

**Artificial Intelligence Systems**

Lab Report # 08

**Submitted by:** Ali Hamza

**Registration No:** B23F0063AI106

**Submitted to:** Dr. Mohsin

Date: 12th April, 2025

# Lab Task:

# In this assignment, you will be working with the Hill Climbing Algorithm. Your task is to analyze the provided code, understand its functionality, and document your findings in a detailed lab report. Below are the steps you should follow to complete this assignment successfully.

# Step 1: Run the Code

# Set Up Your Environment:

# Ensure you have the necessary software installed to run the code. This may include a specific programming language environment or an integrated development environment (IDE).

# If you are unsure about the setup, consult your teacher or refer to any provided resources.

# Execute the Code:

# Open the code file in your IDE or text editor.

# Run the code by following the appropriate commands for your environment. Make sure to observe any output or results that the code generates.

# Take Notes:

# As you run the code, take notes on what happens. Pay attention to any inputs, outputs, and any errors that may occur. This will help you in your analysis later.

# Step 2: Analyze the Code

# Understand the Structure:

# Break down the code into smaller sections. Identify the main components such as functions, loops, and conditionals.

# Look for comments within the code that may explain what certain parts do.

# Identify Key Concepts:

# Focus on the key concepts of the Hill Climbing Algorithm. What is its purpose? How does it work? What are its strengths and weaknesses?

# Research any unfamiliar terms or functions that you encounter in the code.

# Document Your Observations:

# Write down your observations about how the code operates. What are the inputs and outputs? How does the algorithm make decisions?

# Consider how the algorithm could be applied to real-world problems.

# Step 3: Write the Lab Report

# Title Page:

# Start your report with a title page that includes the title of the assignment, your name, the date, and any other required information.

# Introduction:

# Write an introduction that explains what the Hill Climbing Algorithm is and why it is important. Provide context for your analysis.

# Code Description:

# In this section, provide a detailed description of the code you analyzed.

# Break it down into parts, explaining the purpose of each section. Use bullet points or numbered lists to organize your thoughts clearly.

# Analysis of Results:

# Discuss the results you observed when running the code. Did it perform as expected? Were there any surprises?

# Include any graphs or charts if applicable to illustrate your findings.

# Conclusion:

# Summarize your findings and reflect on what you learned from this assignment.

# Consider how the Hill Climbing Algorithm could be improved or applied in different scenarios.

# References:

# If you used any external sources for your research, be sure to include a references section at the end of your report.

import random  
import numpy as np  
  
#coordinate of the points/cities  
coordinate = np.array([[1,2], [30,21], [56,23], [8,18], [20,50], [3,4], [11,6], [6,7], [15,20], [10,9], [12,12]])  
  
#adjacency matrix for a weighted graph based on the given coordinates  
def generate\_matrix(coordinate):  
 matrix = []  
 for i in range(len(coordinate)):  
 for j in range(len(coordinate)) :  
 p = np.linalg.norm(coordinate[i] - coordinate[j])  
 matrix.append(p)  
 matrix = np.reshape(matrix, (len(coordinate),len(coordinate)))  
 #print(matrix)  
 return matrix  
  
#finds a random solution  
def solution(matrix):  
 points = list(range(0, len(matrix)))  
 solution = []  
 for i in range(0, len(matrix)):  
 random\_point = points[random.randint(0, len(points) - 1)]  
 solution.append(random\_point)  
 points.remove(random\_point)  
 return solution  
  
  
#calculate the path based on the random solution  
def path\_length(matrix, solution):  
 cycle\_length = 0  
 for i in range(0, len(solution)):  
 cycle\_length += matrix[solution[i]][solution[i - 1]]  
 return cycle\_length  
  
#generate neighbors of the random solution by swapping cities and returns the best neighbor  
def neighbors(matrix, solution):  
 neighbors = []  
 for i in range(len(solution)):  
 for j in range(i + 1, len(solution)):  
 neighbor = solution.copy()  
 neighbor[i] = solution[j]  
 neighbor[j] = solution[i]  
 neighbors.append(neighbor)  
  
 #assume that the first neighbor in the list is the best neighbor  
 best\_neighbor = neighbors[0]  
 best\_path = path\_length(matrix, best\_neighbor)  
  
