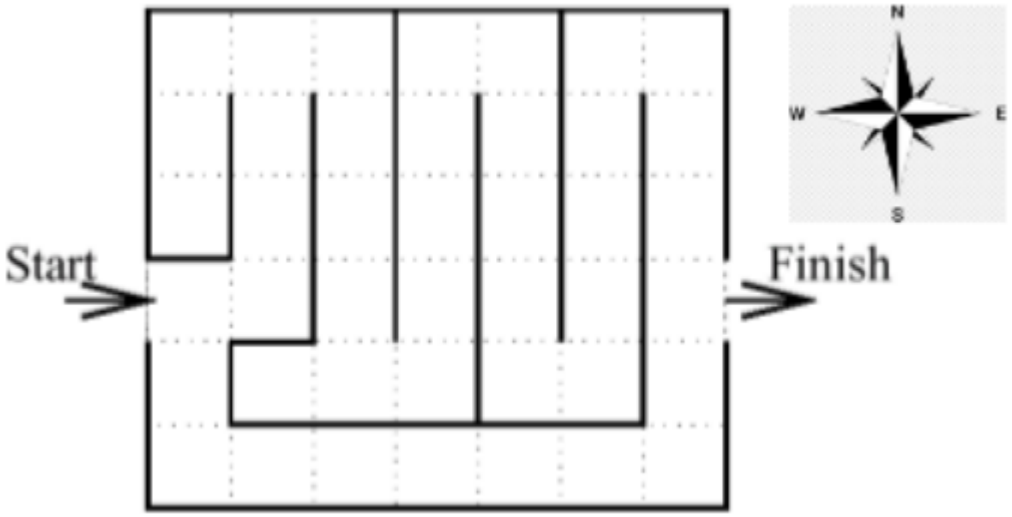
Path finding Maze Agent Using IDA* & Hill Climbing Algorithm)

Problem statement

Given the below maze configuration, the task of the robot is to navigate in the maze and find the optimal path to reach the finish position. It can move to the north, south, west and east direction. While navigating through the environment it has obstacles like walls. For each transition, a path cost of +3 is added in search. Assume that the robot's vision sensors are sensitive to the exposure to the sunlight and whenever it tries to move towards the east direction resulting in incurring an additional penalty of +5 cost. Use Manhattan distance as a heuristic wherever necessary.



Use the following algorithms to find the optimal path.

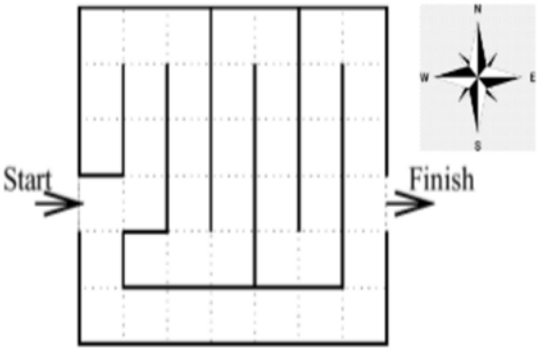
a. Iterative Deepening A* Algorithm

b. Hill Climbing Algorithm

Analysis and Design

Approach

To get started, we converted the given maze into a Matrix form as per below pictures. Each cell of this matrix is internally assumed as node. This helps to start coding for the maze agent as the grid is easily interpretable. Coding came naturally as agent can navigate through the nodes. Obstructions in the form of walls are also taken into account. Defined the node attributes to represent directions (North, South, East, West) in this sequence and every node has defined degree of freedom in terms of (1 – OK to proceed in the direction and 0 – Wall). To solve the IDA* we converted this matrix into a Tree form. The tree form is also given below for visual understanding.



	1	2	3	4	5	6	7
1	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)	(1, 7)
2	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6)	(2, 7)
3	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)	(3, 7)
4	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	(4, 6)	(4, 7)
5	(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)	(5, 6)	(5, 7)
6	(6, 1)	(6, 2)	(6, 3)	(6, 4)	(6, 5)	(6, 6)	(6, 7)

N,S,E,W

	1	2	3	4	5	6	7
1	0,1,1,0	0,1,1,1	0,1,0,1	0,1,1,0	0,1,0,1	0,1,1,0	0,1,0,1
2	1,1,0,0	1,1,0,0	1,1,0,0	1,1,0,0	1,1,0,0	1,1,0,0	1,1,0,0
3	1,0,0,0	1,1,0,0	1,1,0,0	1,1,0,0	1,1,0,0	1,1,0,0	1,1,0,0
4	0,1,1,0	1,0,0,1	1,1,0,0	1,1,0,0	1,1,0,0	1,1,0,0	1,1,0,0
5	1,1,0,0	0,0,1,0	1,0,1,1	1,0,0,1	1,0,1,0	1,0,0,1	1,1,0,0
6	1,0,1,0	0,0,1,1	0,0,1,1	0,0,1,1	0,0,1,1	0,0,1,1	1,0,0,1

Figure 1 : Given Maze - Matrix Form with Directions

Language/Tools Used

- Python with standard libraries and Excel

Problem Solving Agent (PSA) – Problem Formulation

AI problem solving requires a well-formulated problem, as it allows the AI algorithm to identify patterns and relationships that can be used to solve the problem effectively and efficiently.

