Group Assignment-3 (Hill Climb)
Members:

- Avinash Aanand-(Roll No.: 2403RES99)
- Aditya Gupta-(Roll No.:2403RES85)
- Aman Kumar-(Roll No.:2403RES10)
- Sanjeev Kumar (Roll No.: 2403RES117)

- Due Date: April 2024
- Submitted To: Dr. Asif Ekbal

Artificial Intelligence & Machine Learning Lab, CS561/571 Assignment 03

Hill Climbing

Table of Content

Problem Statements: 2
Statement 2
Questions: 2
Instructions 2
Solutions (Questions): 2
Tasks 4
Heuristics: 5
Hill Climb Feasibility: 5
Introduction: 6
Analysis of Hill Climb 6
Comparatively Table Breakdown (BFS/DFS/A*/Hill Climb) 7
Hill Climb Search Properties 7
Hill Climb Analysis 8
Through Pseudo Code 8
Hill Climb Algorithm 8
Analysis Through Code 9
Hill Climb Execution: 10
Real time Implementation (In Python) 10
Introduction: 10
HillClimb Implementation: 11
Post Code Analysis 12
Analysis of Hill Climb Algorithm: 12
Results/Output 12
Welcome Window 12
Heuristics Outputs 12
Observation /Conclusion 14

Problem Statements:

Statement

- The assignment targets to implement Hill Climb search for the 8-puzzle problem.

Questions:

A local search algorithm tries to find the optimal solution by exploring the states in the local region. Hill climbing is a local search technique that constantly looks for a better solution in its neighborhood.

- Implement the Hill Climbing Search Algorithm for solving the 8-puzzle problem.
- Check the algorithm for the following heuristics:
    - h1(n) = number of tiles displaced from their destined position.
    - h2(n) = sum of the Manhattan distance of each tile from the goal position.

Instructions:

- Take the input and store the information in a matrix. Configuration of the start and goal states are random (try 100 times, stop on 1st success). For example, T1, T2, …, and T8 are tile numbers, and B is blank space. (Success is when the goal is reachable from the initial state).
- The output should have the following information for each h1 and h2:
    - On success:
        - Success Message
        - Start State / Goal State
        - Total number of states explored
        - Total number of states to the optimal path using h1 h2
        - Optimal Path
    - On failure:
        - Failure Message
        - Start State / Goal State
        - Total number of states explored before termination

Solutions (Questions):

Case:1

In a general search algorithm, each state (n) maintains a function f(n) = g(n) + h(n) where g(n) is the least cost from the source state to state n found so far and h(n) is the estimated cost of the optimal path from state n to the goal state.

- Key Components

f(n): Estimated Total Path Cost

The core idea is to estimate the total cost of a solution path that goes through node 'n'. This helps the algorithm prioritize which nodes to explore.

g(n): Actual Cost So Far

This is the exact cost incurred to reach node 'n' from the starting state by following the path discovered so far. It's based on real moves made.

h(n): Estimated Cost to Goal (Heuristic)

This is the "educated guess" part. It's an estimate of the cost to reach the goal state from node 'n'. A good heuristic is crucial for guiding the search in the right direction.

How f(n) Guides A* Search

- Imagine we're trying to solve a maze. Here's how f(n), g(n), and h(n) might interact with a search algorithm:
- Prioritization: At each step, the search algorithm needs to decide which node to explore next. It picks nodes with the lowest f(n) value.

Balance:

- Low g(n): Means the node is close to the start. This represents progress already made.
- Low h(n): Means the node *seems* to be closer to the goal (based on the heuristic). This represents potential for future progress.
- f(n) finds a balance between exploiting known progress (g(n)) and exploring promising directions (h(n)).

- Why This Works

Focus:

- The search is biased towards nodes that seem to be on paths likely to yield low-cost solutions.
- Heuristics are Key: The quality of the heuristic function h(n) significantly impacts the search.
- A good heuristic directs the search towards the goal efficiently.
- A poor heuristic can mislead the search and waste time exploring unpromising areas.

Example (Simplified)
Imagine a simple graph search. Let's say:
Cost of moving between any two nodes = 1
We can provide h(n) as the straight-line distance to the goal
Scenario:

- Node A: g(A) = 3 (3 steps from the start), h(A) = 5 (estimated distance to goal) -> f(A) = 8
- Node B: g(B) = 5 (5 steps from the start), h(B) = 2 (seems closer to the goal) -> f(B) = 7

In this case, the algorithm would likely choose to explore Node B next, even though it's further from the start, because its lower f(n) indicates a higher potential for a shorter overall path.

