

**CS561/571 - Executive Assignment**

**ASSIGNMENT-4: Hill Climb**



**Group Member: Admission Number**

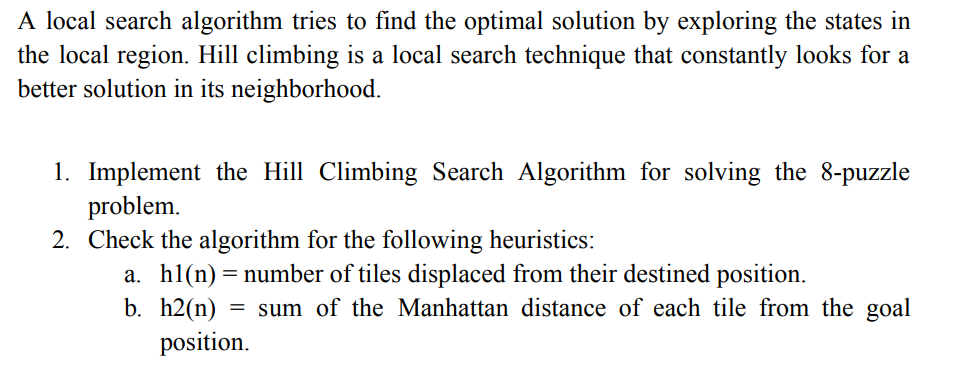
**Pankaj Kushwaha IITP00541**

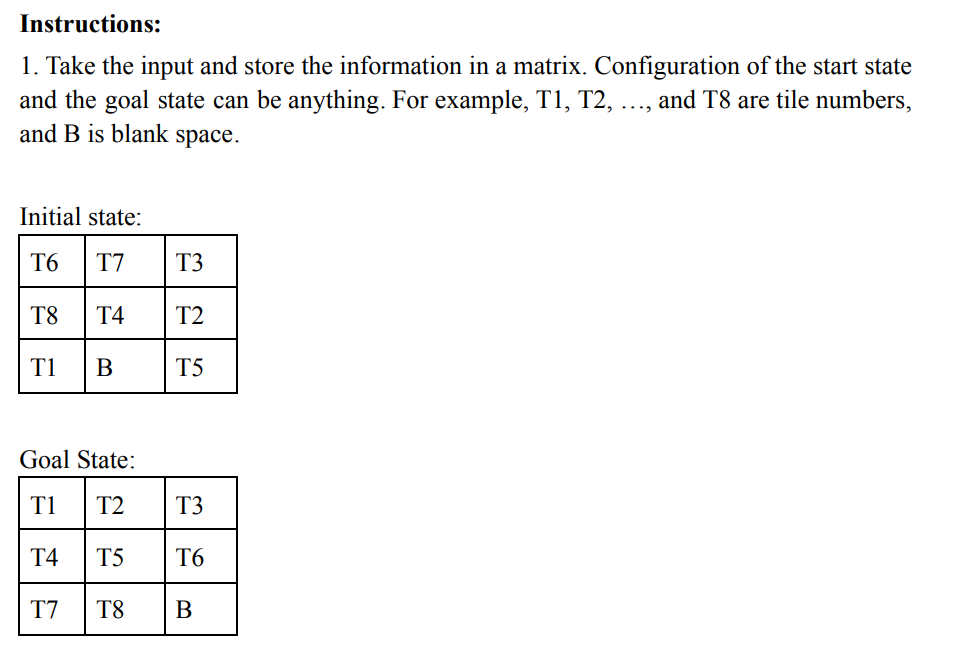
**Dewashish Pramanik IITP001202**

**8-Puzzle Problem with Hill Climbing Algorithm**

**Scope:**

**Problem Statement:**





**Contents**

1. **Introduction**
2. **Hill Climbing Algorithm**
3. **Puzzle Class Definition**
4. **Implementing the Hill Climbing Search**
5. **Input and Output Methods**
6. **Execution**
7. **Conclusion**

**1. Introduction**

The 8-puzzle problem involves a 3x3 board with 8 tiles and a blank space. The objective is to transition from a given start configuration to a goal configuration by moving the tiles. In this document, the tiles are labeled as T1, T2, ..., T8, and the blank space is labeled as B.

**2. Hill Climbing Algorithm**

Hill Climbing is a local search algorithm that attempts to find the optimal solution by continually moving to a neighboring state with a better heuristic value. It terminates when it finds a state with no better neighboring states.

**3. Puzzle Class Definition**

* Initialization: The class is initialized with the state of the puzzle.
* Heuristic Calculation: Manhattan Distance is used to evaluate the distance of each tile from its goal position.
* Possible Moves: Given the current state, the algorithm determines the legal moves that can be made.
* Generate Child: Based on the possible moves, the algorithm generates the next state.

