6	5	9	150
S3	8	3	3	2	500
DEMAND	250	150	400	200	

CODE:

```
supply = [300,150,500]
demand = [250,150,400,200]
costs = np.array([[3,2,8], [1,6,3],[7,5,3],[4,9,2]])

allocation, total_cost = transportation_problem(supply, demand, costs)

print("Optimal Allocation:")
print(allocation)
print("Total Cost:", total_cost)
```

OUTPUT:

Optimal Allocation:
[[100. 150. 0. 50.]
[150.   0. 0 .0.]
[0.   0. 350. 150.]
[0.0. 50.0.]]

PROBLEM 5:

Consider the transportation problem:

	D1	D2	D3	D4	D5	SUPPLY
S1	10	2	16	14	10	300
S2	6	18	12	13	16	500
S3	8	4	14	12	10	725
S4	14	22	20	8	18	375
	350	200	250	150	400	

CODE:

```
supply = [300,500,725,375]
demand = [350,200,250,150,400]
costs = np.array([[10,6,8,14], [2,18,4,22],[16,12,14,20],[14,13,12,8],[10,16,10,18]])

allocation, total_cost = transportation_problem(supply, demand, costs)

print("Optimal Allocation:")
print(allocation)
print("Total Cost:", total_cost)
```

OUTPUT:

```
Optimal Allocation:
[[   0.      0.      0.      0.    300.      0.]
 [350.      0.      0.      0.      0.    150.]
 [   0.    200.    250.    150.    100.     25.]
 [   0.      0.      0.      0.      0.    375.]]
Total Cost: 13200.0
```

RESULT:

Thus the Transportation Problem for minimizing the cost of transporting goods from multiple sources to multiple destinations using Python was executed successfully.

Ex.No:2 Date:	**ASSIGNMENT PROBLEM**

Assignment Problem-Assignment with team of workers-Assignment with task size

AIM:

To understand and solve the assignment problem using python, assignment team of workers.

PROCEDURE:

1. **Input Data:** Define the cost matrix where each element represents the cost of assigning a particular task to a specific worker.

2. **Algorithm:** Use an optimization algorithm like the Hungarian algorithm (or the Munkres algorithm) to find the optimal assignment that minimizes the total cost.

3. **Output:** Display the optimal assignment along with the minimum total cost.

Problem 1: Solve the Assignment problem of 4 jobs and 5 Machines for the following matrix

$$\begin{pmatrix} 10 & 7 & 5 & 13 \\ 11 & 11 & 6 & 15 \\ 4 & 10 & 9 & 11 \\ 2 & 14 & 12 & 10 \\ 8 & 12 & 14 & 7 \end{pmatrix}$$

Code:

```
import numpy as np
from scipy.optimize import linear_sum_assignment

def solve_assignment_problem(cost_matrix, problem_type="balanced"):
if problem_type == "unbalanced":
print("Unbalanced assignment problem detected.")

# Pad the cost matrix with zeros for unbalanced problems

num_workers, num_tasks = cost_matrix.shape
```

```
max_dim = max(num_workers, num_tasks)
padded_cost_matrix = np.zeros((max_dim, max_dim))
padded_cost_matrix[:num_workers, :num_tasks] = cost_matrix
print("padded cost matrix(add dummy rows):")
print(padded_cost_matrix)
cost_matrix = padded_cost_matrix

# Solve the assignment problem
row_ind, col_ind = linear_sum_assignment(cost_matrix)

if problem_type = = "unbalanced":
# Filter out assignments to dummy tasks/workers
valid_assignments = col_ind < cost_matrix.shape[1] # Assuming more tasks than
workers
row_ind = row_ind[valid_assignments]
col_ind = col_ind[valid_assignments]
# Recalculate total cost using the original cost matrix (passed as argument)
# to avoid including dummy costs.
# Use cost_matrix instead of cost_matrix_unbalanced, which is a global variable
total_cost = cost_matrix[row_ind, col_ind].sum()

# Calculate total cost for 'balanced' case (this was missing)
else:
total_cost = cost_matrix[row_ind, col_ind].sum()

return row_ind, col_ind, total_cost

# Example usage with your provided matrix:
cost_matrix = np.array([[10, 11, 4, 2, 8],

[7, 11, 10, 14, 12],
[5, 6, 9, 12, 14],
[13, 15, 11, 10, 7]])

# Calling the correct function name: solve_assignment_problem
row_ind, col_ind, total_cost = solve_assignment_problem(cost_matrix,
problem_type="unbalanced")
print("\nOptimal Assignment:")
for i in range(len(row_ind)):
print(f"Job {row_ind[i]+1} assigned to Machine {col_ind[i]+1}")
print(f"\nMinimum Total Cost: {total_cost}")
```

Output:

Unbalanced assignment problem detected.
Padded cost matrix (added dummy rows):

[[10. 11. 4. 2. 8.]
[7. 11. 10. 14. 12.]
[5. 6. 9. 12. 14.]
[13. 15. 11. 10. 7.]
[0. 0. 0. 0. 0.]]

Optimal Assignment:
Job 1 assigned to Machine 4
Job 2 assigned to Machine 1
Job 3 assigned to Machine 2
Job 4 assigned to Machine 5
Job 5 assigned to Machine 3

Minimum Total Cost: 22.0

2. Solve the Assignment problem of 4 jobs and 4 workers for the following matrix

$$\begin{pmatrix} 1 & 4 & 6 & 3 \\ 9 & 7 & 10 & 9 \\ 4 & 5 & 11 & 7 \\ 8 & 7 & 8 & 5 \end{pmatrix}$$

```
cost_matrix = np.array([ [1, 4, 6, 3], [9, 7, 10, 9], [4, 5, 11, 7], [8, 7, 8, 5] ])

#Calling the correct function name: solve_assignment_problem
row_ind, col_ind, total_cost = solve_assignment_problem(cost_matrix,
problem_type="unbalanced")
print("\nOptimal Assignment:")
for i in range(len(row_ind)):
print(f"Job {row_ind[i]+1} assigned to Machine {col_ind[i]+1}")
print(f"\nMinimum Total Cost: {total_cost}")
```

Output:

Optimal Assignment:
Job 1 assigned to Machine 1
Job 2 assigned to Machine 3
Job 3 assigned to Machine 2
Job 4 assigned to Machine 4

Minimum Total Cost: 21.0

3. Solve the Assignment problem

$$\begin{pmatrix} 30 & 39 & 31 & 38 & 40 \\ 43 & 37 & 32 & 35 & 38 \\ 34 & 41 & 33 & 41 & 34 \\ 39 & 36 & 43 & 32 & 36 \\ 32 & 49 & 35 & 40 & 37 \\ 36 & 42 & 35 & 44 & 42 \end{pmatrix}$$

```
cost_matrix = np.array([ [30, 39, 31, 38, 40], [43, 37, 32, 35, 38], [34, 41, 33, 41, 34], [39,
36, 43, 32, 36], [32, 49, 35, 40, 37], [36, 42, 35, 44, 42] ])

# Calling the correct function name: solve_assignment_problem
row_ind, col_ind, total_cost = solve_assignment_problem(cost_matrix,
problem_type="unbalanced")
print("\nOptimal Assignment:")
for i in range(len(row_ind)):
print(f"Job {row_ind[i]+1} assigned to Machine {col_ind[i]+1}")
print(f"\nMinimum Total Cost: {total_cost}")
```