Case:2:

8-Puzzle:

- The 8-puzzle is a classic sliding puzzle with a 3x3 grid. It has eight numbered tiles and one empty space. we slide tiles to re-arrange them into the goal configuration.

State Representation:

- In the code, a state is likely represented as a list or an array like this: `[1, 2, 3, 4, 5, 6, 8, 0, 7]` (where 0 represents the empty space).

Start and Target States:

- We need to define these based on the problem instance.

Search Algorithm

- The `informed_astar_search` function in the provided code likely implements the core A* search algorithm. Here's how to adapt it:

Move Generation:

- The `get_possible_moves` function already handles generating valid moves from a given puzzle state (up, down, left, right).

Open/Closed Lists:

- A* uses an `open_list` for nodes to be explored, and a `closed_set` to avoid re-examining states. The provided code should have structures for these.

Termination:

Success:

- Check if the current state matches the goal state during each iteration of the search loop.

Failure:

- If the `open_list` is empty and no solution is found, the search fails.

Calculating g(n)

- In the 8-puzzle, assuming every move has a cost of 1, g(n) is straightforward:

g(n) = Depth in the search tree. Each move away from the start state increases the cost by 1.

Heuristics

- The provided code has implementations for these heuristics:
  - h1(n) = 0: Always returns zero (`heuristic_function_1`)
  - h2(n) = Number of misplaced tiles: Counts how many tiles are not in their goal positions (`heuristic_function_2`)
  - h3(n) = Sum of Manhattan Distances: Calculates the Manhattan distance of each tile to its goal position and sums those distances (`heuristic_function_3`)
  - h4(n) = h3(n) + 1: Adds 1 to h3(n) (`heuristic_function_4`)

Devising h(n) > h(n)*

- h(n): * The true optimal cost to reach the goal state from node n. This is generally unknown during the search.

Overestimating Heuristic:

- To make h(n) > h*(n), we need to inflate the estimate. One way could be:

Multiply h3(n) by a factor:

- For example, `h(n) = 2 * h3(n)`. This might lead to finding the solution faster but sacrifices the guarantee of optimality.

Random States (if needed)

- The `random_array` function likely provides this functionality. Here's how to integrate it:

Check if solution possible:

- There's some theory about which 8-puzzle configurations are solvable. We might want to implement a solvability checker.

Loop until solvable:

- Generate a random start state.
- Attempt to solve it using A*.
- If a solution is not found, regenerate the start state

Tasks

Task 1: Algorithm Implementation

Data Representation:

- Choose a suitable programming language (e.g., Python, Java, C++).
- Define a data structure to represent the 8-puzzle board (e.g., a 3x3 matrix).

Input Handling:

- Design a function to take user input for the initial state and goal state.
- Implement a mechanism for random state generation (start and goal).

Success Check:

- Create a function that determines if the goal state has been reached.

Neighbor Generation:

- Write a function to identify all possible moves from a given state (find the blank tile and potential swap positions).

Heuristic Calculations:

- Implement functions to calculate h1(n) and h2(n) for a given state.

Hill Climbing Core:

- Write the main hill climbing function:
- Start at the initial state.

For each neighbor:

- Calculate its heuristic value.
- If the neighbor is better than the current state, move there.
- If stuck in a local optimum, consider variations like random restarts.
- Task 2: Heuristics Evaluation

Experiment Loop:

- Set up a loop to run the hill climbing algorithm 100 times (or until the first success) with random starting states.

Record Data:

On Success:

- Print a clear success message.
- Log the initial state and goal state.
- Store the total number of states explored.
- Store the path length to the optimal solution (for both h1 and h2).
- Visualize the optimal path (print the board at each step).

On Failure:

- Print a clear failure message.
- Log the initial state and goal state.

- Store the total number of states explored.

Reachability Check:

- Include a function to verify if the goal state is even solvable from the initial state (this will help filter out impossible cases in the experiments).

Other Considerations

- Code Structure: Organize your code into well-defined functions with comments for readability.
- Testing: Include test cases to verify the correctness of different components.