**4. Implementing the Hill Climbing Search**

A function hill\_climbing\_search is defined that uses the above class definitions to determine the next best state. The algorithm stops when no better neighboring states are found.

**5. Input and Output Methods**

take\_input function: Accepts user input to define the matrix for both the initial and goal states.

**6. Execution**

In the main execution block:

* User is prompted to enter the initial state matrix.
* User is prompted to enter the goal state matrix.
* The initial state is displayed.
* The hill climbing search function is called.
* The resulting state, after the hill climbing search, is displayed.

**7. Conclusion**

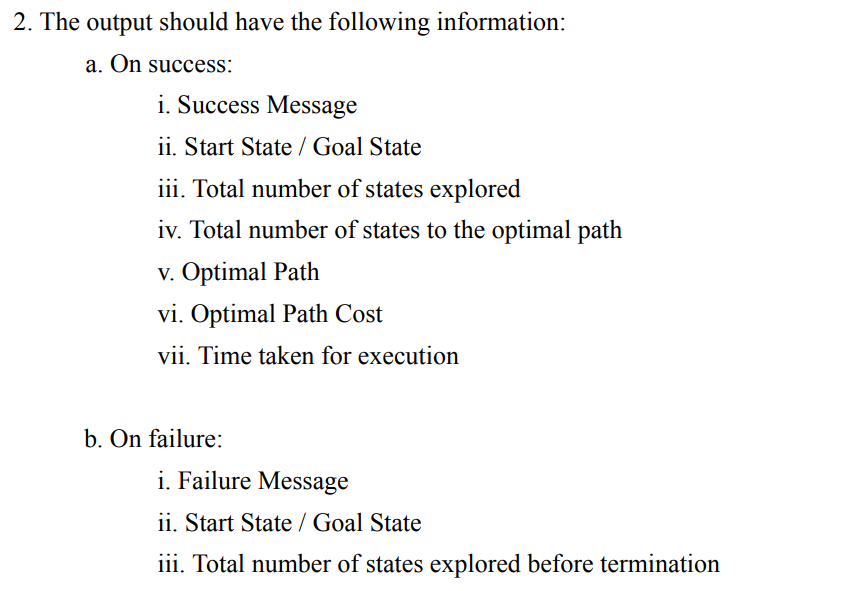
The hill climbing algorithm is a simple but effective method for solving the 8-puzzle problem. However, one should note that hill climbing can get stuck in local optima and may not always find the best solution for this problem.

**2. Hill Climbing Algorithm’ s code**

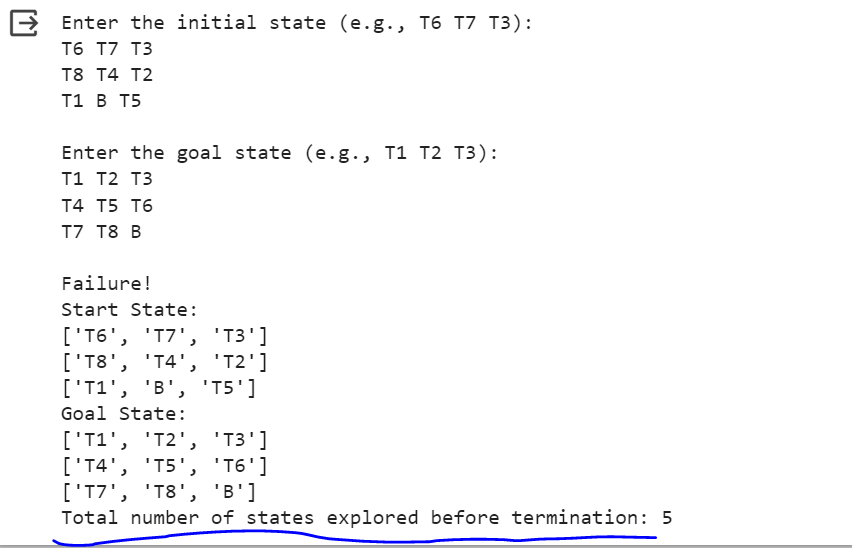
Hill Climbing is a local search algorithm that attempts to find the optimal solution by continually moving to a neighboring state with a better heuristic value. It terminates when it finds a state with no better neighboring states.