Components	IDA* Maze Agent	Hill Climbing Maze Agent
Initial State	• As per problem statement – (4,1) • Can be dynamically defined in main code	
Possible Actions	[Move North (Up, Move South (Down), Move East (Right), Move West (Left)]	
Successor Function / Transition Model	[If open direction, then explore] Example: At (4,1), explore (4,2) and (5,1) open directions	
Goal Test	Is Goal reached? (4,7) reached as per problem statement	Is Goal or local minima/plateau reached?
Path Cost	• Cost of travelling every step + penalty (West to East) • Number of cells traversed	

Table 1: Problem Solving Agent (PSA) – Problem Formulation

PEAS (Performance, Environment, Actuators, Sensors)

PEAS	Path Finding Agent for IDA*	Path Finding Agent for Hill Climbing
Performance	• Optimal Path is guaranteed • Low memory consumption as it doesn't store which increases linearly as it doesn't store and forgets after it reaches a certain depth and start over again • Slower due to repeating the exploring of explored nodes	• Optimal Path is not guaranteed • Low memory consumption as it evaluates the neighbour node state at a time and selects the first one which optimizes current cost and set it as a current state • Faster since it is typically not a complete algorithm i.e. Can stop at best possible solution
Environment	• Presence of rectangular maze • Every cell has open directions in which agent can proceed and walls as obstacles • Start (Source) and End (Goal) cell is defined • Maze is static and fully observable	• Presence of rectangular maze • Every cell has open directions in which agent can proceed and walls as obstacles • Start (Source) and End (Goal) cell is defined • Maze is static and fully observable
Actuator	Keeps moving towards North/South/ East/West directions as per sensor and algorithm input, till goal is reached. Restart as per threshold value	Keeps moving towards North/South/ East/West as per sensor and algorithm input, till goal/local minima (best possible solution) is reached
Sensor	Identify blockage/obstacle and available free directions *Tree Nodes attributes contain this input in the form of heuristic cost	Identify blockage/obstacle and available free directions *Maze Matrix contains this input in the form of heuristic cost

Table 2: PEAS (Performance, Environment, Actuators, Sensors)

Key Code and Explanation

- Imported the required libraries.

```
#Code Block : Set Initial State (Must handle dynamic inputs)

from queue import PriorityQueue
from tkinter import *
from enum import Enum

INFINITY = 10000000
cameFrom = {}
```

Figure 2 : Python Libraries used in Algorithm

- The queue is used for hill climbing algorithm
- Tkinter is used for GUI creation to display maze visually (Limitation: Working on Windows platform only)
- Enum is used to denote colors

- Defined the below colors (Limitation: Works only on Windows platforms and the editors that use ipynb files. Tested on Jupyter Notebook using Anaconda environment and using PyCharm in Windows 11 with Python 3.7+)

```
[2]  #Code Block : Set the matrix for transition & cost (as relevant for the given problem)

    class COLOR(Enum):
        '''
        This class is created to use the Tkinter colors easily.
        Each COLOR object has two color values.
        The first two objects (dark and light) are for theme and the two color
        values represent the Canvas color and the Maze Line color respectively.
        The rest of the colors are for Agents.
        The first value is the color of the Agent and the second is the color of
        its footprint
        '''
        dark = ('gray11', 'white')
        light = ('white', 'black')
        black = ('black', 'dim gray')
        red = ('red3', 'tomato')
        cyan = ('cyan4', 'cyan4')
        green = ('green4', 'pale green')
        blue = ('DeepSkyBlue4', 'DeepSkyBlue2')
        yellow = ('yellow2', 'yellow2')
```

Figure 3 : Maze GUI Colours definition

- Colour codes for agent, maze, path, start point and end point is defined as below:
  - Light/Dark → Maze board background
  - Green → Goal cell
  - Yellow → Start cell
  - Blue → Agent's final path (Maybe optimal or good)
  - Red → Agent's All tested paths

- Agent class is used to define the Agent which will navigate from source to destination.

  An agent can be thought of as a physical agent like a robot or it can simply be used to highlight or point a cell in the maze. For this, agent class is defined. Agent starts inside the Maze at the source location with an aim to reach destination using optimum / best possible path with discrete steps.

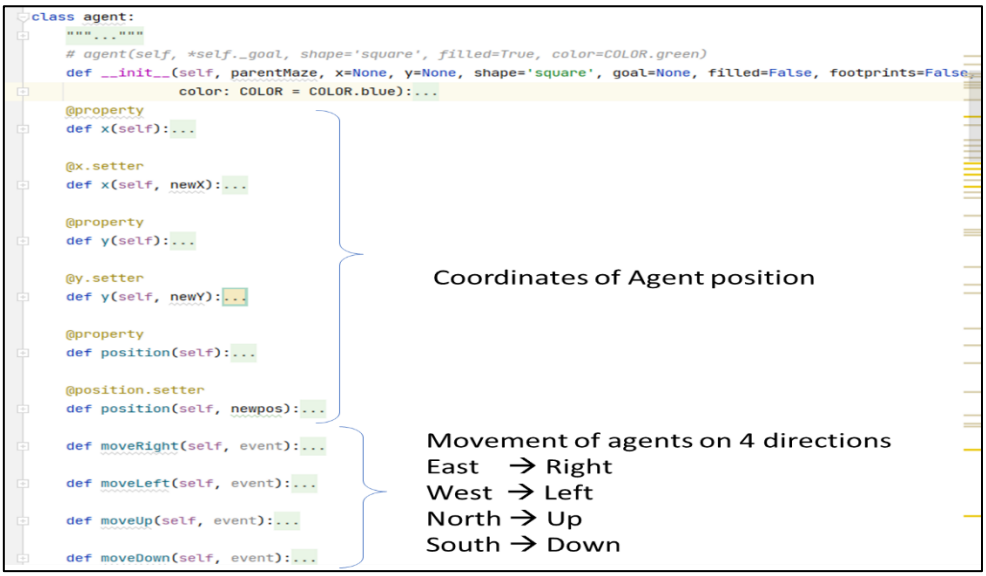


Figure 4 : Agent Class functions

- Maze class is created to define the rectangular maze board.