Output:

```
Unbalanced assignment problem detected.
[[30. 39. 31. 38. 40.  0.]
 [43. 37. 32. 35. 38.  0.]
 [34. 41. 33. 41. 34.  0.]
 [39. 36. 43. 32. 36.  0.]
 [32. 49. 35. 40. 37.  0.]
 [36. 42. 35. 44. 42.  0.]]

Optimal Assignment:
Job 1 assigned to Machine 3
Job 2 assigned to Machine 2
Job 3 assigned to Machine 5
Job 4 assigned to Machine 4
Job 5 assigned to Machine 1
Job 6 assigned to Machine 6

Minimum Total Cost: 166.0
```

4. Consider the problem of assigning 4 sales persons to 4 different sales regions as shown below. Find the optimal allocation

$$\begin{pmatrix} 5 & 11 & 8 & 9 \\ 5 & 7 & 9 & 7 \\ 7 & 8 & 9 & 9 \\ 6 & 8 & 11 & 12 \end{pmatrix}$$

```
cost_matrix= np.array([ [5, 11, 8, 9], [5, 7, 9, 7], [7, 8, 9, 9], [6, 8, 11, 12] ])
```

```
# Calling the correct function name: solve_assignment_problem
row_ind, col_ind, total_cost = solve_assignment_problem(cost_matrix,
problem_type="unbalanced")
print("\nOptimal Assignment:")
for i in range(len(row_ind)):
print(f"Job {row_ind[i]+1} assigned to Machine {col_ind[i]+1}")
print(f"\nMinimum Total Cost: {total_cost}")
```

Output:

Optimal Assignment:
Job 1 assigned to Machine 1
Job 2 assigned to Machine 4
Job 3 assigned to Machine 3
Job 4 assigned to Machine 2

Minimum Total Cost: 29.0

5. solve the assignment problem. The cell entries represent the processing time of the job i if it is assigned to the operator j

$$\begin{pmatrix} 13 & 5 & 8 & 10 \\ 9 & 15 & 18 & 10 \\ 12 & 14 & 10 & 10 \\ 10 & 14 & 9 & 12 \end{pmatrix}$$

```
cost_matrix = np.array([ [13, 5, 8, 10], [9, 15, 18, 10], [12, 14, 10, 10], [10, 14, 9, 12] ])
# Calling the correct function name: solve_assignment_problem
row_ind, col_ind, total_cost = solve_assignment_problem(cost_matrix,
problem_type="unbalanced")
print("\nOptimal Assignment:")
for i in range(len(row_ind)):
print(f"Job {row_ind[i]+1} assigned to Machine {col_ind[i]+1}")
print(f"\nMinimum Total Cost: {total_cost}")
```

Output:

Optimal Assignment:
Job 1 assigned to Machine 2
Job 2 assigned to Machine 1
Job 3 assigned to Machine 4
Job 4 assigned to Machine 3

Minimum Total Cost: 33.0

RESULT:

Thus understanding and solving of the assignment problem using python, assignment team of workers

Ex. No: 3 Date:	CRITICAL PATH METHOD - ANALYSIS

AIM:

To perform Critical Path Method (CPM) Analysis for project scheduling and management using Python.

PROCEDURE:

1. **Install Required Libraries:**
    - Make sure you have pandas, networkx, and matplotlib installed. You can install them using pip if not already installed.

      bash

      Copy the code:

      pip install pandas networkx matplotlib

2. **Define the Problem:**
    - Identify the tasks, their durations, and dependencies.
3. **Set Up the Problem in Python:**
    - Use NetworkX to represent the project tasks and their dependencies.
    - Use the topological sort to identify the critical path.
4. **Run the Code:**
    - Execute the Python code to get the critical path and project duration.
5. **Analyze the Output:**
    - Interpret the results and validate the solution against the project schedule.

MAIN CODE:

```
import networkx as nx
import matplotlib.pyplot as plt
```

```
class Task:
  def __init__(self,name,duration):
    self.name = name
    self.duration = duration
    self.early_start = 0
    self.early_finish = 0
    self.late_start = float('inf')
    self.late_finish = float('inf')
    self.successors = []

class CPM:
  def __init__(self,tasks,dependencies):
    self.tasks = {name:Task(name,duration)for name, duration in tasks.items()}
    self.dependencies = dependencies
    self.build_graph()

  def build_graph(self):
    for task, deps in self.dependencies.items():
        for dep in deps:
          self.tasks[dep].successors.append(self.tasks[task])

  def forward_pass(self):
    for task in self.tasks.values():
      if not self.dependencies[task.name]:
        task.early_start = 0
        task.early_finish = task.duration
```

```
for task in sorted(self.tasks.values(), key=lambda t: t.early_start):
      for succ in task.successors:
        succ.early_start = max(succ.early_start, task.early_finish)
        succ.early_finish = succ.early_start + succ.duration

  def backward_pass(self):
    max_finish = max(task.early_finish for task in self.tasks.values())
    for task in self.tasks.values():
      if not task.successors:
        task.late_finish = max_finish
        task.late_start = max_finish - task.duration

    for task in sorted(self.tasks.values(), key=lambda t: -t.early_finish):
      for succ in task.successors:
        task.late_finish = min(task.late_finish, succ.late_start)
        task.late_start = task.late_finish - task.duration

  def find_critical_path(self):
    return [task.name for task in self.tasks.values() if task.early_start ==
task.late_start]

  def run(self):
    self.forward_pass()
    self.backward_pass()
    return self.find_critical_path()
```

```
def visualize_network(self):
    G = nx.DiGraph()
    for task in self.tasks.values():
      G.add_node(task.name, label=f"{task.name}\nES:{task.early_start},
EF:{task.early_finish}\nLS:{task.late_start}, Lf:{task.late_finish}")
      for succ in task.successors:
        G.add_edge(task.name,succ.name)
    pos = nx.spring_layout(G)
    labels = {node:data['label'] for node,data in G.nodes(data=True)}
    plt.figure(figsize = (10,6))
    nx.draw(G, pos, node_size = 3000, node_color = 'lightblue',edge_color =
'gray',font_size = 10)
    nx.draw_networkx_labels(G, pos, labels = labels, font_size = 8)
    plt.title("Project Network Diagram")
    plt.show()
```

Problem: Find the critical path for the following activities with duration (in days):

Activities	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Immediate Predecessor	**-**	**A**	**A**	**B,C**	**C**	**D,E**	**F**
Duration	**3**	**2**	**4**	**2**	**3**	**1**	**2**