Heuristics:

Heuristics holds crucial in hill climbing

- Efficiency: Hill climbing only explores the immediate neighborhood of the current state. Heuristics help prioritize the search direction, leading to faster convergence towards a solution.
- Overcoming Local Optima: Hill climbing can get stuck in local optima (a solution better than its neighbors but not the global best solution). A well-designed heuristic offers a chance to escape these plateaus and find better solutions elsewhere in the search space.
- Common Heuristic Types for Hill Climbing

Problem-Specific Heuristics:

8-Puzzle Problem:

- Number of misplaced tiles
- Sum of Manhattan distances of tiles from their goal positions
- Traveling Salesperson Problem:
- Sum of distances between consecutive cities on the tour
- General Heuristics:
- Relaxed Problem: Calculate the solution cost for a simplified version of the problem (a heuristic can be admissible if it never overestimates the cost to reach the goal).
- Heuristic Design Considerations
- Accuracy: A more accurate heuristic typically leads to better performance.
- Computational Cost: The heuristic should be relatively fast to calculate, as it will be evaluated repeatedly during the search.
- Balance: Finding the right balance between accuracy and computational efficiency is key.
- Domain Knowledge: Leveraging domain-specific knowledge often helps design highly effective heuristics.

Important Note:

- While good heuristics are immensely helpful, they cannot guarantee that hill climbing will find the global optimal solution. Hill climbing remains susceptible to local optima.

Hill Climb Feasibility:

Strengths of Hill Climbing

- Simplicity: Hill climbing is relatively easy to understand and implement. Its core concept of iteratively exploring better neighbors makes it straightforward.
- Efficiency: With a well-chosen heuristic, hill climbing can be computationally efficient, especially for problems with a large search space. It focuses on local improvements, reducing the number of states that need to be explored.
- Memory Friendly: Hill climbing generally has a low memory footprint as it primarily needs to store the current state and its neighbors.

Limitations of Hill Climbing

- Local Optima: The fundamental weakness of hill climbing is its susceptibility to getting stuck in local optima. Once it finds a state better than all its neighbors, it might end the search even if a much better global solution exists elsewhere.
- Plateaus: Plateaus, where neighboring states have the same heuristic value, can stall progress and require modifications like random restarts

- Ridges: Ridges, where a sequence of moves must occur in a specific order to improve, can be challenging for hill climbing's step-by-step approach.
- Guarantee of Solution: Hill climbing does not guarantee finding an optimal solution, or even finding a solution at all if one exists.

When Hill Climbing is Feasible

- Problem Complexity: Appropriate for problems were finding a "good enough" solution quickly is more important than guaranteeing the absolute best solution.
- Resource Constraints: Useful when memory or computational resources are limited.
- Dynamic Environments: Suitable for problems where the environment or goal state changes, requiring fast adaptation.
- Initial Step: Can be used as a starting point for more complex optimization techniques or paired with strategies like random restarts to improve its exploration capabilities.

When Hill Climbing is Less Desirable

- Guarantee Required: Not ideal when finding the global optimal solution is crucial.
- Known Search Space Structure: If the search landscape is well-understood, more sophisticated algorithms that avoid local optima can be superior.

Introduction:

The 8-Puzzle

Problem Statement:

- A 3x3 grid contains eight numbered tiles and one empty space. The goal is to rearrange the tiles, using the empty space for movement, to reach a specific goal configuration.

Moves:

- We can slide any tile adjacent to the empty space (up, down, left, right) into that space.

Representing the puzzle

We can represent the state of the puzzle in various ways:

- **List or array**: A simple list of 9 numbers represents the tile configuration, where 0 (or a blank character) denotes the empty space.
- **Matrix**: A 3x3 matrix can also be used.

Search Algorithms: Hill Climbing

Analysis of Hill Climb

Weaknesses

- Local Optima: The core weakness is its tendency to get trapped in local optima. Once it finds a solution better than all its immediate neighbors, it might end the search prematurely, missing potentially better solutions elsewhere in the search space.
- Plateaus: Neighboring states with the same heuristic value can stall the algorithm's progress.
- Ridges: Situations where improvement requires a sequence of moves with no intermediate gain can be challenging for the incremental nature of hill climbing.
- No Guarantee of Optimality: In general, hill climbing does not guarantee finding the global optimum, or even a solution at all if one exists.

Common Variations

- Steepest Ascent: Selects the neighbor with the greatest improvement in each iteration.
- Stochastic Hill Climbing: Randomly chooses among equally good or better neighbors, helping avoid some local optima.
- Random Restarts: Periodically restarts the search from a new random point to increase the chance of finding a better region of the search space.
- Simulated Annealing: A probabilistic variation that initially allows moves that worsen the solution, progressively reducing this probability to escape local optima and explore more widely.

Considerations for Use

- Problem Nature: Assess whether the search space is likely to have many local optima hindering simple hill climbing.
- Solution Quality vs. Time: Determine if finding a good-enough solution quickly is more important than guaranteeing the best possible solution.
- Heuristic Design: The choice of heuristic significantly impacts the algorithm's performance. A good heuristic should guide the search effectively.
- Potential Combination: Hill climbing can be used as a part of more sophisticated optimization techniques, providing an initial solution or local refinement.