  A Maze is given as MxN matrix of cells where source cell and destination cell is given. An agent starts from source and has to reach the destination cell. The agent can move in four directions: north, south, east and west if available. Availability of direction is defined with binary value 1/0 for each direction, where '1' means, the agent is allowed to move in that direction, while in case of '0' - wall act as an obstacle in the path of agent (ex: For cell (1,1) → (0,1,1,0) as (N,S,E,W) meaning agent can move in South (2,1) and East (1,2) direction)

```
class maze:
    """..."""
    def __init__(self, rows=10, cols=10):...

    @property
    def grid(self):...

    @grid.setter
    def grid(self, n):...

    def _Open_East(self, x, y):...

    def _Open_West(self, x, y):...

    def _Open_North(self, x, y):...

    def _Open_South(self, x, y):...

    def CreateMaze(self, start_=(1, 1), fin_=(1, 1), pattern=None, loopPercent=0, saveMaze=False, loadMaze
                   theme: COLOR = COLOR.dark, maze_map_raw={}):...

    def _drawMaze(self, theme):...

    def _redrawCell(self, x, y, theme):...

    _tracePathList = []
    def _tracePathSingle(self, a, p, kill, showMarked, delay):...

    def tracePath(self, d, kill=False, delay=500, showMarked=False):...

    def run(self):...
```

To define the degree of freedom of each cell

To construct the dynamic maze on a given input

To trace and display final path covered by the agent

Figure 5 : Maze Class and its functions

- As per the problem statement the given maze is converted to matrix format (6 rows x 7 columns). Matrix index is started with 1. Directions (both open and blocked) are defined with binary variables as defined before. GUI of the same is given in Figure 6 : Maze Matrix and its definition

Figure 6 : Maze Matrix and its definition

➢ **Driver functions:**

Multiple driver functions are written to support the two algorithms – IDA* and Hill Climb algorithm functioning

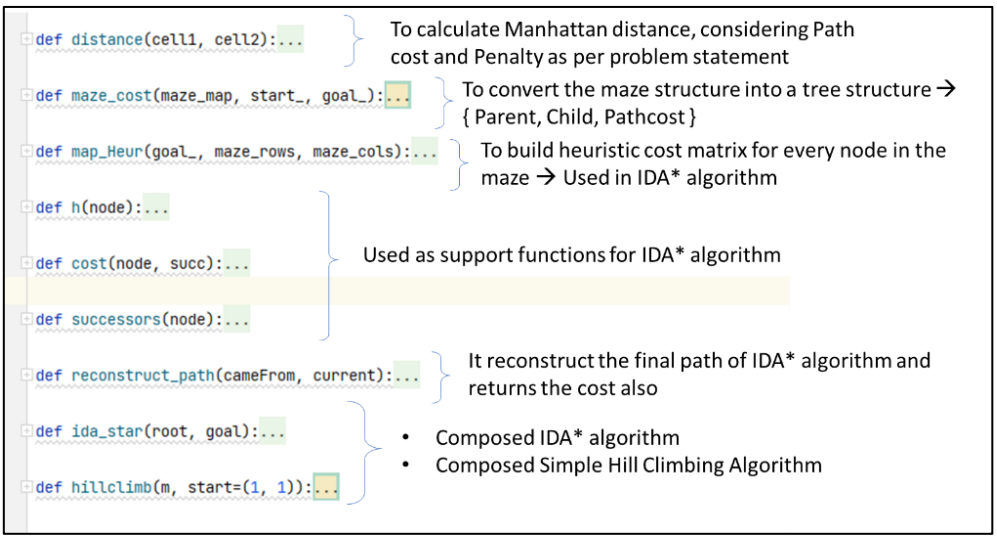


Figure 7 : Driver functions

➢ **Heuristic Function:**

Manhattan distance function is used to calculate heuristic value of every cell up to goal cell. Cost of every step as per the problem statement is given as '3'. Along with this, penalty of '5' is applied to every step taken in the East direction. As shown in the Figure 8, heuristic cost of start (source) cell i.e. (4,1) is calculated to be 48 (8 + 8 + 8 + 8 +8 + 8) and for goal cell (4,7) is calculated to be 0.

Heuristic function so defined is *admissible* since heuristic value of every cell is lesser than or equal to actual cost incurred from that cell to goal cell.

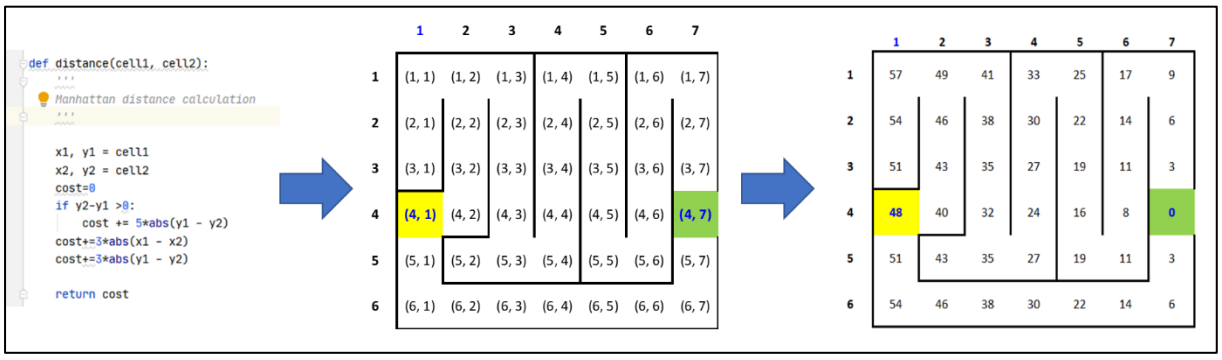


Figure 8: Heuristic Code and Heuristic Value Matrix corresponding to Maze Matrix as per Problem Statement

