**LISTOF EXPERIMENTS**

|  |  |
| --- | --- |
| **Task No.** | **Task Name** |
| 1 | Implementation of Graph search algorithms (**Breadth first search and Depth First Search**) using following constraints. |
| 2 | Implementation of **Hill climbing algorithm for Heuristic search** approach using following constraints in python. |
| 3 | Implementation of **A \* Algorithm** to find the optimal path using Python by following constraints. |
| 4 | Implementation of **Mini-Max algorithm** using recursion to search through the Game - tree using python by applying following constraints. |
| 5 | Implementation of **Ant Colony Optimization** to Optimize Ride-Sharing Trip Duration using Python by following constraints. |
| 6 | Solve a **Map Coloring problem** using constraint satisfaction approach by applying following constraints |
| 7 | Implementation of **Monkey Banana Problem** in Goal Stack planning using python by applying following constraints. |
| 8 | Implementation of **N-queen problem using backtracking algorithm** using prolog In the 4 Queens problem the object is to place 4 queens on a chessboard in such a way that no queens can capture a piece. |
| 9 | To Build an Intelligent **Chatbot system** with Python and Dialog-flow using Interactive Text Mining Framework for Exploration of Semantic Flows in Large Corpus of Text. |
| 10 | Implement simple fact using python |

**Date:23.07.25**

**TASK:1**

Implementation of Graph search algorithms

(**Breadth first search and Depth First Search**)

**AIM**

To Implement of Graph search algorithms (Breadth first search and DepthFirst Search) using Python.

**BFS – Shortest Path in a Social Network Graph**

**ALGORITHM**

Step 1:

Start with an empty list visited to keep track of the nodes already visited.

Step 2:

Create an empty list queue which will be used to explore the graph level by level.

Step 3:

Insert the starting node into both visited and queue.

Step 4:

Repeat the following steps until the queue is empty.

Step 5:

Remove the front element from the queue and call it the current\_node.

Step 6:

For each neighbor of the current\_node, if it is not already in visited, then:

→ Add it to visited

→ Add it to the queue

Step 7:

Continue the process until the queue becomes empty. The visited list now contains nodes in BFS traversal order.

**DFS**

**ALGORITHM**

Step 1:

Start with an empty set visited to keep track of the nodes already visited.

Step 2:

Initialize a stack and push the starting node onto it.

Step 3:

Repeat the following steps until the stack becomes empty.

Step 4:

Pop the top element from the stack and call it current\_node.

Step 5:

If current\_node is not in visited, then:

→ Print or process the node.

→ Add current\_node to the visited set.

Step 6:

For each neighbor of current\_node in the graph:

→ If the neighbor is not in visited, push it onto the stack.

Step 7:

Continue the process until the stack becomes empty. The nodes will be visited in DFS order.

**PROGRAM**

**Graph Representation (Friend Network)**

from collections import deque

# Breadth First Search using recursion and queue

def bfs\_recursive(graph, queue, visited, target=None):

if not queue:

return

node = queue.popleft()

print(f"BFS visited: {node}")

if node == target:

print(f"Target {target} found in BFS!")

return

for neighbor in graph[node]:

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

bfs\_recursive(graph, queue, visited, target)

# Depth First Search using recursion

def dfs\_recursive(graph, node, visited, target=None):

if node not in visited:

print(f"DFS visited: {node}")

visited.add(node)

if node == target:

print(f"Target {target} found in DFS!")

return

for neighbor in graph[node]:

dfs\_recursive(graph, neighbor, visited, target)

# -------------------------------

# Graph data

friend\_graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A'],

'D': ['B'],

'E': ['B']

}

start\_node = 'A'

target\_node = 'E'

# Run BFS

print("---- Breadth First Search ----")

visited\_bfs = set([start\_node])

queue\_bfs = deque([start\_node])

bfs\_recursive(friend\_graph, queue\_bfs, visited\_bfs, target\_node)