Overall

Hill climbing is a foundational optimization algorithm that offers a trade-off between simplicity and the potential issue of local optima. Understanding its strengths, weaknesses, and variations is crucial for determining its suitability for specific problems.

Comparatively Table Breakdown (BFS/DFS/A*/Hill Climb)

Criterion	BFS (Breadth-First Search)	DFS (Depth-First Search)	A* Search	Hill Climbing
Completeness	Yes (finite branching factor)	No (may get stuck in infinite paths)	Yes (finite branching factor)	No (may get stuck in local optima)
Time Complexity	O(b^d)	O(b^m)	Varies (depends on heuristic)	Varies (depends on problem and restarts)
Space Complexity	O(b^d)	O(b^m)	Varies (open list)	O(b) (typically)
Optimality	Yes (equal step costs)	No	Yes (admissible heuristic)	No (local optima)

Reason behind it

Completeness:

- BFS and A* guarantee finding a solution if one exists (provided there are a finite number of possibilities at each step).
- DFS and Hill Climbing might not terminate when the search space is infinite or they get trapped in cycles or local optima.

Time Complexity:

- BFS and DFS have exponential worst-case time complexity, where 'b' is the branching factor (number of child nodes from each node) and 'd' and 'm' represent maximum depth and maximum possible path length respectively.
- A*'s time complexity depends on the heuristic. A good heuristic can guide it towards the goal node quickly.
- Hill Climbing's time complexity varies; it can find good solutions quickly in some cases, but can also take excessively long if it gets stuck.

Space Complexity:

- BFS and DFS can have exponential worst-case space complexity, as they need to store nodes in a queue or stack respectively.
- A* can have large memory requirements for its 'open list' containing potential nodes for exploration.
- Hill Climbing typically has very low space complexity, usually storing only the current state and its neighbors.

Optimality:

- BFS finds the shortest path if all step costs are equal.
- A* finds the optimal path if the heuristic is admissible (never overestimates the cost to reach the goal).
- DFS and Hill Climbing do not guarantee finding optimal solutions.

Hill Climb Search Properties

Property	Description
• Completeness	Hill climbing is not complete. It may get stuck in local optima, even if a solution exists.

• Time Complexity	Varies greatly depending on the problem structure and the quality of the heuristic. In the worst case, it might require exploring a significant portion of the search space.
• Space Complexity	Generally low. Hill climbing typically stores only the current state, its neighbors, and heuristic evaluations.
• Optimality	Hill climbing does not guarantee optimality. It is susceptible to converging on local optima rather than the global best solution.

Hill Climb Analysis

Through Pseudo Code

Hill Climb Pseudo Code:

```
function HillClimbingSearch(matrix, goal):
    Initialize root node using the initial matrix state
    Initialize visited set to keep track of visited states
    Initialize states_explored to count the number of explored states
    Define HEURISTICS dictionary mapping heuristic names to their corresponding functions
    for each cell in the initial matrix:
        if the cell contains the blank tile 'B':
            Set the root node using the cell coordinates
    Initialize goal_positions dictionary mapping each goal tile to its position in the goal state
    while True:
        Add the current state to the visited set
        if the current state is the goal state:
            Return the current node
        Initialize local_min_heuristic to the heuristic value of the current state
        Initialize best_neighbor to None
        Generate neighboring nodes for the current node
        for each neighbor in neighbors:
            if the neighbor state is not visited:
                Calculate the heuristic value of the neighbor state
                if the neighbor state heuristic is less than local_min_heuristic:
                    Update local_min_heuristic with the neighbor state heuristic
                    Set best_neighbor to the neighbor node
        if no better neighbor is found:
            Return None (hill climbing is stuck at a peak or plateau)
        Move to the best neighbor state
        Increment the states_explored count
```

Hill Climb Algorithm

Analysis Through Code

Our code consists of implementation of Hill Climb algorithms for a 3x3 random initial matrix.
Breakdown of the Hill Climbing Search code, along with explanations, insights, and potential areas for

Classes

Node:

- Represents a single state in the search space of the puzzle.

Attributes:

- `i`: The i-coordinate of the blank tile (or another relevant element)
- `j`: The j-coordinate of the blank tile
- `state`: The current configuration of the puzzle (likely a list of lists)
- `parent`: A reference to the parent node, used to trace the solution path.

HillClimbingSearch:

- Represents the core implementation of the Hill Climbing Search Algorithm

Attributes:

- `matrix`: Initial puzzle configuration
- `goal`: The desired goal state
- `visited`: A set to store visited states, preventing revisiting.
- `states_explored`: Counter for the number of states explored.
- `HEURISTICS`: A dictionary mapping heuristic function names to their implementations.
- `root`: The starting node.
- `goal_positions`: Maps tile values in the goal state to their (i, j) positions (useful for Manhattan distance calculation).