- **Main Function:**

  Initializing Start and goal position. Invoking the algorithms and GUI functions.

```
try:
    print('Press 1 - To Display IDA* Algorithm Path using GUI or Press 2 - To Display Hill CLimb Algorithm Path using GUI')
    #print('\n')
    switch_ = int(input())
    if switch_==1:
        print('Selected GUI is IDA* Algorithm')
    elif switch_==2:
        print('Selected GUI is Hill Climb Algorithm')

    # User can change this to try out different start cell
    start_ = (4, 1)

    # User can change this to try out different goal cell
    goal_ = (4, 7)

    if (start_ not in maze_map) or (goal_ not in maze_map):
        print('Invalid range of start or end cell')
        print('Please mention correct Start & End Cell')

    else:
        print('Start cell is', start_)
        print('Goal cell is', goal_)
        m=maze(maze_rows,maze_cols)

        # Creating maze as per given start and goal cell position
        m.CreateMaze(start_=(start_[0], start_[1]), fin_=(goal_[0], goal_[1]), loadMaze=1, maze_map_raw=maze_map)

        # IDA Star Algorithm Execution
        E = maze_cost(maze_map, start_, goal_) # Maze matrix converted to tree structure(E) for IDA* algorithm
        V = map_Heur(goal_, maze_rows=maze_rows, maze_cols=maze_cols) # Heuristic matrix
        bound, cameFrom, total_path, all_nodes, cost_idastar = ida_star(start_, goal_) # Calling IDA* algorithm function
        total_path.pop()
        total_path.reverse()

        idas_all_nodes = all_nodes.copy()
        idas_total_path = total_path.copy()

        # Simple Hill Climb Algorithm Execution
        searchPath, aPath, fwdPath, tot_cost = hillclimb(m, start=(start_[0], start_[1])) # Calling Hill Climbing algorithm funct

        shc_searchPath = searchPath.copy()
        shc_fwdPath = fwdPath.copy()

        invalid_input=0
        # As per GUI input selected (1 - for IDA* and 2 - for Hill Climbing)
        if switch_ == 1:
            print('Please check the GUI popup window for maze trace path, it will close automatically after 5 seconds')
            a = agent(m, start_[0], start_[1], footprints=True, color=COLOR.red, filled=FALSE)
            c = agent(m, start_[0], start_[1], footprints=True, color=COLOR.blue, filled=FALSE)
            m.tracePath({a: all_nodes}, delay=250,kill=False) # draw path in GUI with every step taking delay 150 miliseconds
            m.tracePath({c: total_path}, delay=50,kill=True) # draw final path in GUI with every step taking delay 50 miliseconds
            l = textLabel(m, 'IDA* Algorithm GUI','')
            m.run()
        elif switch_ == 2:
            print('Please check the GUI popup window for maze trace path, it will close automatically after 5 seconds')
            a = agent(m, start_[0], start_[1], footprints=True, color=COLOR.red, filled=FALSE)
            c = agent(m, start_[0], start_[1], footprints=True, color=COLOR.blue, filled=FALSE)
            m.tracePath({a: searchPath}, delay=250,kill=False) # draw path in GUI with every step tak
            m.tracePath({c: fwdPath}, delay=50,kill=True) # draw final path in GUI with every step tal
            l = textLabel(m, 'Hill Climbing Algorithm GUI','')
            m.run()
        else:
            print('No display using GUI')
except Exception as e:
    print('Error observed while processing',e)
```

Define Start and Goal Position

GUI for IDA*

GUI for Hill Climb

Figure 9: Main Function

IDA* Algorithm

IDA* stands for Iterative Deepening A Star Algorithm. IDA* algorithm is an informed search algorithm. It is a variant of iterative deepening depth-first search that borrows the idea to use a heuristic function to evaluate the remaining cost to get to the goal from the A* search algorithm. Since it is a depth-first search algorithm, its memory usage is lower than in A*, but unlike ordinary iterative deepening search, it concentrates on exploring the most promising nodes and thus doesn't go to the same depth everywhere in the search tree. Unlike A*, IDA* doesn't utilize dynamic programming and therefore often ends up exploring the same nodes many times.

