**Table of contents**

[**PEAS Environment** 2](#_Toc196752530)

[**Task Environment** 2](#_Toc196752531)

[**Choice of algorithms** 3](#_Toc196752532)

[**Complexity Analysis** 3](#_Toc196752533)

[**Data Structures** 4](#_Toc196752534)

[**Algorithm 1: Iterative Deepening A\*** 4](#_Toc196752535)

[**Heuristic function for IDA\*** 4](#_Toc196752536)

[**Fitness function for IDA\*** 5](#_Toc196752537)

[**Fringe List** 5](#_Toc196752538)

[**Step by step code flow** 5](#_Toc196752539)

[**IDA\* Output** 8](#_Toc196752540)

[**Algorithm 2: Hill Climbing** 9](#_Toc196752541)

[**Heuristic function for Hill Climbing** 10](#_Toc196752542)

[**Fitness function for Hill Climbing** 10](#_Toc196752543)

[**Step by step code flow** 10](#_Toc196752544)

[**Hill Climbing Output** 11](#_Toc196752545)

[**Conclusion** 12](#_Toc196752546)

## **PEAS Environment**

|  |  |
| --- | --- |
| **Performance** | * The robot must efficiently locate and reach the dirtiest spot in the room. * Minimize the energy consumed (number of moves). * Terminate at the global maximum (highest dirt value, 9 at cell (2,2)). * Avoid unnecessary moves to adjacent cells if a local maximum is reached. |
| **Environment** | * A 2D grid representing the room, where:   + Each cell contains a value representing the dirt amount.   + The robot can only move up, down, left, or right (no diagonal movement). * The robot starts at (0, 0). * Goal: Find and stop at the cell with the highest dirt value. |
| **Actuators** | Robot wheels which perform the following actions: *Move Up, Move Down, Move Right, Move Left, Suck, No operation* |
| **Sensors** | Robot has optical sensors which reads the location of adjacent cells and detect dirt |

## **Task Environment**

|  |  |
| --- | --- |
| **Fully vs Partially Observable** | **Fully observable**  The robotic vacuum cleaner can sense the dirt value of its current cell and the dirt values of all adjacent cells. It has complete information about the immediate environment needed to decide. There are no hidden or unknown states in the grid for the robot. Hence it is fully observable. |
| **Single vs Multi Agent** | **Single Agent**  The problem involves only one robotic vacuum cleaner navigating the grid. There are no other agents involved in it. |
| **Deterministic vs Stochastic** | **Deterministic**  The outcome of each action (move up, down, left, or right) is predictable and deterministic.  For example, if the robot moves from (0, 0) to (1, 0), it will always end up at (1, 0) with no randomness involved. The dirt values in the grid remain constant and do not change over time. |
| **Episodic vs Sequential** | **Sequential**  Each action taken by the robot (moving to a new cell) depends on the previous actions and affects future decisions. For example, the robot’s path to the dirtiest spot is influenced by its current position, so the problem is sequential. |
| **Static vs Dynamic** | **Static**  The environment (the grid and dirt values) does not change while the robot is operating. The dirt values are fixed, and there are no moving obstacles or external changes to the grid during the robot’s operation. |
| **Discrete vs Continuous** | **Discrete**  The robot operates in a discrete grid environment where the positions and actions are clearly defined (e.g., moving from (0, 0) to (1, 0) involves a single discrete step).  There are no continuous movements or infinitely varying states. |

## **Choice of algorithms**

We can use the following two algorithms to solve the given problem statement:

1. Iterative Deepening A\*
2. Hill Climbing

## **Complexity Analysis**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Space Complexity** |
| Iterative Deepening A\* | ) where b is the branching factor and d is the depth of the goal | O(d) as it stores only nodes in the current path |
| Hill Climbing | O(b.d) where b is the branching factor and d is the depth of the goal | O(1) as it stores only the current state |

## **Data Structures**

1. **Grid Representation:** A 2D list to represent the room
2. **Fringes of A\*:** Priority queue to manage the states sorted by cost plus heuristic
3. **Visited States:** A set to store visited cells

## **Algorithm 1: Iterative Deepening A\***

The Iterative Deepening A\* (IDA\*) algorithm is a graph search algorithm that combines the depth-first search approach with iterative deepening, ensuring optimal pathfinding using an adaptive cost threshold. It is particularly useful in grid-based pathfinding scenarios where a goal state is defined by a specific criterion, such as reaching the maximum value in the grid.

**Algorithm Overview:**

**Identify the Goal State:** The goal state is determined by finding the cell in the grid that has the highest value.

**Initialize the Threshold:** The threshold is set as the heuristic cost from the start position to the goal, computed using the Manhattan distance.