Methods

init(self, matrix, goal):

- Initializes the search object.
- Sets up the initial state, goal state, data structures, and calculates the goal positions for heuristic calculations.

swap(self, node, move)

- Swaps tiles to create a new state based on a given move direction.

move_up(self, node), move_down(...), move_left(...), move_right(...):

- Implement the actions of moving the blank tile (or another specified element).
- Returns a new `Node` if the move is valid, otherwise returns `None`.

generate_neighbors(self, node):

- Generates a list of possible next states (neighboring nodes) from the given node.

print_path(self, node):

- Traces the path from the given node (usually the goal node) back to the root, reconstructing the solution sequence.

h1(self, state), h2(self, state):

- Implementations of two heuristic functions:
- `h1`: Counts the number of misplaced tiles.
- `h2`: Calculates the sum of Manhattan distances of tiles from their goal positions.

Solve (self, heuristic):

- The primary function to execute the hill climbing search.
- Returns the solution path if the goal is found.

Search (self, heuristic):

- The core of the hill climbing algorithm.

Key Logic:

- Maintain the current state.
- Check if the current state matches the goal; if so, return success.
- Generate neighbors of the current state.
- Evaluate neighbors using the specified heuristic.
- Select the neighbor with the lowest heuristic value as the new current state.

- Repeat until no better neighbor is found (stuck in a local optimum) or the goal is reached.

Few Observations

- **Clear Structure:** The code is well-structured, with classes and methods designed for their intended purposes.
- **Good Heuristics:** The provided heuristics `h1` and `h2` are common choices for slide puzzles.
- **Efficient State Tracking:** The code was likely updated to use hashing for storing visited states, which can significantly improve the speed of checking if a state has already been explored.
- **Restart Strategy:** Consider adding random restarts to help escape local optima. This means periodically restarting the search from a new randomly generated state.
- **Alternative Heuristics:** Experiment with different heuristics or combinations of heuristics to see if they improve search performance.

Hill Climb Execution:

This execution details description provides a step-by-step overview of how we have gone through this Hill Climbing algorithm, initialized, executed it, and displayed its output.

Here's an execution details description for the Hill Climbing Search algorithm based on the Python code we have:

Initialization:

- Start state:
  - A randomly generated 3x3 array.
- Goal state:
  - Predefined goal state.
- Hill Climbing Initialization and Execution:
  - Hill Climbing algorithm initialized with the start state and goal state.
  - Hill Climbing solver instance created.
  - Hill Climbing solver's `solve()` method executed.

During the search:

- States are explored based on their heuristic values.
- Nodes are generated and explored iteratively.
- The algorithm selects the best neighboring state based on the heuristic function.
- If no better neighbor is found, the algorithm terminates.

Once the goal state is reached or no more nodes to explore:

- If the goal state is reached, the solution path is extracted.
- Number of explored states is recorded.

Output Display:

- We've used file handling to generate a rewritable output.txt file which contains results for all heuristics.
- Start and goal states are displayed.
- The number of explored states and execution details are printed.
- If a solution is found, the solution path is displayed.
- If the Hill Climbing process is completed without finding a solution, a failure message is displayed.

Real time Implementation (In Python)

Introduction:

- **The course_det function provides an introduction message, group details and course details in the output window.**

```
 ------------------------------------------------
1. Course Name: Ex. M.Tech. Program
**********************************
2. Session: 2024-2026
**********************
3. Subject: Artificial Intelligence
**********************************
4. Subject Code: CS561
**********************
5. Faculty Name: Dr. Asif Ekbal
*******************************
6. Team Members:
*****************
   a. Sanjeev Kumar (2403res117 - IITP002297)
*****************************************
   b. Aditya Gupta (2403res85 - IITP002204)
*****************************************
   c. Avinash Aanand (2403res99 - IITP002223)
*********************************************
   d. Aman Kumar (2403res10 - IITP002012)
*****************************************
************************************************************************************************************
```

HillClimb Implementation:

Our code defines a class `HillClimbingSearch` that implements the Hill Climbing search algorithm for the 8-puzzle problem and a main function `run()` the drives the code.
Here's a summary of the classes and their functionalities:

1. Node Class:
    a. Represents a node in a search tree.
    b. Attributes:
        a) i (int): The value of i^{th} coordinate.
        b) j (int): The value of j^{th} coordinate.
        c) state (object): The state associated with the node.
        d) parent (Node, optional): The parent node of the current node.
2. HillClimbingSearch class:
    a. Initializes with initial and goal state matrix.
    b. Attributes:
        a) matrix (list): The initial matrix state.
        b) goal (list): The goal matrix state.
        c) visited (set): A set to keep track of visited states.
        d) states_explored (int): The number of states explored.
        e) HEURISTICS (dict): A dictionary of available heuristics.
        f) root (Node): The root node of the search tree.
        g) goal_positions (dict): A dictionary mapping goal positions.
    c. Methods:
        a) swap(node, move): Swaps the current node's state with the state obtained by making a move in a specified direction.
        b) move_up(node): Moves the given node up by one position.
        c) move_down(node): Moves the given node down by swapping it with the element below it.
        d) move_left(node): Moves the given node to the left.
        e) move_right(node): Moves the empty tile to the right in the puzzle.
        f) generate_neighbors(node): Generates the neighbouring nodes for a given node.
        g) print_path(node): Prints the path from the given node to the root node.
        h) h1(state): Heuristic function that calculates the number of misplaced tiles.
        i) h2(state): Heuristic function that calculates the Manhattan distance.
        j) solve(heuristic): Solves the puzzle using the hill climbing search algorithm.

k) search(heuristic): Performs the hill climbing search to find the goal state.

3. Course detail's function:
   A course_det() is defined for the below tasks:
   a. Displays the course details.
   b. This method prints out the details of the course, including the course name, session, subject, subject code, faculty name, and team members. Each detail is printed with a line of asterisks below it to align subsequent lines with the numbers.
4. Run function:
   a. It is our main function.
   b. Calls the `HillClimbingSearch` method for given initial and goal state to solve the 8-puzzle problem with different heuristics so that we can perform a comparative analysis based on multiple parameters like number of states explored, time consumed, memory efficiency.
   c. Prints the results of each heuristic function in an output.txt file, including the start state, goal state, states explored, optimal path length, execution time, memory usage, path, and monotonicity check.
5. random_array function:
   a. Generates a random 3x3 array of numbers.

Post Code Analysis

Hill Climb search are ways a computer can solve puzzles of the 3x3 slide puzzle.

Analysis of Hill Climb Algorithm:

Efficiency:

- The efficiency of Hill Climbing algorithm as compared to A*, BFS & DFS in terms of number of nodes explored varies as it may get stuck at local optima and never reach goal state.
- The Hill Climbing algorithm does not guarantee finding the optimal solution but A* and BFS do.

Space Analysis:

- Hill climbing outperforms A*, BFS and DFS in terms of space complexity as it only needs to store the current state and goal state at each step. Hence it requires less memory.
- It does not need to maintain a queue like A*, BFS & DFS to store the list of visited nodes.
- As Hill climbing used constant amount of extra space, its space complexity is O (1).

Time Analysis:

- The Hill Climbing algorithm is faster as compared to A* and BFS but it is prone to local optima hence not reliable whereas A* and BFS are comparatively slower but reliable.
- The time complexity of Hill Climbing algorithm is O (∞) as it may get stuck at local optima and never reach globally optimal solution.

Optimality:

- Hill climbing can be optimal for certain cases where it reaches the globally optimal solution but that usually happens once in 1000 trials. Generally, A* and BFS are considered optimal solutions as they guarantee optimal solutions.
- Hill climbing is not an optimal solution because of its limitations.

Results/Output

Welcome Window:

Heuristics Outputs		
Invalid States:		Result with all details:
Going through Heuristic 1 (Misplaced tiles):		
Total number of invalid states for h1 : 863		Solution found Starting state: T4 T1 T3