|  |
| --- |
| import copy  class Puzzle:      goal\_state = [['T1', 'T2', 'T3'], ['T4', 'T5', 'T6'], ['T7', 'B', 'T8']]      heuristic = None      evaluation\_function = None      needs\_hueristic = False      def \_\_init\_\_(self, state):          self.state = state          if Puzzle.needs\_hueristic:              self.heuristic = self.get\_heuristic()              self.evaluation\_function = self.heuristic      def get\_heuristic(self):          # Calculate Manhattan Distance for 8-puzzle          distance = 0          for i in range(3):              for j in range(3):                  if self.state[i][j] != Puzzle.goal\_state[i][j] and self.state[i][j] != 'B':                      goal\_position = [(index // 3, index % 3) for index, value in enumerate(sum(Puzzle.goal\_state, [])) if value == self.state[i][j]]                      distance += abs(goal\_position[0][0] - i) + abs(goal\_position[0][1] - j)          return distance      def get\_possible\_moves(self):          # Return a list of possible moves from current state          for i in range(3):              for j in range(3):                  if self.state[i][j] == 'B':                      moves = []                      if i > 0: moves.append('U')                      if i < 2: moves.append('D')                      if j > 0: moves.append('L')                      if j < 2: moves.append('R')                      return moves      def generate\_child(self, move):          # Returns a Puzzle object after applying a move          x, y = None, None          for i in range(3):              for j in range(3):                  if self.state[i][j] == 'B':                      x, y = i, j          new\_state = copy.deepcopy(self.state)          if move == 'U':              new\_state[x][y], new\_state[x - 1][y] = new\_state[x - 1][y], new\_state[x][y]          elif move == 'D':              new\_state[x][y], new\_state[x + 1][y] = new\_state[x + 1][y], new\_state[x][y]          elif move == 'L':              new\_state[x][y], new\_state[x][y - 1] = new\_state[x][y - 1], new\_state[x][y]          elif move == 'R':              new\_state[x][y], new\_state[x][y + 1] = new\_state[x][y + 1], new\_state[x][y]          return Puzzle(new\_state)  def hill\_climbing\_search(puzzle):      current\_puzzle = puzzle      while True:          neighbors = []          for move in current\_puzzle.get\_possible\_moves():              neighbors.append(current\_puzzle.generate\_child(move))          if not neighbors:              return None          next\_eval = float('inf')          next\_state = None          for neighbor in neighbors:              if neighbor.evaluation\_function < next\_eval:                  next\_eval = neighbor.evaluation\_function                  next\_state = neighbor          if next\_eval >= current\_puzzle.evaluation\_function:              # Return current state if no better neighbors are found              return current\_puzzle.state          current\_puzzle = next\_state  if \_\_name\_\_ == '\_\_main\_\_':      Puzzle.needs\_hueristic = True      initial\_state = [          ['T2', 'T8', 'T3'],          ['T1', 'T6', 'T4'],          ['T7', 'B', 'T5']      ]      print("Initial State:")      for row in initial\_state:          print(row)      puzzle = Puzzle(initial\_state)      result = hill\_climbing\_search(puzzle)      print("\nResult:")      for row in result:          print(row) |