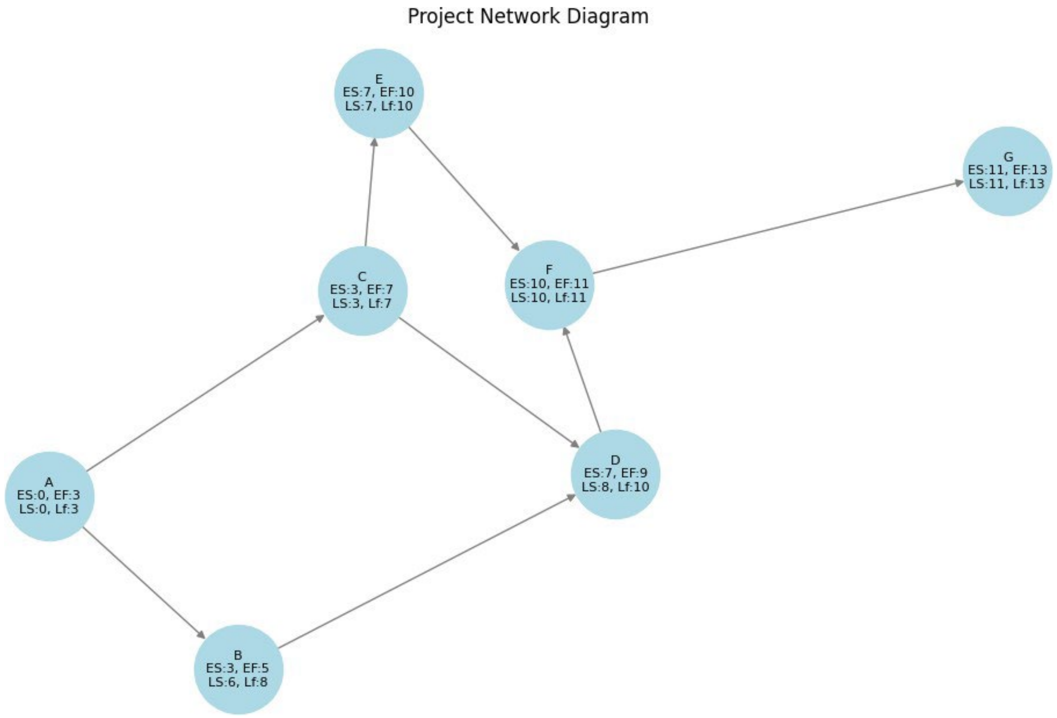
CODE:

```
if __name__== "__main__":
  tasks = {
      'A': 3,'B': 2,'C': 4,'D': 2,'E': 3,'F': 1,'G': 2
      }
```

```
dependencies = {
    'A': [ ],'B': ['A'],'C': ['A'],'D': ['B', 'C'],'E': ['C'],'F': ['D', 'E'],'G': ['F']
    }
cpm = CPM(tasks, dependencies)
critical_path = cpm.run()
print("Critical Path:", " -> ".join(critical_path))
cpm.visualize_network()
```

OUTPUT:

Critical Path: A -> C -> E -> F -> G



Ex. No: 4 Date:	PROJECT EVALUATION AND REVIEW TECHNIQUES - ANALYSIS

AIM:

To perform Program Evaluation Review Technique (PERT) Analysis for project scheduling and management using Python.

PROCEDURE:

1. **Install Required Libraries:**
   - Make sure you have pandas, networkx, matplotlib, and numpy installed. You can install them using pip if not already installed.

     bash

     Copy the code

     pip install pandas networkx matplotlib numpy
2. **Define the Problem:**
   - Identify the tasks, their optimistic, most likely, and pessimistic durations, and dependencies.
3. **Set Up the Problem in Python:**
   - Use NetworkX to represent the project tasks and their dependencies.
   - Calculate the expected duration and variance for each task.
   - Use topological sort to identify the critical path and perform PERT analysis.
4. **Run the Code:**
   - Execute the Python code to get the critical path, project duration, and associated uncertainty.
5. **Analyze the Output:**
   - Interpret the results and validate the solution against the project schedule.

MAIN CODE:

```
import networkx as nx
import matplotlib.pyplot as plt

class Task:
  def __init__(self, name, optimistic, most_likely, pessimistic):
    self.name = name
    self.optimistic = optimistic
    self.most_likely = most_likely
    self.pessimistic = pessimistic
    self.duration = (optimistic + 4 * most_likely + pessimistic) / 6 # PERT
formula
    self.early_start = 0
    self.early_finish = 0
    self.late_start = float('inf')
    self.late_finish = float('inf')
    self.successors = []

class PERT:
  def __init__(self, tasks, dependencies):
    self.tasks = {name: Task(name, *durations) for name, durations in
tasks.items()}
    self.dependencies = dependencies
    self.build_graph()

  def build_graph(self):
    for task, deps in self.dependencies.items():
      for dep in deps:
```

```
        self.tasks[dep].successors.append(self.tasks[task])

  def forward_pass(self):
    for task in self.tasks.values():
      if not self.dependencies[task.name]:
        task.early_start = 0
        task.early_finish = task.duration

    for task in sorted(self.tasks.values(), key=lambda t: t.early_start):
      for succ in task.successors:
        succ.early_start = max(succ.early_start, task.early_finish)
        succ.early_finish = succ.early_start + succ.duration

  def backward_pass(self):
    max_finish = max(task.early_finish for task in self.tasks.values())
    for task in self.tasks.values():
      if not task.successors:
        task.late_finish = max_finish
        task.late_start = task.late_finish - task.duration

    for task in sorted(self.tasks.values(), key=lambda t: -t.early_finish):
      for succ in task.successors:
        task.late_finish = min(task.late_finish, succ.late_start)
        task.late_start = task.late_finish - task.duration

  def find_critical_path(self):
    return [task.name for task in self.tasks.values() if task.early_start ==
task.late_start]
```

```
  def run(self):
    self.forward_pass()
    self.backward_pass()
    return self.find_critical_path()

  def visualize_network(self):
    G = nx.DiGraph()
    for task in self.tasks.values():
      G.add_node(task.name, label=f"{task.name}\nES: {task.early_start},
EF:{task.early_finish}\nLS: {task.late_start}, LF: {task.late_finish}")
      for succ in task.successors:
        G.add_edge(task.name, succ.name)
    pos = nx.spring_layout(G)
    labels = {node: data['label'] for node, data in G.nodes(data=True)}
    plt.figure(figsize=(12, 7))
    nx.draw(G, pos, with_labels=False, node_size = 3000, node_color =
'lightblue',edge_color = 'gray', font_size = 10)
    nx.draw_networkx_labels(G, pos, labels=labels, font_size=8)
    plt.title("PERT Network Diagram")
    plt.show()
```

Problem: Consider the table with details of project involving 7 activities:

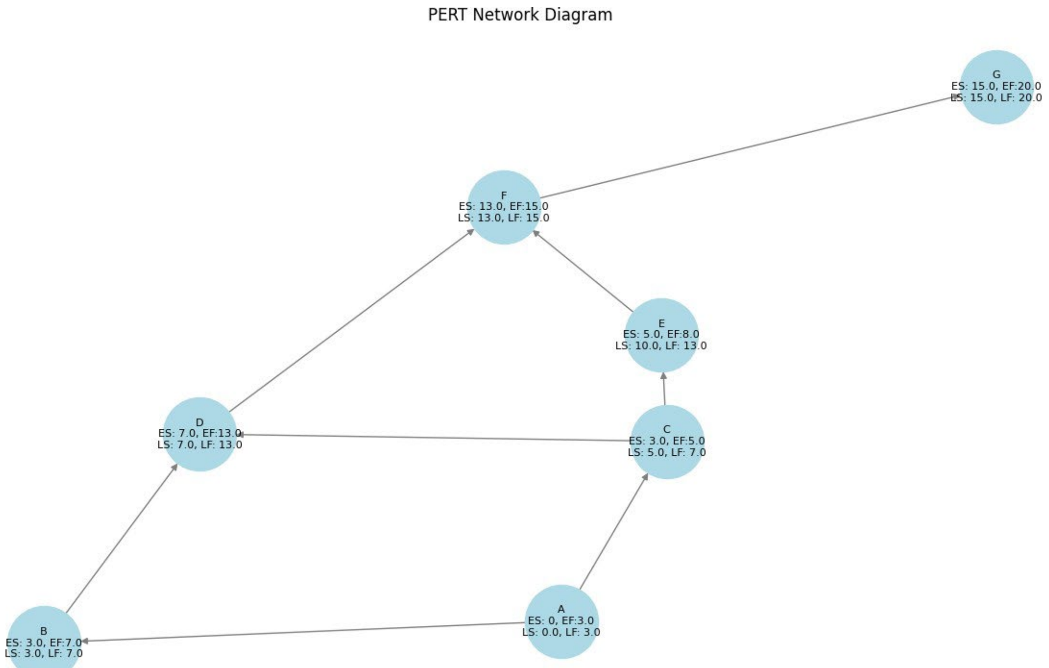
Activities	**A**	**B**	**C**	**D**	**E**	**F**	**G**
Immediate Predecessor	**-**	**A**	**A**	**B,C**	**C**	**D,E**	**F**
a	**1**	**2**	**1**	**3**	**2**	**1**	**2**
m	**3**	**4**	**2**	**6**	**3**	**2**	**5**
b	**5**	**6**	**3**	**9**	**4**	**3**	**8**

CODE:

```
if __name__ == "__main__":
  tasks = {
      'A': (1, 3, 5), # Optimistic, Most Likely, Pessimistic
      'B': (2, 4, 6),
      'C': (1, 2, 3),
      'D': (3, 6, 9),
      'E': (2, 3, 4),
      'F': (1, 2, 3),
      'G': (2, 5, 8),
      }
  dependencies = {
      'A': [], 'B': ['A'], 'C': ['A'], 'D': ['B', 'C'], 'E': ['C'], 'F': ['D', 'E'], 'G': ['F']
      }
  pert = PERT(tasks, dependencies)
  critical_path = pert.run()
  print("Critical Path:", " -> ".join(critical_path))
  pert.visualize_network()
```

OUTPUT:

Critical Path: A -> B -> D -> F -> G



Ex. No: 5.a Date:	LINEAR PROGRAMMING PROBLEM - CONSTRAINT OPTIMIZATION

AIM:

To solve a Linear Programming Problem (LPP) for constraint optimization using Python.

PROCEDURE:

1. **Install Required Libraries:**
   - Make sure you have numpy and scipy installed. You can install them using pip if not already installed.

     bash

     Copy the code

     pip install numpy scipy
2. **Define the Problem:**
   - Identify the objective function to maximize or minimize.
   - Determine the constraints (equality and inequality).
3. **Set Up the Problem in Python:**
   - Use the scipy.optimize.linprog function to set up and solve the LPP.
4. **Run the Code:**
   - Execute the Python code to get the optimized result.
5. **Analyze the Output:**
   - Interpret the results and validate the solution against the problem constraints.

Problem - 1:

Solve the following Linear Programming Problem

Maximize $z = 3x_1 + 2x_2$

Subject to Constraints: $2x_1 + x_2 \leq 20$,

$4x_1 - 5x_2 \leq 10$,

$x_1 \geq 0, x_2 \geq 0$

CODE:

```
# Define the coefficients of the objective function
# For example, maximize: z = 3x1 + 2x2
c = [-3,-2]
# Define the coefficients of the inequality constraints
# For example:
# 2x1 + x2 <= 20
# 4x1 - 5x2 <= 10
# x1, x2 >= 0 (non-negativity constraints)
A = [
    [2,1],
    [4,-5]
]
# Define the right-hand side of the inequality constraints
b = [20, 10]
# Define bounds for variables (non-negativity constraints)
x_bounds = (0, None)
y_bounds = (0, None)
# Solve the linear programming problem
result = linprog(c, A_ub=A, b_ub=b, bounds=[x_bounds, y_bounds], method = 'highs')
# Print the result
print('Optimal value:',-result.fun)
print('Values of x:',result.x)
```

OUTPUT:

Optimal value: 40.0

Values of x: [0. 20.]

Problem - 2:

Solve the following Linear Programming Problem

Maximize $z = 3x + 5y$

Subject to Constraints: $2x + 3y \leq 12$,

$x + y \leq 5$,

$x \geq 0, y \geq 0$

CODE:

```
c = [-3,-5]
A = [
    [2,3],
    [1,1]
]
b = [12, 5]
x_bounds = (0, None)
y_bounds = (0, None)
result = linprog(c, A_ub=A, b_ub=b, bounds=[x_bounds, y_bounds], method = 'highs')
print('Optimal value:',-result.fun)
print('Values of x:',result.x)
```

OUTPUT:

Optimal value: 20.0

Values of x: [0. 4.]

Problem – 3:

Solve the following Linear Programming Problem

Minimize $z = 20x + 10y$

Subject to Constraints: $x + 2y \leq 40$,

$3x + y \geq 30$,

$4x + 3y \geq 60$,

$x \geq 0, y \geq 0$

CODE:

```
c = [20,10]
A = [
    [1,2],
    [-3,-1],
    [-4,-3]
]
b = [40, -30, -60]
x_bounds = (0, None)
y_bounds = (0, None)
result = linprog(c, A_ub=A, b_ub=b, bounds=[x_bounds, y_bounds], method = 'highs')
print('Optimal value:',result.fun)
print('Values of x:',result.x)
```

OUTPUT:

```
Optimal value: 240.0
Values of x: [ 6. 12.]
```

RESULT:

Thus, the **Linear Programming Problem (**LPP) by **Constraint Optimization** using Python program has been implemented successfully.

Ex. No: 6 Date:	INTEGER PROGRAMMING PROBLEM- BRANCH AND BOUND METHOD

AIM:

To solve any integer programming problem using the Branch and Bound method in Python.

PROCEDURE;

1. Formulate the integer programming problem.
2. Implement the Branch and Bound method in Python
3. Solve the problem using the implemented method.
4. Verify the solution.