Goal state: T1 T2 T3 T4 T5 T6 T7 T8 B Total number of states explored before termination: 0 --- Solution not found Starting state: T1 T6 T7 T3 T8 B T4 T5 T2 Goal state: T1 T2 T3 T4 T5 T6 T7 T8 B Total number of states explored before termination: 2 --- Solution not found Starting state: T5 T3 T7 T4 T1 T8 T2 T6 B Goal state: T1 T2 T3 T4 T5 T6 T7 T8 B Total number of states explored before termination: 6 --- Solution not found Starting state: B T3 T1 T5 T7 T4 T8 T2 T6 Goal state: T1 T2 T3 T4 T5 T6 T7 T8 B Total number of states explored before termination: 0 --- Solution not found Starting state: T6 T1 T5 B T2 T4 T7 T3 T8 Goal state: T1 T2 T3 T4 T5 T6 T7 T8 B Total number of states explored before termination: 5 --- Solution not found Starting state: T5 B T1	B T2 T5 T7 T8 T6 Goal state: T1 T2 T3 T4 T5 T6 T7 T8 B Total number of states explored: 5 Total number of states to the optimal path using h1: 6 Optimal path: T4 T1 T3 B T2 T5 T7 T8 T6 B T1 T3 T4 T2 T5 T7 T8 T6 T1 B T3 T4 T2 T5 T7 T8 T6 T1 T2 T3 T4 B T5 T7 T8 T6 T1 T2 T3 T4 T5 B T7 T8 T6 T1 T2 T3 T4 T5 T6 T7 T8 B
Going through Heuristic 2 (Manhattan distance):	
Total number of invalid states for h2: 56	Solution found Starting state: T2 T3 T6 T1 T7 B T5 T4 T8 Goal state: T1 T2 T3 T4 T5 T6 T7 T8 B Total number of states explored: 11 Total number of states to the optimal path using h2: 12 Optimal path: T2 T3 T6 T1 T7 B T5 T4 T8 T2 T3 B T1 T7 T6 T5 T4 T8

```
Total number of states explored before termination: 4
--------------------------------------------------
Solution found
Starting state:
T1 T3 T6
T5 T2 B
T4 T7 T8
Goal state:
T1 T2 T3
T4 T5 T6
T7 T8 B
Total number of states explored: 7
Total number of states to the optimal path using h2: 8
Optimal path:
T1 T3 T6
T5 T2 B
T4 T7 T8

T1 T3 B
T5 T2 T6
T4 T7 T8

T1 B T3
T5 T2 T6
T4 T7 T8

T1 T2 T3
T5 B T6
T4 T7 T8

T1 T2 T3
B T5 T6
T4 T7 T8

T1 T2 T3
T4 T5 T6
B T7 T8

T1 T2 T3
T4 T5 T6
T7 B T8

T1 T2 T3
T4 T5 T6
T7 T8 B

--------------------------------------------------
Total number of invalid states for h2 : 380
==================================================
```

T2 B T3
T1 T7 T6
T5 T4 T8

B T2 T3
T1 T7 T6
T5 T4 T8

T1 T2 T3
B T7 T6
T5 T4 T8

T1 T2 T3
T5 T7 T6
B T4 T8

T1 T2 T3
T5 T7 T6
T4 B T8

T1 T2 T3
T5 B T6
T4 T7 T8

T1 T2 T3
B T5 T6
T4 T7 T8

T1 T2 T3
T4 T5 T6
B T7 T8

T1 T2 T3
T4 T5 T6
T7 B T8

T1 T2 T3
T4 T5 T6
T7 T8 B

So above we have portrayed the success cases. We have written all the actual output of the success and failure cases into an output file. We have attached it in the zip for your kind perusal.

Observation /Conclusion

Conclusion of Hill Climbing for 8-Puzzle Problem:

Efficiency:

- Hill climbing is known for its simplicity and low memory requirements since it only needs to maintain the current state and explore its neighbour's.
- This makes it suitable for memory-constrained environments. However, the efficiency of hill climbing varies significantly based on the choice of heuristic and the characteristics of the problem instance.

- For both the misplaced tile and Manhattan distance heuristics, the efficiency depends on factors such as the initial state's proximity to the goal state and the complexity

Heuristic Choice:

- The effectiveness of hill climbing is significantly influenced by the choice of heuristic function.
- Two commonly used heuristics for the 8-puzzle problem are the number of misplaced tiles and the Manhattan distance.
- The number of misplaced tiles heuristic counts the number of tiles that are not in their correct positions, while the Manhattan distance heuristic calculates the sum of the distances of each tile from its goal position.
- In general, the Manhattan distance heuristic tends to provide better guidance, especially in larger search spaces, as it considers the actual distance between tiles and their goal positions.

Effectiveness with Misplaced Tile Heuristic:

- The misplaced tile heuristic evaluates the number of tiles that are not in their correct positions.
- Hill climbing utilizing this heuristic incrementally improves the current state by moving tiles to their correct positions.
- It is effective when the solution is nearby and the heuristic accurately estimates the distance to the goal state. However, its effectiveness diminishes as the puzzle complexity increases or when the initial state is far from the goal state.
- Hill climbing may get trapped in local optima, where further moves worsen the solution rather than improving it.

Effectiveness with Manhattan Distance Heuristic:

- The Manhattan distance heuristic calculates the sum of the distances of each tile from its goal position, providing a more informed evaluation of the distance to the goal state.
- Hill climbing employing this heuristic tends to outperform the misplaced tile heuristic, especially in larger search spaces.
- It considers the actual distance between tiles and their goal positions, guiding the search more effectively towards the optimal solution.
- However, like any heuristic-driven search, its effectiveness is limited by the quality of the heuristic function and the initial state. of the puzzle.