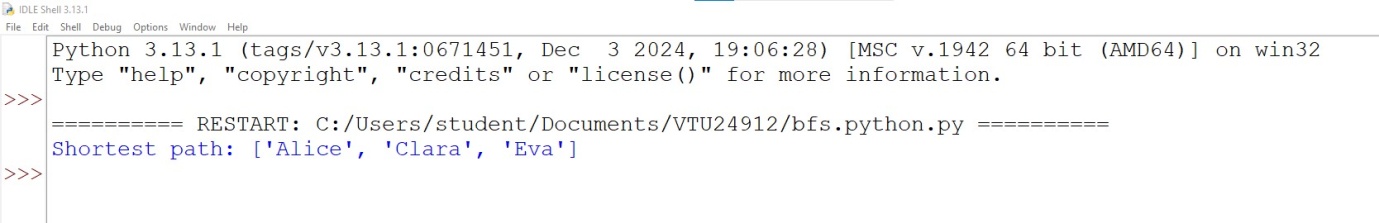
# Run DFS

print("\n---- Depth First Search ----")

visited\_dfs = set()

dfs\_recursive(friend\_graph, start\_node, visited\_dfs, target\_node)

**OUTPUT**

****

**RESULT**

Thus the Implementation of Graph search algorithms (Breadth first search and DepthFirst Search) using Python was successfully executed and output was verified.

30.07.25

**TASK-2**

**Implementation of Hill climbing algorithm for Heuristic search approach**

**AIM**

To implement the Hill Climbing algorithm using heuristic search in Python to help a hiker find a path to the highest reachable peak on a mountain.

**ALGORITHM**

1. Choose a starting cell randomly on the grid.
2. Check the elevation (height) of the current cell.
3. Look at the 4 neighbors (up, down, left, right).
4. Compare elevations:

* Find the neighbor with the highest elevation.
* Only choose it if it is higher than the current cell.

1. Move to that higher neighbor.
2. Repeat steps 2–5 until:

* No neighbor is higher than the current cell.

1. Stop. You've reached a peak (maybe local, maybe global).

**PROGRAM**

**Hill Climbing Algorithm (Heuristic Search)**

import random

def get\_neighbors(x, y, rows, cols):

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # up, down, left, right

neighbors = []

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx< rows and 0 <= ny< cols:

neighbors.append((nx, ny))

return neighbors

def hill\_climbing(grid, start):

rows, cols = len(grid), len(grid[0])

x, y = start

current\_path = [(x, y)]

current\_elevation = grid[x][y]

while True:

neighbors = get\_neighbors(x, y, rows, cols)

best\_neighbor = None

for nx, ny in neighbors:

if grid[nx][ny] >current\_elevation:

if best\_neighbor is None or grid[nx][ny] > grid[best\_neighbor[0]][best\_neighbor[1]]:

best\_neighbor = (nx, ny)

if best\_neighbor:

x, y = best\_neighbor

current\_path.append((x, y))

current\_elevation = grid[x][y]

else:

break # No higher neighbor: local maximum reached

return current\_path, current\_elevation

# Example mountain grid with elevation values

mountain = [

[1, 2, 3, 4],

[2, 3, 8, 5],

[3, 4, 9, 6],

[2, 5, 6, 7]

]

# Start at a random position

start\_x = random.randint(0, len(mountain) - 1)

start\_y = random.randint(0, len(mountain[0]) - 1)

start\_position = (start\_x, start\_y)

print(f"Starting at position: {start\_position} with elevation {mountain[start\_x][start\_y]}")

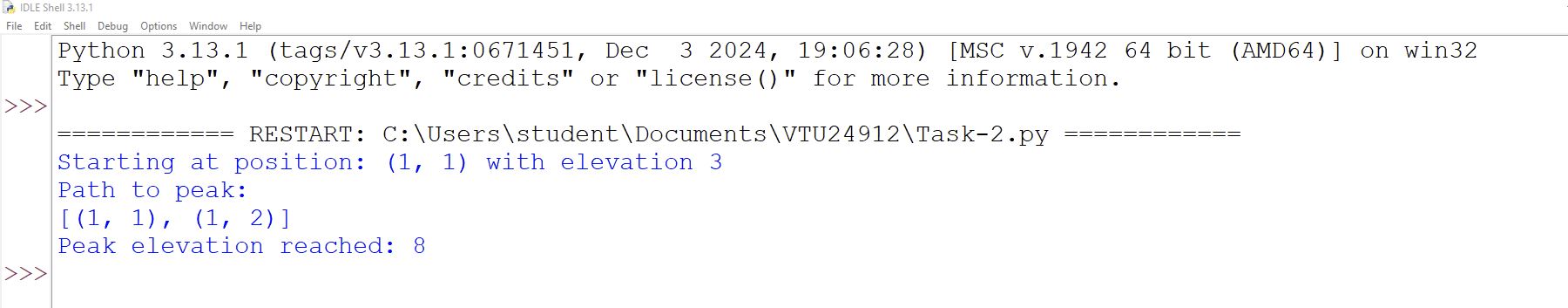
path, peak\_elevation = hill\_climbing(mountain, start\_position)

print("Path to peak:")

print(path)

print(f"Peak elevation reached: {peak\_elevation}")