MAIN CODE:

```
import numpy as np
from scipy.optimize import linprog
from queue import Queue

#Function to solve the linear programming relaxation
def solve_lp(c,A,b,bounds):
  res = linprog(c,A_ub=A,b_ub=b,bounds=bounds,method='highs')
  return res

#Branch and Bound method
def branch_and_bound(c,A,b,bounds):
  Q = Queue()
  Q.put((c,A,b,bounds))
  best_solution = None
  best_value = float('-inf')

  while not Q.empty():
    current_problem = Q.get()
    res = solve_lp(*current_problem)
```

```
    if res.success and -res.fun > best_value:
      solution = res.x
      if all(np.isclose(solution,np.round(solution))):
        value = -res.fun #Objective Function Value
        if value > best_value:
          best_value = value
          best_solution = solution
      else:
        #branching
        for i in range(len(solution)):
          if not np.isclose(solution[i],np.round(solution[i])):
            lower_bound= current_problem[3].copy()
            upper_bound= current_problem[3].copy()

            lower_bound[i]=(lower_bound[i][0],np.floor(solution[i]))
            upper_bound[i]=(np.ceil(solution[i]),upper_bound[i][1])

            Q.put((current_problem[0],current_problem[1],current_problem[2],lower_bound))
            Q.put((current_problem[0],current_problem[1],current_problem[2],upper_bound))
            break

  return best_solution,best_value

# Example usage
c = [-4, -3] #Coefficients for the objective function (maximize)
A = [[2, 1], [1, 2]] #Coefficients for the constraints
b = [8, 6] #RHS values for the constraints
bounds = [(0, None), (0, None)] #Bounds for the variables

#Solve the integer programming problem
solution, value = branch_and_bound(c, A, b, bounds)
print(f"Optimal solution: {solution}")
```

```
print(f"Optimal value: {value}")
```

OUTPUT:

Optimal solution: [4. -0.]

Optimal value: 16.0

Problem – 1:

Solve the Integer Programming Problem using Branch and Bound method:

Max Z = 3x + 5y;

Subject to 2x + 4y ≤ 25;

x ≤ 8;

2y ≤ 10;

x, y ≥ 0

CODE:

```
# Problem - 1:
c = [-3, -5] # Coefficients for the objective function (maximize)
A = [[2, 4], [1, 0],[0, 2]] # Coefficients for the constraints
b = [25, 8, 10] # RHS values for the constraints
bounds = [(0, None), (0, None)] # Bounds for the variables

# Solve the integer programming problem
solution, value = branch_and_bound(c, A, b, bounds)
print(f"Optimal solution: {solution}")
print(f"Optimal value: {value}")
```

OUTPUT:

Optimal solution: [8. 2.]

Optimal value: 34.0

Problem – 2:

Solve the Integer Programming Problem using Branch and Bound method:

Max Z = 7x + 9y;

Subject to $-x + 3y \leq 6$;

$7x + y \leq 35$;

$y \leq 7$;

$x, y \geq 0$.

CODE:

```
# Problem - 2:
c = [-7, -9] # Coefficients for the objective function (maximize)
A = [[-1, 3], [7, 1],[0, 1]] # Coefficients for the constraints
b = [6, 35, 7] # RHS values for the constraints
bounds = [(0, None), (0, None)] # Bounds for the variables

# Solve the integer programming problem
solution, value = branch_and_bound(c, A, b, bounds)
print(f"Optimal solution: {solution}")
print(f"Optimal value: {value}")
```

OUTPUT:

```
Optimal solution: [4. 3.]
Optimal value: 55.0
```

Problem – 3:

Solve the Integer Programming Problem using Branch and Bound method:

Min Z = 5x + 4y;

Subject to $3x + 2y \geq 5$;

$2x + 3y \leq 7$;

$x, y \geq 0$.

CODE:

```
# Problem - 3:
c = [5, 4] # Coefficients for the objective function (minimize)
A = [[-3, 2], [2, 3]] # Coefficients for the constraints
b = [-5, 7] # RHS values for the constraints
bounds = [(0, None), (0, None)] # Bounds for the variables

# Solve the integer programming problem
solution, value = branch_and_bound(c, A, b, bounds)
print(f"Optimal solution: {solution}")
print(f"Optimal value: {value}")
```

OUTPUT:

```
Optimal solution: [2. 0.]
Optimal value: -10.0
```

RESULT:

Thus, solving any **Integer Programming Problem** using the **Branch and Bound** method in Python has been implemented and executed successfully.

Ex. No: 7 Date:	DYNAMIC PROGRAMMING – KNAPSACK PROBLEM, SUBSET SUM PROBLEM, LONGEST COMMON SUBSEQUENCE PROBLEMS

AIM:

1. To maximize the total value of items included in a knapsack without exceeding its capacity, given weights and values of the items
2. To determine if a subset of a given set of integers sums up to a specified value.
3. To find the length of the longest subsequence common to two sequences, where the subsequence maintains the order of elements.

PROCEDURE:

1. Knapsack Problem:

To maximize the total value of items included in a knapsack without exceeding its capacity, given weights and values of the items

MAIN CODE:

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

  for i in range(n+1):
    for w in range(W + 1):
      if i==0 or w==0:
        dp[i][w] = 0
      elif weights[i-1] <= w:
        dp[i][w] = max(values[i-1] + dp[i-1][w-weights[i-1]], dp[i-1][w])
      else:
        dp[i][w] = dp[i -1][w]
  return dp[n][w]
```

Problem - 1:

Using dynamic programming, Find the maximum value in the knapsack with capacity of

W=7 for 4 objects.

Weights	1	3	4	5
Values	**1**	**4**	**5**	**7**

CODE:

```
# Example Usage
weights = [1, 3, 4, 5]
values = [1, 4, 5, 7]
W = 7
max_value = knapsack(weights, values, W)
print("Maximum value in Knapsack:", max_value)
```

OUTPUT:

Maximum value in Knapsack: 9

Problem – 2:

Using dynamic programming, Find the maximum value in the knapsack with capacity of W=5 for 4 objects.

Weights	2	3	4	5
Values	**3**	**4**	**5**	**6**

CODE:

```
# Problem 2
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
W = 5
max_value = knapsack(weights, values, W)
print("Maximum value in Knapsack:", max_value)
```

OUTPUT:

Maximum value in Knapsack: 7

Problem – 3:

Using dynamic programming, Find the maximum value in the knapsack with capacity of W=15 for 7 objects.

Weights	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Values	**3**	**4**	**8**	**8**	**10**	**13**	**15**

CODE:

```
# Problem 3
weights = [2, 3, 4, 5, 6, 7, 8]
values = [3, 4, 8, 8, 10, 13, 15]
W = 15
max_value = knapsack(weights, values, W)
print("Maximum value in Knapsack:", max_value)
```

OUTPUT:

Maximum value in Knapsack: 28

Problem – 4:

Using dynamic programming, Find the maximum value in the knapsack with capacity of W=15 for 10 objects.

Weights	**1**		**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Values	**10**		**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**

CODE:

```
# Problem 4
weights = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
values = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
W = 15
max_value = knapsack(weights, values, W)
print("Maximum value in Knapsack:", max_value)
```

OUTPUT:

Maximum value in Knapsack: 40

2. Subset Sum Problem:

To determine if there is a subset of the given set with a sum equal to a given sum.

MAIN CODE:

```
def is_subset_sum(nums, target):
  n = len(nums)
  dp = [[False]*(target + 1) for _ in range(n + 1)]

  # A sum of 0 can always be made with an empty subset
  for i in range(1, n + 1):
    dp[i][0] = True

  # Fill the dp table
  for i in range(1, n + 1):
    for j in range(1, target + 1):
      if nums[i - 1] > j:
        dp[i][j] = dp[i - 1][j]
      else:
        dp[i][j] = dp[i - 1][j] or dp[i - 1][j - nums[i - 1]]

  return dp[n][target]
```

Problem – 1:

To check whether the target sum=9 is a subset of the given numbers

3, 34, 4, 12, 5, 2.

CODE:

```
#Example usage
numbers = [3, 34, 4, 12, 5, 2]
```

```
target_sum = 9
print(is_subset_sum(numbers, target_sum))
```

OUTPUT:

True

Problem – 2:

To check whether the target sum=6 is a subset of the given numbers
1, 2, 3, 7.

CODE:

```
# Problem 2
numbers = [1, 2, 3, 7]
target_sum = 6
print(is_subset_sum(numbers, target_sum))
```

OUTPUT:

True

Problem – 3:

To check whether the target sum=4 is a subset of the given numbers
1, 2, 5.

CODE:

```
# Problem 3
numbers = [1, 2, 5]
target_sum = 4
print(is_subset_sum(numbers, target_sum))
```

OUTPUT:

False

Problem – 4:

To check whether the target sum=8 is a subset of the given numbers
3, 5, 9, 12.

CODE:

```
# Problem 4
numbers = [3, 5, 9, 12]
target_sum = 8
print(is_subset_sum(numbers, target_sum))
```

OUTPUT:

False

3. Longest Common Subsequence Problem:

To find the length of the longest subsequence common to two sequences, where the subsequence maintains the order of elements.

MAIN CODE:

```
def longest_common_subsequence(X, Y):
  m = len(X)
  n = len(Y)
  dp = [[0] * (n + 1) for _  in range(m + 1)]
for i in range(1, m + 1):
    for j in range(1, n + 1):
      if X[i - 1] == Y[j - 1]:
        dp[i][j] = dp[i - 1][j - 1] + 1
      else:
        dp[i][j] = max(dp[i-1][j], dp[i][j - 1])
  return dp[m][n]
```

Problem – 1:

Find the length of the sequences given

X: AGGTAB

Y: GXTXAYB

CODE:

```
#Example Usage
X = "AGGTAB"
Y = "GXTXAYB"
print(longest_common_subsequence(X, Y))
```

OUTPUT:

4

Problem – 2:

Find the length of the sequences given

X: ABCBDAB

Y: BDCAB

CODE:

```
# Problem 2
X = "ABCBDAB"
Y = "BDCAB"
print(longest_common_subsequence(X, Y))
```

OUTPUT:

4

Problem – 3:

Find the length of the sequences given

X: ABC

Y: AC

CODE:

```
# Problem 3
X = "ABC"
Y = "AC"
print(longest_common_subsequence(X, Y))
```

OUTPUT:

```
2
```

Problem – 4:

Find the length of the sequences given

X = AXYT

Y = AYZX

CODE:

```
# Problem 4
X = "AXYT"
Y = "AYZX"
print(longest_common_subsequence(X, Y))
```

OUTPUT:

```
2
```

RESULT:

Thus, solving Knapsack Problem, Subset Sum Problem and Longest Common Subsequence Problem using Dynamic Programming in Python has been implemented and executed successfully.

Ex. No: 8 Date:	GRADIENT DESCENT METHOD- STOCHASTIC GRADIENT DESCENT ALGORITHM

AIM:

To iteratively adjust the parameters of a model in order to minimize the error between the predicted and actual values by using the Stochastic Gradient Descent (SGD) algorithm. SGD updates the model parameters based on each training example, allowing for faster convergence and handling larger datasets efficiently compared to batch gradient descent.

PROCEDURE:

1. Initialization:

1. We initialize the weights to zeros.
2. X is the feature matrix, and y is the target vector.
3. Learning rate controls the step size of each update.
4. Epochs are the number of iterations over the entire dataset.

2. Gradient Computation:

1. For each sample in the dataset, we compute the gradient of the loss with respect to the weights.
2. The gradient is calculated as (prediction - target) * feature.

Weights Update:

1. We update the weights by moving them in the opposite direction of the gradient, scaled by the learning rate.

MAIN CODE:

```
import numpy as np
def stochastic_gradient_descent(X, y, learning_rate = 0.01, epochs = 1000):
  """
  Perform Stochastic Gradient Descent to learn the weights for linear regression.

  Parameters:
  X: numpy array, shape (n_samples, n_features)
```

```
      Training Data.
    y: numpy array, shape (n_samples,)
      Target values.
    learning_rate: float
    The Learning rate
    epochs: int
      Number of iterations over the training data

    Returns:
    weights: numpy array, shape (n_features,)
      The learned weights
    """
    n_samples, n_features = X.shape
    weights = np.zeros(n_features)

    for epoch in range(epochs):
      for i in range(n_features):
        gradient = (np.dot(X[i], weights) - y[i]) * X[i]
        weights -= learning_rate * gradient

    return weights
```

Problem - 1:

CODE:

```
#Problem - 1
if __name__ == "__main__":
  #Generating some example data
  np.random.rand(0)
  X = 2 * np.random.randn(100, 1)
  y = 4 + 3 * X + np.random.randn(100, 1)
  X_b = np.c_[np.ones((100, 1)), X] #Add x0 = 1 to each instance

  # Reshape y
```

```
  y = y.ravel()

  # Parameters
  learning_rate = 0.01
  epochs = 1000

  # Running Stocastic Gradient Descent
  weights = stochastic_gradient_descent(X_b, y, learning_rate, epochs)
  print(f"Weights: {weights}")
```

OUTPUT:

Weights: [3.26844642 3.13726567]

Problem – 2:

CODE:

```
#Problem - 2
if __name__ == "__main__":
  # Example - 1: Linear Regression
  np.random.rand(0)
  X = 2 * np.random.rand(100, 1)
  y = 5 + 2 * X + np.random.randn(100, 1)
  X_b = np.c_[np.ones((100, 1)), X] #Add x0 = 1 to each instance

  y = y.ravel()

  learning_rate = 0.01
  epochs = 1000

  weights = stochastic_gradient_descent(X_b, y, learning_rate, epochs)
  print(f"Linear Regression Weights: {weights}")
```

OUTPUT:

Linear Regression Weights: [2.02389407 3.39009053]

Problem – 3:

CODE:

```
#Problem - 3
# Example - 2: Quadratic Function Minimization
def quadratic_loss(x):
  return x**2 + 2*x + 1 # Simple Quadratic Function

def quadratic_gradient(x): # Gradient of the function
  return 2*x + 2

x = np.random.randn() # Random initial point
lr = 0.1
for _ in range(50): #Iterate to minimize
  x -= lr * quadratic_gradient(x)
print("Minimum of quadratic function:", x)
```

OUTPUT:

Minimum of quadratic function: -0.999991101321883

Problem – 4:

CODE:

```
#Problem - 4
# Example - 2: Quadratic Function Minimization
def quadratic_loss(x):
  return (x-3)**2 # Simple Quadratic Function

def quadratic_gradient(x): # Gradient of the function
  return 2*(x - 2)
```

```
x = np.random.randn() # Random initial point
lr = 0.1
for _ in range(50): #Iterate to minimize
  x -= lr * quadratic_gradient(x)
print("Minimum of quadratic function:", x)
```

OUTPUT:

Minimum of quadratic function: 1.9999523866369255

RESULT:

Thus, iteratively adjusting the parameters of a model in order to minimize the error between the predicted and actual values by using the Stochastic Gradient Descent (SGD) algorithm has been implemented and executed successfully.

EXP 9 Date:	UNCONSTRAINED OPTIMIZATION- NONLINEAR LEAST SQUARES

AIM:

The objective of this lab is to understand and implement optimization techniques for solving nonlinear least squares problems without constraints.

PROCEDURE:

a. **Problem Formulation:**
- Define the nonlinear least squares objective function f(x).
- Specify the data points and the model that relates to the data.

b. **Choose an Optimization Library:**
- Select an appropriate optimization library (e.g., scipy.optimize in Python).

c. **Implementing the Optimization:**
- Set up the objective function to be minimized.
- Define any constraints (if the problem extends to constrained NLS).

d. **Running Optimization:**
- Execute the chosen optimization algorithm.
- Monitor convergence and iterate as necessary.

e. **Post-optimization Analysis:**
- Evaluate and interpret the results obtained from the optimization.
- Compare with initial assumptions and data.

CODE:

1. Nonlinear least squares

```
import numpy as np
from scipy.optimize import least_squares
def model(X,t):
  return X[0]*np.exp(-X[1]*t)
def residual(X,t,y):
  return  model(X,t)-y
np.random.seed(123)
t=np.linspace(0,1,50)
y_true=[2.0,0.5]
y=model(y_true,t)+0.1*np.random.randn(50)
x0=[1.0,1.0]
result=least_squares(residual,x0,args=(t,y))
print("optimized parameters:",result.x)
```

Output:

optimized parameters: [1.98254019 0.47917343]

2. Nonlinear least squares for fitting a logistic growth model to data

```
import numpy as np
from scipy.optimize import least_squares
def model(X,t):
  return X[0]/(1+np.exp(-X[1]*(t-X[2])))
def residual(X,t,y):
  return model(X,t)-y
np.random.seed(456)
t=np.linspace(0,5,100)
y_true=[5.0,0.8,3.0]
y=model(y_true,t)+0.1*np.random.randn(100)
x0=[1.0,1.0,1.0]
result=least_squares(residual,x0,args=(t,y))
print("optimized parameters:",result.x)
```

Output:

```
optimized parameters: [5.19893281 0.77489624 3.10139266]
```

3. Nonlinear least squares for fitting a sum of sine waves to data

```
import numpy as np
from scipy.optimize import least_squares
def model(X,t):
  return X[0]*np.sin(X[1]*t + X[2])+X[3]*np.sin(X[4]*t+X[5])
def residual(X,t,y):
  return model(X,t)-y
np.random.seed(789)
t=np.linspace(0,10,200)
y_true=[2.0,0.5,1.0,1.0,2.0,2.5]
y=model(y_true,t)+0.1*np.random.randn(200)
x0=[1.0,1.0,1.0,1.0,1.0,1.0]
result=least_squares(residual,x0,args=(t,y))
print("optimized parameters:",result.x)
```

Output:

```
optimized parameters: [0.97587131 0.54470271 0.77416988 0.97586095 0.54475034
0.77408607]
```

RESULT:

Thus, solving any **Unconstrained Optimization** using the **Non-Linear Least Squares Algorithm** in python has been implemented and executed successfully

EXP 10 Date:	KUHN-TUCKER CONDITIONS - LAGRANGIAN MULTIPLIER METHOD

AIM:

The objective of this lab is to introduce and implement the Kuhn-Tucker conditions for constrained optimization using Python. Participants will learn the theoretical basis of the Kuhn-Tucker conditions and apply them to solve optimization problems with inequality constraints.

PROCEDURE:

a. Objective Function and Constraints:

- Define the objective function and inequality constraints in Python.

b. Kuhn-Tucker Conditions:

- Implement the Kuhn-Tucker conditions using Python functions.
- Compute the gradients of the objective function and constraints.

c. Optimization Setup:

- Use sympy for symbolic mathematics operation or another suitable optimization function.
- Incorporate the Kuhn-Tucker conditions as additional constraints or through custom callback functions.

d. Running the Optimization:

- Execute the optimization procedure and monitor convergence.
- Validate results against theoretical expectations.

CODE:

Problem 1:

Minimize : $f(x_1, x_2) = {x_1}^2 + {x_2}^2$

Subject to the constraints:

$$g(x) = x_1 + x_2 \leq 1$$

$$x_1, x_2 \geq 0$$

```
import sympy as sp
X1,X2=sp.symbols('X1 X2',real=True)
l1=sp.symbols('l1',real=True)
f=X1**2 + X2**2
g1=X1 + X2
L=f+l1*g1
grad_L=[sp.diff(L,var) for var in [X1,X2]]
kkt_eqs=[
    grad_L[0],
    grad_L[1],
    l1*g1
]
solutions=sp.solve(kkt_eqs,[X1,X2,l1],dict=True)
feasible=[]
for sol in solutions:
  g1_val=g1.subs(sol)
  l1_val=sol[l1]
  if g1_val <=0 and l1_val >=0:
    feasible.append(sol)
if feasible:
  for i,sol in enumerate(feasible):
    print(f"\n✅Solution {i+1}:")
    x1_val=sol[X1]
    x2_val=sol[X2]
    l1_val=sol[l1]
    print(f" Optimal point:x1={x1_val},x2={x2_val}")
    print(f" Lagrange multiplier λ={l1_val}")
    grad_f=[sp.diff(f,var) for var in [X1,X2]]
    grad_f_val=[g.subs({X1:x1_val,X2:x2_val}) for g in grad_f]
    print(f" Gradient of f at optimal point:{grad_f_val}")
else:
  print("❌No feasible KKT solution found")
```

OUTPUT:

✅Solution 1:

Optimal point:$x1 = 0, x2 = 0$

Lagrange multiplier $\lambda = 0$

Gradient of f at optimal point:[0, 0]

Problem 2:

Minimize : $f(x_1, x_2, x_3) = x_1{}^2 + 2x_2{}^2 + 3x_3{}^2$

Subject to the constraints

$$g_1 = x_1 - x_2 - 2x_3 \leq 12$$

$$g_2 = x_1 + 2x_2 - 3x_3 \leq 8$$

$$x_1, x_2, x_3 \geq 0$$