**Perform Depth-First Search with Iterative Deepening:**

* The algorithm recursively expands nodes while ensuring the cost does not exceed the current threshold.
* If a solution is found within the threshold, it is returned.
* If not, the minimum exceeded cost becomes the new threshold, and the search is repeated

### **Heuristic function for IDA\***

If we use informed search, we can use the following formula to get the heuristic function:

h(n) = Manhattan distance to the goal node from the current node

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** |
| **0** | 4 | 3 | 2 | 3 |
| **1** | 3 | 2 | 1 | 2 |
| **2** | 2 | 1 | 0 | 1 |
| **3** | 3 | 2 | 1 | 1 |

### **Fitness function for IDA\***

f(n) = g(n) + h(n) where g(n) is the number of moves from the current position to the goal node

### **Fringe List**

|  |  |  |
| --- | --- | --- |
| **Threshold** | **Fringe List** | **Goal State Reached?** |
| 4 | {(0, 1), (1, 0)} | No |
| 6 | {(0, 2), (1, 1), (2, 0)} | No |
| 8 | {(0, 3), (1, 2), (2, 1), (3, 0)} | No |
| 9 | {(1, 3), (2, 2), (3, 1)} | Yes |

### **Step by step code flow**

1. Find the goal state (the cell with the maximum value in the grid).

|  |
| --- |
| goal = None  max\_val = float('-inf')  for i in range(len(grid)):  for j in range(len(grid[0])):  if grid[i][j] > max\_val:  max\_val = grid[i][j]  goal = (i, j) |

grid 2D list is iterated in the nested for loop to find the maximum value (maximum dirt in a cell)

1. Initialize the threshold to the heuristic value of the start position.

|  |
| --- |
| threshold = heuristic(start, goal) |

heuristic is a function which computes the heuristic value using the Manhattan distance between the specified node and goal node. In the beginning, the initial threshold is calculated between the start node and the goal node. **heuristic** function is given below:

|  |
| --- |
| def heuristic(pos, goal):  return abs(pos[0] - goal[0]) + abs(pos[1] - goal[1]) |

1. Iteratively deepen the search until a solution is found or no more paths exist.

|  |
| --- |
| while True:  result = search([start], 0, threshold)  if isinstance(result, list):  return result, len(result), threshold  if result == math.inf:  return None, None, None  threshold = result |

Start the search with the initial threshold. If the path is found, then return the path, cost and threshold. Otherwise, return None for path, cost and threshold if no solutions found.

**search** function is explained below:

|  |
| --- |
| def search(path, g, threshold):  current = path[-1]  f = g + heuristic(current, goal)    if f > threshold:  return f  if is\_goal(current, grid):  return path  min\_threshold = math.inf  for successor in get\_successors(current, grid):  if successor not in path:  result = search(path + [successor], g + get\_cost(current, successor), threshold)  if isinstance(result, list):  return result  min\_threshold = min(min\_threshold, result)  return min\_threshold |

Get the current position from the end of the path. Calculate the fitness function using the formula f(n) = g(n) + h(n). If f exceeds the threshold, return the fitness value as the updated threshold. If the current position is the goal, return the path as the solution.

Then, initialize the minimum threshold for the next iteration to infinity. Explore all valid successor states (neighbors) by calling **get\_successors** function and iterating through the successors. Avoid revisiting nodes in the current path and call recursively to search from the successors by passing next path, next cost function and threshold.

If a path is found, return it and also update the minimum threshold. If path is not found, then return the updated threshold for the next iteration.

**get\_successors** function is explained below:

|  |
| --- |
| def get\_successors(state, grid):  x, y = state  successors = []  for dx, dy in directions:  nx, ny = x + dx, y + dy  if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]):  successors.append((nx, ny))  return successors |

Extract the current position's coordinates. Initialize an empty list to store valid successors. Iterate over all possible movement directions such as up, down, right and left. Calculate the new position after applying the direction. Check if the new position (nx, ny) is within the grid's bounds. If the position is valid, add it to the successors list. Return the list of all valid successor positions.

**get\_cost** function is explained below. This is called to get cost and add to the current cost value to find the next cost value.

|  |
| --- |
| def get\_cost(state1, state2):  return 1 |

The cost function gets the uniform cost of moving between adjacent cells in the grid (room).