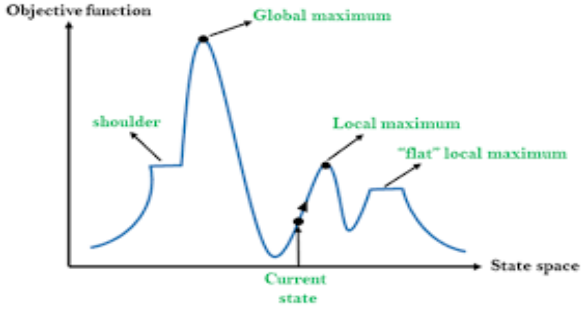
Limitations:

- Despite its simplicity and effectiveness in some scenarios, hill climbing has several limitations.
- One major drawback is its susceptibility to getting stuck in local optima. This occurs when the algorithm reaches a state where no further moves improve the solution, preventing it from reaching the global optimal solution.
- Additionally, hill climbing does not guarantee optimality or completeness. It may terminate prematurely without finding a solution or finding a suboptimal solution.

Disadvantage:

Local Optima:

- One of the primary disadvantages of hill climbing is its susceptibility to getting stuck in local optima.
- Since hill climbing makes greedy choices based solely on the heuristic evaluation function without considering future consequences, it may converge to a suboptimal solution that is locally optimal but not globally optimal.
- This limitation can prevent the algorithm from finding the best possible solution, especially in complex search spaces with multiple local optima.



Limited Exploration:

- Hill climbing explores only a subset of neighbouring states at each iteration, typically selecting the one with the lowest heuristic value.
- This limited exploration can lead to overlooking potentially better solutions located in regions of the search space that are not immediately accessible from the current state.
- As a result, hill climbing may fail to discover the global optimal solution or even miss feasible solutions altogether.

Initial State Sensitivity:

- The effectiveness of hill climbing heavily depends on the choice of initial state.
- If the initial state is far from the goal state or located in a region of the search space with few feasible solutions, hill climbing may struggle to find a solution or take a long time to converge.
- Furthermore, different initial states may lead to vastly different outcomes, making it challenging to predict the algorithm's performance across different problem instances.

Heuristic Dependency:

- The performance of hill climbing is highly dependent on the quality of the heuristic function used to evaluate states.
- While certain heuristics, such as the Manhattan distance, can provide reasonable guidance in guiding the search towards the goal state, they may not always accurately reflect the true distance to the solution.
- If the heuristic function is poorly designed or fails to capture important characteristics of the problem, hill climbing may make suboptimal decisions, leading to inefficient or ineffective search.

Lack of Optimality and Completeness Guarantees:

- Unlike some other search algorithms like A*, hill climbing does not guarantee optimality or completeness.
- There is no assurance that the solution found by hill climbing is the best possible solution, and it may terminate prematurely without finding a solution or converging to a suboptimal solution.
- This limitation can be problematic, especially in critical applications where finding the best possible solution is essential.

Comparison of 8-Puzzle Problem Solving Approaches:

Hill Climbing vs. A*:

- A* combines the advantages of both breadth-first and greedy best-first searches by considering both the cost to reach a state and the estimated cost to the goal.
- It guarantees optimality and completeness when using an admissible heuristic.
- Compared to hill climbing, A* is generally more effective and efficient, particularly with the Manhattan distance heuristic.
- A* explores the search space more systematically, leading to optimal solutions.
- However, A* may require more memory and computation time compared to hill climbing, especially in larger search spaces.

Hill Climbing vs. Depth-First Search (DFS):

- DFS explores the search space by going as deep as possible along each branch before backtracking. It does not use any heuristic information and may get stuck in deep branches without finding a solution.
- Hill climbing focuses on improving the current solution iteratively based on heuristic evaluation. It tends to outperform DFS, especially with the Manhattan distance heuristic.
- However, DFS guarantees completeness while hill climbing does not.

Hill Climbing vs. Breadth-First Search (BFS):

- BFS explores the search space level by level, considering all neighboring states before moving to the next level. It guarantees optimality and completeness but may require more memory compared to hill climbing.
- Hill climbing is more memory-efficient but may struggle to find optimal solutions, especially with the misplaced tile heuristic.

- In terms of effectiveness, BFS tends to outperform hill climbing, particularly when the solution is located at a shallow depth in the search space. However, hill climbing can be faster in finding solutions if the search space is large and the solution is nearby.