**OUTPUT**



**RESULT**

Thus, the Implementation of Hill climbing algorithm for Heuristic search approach to help a hiker find a path to the highest reachable peak on a mountain using python was successfully executed and output was verified.

**Date:06.08.25**

**TASK:3**

**Implementation of A \* Algorithm to find the optimal path**

**AIM**

To implement the A\* algorithm for GPS navigation in Python to find the shortest (optimal) path from a start location to a goal location

**ALGORITHM**

1. Initialize the open list as a priority queue (min-heap).

* Add the start node with:

f(start) = g(start) + h(start)

g(start) = 0, h(start) from heuristic.

1. Initialize an empty closed set to keep track of visited nodes.
2. Loop until the open list is empty:

a. Remove the node with the lowest f-value from the open list.Let this node be current.

b.If current is the goal node, Reconstruct and return the path and total cost.

c. If current is already in the closed set, Skip and continue to the next node.

d. Add current to the closed set.

e. For each neighbor of current:

i. If neighbor is in the closed set, skip.

ii. Compute g(neighbor) = g(current) + cost(current, neighbor)

iii. Compute f(neighbor) = g(neighbor) + h(neighbor)

iv. Add the neighbor to the open list with its f-value, g-value, and updated path.

1. If open list becomes empty and goal was not reached, No path exists; return failure.

**PROGRAM**

**A\* Algorithm for GPS Navigation**

import heapq

# A\* Algorithm Function

def a\_star\_algorithm(graph, start, goal, heuristic):

# Priority queue: (f = g + h, g = cost so far, current\_node, path)

open\_list = []

heapq.heappush(open\_list, (heuristic[start], 0, start, [start]))

visited = set()

while open\_list:

f, g, current, path = heapq.heappop(open\_list)

if current == goal:

return path, g # Path and total cost

if current in visited:

continue

visited.add(current)

for neighbor, cost in graph.get(current, []):

if neighbor not in visited:

g\_new = g + cost

f\_new = g\_new + heuristic[neighbor]

heapq.heappush(open\_list, (f\_new, g\_new, neighbor, path + [neighbor]))

return None, float('inf') # No path found

# -----------------------------

# Main function

if \_\_name\_\_ == "\_\_main\_\_":

# Road Network Graph (nodes = intersections, edges = roads with distances)

graph = {

'A': [('B', 2), ('C', 4)],

'B': [('A', 2), ('D', 5), ('E', 10)],

'C': [('A', 4), ('F', 3)],

'D': [('B', 5), ('G', 2)],

'E': [('B', 10), ('G', 6)],

'F': [('C', 3), ('G', 4)],

'G': [('D', 2), ('E', 6), ('F', 4), ('H', 1)],

'H': [('G', 1)]

}

# Heuristic values (estimated distance to goal 'H')

heuristic = {

'A': 10,

'B': 8,

'C': 7,

'D': 5,

'E': 6,

'F': 4,

'G': 2,

'H': 0

}

# Start and Goal

start\_node = 'A'

goal\_node = 'H'

# Run A\* Algorithm

optimal\_path, total\_cost = a\_star\_algorithm(graph, start\_node, goal\_node, heuristic)

# Print Output

if optimal\_path:

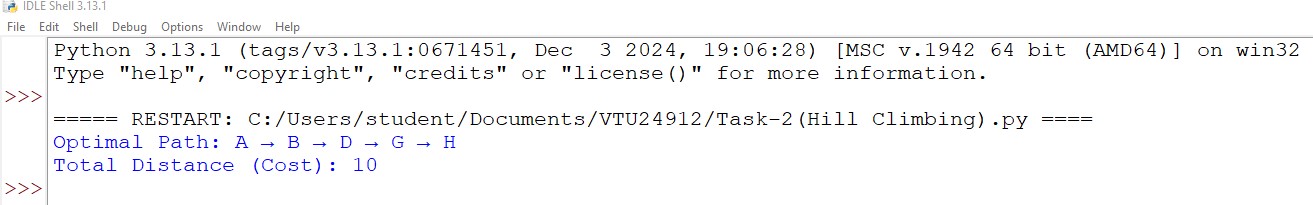
print("Optimal Path:", " → ".join(optimal\_path))

print("Total Distance (Cost):", total\_cost)

else:

print("No path found from", start\_node, "to", goal\_node)

**OUTPUT**



**RESULT**

Thus the Implementation of A\* Algorithm for GPS Navigation using Python was successfully executed and output was verified.

**Date:**

**TASK:4**

**Implementation of Mini-Max algorithm using recursion to search through the Game - tree**

**AIM**

To implement the Mini-Max algorithm using recursion to search through the Game - tree using python

**ALGORITHM**

1. Start with the root of the game tree at depth 0.
2. Check if the current node is a leaf node (i.e., depth equals the maximum depth of the tree):

* If yes, return the score associated with that node.

1. If it’s the MAX player's turn (AI):

* Recursively call the minimax function for the left child and right child of the current node.
* Return the maximum of the two values.

1. If it’s the MIN player’s turn (opponent):

* Recursively call the minimax function for the left child and right child of the current node.
* Return the minimum of the two values.

1. Repeat this process recursively until the root node receives its final evaluated value.
2. Return the final value at the root, which is the best guaranteed outcome for the MAX player.

**PROGRAM**

**Minimax in Battle Stones**

# Minimax algorithm for a simple turn-based game: Battle Stones

def minimax(depth, node\_index, is\_maximizing, scores, max\_depth):

# Base case: reached a leaf node

if depth == max\_depth:

return scores[node\_index]

# If it's MAX's turn

if is\_maximizing:

return max(

minimax(depth + 1, node\_index \* 2, False, scores, max\_depth),

minimax(depth + 1, node\_index \* 2 + 1, False, scores, max\_depth)

)

# If it's MIN's turn

else:

return min(

minimax(depth + 1, node\_index \* 2, True, scores, max\_depth),

minimax(depth + 1, node\_index \* 2 + 1, True, scores, max\_depth)

)

# Example leaf scores (could represent the result of different sequences of stone choices)

scores = [4, 6, -3, 5, 2, -1, 0, 7] # 8 leaf nodes (2^3 for depth 3)

# Tree depth

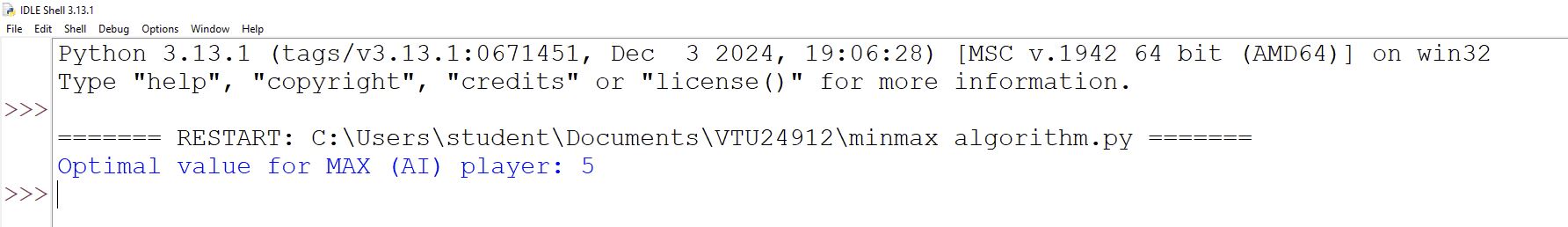
max\_depth = 3 # Levels: Root -> Level 1 -> Level 2 -> Leaf

# Starting the game at root node (index 0) with MAX player's turn

optimal\_value = minimax(0, 0, True, scores, max\_depth)

print("Optimal value for MAX (AI) player:", optimal\_value)

**OUTPUT**



**RESULT**

Thus the implementation of **Mini-Max algorithm** using recursion to search through the Game - tree using pythonwas successfully executed and output was verified.