As per given problem statement, maze matrix is converted to tree structure using function defined in the main code only as shown in Figure 10 : Maze as tree structure with path cost

Conversion of maze to tree structure

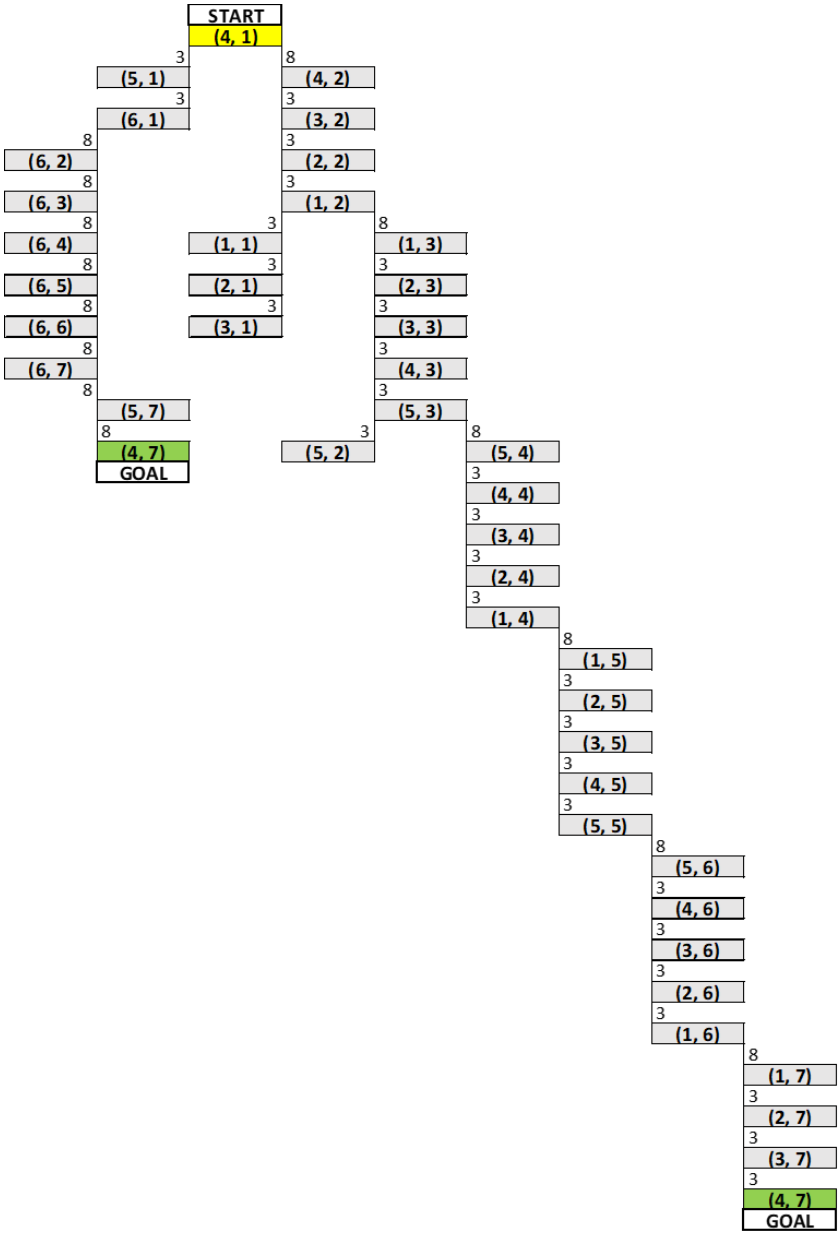
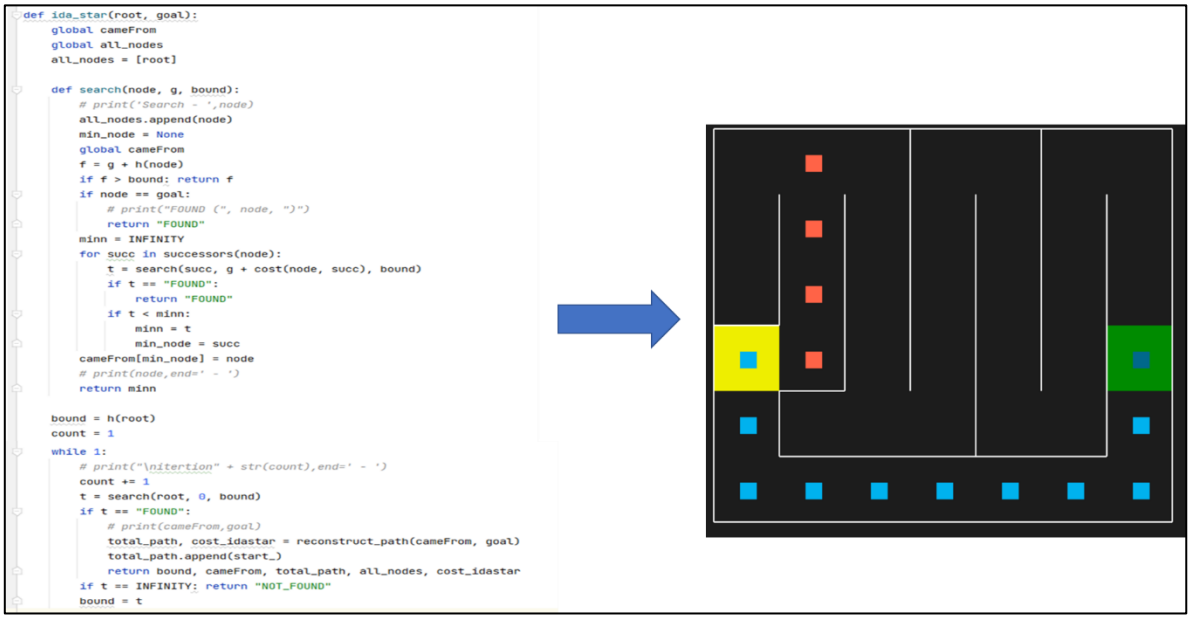


Figure 10 : Maze as tree structure with path cost

IDA* - Code Used and Its Output:

IDA* decides the depth using the f-score known as the 'threshold' which increases when a node with greater f-score is reached and the algorithm starts all over again from the beginning up to the new depth.