 #check if there is a better neighbor  
 for neighbor in neighbors:  
 current\_path = path\_length(matrix, neighbor)  
 if current\_path < best\_path:  
 best\_path = current\_path  
 best\_neighbor = neighbor  
 return best\_neighbor, best\_path  
  
  
def hill\_climbing(coordinate):  
 matrix = generate\_matrix(coordinate)  
  
 current\_solution = solution(matrix)  
 current\_path = path\_length(matrix, current\_solution)  
 neighbor = neighbors(matrix,current\_solution)[0]  
 best\_neighbor, best\_neighbor\_path = neighbors(matrix, neighbor)  
  
 while best\_neighbor\_path < current\_path:  
 current\_solution = best\_neighbor  
 current\_path = best\_neighbor\_path  
 neighbor = neighbors(matrix, current\_solution)[0]  
 best\_neighbor, best\_neighbor\_path = neighbors(matrix, neighbor)  
  
 return current\_path, current\_solution  
final\_solution = hill\_climbing(coordinate)  
print("The solution is \n", final\_solution[1])

## **Introduction**

The Hill Climbing algorithm is a fundamental local search technique used in optimization problems. It starts with a random solution and attempts to incrementally improve it by moving to a neighboring solution with a better score, or in this case, a shorter path. This report documents the implementation of the Hill Climbing algorithm applied to a variant of the Travelling Salesman Problem (TSP), where the goal is to find the shortest path that visits all points.

Hill Climbing is simple, memory-efficient, and suitable for problems where the search space is well-behaved. However, it suffers from local optima and lacks backtracking, making it sensitive to the initial random solution.

## **Code Description**

This section breaks down the functionality of the provided Python code:

### 2.1 Coordinates Initialization

coordinate = np.array([[1,2], [30,21], [56,23], [8,18], [20,50], [3,4], [11,6], [6,7], [15,20], [10,9], [12,12]])

* Represents 11 cities or points using their (x, y) coordinates.

### 2.2 Distance Matrix Generation

def generate\_matrix(coordinate):

* Calculates the Euclidean distance between each pair of points using np.linalg.norm.
* Outputs an 11x11 adjacency matrix of distances.

### 2.3 Initial Random Solution

def solution(matrix):

* Creates a random permutation of the point indices (0 to 10) representing a tour.
* Ensures that each city is visited exactly once.

### 2.4 Path Length Calculation

def path\_length(matrix, solution):

* Computes the total cost (path length) by summing the distances between each consecutive pair of cities in the tour.
* Uses cyclic indexing: solution[i - 1] connects the last city to the first.

### 2.5 Neighbors Generation and Selection

def neighbors(matrix, solution):

* Generates neighboring solutions by swapping every pair of cities in the current solution.
* Evaluates each neighbor’s path length.
* Returns the one with the shortest path (greedy choice).

### 2.6 Hill Climbing Logic

def hill\_climbing(coordinate):

* Initializes the algorithm with a random solution.
* Repeatedly replaces the current solution with a better neighbor until no improvement is possible (local optimum).
* Returns the best path length and the best solution found.

## **Analysis of Results**

### 3.1 Execution

* Upon running, the program prints the best route (city indices) it found using the Hill Climbing approach.
* Example Output:

The solution is

[2, 1, 4, 0, 5, 3, 9, 6, 10, 7, 8]

### 3.2 Observations

* The final solution is local, not necessarily the global shortest path.
* Since the initial solution is random, different runs can yield different results.
* No graphs are generated, but if plotted, the solution would resemble a connected tour between 11 cities.

### 3.3 Strengths and Weaknesses

|  |  |
| --- | --- |
| **Strengths** | **Weaknesses** |
| Simple to implement | Prone to local optima |
| Fast convergence | No memory of past states |
| Suitable for small problems | Random start means inconsistency |

## **Conclusion**

The Hill Climbing algorithm implemented here effectively demonstrates a basic local search heuristic applied to a simplified TSP. It iteratively improves the solution by selecting the best neighbor. However, due to its greedy nature, it is prone to getting stuck in local optima. Future enhancements could involve **random restarts**, **simulated annealing**, or **genetic algorithms** to overcome its limitations.

## **References**

Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.).

[Hill Climbing – GeeksforGeeks](https://www.geeksforgeeks.org/hill-climbing-algorithm-in-ai/)

[TSP and Hill Climbing – TutorialsPoint](https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_hill_climbing.htm)