```
import sympy as sp
X1,X2,X3=sp.symbols('X1 X2 X3',real=True)
l1=sp.symbols('l1',real=True)
l2=sp.symbols('l2',real=True)
f=X1**2 + 2*X2**2 + 3*X3**2
g1=X1 - X2 - 2*X3 - 12
g2=X1 + 2*X2 - 3*X3 - 8
L=f+l1*g1+l2*g2
grad_L=[sp.diff(L,var) for var in [X1,X2,X3]]
kkt_eqs=[
    grad_L[0],
    grad_L[1],
    grad_L[2],
    l1*g1,
    l2*g2
]
solutions=sp.solve(kkt_eqs,[X1,X2,X3,l1,l2],dict=True)
feasible=[]
for sol in solutions:
  g1_val=g1.subs(sol)
  l1_val=sol.get(l1,0)
  if g1_val <=0 and l1_val >=0:
    g2_val=g2.subs(sol)
    l2_val=sol.get(l2,0)
    if g2_val <=0 and l2_val >=0:
        feasible.append(sol)
if feasible:
  for i,sol in enumerate(feasible):
    print(f"\n✅Solution {i+1}:")
    x1_val=sol.get(X1,0)
    x2_val=sol.get(X2,0)
    x3_val=sol.get(X3,0)
    l1_val=sol.get(l1,0)
```

```
    l2_val=sol.get(l2,0)
    print(f" Optimal point:x1={x1_val},x2={x2_val},x3={x3_val}")
    print(f" Lagrange multiplier λ1={l1_val}, λ2={l2_val}")
    grad_f=[sp.diff(f,var) for var in [X1,X2,X3]]
    grad_f_val=[g.subs({X1:x1_val,X2:x2_val,X3:x3_val}) for g in grad_f]
    print(f" Gradient of f at optimal point:{grad_f_val}")
else:
  print("❌No feasible KKT solution found")
```

OUTPUT:

✅Solution 1:

Optimal point:$x1 = 0, x2 = 0, x3 = 0$

Lagrange multiplier $\lambda 1 = 0, \lambda 2 = 0$

Gradient of f at optimal point:[0, 0, 0]

Problem 3:

Maximize $z = -{x_1}^2 + 2x_1 + x_2$

Subject to the constraints

$$2x_1 + 3x_2 \leq 6$$

$$2x_1 + x_2 \leq 4$$

$$x_1, x_2 \geq 0$$

```
import sympy as sp
X1,X2=sp.symbols('X1 X2',real=True)
l1=sp.symbols('l1',real=True)
l2=sp.symbols('l2',real=True)
f=-X1**2 + 2*X1 +X2
g1=2*X1 + 3*X2 - 6
g2=2*X1 + X2 - 4
L=f-l1*g1-l2*g2
grad_L=[sp.diff(L,var) for var in [X1,X2]]
kkt_eqs=[
    grad_L[0],
    grad_L[1],
    l1*g1,
    l2*g2 ]
solutions=sp.solve(kkt_eqs,[X1,X2,l1,l2],dict=True)
feasible=[]
```

```
for sol in solutions:
  g1_val=g1.subs(sol)
  l1_val=sol.get(l1,0)
  if g1_val <=0 and l1_val >=0:
    g2_val=g2.subs(sol)
    l2_val=sol.get(l2,0)
    if g2_val <=0 and l2_val >=0:
        feasible.append(sol)
if feasible:
  for i,sol in enumerate(feasible):
    print(f"\n✅Solution {i+1}:")
    x1_val=sol.get(X1,0)
    x2_val=sol.get(X2,0)
    l1_val=sol.get(l1,0)
    l2_val=sol.get(l2,0)
    print(f" Optimal point:x1={x1_val},x2={x2_val}")
    print(f" Lagrange multiplier λ1={l1_val}, λ2={l2_val}")
    grad_f=[sp.diff(f,var) for var in [X1,X2]]
    grad_f_val=[g.subs({X1:x1_val,X2:x2_val}) for g in grad_f]
    print(f" Gradient of f at optimal point:{grad_f_val}")
else:
  print("❌No feasible KKT solution found")
```

OUTPUT:

✅Solution 1:

Optimal point:$x1 = 2/3, x2 = 14/9$

Lagrange multiplier $\lambda 1 = 1/3, \lambda 2 = 0$

Gradient of f at optimal point:[2/3, 1]

Problem 4:

Minimize:

$$f(x_1, x_2) = (x_1 - 1)^2 + (x_2 - 2)^2$$

Subject to two inequality constraints

$$g_1(x) = x_1 + 2x_2 - 4 \leq 0$$

$$g_2(x) = -x_1 \leq 0 \quad (i.e., x_1 \geq 0)$$

```
import sympy as sp
X1,X2=sp.symbols('X1 X2',real=True)
l1=sp.symbols('l1',real=True)
l2=sp.symbols('l2',real=True)
f = (X1 - 1)**2 + (X2 - 2)**2
g1 = X1 + 2*X2 - 4
g2 = -X1
L = f + l1 * g1 + l2 * g2
grad_L=[sp.diff(L,var) for var in [X1,X2]]
kkt_eqs=[
   grad_L[0],
   grad_L[1],
   l1*g1,
   l2*g2 ]
solutions=sp.solve(kkt_eqs,[X1,X2,l1,l2],dict=True)
feasible=[]
for sol in solutions:
  g1_val=g1.subs(sol)
  l1_val=sol.get(l1,0)
  if g1_val <=0 and l1_val >=0:
    g2_val=g2.subs(sol)
    l2_val=sol.get(l2,0)
    if g2_val <=0 and l2_val >=0:
        feasible.append(sol)
if feasible:
  for i,sol in enumerate(feasible):
    print(f"\n✅Solution {i+1}:")
    x1_val=sol.get(X1,0)
    x2_val=sol.get(X2,0)
    l1_val=sol.get(l1,0)
    l2_val=sol.get(l2,0)
    print(f" Optimal point:x1={x1_val},x2={x2_val}")
    print(f" Lagrange multiplier λ1={l1_val}, λ2={l2_val}")
    grad_f=[sp.diff(f,var) for var in [X1,X2]]
    grad_f_val=[g.subs({X1:x1_val,X2:x2_val}) for g in grad_f]
    print(f" Gradient of f at optimal point:{grad_f_val}")
else:
  print("❌No feasible KKT solution found")
```

OUTPUT:

Optimal point: $x1 = 4/5, x2 = 8/5$

Lagrange multiplier $\lambda1 = 2/5, \lambda2 = 0$

Gradient of f at optimal point:[-2/5, -4/5]

RESULT:

Thus the experiment Kuhn Tucker Conditions- Largrangian Multiplier Method has been successfully implemented and excuted