Name: Tejas Redkar
PRN: 1032210937
Panel - C
Roll No: PC-44

AIES Assignment 5

* Aim: Implement Hill climbing algorithm for Travelling salesman problem

* Objectives: Write a program in C / C++ / Java / Python to solve the hill climbing algorithm for travelling salesman problem

* Theory:

- Local search Algorithm:
  - It is a systematic search algorithm to find the goal
  - The path to the goal is irrelevant for eg. n-queens state space = set of "complete" configurations
  - Advantages: Use very little memory
    Find reasonable solutions in large or infinite ~~data sets~~ state spaces.
- Optimization: To find the best state based on an objective function ~~i.e~~ e.g. fitness - no goal test & path.
- Local Search algorithm: are used when we care only about a solution but not the path to a solution.

- Hill Climbing algorithm:
  - It is a local search algorithm which continuosly moves in the direction of increasing elevation to find the peak of the mountain our best solution to the problem. It terminates when it reaches a peak value where no neighbour has a higher value
  - A node of hill climbing algorithm has two components which one state & value
  - Hill climbing is mostly used when a good heuristic is available.

* Input : N * N matrix of distance for Travelling Salesman problem

* Output : An optical distance between two cities.

* Algorithm:

Hill Climbing algorithm

function HILL-CLIMBING (problem) returns a state that is local maximum

inputs : problem, a problem

local variables : current, a node
neighbour, a node

current ← MAKE - NODE [INITIAL - STATE (Problem)])

loop do

neighbour ← a highest-valued successour of current if VALUE [neighbour] < VALUE [current]

then return state [CURRENT]

current ← neighbour
end

* Platform : Linux/ Windows

* FAQ's

Q1) Explain Hill Climbing algorithm in detail with example

Ans Hill Climbing algorithm is a local search method that aims to find the best solution by continuously making greedy improvements in the search space. It doesn't backtrack, only explores immediate neighbours & terminates when it reaches a peak value. It's commonly used for optimization problems when a good heuristic is available, like minimizing travel distance in the Travelling salesman Problem

Example :

2 8 3
1 6 4 $h=4$
7   5

-5   -5

2 8 3
1   4
7 6 5

-3   -4

2   3
1 8 4
7 6 5

$h=-3$

-4

2   3
1 8 4 $h=-2$
7 6 5

1 2 3
8   4 $h=-1$
7 6 5

-2

1 2 3
goal 8   4 $h=0$
7 6 5

$F(n) = -$(no. of tiles out of place)

Q2) Explain the limitations of hill climbing & solutions to it.

Ans 1) Local optima Problem : Hill climbing ~~& solutions~~ tends to get stuck in local optima & may not find the global optimum.
Soln: Random restarts, simulated annealing.

2) Sensitivity to initial state : The quality of the solution depends heavily on the initial state, making it less robust.

3) Limited Memory: Hill Climbing does not remember previous states, which can be hinder its ability to backtrack when needed.

4) Plateau Problem : Hill climbing can become stuck on plateaus where the objective function remains constant for a range of neighbouring states.
Soln:- Random Perturbation, Tabu search.

Q3) Solve n queen problem using local search algorithm

Ans To solve the N-Queen problem using a local search algorithm like Hill Climbing:

1) Start with an initial queen placement on an NXN chessboard.

2) Define a cost function to measure how many queens threaten each other.

3) Evaluate the initial solutions cost.

4) Repeatedly generate neighbouring solutions by moving one queen & evaluate their costs.

5) Move to the neighbour with the lowest cost, repeating this process until no better solution is found or a termination condition is met.

6) The final solution, with a cost of 0, represents a valid N-Queens placement

Solution is:

	A	B	C	D
0			X	
1	X			
2				X
3		X		

1/12/23

```python
import random

# Function to create a random solution generator
def randomSolution(num_cities):
    cities = list(range(num_cities))
    solution = random.sample(cities, num_cities)
    return solution

# Function for calculating the length of a route
def routeLength(tsp, solution):
    route_length = 0
    num_cities = len(tsp)
    for i in range(num_cities):
        route_length += tsp[solution[i - 1]][solution[i]]
    return route_length

# Function for generating all neighbors of a solution
def getNeighbours(solution):
    neighbours = []
    num_cities = len(solution)
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            neighbour = solution[:]
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)
    return neighbours

# Function for finding the best neighbor
def getBestNeighbour(tsp, neighbours):
    best_route_length = routeLength(tsp, neighbours[0])
    best_neighbour = neighbours[0]
    for neighbour in neighbours:
        current_route_length = routeLength(tsp, neighbour)
        if current_route_length < best_route_length:
            best_route_length = current_route_length
            best_neighbour = neighbour
    return best_neighbour, best_route_length

# Hill climbing algorithm
def hillClimbing(tsp, num_cities):
    current_solution = randomSolution(num_cities)
    current_route_length = routeLength(tsp, current_solution)
    neighbours = getNeighbours(current_solution)
    best_neighbour, best_neighbour_route_length =
getBestNeighbour(tsp, neighbours)

    while best_neighbour_route_length < current_route_length:
        current_solution = best_neighbour
        current_route_length = best_neighbour_route_length
```

```
        neighbours = getNeighbours(current_solution)
        best_neighbour, best_neighbour_route_length = 
getBestNeighbour(tsp, neighbours)

    return current_solution, current_route_length

def main():
    num_cities = int(input("Enter the number of cities: "))
    tsp = []

    for i in range(num_cities):
        row = list(map(int, input(f"Enter the distances from city 
{i+1} to all cities separated by spaces: ").split()))
        tsp.append(row)

    solution, route_length = hillClimbing(tsp, num_cities)

    print("Optimal Route:", solution)
    print("Optimal Route Length:", route_length)

if __name__ == "__main__":
    main()

Enter the number of cities: 5
Enter the distances from city 1 to all cities separated by spaces: 10 
12 13 19 9
Enter the distances from city 2 to all cities separated by spaces: 12 
13 10 7 9
Enter the distances from city 3 to all cities separated by spaces: 13 
14 12 10 8
Enter the distances from city 4 to all cities separated by spaces: 12 
13 13 10 7
Enter the distances from city 5 to all cities separated by spaces: 12 
13 10 8 9
Optimal Route: [0, 1, 3, 4, 2]
Optimal Route Length: 49
```