**is\_goal** function is called to evaluate if the current node is the same as the goal node. This function is explained below:

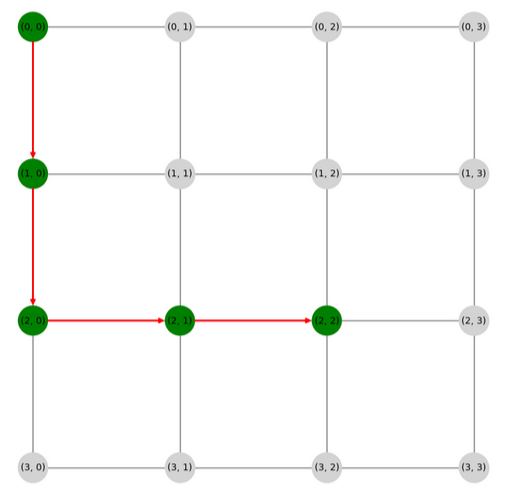
|  |
| --- |
| def is\_goal(state, grid):  max\_val = max(max(row) for row in grid)  return grid[state[0]][state[1]] == max\_val |

This function gets the maximum value in a row from the grid and then maximum from all the maximums found. If the current value is the maximum value, then true is returned to specify that goal was reached. Otherwise, false is returned.

### **IDA\* Output**

For a starting position (0, 0), the following output is captured. The output will vary based on the user input for the starting position.

|  |  |
| --- | --- |
| **Path** | [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2)] |
| **Cost** | 5 |
| **Threshold** | 4 |
| **Execution Time** | 0.000123 seconds |



## **Algorithm 2: Hill Climbing**

The Hill Climbing algorithm is a heuristic search algorithm used to find a local maximum in a search space. It is particularly effective in scenarios where a greedy approach can be applied to iteratively improve a solution. In the context of this implementation, the Hill Climbing algorithm is used to find the position with the highest dirt value in a 2D grid starting from a given position.

**Algorithm Overview:**

**Start from the Initial Position:** The algorithm begins at the specified starting position in the grid.

**Evaluate Neighbors:** For the current position, the algorithm evaluates all valid neighbors (adjacent cells in the grid).

**Select the Best Successor:** Among the neighbors, the one with the highest dirt value is chosen as the next position.

**Check for Termination:** The algorithm terminates when no valid successors are available (dead-end) and the dirt value of the best successor is less than or equal to the current position (local maximum).

### **Heuristic function for Hill Climbing**

h(n) = Amount of dirt at each cell

### **Fitness function for Hill Climbing**

Same as the heuristic function for Hill Climbing. That is, the amount of dirt at each cell

### **Step by step code flow**

1. Initialize the current position to the starting position.

|  |
| --- |
| current = start |

1. Initialize the path to include the starting position.

|  |
| --- |
| path = [current] |

1. Start the Hill Climbing process.

|  |
| --- |
| while True:  successors = get\_successors(current, grid)  next\_state = max(successors, key=lambda s: grid[s[0]][s[1]], default=None)  if next\_state is None or grid[next\_state[0]][next\_state[1]] <= grid[current[0]][current[1]]:  break  current = next\_state  path.append(current) |

Get all valid successors (neighbors) of the current position by calling [**get\_successors**](#241ywjuizukz)function. Select the neighbor with the maximum dirt value. If there are no successors, default to None.

If the following termination conditions are satisfied, terminate the search (local maximum reached) : [1] If there are no valid successors (next\_state is None). [2] If the best successor's value is not greater than the current cell's value.

Move to the best successor (next\_state) and add it to the path.

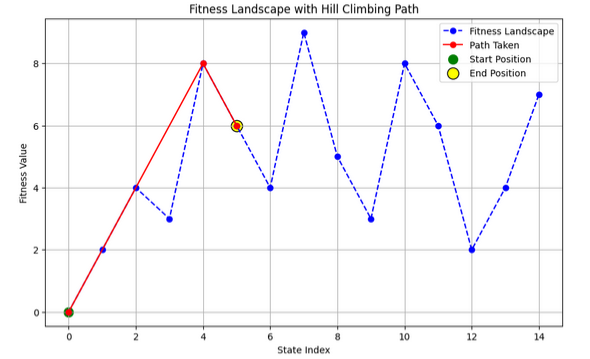
1. Return the path, number of steps, and the dirt value at the final position.

|  |
| --- |
| return path, len(path), grid[current[0]][current[1]] |

### **Hill Climbing Output**

For a starting position (0, 0), the following output is captured. The output will vary based on the user input for the starting position.

|  |  |
| --- | --- |
| **Path** | [(0, 0), (1, 0), (1, 1)] |
| **Cost** | 3 |
| **Local Maxima** | 8 |
| **Execution Time** | 0.000202 seconds |



## 

## **Conclusion**

IDA\* is an effective search algorithm for large graphs and grid environments where memory efficiency is crucial. It combines the benefits of depth-first search with cost-based pruning, ensuring optimality while limiting resource usage.

The Hill Climbing algorithm is a fast and memory-efficient approach for finding local maxima in a grid. It is well-suited for problems with simple heuristics but may require enhancements (like random restarts or simulated annealing) to avoid local optima in complex search spaces.