**Implementation of approximation algorithms**

**for travelling salesman problem**

**Aim:**

To implement the traveling salesman problem using an approximation algorithm.

**Problem description:**

The traveling salesman problem states that we need to find the least cost incurred while visiting all the cities exactly once, starting and ending at the same city.

**Algorithm:**

* Create a minimum spanning tree (MST) of the given graph representing cities.
* Find all the vertices with odd degrees in the MST.
* Create a minimum-weight perfect matching among these odd-degree vertices.
* Combine the MST and the minimum-weight perfect matching to form a connected graph.
* Find an Eulerian circuit in the resulting graph.
* Convert the Eulerian circuit into a Hamiltonian circuit (TSP tour) by skipping repeated vertices and returning to the starting city.

**Code:**

import math

def distance(city1, city2, city\_distances):

    return city\_distances[city1][city2]

def nearest\_neighbor(city\_distances, starting\_city):

    num\_cities = len(city\_distances)

    tour = [starting\_city]

    remaining\_cities = list(range(num\_cities))

    remaining\_cities.remove(starting\_city)

    while remaining\_cities:

        current\_city = tour[-1]

        nearest\_city = min(remaining\_cities, key=lambda c: distance(current\_city, c, city\_distances))

        tour.append(nearest\_city)

        remaining\_cities.remove(nearest\_city)

    return tour

# Get input from the user

num\_cities = int(input("Enter the number of cities: "))

city\_distances = []

for i in range(num\_cities):

    distances = input(f"Enter the distances from city {i+1} to all other cities separated by spaces: ").split()

    distances = [int(d) for d in distances]

    city\_distances.append(distances)

starting\_city = int(input("Enter the starting city (1 to N): ")) - 1

# Solve TSP using nearest neighbor heuristic

tour = nearest\_neighbor(city\_distances, starting\_city)

total\_distance = sum(distance(tour[i], tour[i+1], city\_distances) for i in range(len(tour)-1))

total\_distance += distance(tour[-1], tour[0], city\_distances)

# Adjust tour and starting\_city index for printing

tour = [city + 1 for city in tour]

starting\_city += 1

# Print the result

print("Tour:", tour)

print("Total distance:", total\_distance)

**Output:**

**A screenshot of a computer

Description automatically generated with medium confidence**

• **Time Complexity:** Approximation algorithms for the TSP also have polynomial time complexities, typically ranging from O(n^2) to O(n^3) where n is the number of cities.

**• Algorithm Analysis:** These algorithms aim to find tours that are reasonably close to the optimal tours in terms of length or cost. They achieve this by using heuristics and greedy techniques to construct tours iteratively, considering factors like proximity. Although the solutions may not be optimal, they are usually efficient and provide reasonably good approximations.

**Result:**

Thus, travelling salesman problem has been solved succeffully using approximation algorithm

**Implementation of approximation algorithms**

**for knapsack problem**

**Aim:**

To implement the Knapsack problem using approximation.

**Problem Description:**

The Knapsack problem is an NP-hard problem that requires finding the maximum profit possible by sequentially placing items without exceeding the maximum capacity.

**Algorithm:**

* + Calculate the value-to-weight ratio for each item in the knapsack: ratio = value / weight.
  + Sort the items in descending order based on their value-to-weight ratio.
  + Initialize the total value and total weight variables to 0.
  + Iterate through the sorted items:
  + If the weight of the current item is less than or equal to the remaining capacity of the knapsack, include the whole item.
  + Add the value of the current item to the total value.
  + Subtract the weight of the current item from the remaining capacity of the knapsack.
  + If the weight of the current item is greater than the remaining capacity, include a fraction of the item.
  + Calculate the fraction by dividing the remaining capacity by the weight of the current item.
  + Break out of the loop since the knapsack is full.
  + Return the total value as the approximate solution to the Knapsack problem.

**Code:**

def knapsack\_approximation(values, weights, capacity):

    items = list(zip(values, weights))

    items.sort(key=lambda x: x[0] / x[1], reverse=True)

    total\_value = 0

    knapsack = []

    for value, weight in items:

        if weight <= capacity:

            knapsack.append((value, weight))

            total\_value += value

            capacity -= weight

    return knapsack, total\_value

# Get user input for values and weights

values = input("Enter the values (comma-separated): ").split(",")

weights = input("Enter the weights (comma-separated): ").split(",")

# Convert input values and weights to integers

values = [int(value) for value in values]

weights = [int(weight) for weight in weights]

capacity = int(input("Enter the capacity of the knapsack: "))

knapsack, total\_value = knapsack\_approximation(values, weights, capacity)

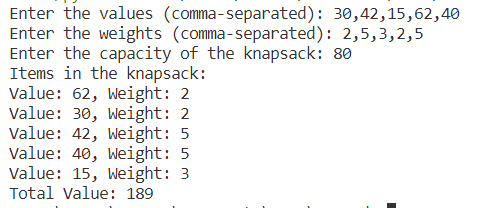
print("Items in the knapsack:")

for value, weight in knapsack:

    print(f"Value: {value}, Weight: {weight}")

print(f"Total Value: {total\_value}")

**Output:**

****

* **Time Complexity:** Approximation algorithms for the Knapsack Problem have polynomial time complexities, typically around O(n^2) where n is the

number of items.

* **Algorithm Analysis:** These algorithms provide efficient solutions by using

heuristics and greedy strategies to select items based on value and weight

ratios. While the solutions may not be optimal, they are generally close

enough to be considered satisfactory.

**Result:**

Thus, knapsack has been solved using approximation algorithm.