So, we use an infinite loop which is the base on recursion. The threshold is not just randomly increased but it depends on what the recursive function returns as the new threshold. During recursion whenever a node with higher f score than the threshold is reached, that node is not explored further, but the f-score is noted. Since we encounter many such nodes, the minimum of this f-score is returned as the new threshold. Code and GUI of IDA* is given in Figure 11



```
def ida_star(root, goal):
    global cameFrom
    global all_nodes
    all_nodes = [root]

    def search(node, g, bound):
        # print('Search - ',node)
        all_nodes.append(node)
        min_node = None
        global cameFrom
        f = g + h(node)
        if f > bound: return f
        if node == goal:
            # print("FOUND (", node, ")")
            return "FOUND"
        minn = INFINITY
        for succ in successors(node):
            t = search(succ, g + cost(node, succ), bound)
            if t == "FOUND":
                return "FOUND"
            if t < minn:
                minn = t
                min_node = succ
        cameFrom[min_node] = node
        # print(node,end=' - ')
        return minn

    bound = h(root)
    count = 1
    while 1:
        # print("\nitertion" + str(count),end=' - ')
        count += 1
        t = search(root, 0, bound)
        if t == "FOUND":
            # print(cameFrom,goal)
            total_path, cost_idastar = reconstruct_path(cameFrom, goal)
            total_path.append(start_)
            return bound, cameFrom, total_path, all_nodes, cost_idastar
        if t == INFINITY: return "NOT_FOUND"
        bound = t
```

Figure 11: IDA Algorithm Main Code and GUI with result keeping Start (4,1) and Goal (4,7)*

Optimal Path Sequence and Cost:

```
------------------------- IDA* Algo ---------------------------------------
As per IDAStar Algo, Optimal/Best Possible Path Sequence:
 [(4, 1), (5, 1), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (5, 7), (4, 7)]
As per IDAStar Algo, Total Cost of final path: 60

---------------------------------------------------------------------------
```

Figure 12: Optimal Path and cost as per given problem statement

IDA* - Applications:

- Industrial AI planning
- Solving games for example shortest path in a maze or on a map, solving the 15-puzzle game, and solving Sudoku puzzles
- Shortest path problems with large search space

IDA* - Pros and Cons:

Pros:

- It will always find the optimal solution provided that it exists and that if a heuristic is supplied it must be admissible
- Heuristic is not necessary; it is used to speed up the process
- Various heuristics can be integrated to the algorithm without changing the basic code
- The cost of each move can be tweaked into the algorithms as easily as the heuristic
- Uses a lot less memory which increases linearly as it doesn't store and forgets after it reaches a certain depth and start over again

Cons:

- Doesn't keep track of visited nodes and thus explores already explored nodes again
- Slower due to repeating the exploring of explored nodes
- Requires more processing power and time than A*

IDA* - Time and Space complexity

Time complexity of the algorithm: The runtime complexity of Iterative Deepening A Star is in principle the same as Iterative Deepening DFS. It is determined by the number of nodes expanded during the search process. In the worst-case scenario, IDA* will expand all nodes in the search space, which is equivalent to a depth-first search. Therefore, the worst-case time complexity of IDA* is **O(b^d)**, where b is the branching factor and d is the depth of the solution.

Space complexity of the algorithm: The space complexity of IDA* is determined by the number of nodes stored in memory during the search process. IDA* uses an iterative deepening strategy, so at any given point in time, only the nodes on the current depth level are stored in memory. Therefore, the space complexity of IDA* is **O(b*d)**, where b is the branching factor and d is the current depth level.

Hill Climbing Algorithm

Hill Climbing is a heuristic search used for mathematical optimisation problems in the field of Artificial Intelligence. So, given a large set of inputs and a good heuristic function, the algorithm tries to find the best possible solution to the problem in the most reasonable time period. This solution may not be the absolute best (global optimal maximum) but it is sufficiently good considering the time allotted. At any point, the search moves in that direction, which optimises the cost of function with the hope of finding the most optimum solution at the end.

There are several variations of the hill climbing algorithm, including: **Steepest Ascent Hill** Climbing: This algorithm moves to the neighbour that has the highest value of the function. **Random-Restart Hill Climbing***:* This algorithm restarts the search from a random point if it gets stuck in a local maximum. **Simulated Annealing**: This algorithm also allows for some "downhill" moves, in order to escape local maxima. Hill climbing is relatively simple to implement and can find good solutions quickly for some problems, but it has some limitations. It can get stuck at local maxima, and it does not guarantee that the global maximum or minimum will be found. It's also computationally expensive in high dimensions.

Code Used and output:

Hill climbing function is based on the principle of selection of neighbour based on minimum heuristic cost value.