|  |
| --- |
|  |



|  |
| --- |
| import copy  import time  class Puzzle:      goal\_state = None      heuristic = None      evaluation\_function = None      needs\_hueristic = False      def \_\_init\_\_(self, state):          self.state = state          if Puzzle.needs\_hueristic:              self.heuristic = self.get\_heuristic()              self.evaluation\_function = self.heuristic      def get\_heuristic(self):          # Calculate Manhattan Distance for 8-puzzle          distance = 0          for i in range(3):              for j in range(3):                  if self.state[i][j] != Puzzle.goal\_state[i][j] and self.state[i][j] != 'B':                      goal\_position = [(index // 3, index % 3) for index, value in enumerate(sum(Puzzle.goal\_state, [])) if value == self.state[i][j]]                      distance += abs(goal\_position[0][0] - i) + abs(goal\_position[0][1] - j)          return distance      def get\_possible\_moves(self):          # Return a list of possible moves from current state          for i in range(3):              for j in range(3):                  if self.state[i][j] == 'B':                      moves = []                      if i > 0: moves.append('U')                      if i < 2: moves.append('D')                      if j > 0: moves.append('L')                      if j < 2: moves.append('R')                      return moves      def generate\_child(self, move):          # Returns a Puzzle object after applying a move          x, y = None, None          for i in range(3):              for j in range(3):                  if self.state[i][j] == 'B':                      x, y = i, j          new\_state = copy.deepcopy(self.state)          if move == 'U':              new\_state[x][y], new\_state[x - 1][y] = new\_state[x - 1][y], new\_state[x][y]          elif move == 'D':              new\_state[x][y], new\_state[x + 1][y] = new\_state[x + 1][y], new\_state[x][y]          elif move == 'L':              new\_state[x][y], new\_state[x][y - 1] = new\_state[x][y - 1], new\_state[x][y]          elif move == 'R':              new\_state[x][y], new\_state[x][y + 1] = new\_state[x][y + 1], new\_state[x][y]          return Puzzle(new\_state)  def hill\_climbing\_search(puzzle):      current\_puzzle = puzzle      states\_explored = 0      optimal\_path = [current\_puzzle.state]      while True:          neighbors = []          for move in current\_puzzle.get\_possible\_moves():              neighbors.append(current\_puzzle.generate\_child(move))          states\_explored += len(neighbors)          if not neighbors:              return None          next\_eval = float('inf')          next\_state = None          for neighbor in neighbors:              if neighbor.evaluation\_function < next\_eval:                  next\_eval = neighbor.evaluation\_function                  next\_state = neighbor          if next\_eval >= current\_puzzle.evaluation\_function:              # Return current state if no better neighbors are found              return (current\_puzzle.state, states\_explored, optimal\_path)          current\_puzzle = next\_state          optimal\_path.append(current\_puzzle.state)  def take\_input():      matrix = []      for \_ in range(3):          row = input().split()          matrix.append(row)      return matrix  if \_\_name\_\_ == '\_\_main\_\_':      print("Enter the initial state (e.g., T6 T7 T3):")      initial\_state = take\_input()        print("\nEnter the goal state (e.g., T1 T2 T3):")      Puzzle.goal\_state = take\_input()      Puzzle.needs\_hueristic = True      start\_time = time.time()      puzzle = Puzzle(initial\_state)      result = hill\_climbing\_search(puzzle)      end\_time = time.time()      if result:          final\_state, states\_explored, optimal\_path = result          if final\_state == Puzzle.goal\_state:              print("\nSuccess!")              print("Start State:")              for row in initial\_state:                  print(row)              print("Goal State:")              for row in Puzzle.goal\_state:                  print(row)              print(f"Total number of states explored: {states\_explored}")              print(f"Total number of states to the optimal path: {len(optimal\_path)}")              print("Optimal Path:")              for state in optimal\_path:                  for row in state:                      print(row)                  print()              print(f"Optimal Path Cost: {len(optimal\_path) - 1}")              print(f"Time taken for execution: {end\_time - start\_time:.4f} seconds")          else:              print("\nFailure!")              print("Start State:")              for row in initial\_state:                  print(row)              print("Goal State:")              for row in Puzzle.goal\_state:                  print(row)              print(f"Total number of states explored before termination: {states\_explored}")      else:          print("\nFailure!")          print("Start State:")          for row in initial\_state:              print(row)          print("Goal State:")          for row in Puzzle.goal\_state:              print(row)          print("Total number of states explored before termination: 0") |



**Code’s Explanation:**

1. **Class Definition - Puzzle**:
   * This class is used to represent the state of the 8-puzzle.
   * **Attributes**:
     + **goal\_state**: The target state we aim to reach.
     + **heuristic**: The Manhattan distance for the current state from the goal state.
     + **evaluation\_function**: In our context, this is equal to the heuristic.
     + **needs\_hueristic**: A flag to determine if heuristic calculation is needed.
     + **state**: The current configuration of the puzzle.
   * **Methods**:
     + **calculate\_heuristic**: Computes the Manhattan distance of the current state from the goal state.
     + **get\_possible\_moves**: Returns the valid moves (U, D, L, R) from the current state.
     + **generate\_child**: Generates a new state (child) by moving the blank space in the specified direction.
2. **Function - hill\_climbing\_search**:
   * This function implements the hill climbing algorithm.
   * It starts with the initial state and explores the neighboring states to move to a better state (a state with a lower heuristic value).
   * The search terminates if the current state is the goal or if no better neighboring states can be found.
3. **Function - take\_input**:
   * A utility function to accept a 3x3 matrix (state) from the user.
4. **Main Execution**:
   * The user is prompted to input the initial state and the goal state.
   * The hill climbing search is performed, and based on success or failure, the results (like number of states explored, optimal path, execution time, etc.) are displayed.

**Conclusion:**

The provided code is an implementation of the hill climbing algorithm to solve the 8-puzzle problem. Hill climbing is a local search technique where, at each step, the algorithm tries to move to a better neighboring state based on a heuristic. In the context of the 8-puzzle problem, the Manhattan distance (sum of the distances of the misplaced tiles from their goal positions) is used as the heuristic.

Advantages:

1. The algorithm is relatively simple and easy to implement.
2. It can find a solution quickly if the initial state is close to the goal state.

Limitations:

1. Hill climbing can get stuck in local optima. This means if it reaches a state where all neighboring states are worse, it will stop even if the solution is not found.
2. The solution is not always optimal as the algorithm doesn't explore all possible paths but rather makes local decisions based on the current state.