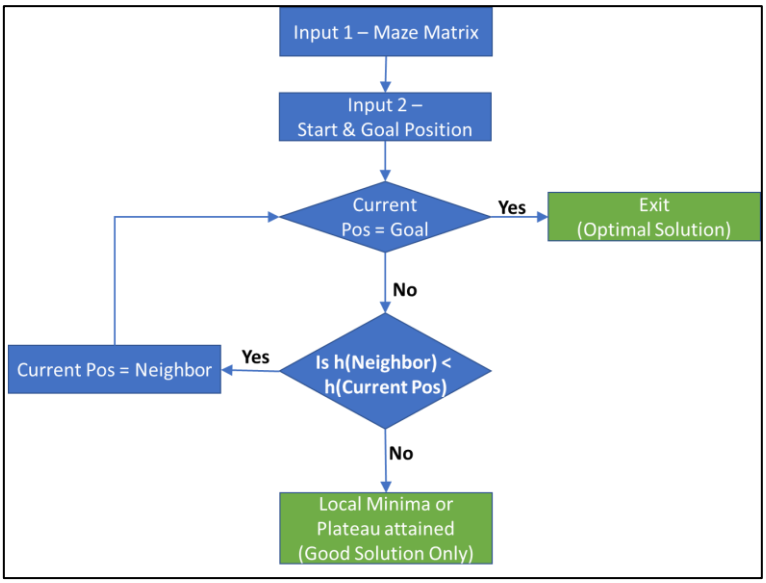


Figure 13: Flowchart of Simple Hill Climb Algorithm used in ACI Assignment

Since hill climbing can get stuck at local minima (heuristic cost), Figure 14 shows that algorithm is stuck at (4,2). Neighbours of (4,2) are (3,2) with heuristic cost 43 and (4,1) which is the source cell with heuristic cost 48. That is why the hill climb algorithm could not reach the goal cell.

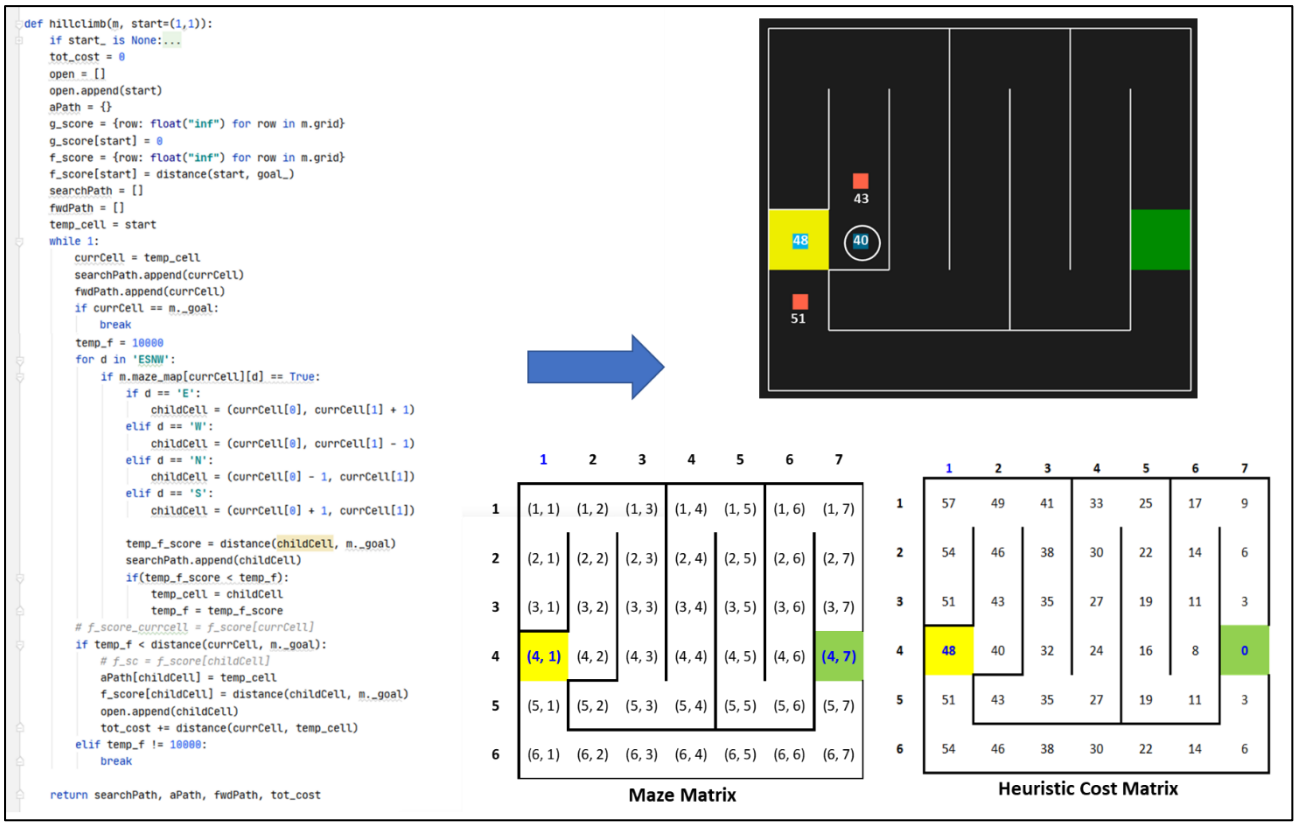


Figure 14: Simple Hill Climb Algorithm Code and GUI as per Problem Statement

Optimal Path Sequence and Cost:

```
---------------------- Simple Hill Climb Algo -------------------------
As per Simple Hill Climb Algo, Optimal/Best Possible Path Sequence:
 [(4, 1), (4, 2)]
As per Simple Hill Climb Algo, Total Cost of final path: 8
-----------------------------------------------------------------------
```

Figure 15: Simple Hill Climb Algorithm Best Possible Path and Cost

Applications of Hill Climbing Technique

- Hill Climbing technique can be used to solve many problems, where the current state allows for an accurate evaluation function, such as Network-Flow, Travelling Salesman problem, 8-Queens problem, Integrated Circuit design, etc.
- Hill Climbing is used in inductive learning methods and in robotics for coordinating multiple robots in a team.
- It is also used in various application such as engineering, physics, economics, and statistics

Hill Climbing - Pros and Cons:

Pros:

- Hill climbing algorithm is easy to implement and understand
- It can quickly find good solutions for some problems, particularly those with a small search space and a smooth objective function
- It is efficient in terms of memory usage as it only needs to keep track of the current and next state

Cons:

- Hill climbing algorithm can get stuck in local maxima or minima, meaning it may not find the global optimal solution
- The quality of the solution found depends heavily on the starting point
- Hill climbing algorithm is sensitive to the quality of the heuristic function. If heuristic function is not well designed, the algorithm may converge to suboptimal solutions.
- It is not suitable for problems with large or complex search spaces
- It is computationally expensive in high dimensions

Hill Climbing - Time and Space complexity:

- **Time complexity of algorithm:** The time complexity of hill climbing algorithms depends on the number of steps required to find a local maximum (or minimum) and the time required to evaluate the function and generate the next state. For simple hill climbing algorithm, the worst-case time complexity is O(d*b^m) where d is the maximum depth of the search tree, b is the branching factor and m is the number of iterations. For our case, using simple hill climb algorithm, m is 1 since, once no good neighbour is available in comparison to current cell's heuristic value, algorithm returns the best possible path.

- **Space complexity of the algorithm:** The space complexity of hill climbing algorithms is determined by the number of states stored in memory during the search process. Simple hill climbing algorithm, have a space complexity of O(d) where d is the maximum depth of the search tree. This is because at any given point in time, only the current state and the next state need to be stored in memory.

Test with different start and goal nodes on the same maze

1. Start (1,1) and Goal (4,7)

```
------------------------ IDA* Algo ---------------------------------------
As per IDAStar Algo, Optimal/Best Possible Path Sequence:
 [(1, 1), (1, 2), (2, 2), (3, 2), (4, 2), (4, 1), (5, 1), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (5, 7), (4, 7)]
As per IDAStar Algo, Total Cost of final path: 80
--------------------------------------------------------------------------

---------------------- Simple Hill Climb Algo -------------------------
As per Simple Hill Climb Algo, Optimal/Best Possible Path Sequence:
 [(1, 1), (1, 2), (1, 3), (2, 3), (3, 3), (4, 3)]
As per Simple Hill Climb Algo, Total Cost of final path: 25
--------------------------------------------------------------------------


------------------------ IDA* Algo ---------------------------------------
As per IDAStar Algo, Space Complexity: 16 Nodes
As per IDAStar Algo, Time Complexity: 136 Nodes
--------------------------------------------------------------------------

---------------------- Simple Hill Climb Algo -------------------------
As per Simple Hill Climb Algo, Space Complexity: 6 Nodes
As per Simple Hill Climb Algo, Time Complexity: 19 Nodes
--------------------------------------------------------------------------
```

IDA* Algorithm Output **Simple Hill Climb Algorithm Output**

Figure 16: Test Case 1 : Start Cell (1,1) and Goal Cell (4,7)

2. Start (6,2) and Goal (2,7)

```
------------------------ IDA* Algo ---------------------------------------
As per IDAStar Algo, Optimal/Best Possible Path Sequence:
 [(6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (5, 7), (4, 7), (3, 7), (2, 7)]
As per IDAStar Algo, Total Cost of final path: 52
--------------------------------------------------------------------------

---------------------- Simple Hill Climb Algo -------------------------
As per Simple Hill Climb Algo, Optimal/Best Possible Path Sequence:
 [(6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (5, 7), (4, 7), (3, 7), (2, 7)]
As per Simple Hill Climb Algo, Total Cost of final path: 52
--------------------------------------------------------------------------


------------------------ IDA* Algo ---------------------------------------
As per IDAStar Algo, Space Complexity: 10 Nodes
As per IDAStar Algo, Time Complexity: 11 Nodes
--------------------------------------------------------------------------

---------------------- Simple Hill Climb Algo -------------------------
As per Simple Hill Climb Algo, Space Complexity: 10 Nodes
As per Simple Hill Climb Algo, Time Complexity: 28 Nodes
--------------------------------------------------------------------------
```

Figure 17: Test Case 2 : Start Cell (6,2) and Goal Cell (2,7)

Comparison of IDA* and Hill Climb Algorithm:

IDA* (Iterative Deepening A*) and hill climbing are both search algorithms used to find solutions to problems, but they have some key differences.

IDA* is a variant of the A* algorithm and is used for solving problems in which the goal state is not known in advance. It works by repeatedly applying a depth-first search with a cost bound, increasing the bound each time the search fails to find a solution. IDA* is considered to be both complete and optimal, meaning that it will always find a solution if one exists and that the solution it finds will be the best possible.

Hill climbing, on the other hand, is a type of local search algorithm that starts with an initial solution and repeatedly makes small changes to the solution in an attempt to improve it. It stops when it reaches a local maximum, which may not necessarily be the global maximum. Hill climbing is not guaranteed to find the best solution and can get stuck in local optima.

Conclusion:

To conclude that, out of the 2 given algorithms, we could say that IDA* is the optimal and best solution for this problem. We tested it well with mxn (6x7) maze board, with agent's start position at cell position (4,1) and goal cell as (4,7). Maze is constructed exclusively to meet the exact requirement as per problem statement. Start and end positions can be dynamic on the board. Few were tested and worked well as expected.

We are thankful to BITS, for giving us the opportunity to learn and explore the algorithms discussed inside the class. It helped us collaborate and complete the assignment well under the given time line and achieve the desired output. We are sure and confident enough to learn and solve complex problems in upcoming assignments.

Thank you