Artificial Intelligence Lab Report 1

1st Dipean Dasgupta
202151188
BTech CSE
IIIT,Vadodara

2nd Shobhit Gupta
202151149
BTech CSE
IIIT,Vadodara

3rd Rahul Rathore
202151126
BTech CSE
IIIT,Vadodara

4th Rohan Deshpande
202151133
BTech CSE
IIIT,Vadodara

***Abstract*—This lab assignment emphasises state space search strategies using hash tables and queues while focusing on creating a graph search agent for the Puzzle-8 issue. Development of pseudocode, implementation of environment functions, and investigation of iterative deepening search are among the main goals. Furthermore, a function for retracing the best route at constant cost is included. This assignment focuses at how Puzzle-8 instances are generated at different depths and analyses how much memory and time are needed to solve examples.**

I. Introduction

Graph search agents are crucial for methodically exploring state domains and solving complex problems. The focus of this project is to create such an agent for the Puzzle-8 problem, in which a goal state is to be achieved by rearranging tiles. The theoretical basis is guided by reference materials [1] and [2], which place special emphasis on hash tables, queues, and state space exploration. Goals include developing flowcharts and pseudocode, implementing environment functions, and investigating iterative deepening search. The assignment also covers the creation of Puzzle-8 instances, backtracking with uniform cost, and an examination of the memory and processing needs involved in solving instances. The objective of this task is to improve comprehension and utilization of graph search algorithms in practical problem-solving scenarios.

II. Solution to Problem Statement

A. Pseudocode for a graph search agent and Flow chart

Algorithm 1 Graph Search Agent

```
0: function GRAPHSEARCHAGENT(problem)
0:     Initialize data structures, including an empty queue, a hash table, and other necessary variables
0:     Enqueue initial state of the problem to the queue
0:     Add initial state to the hash table
0:     while queue is not empty do
0:         current_state ← Dequeue from the queue
0:         if current_state is the goal state then
0:             return success and the path to the goal
0:         end if
0:         for all successor_state in GETSUCCESSORS(current_state) do
0:             if successor_state is not in hash table then
0:                 Add successor_state to hash table
0:                 Enqueue successor_state to the queue
0:             end if
0:         end for
0:     end while
0:     return failure
0: end function
0: function GETSUCCESSORS(state) {Returning a list of successor states from the current state} {Implementing logic specific to the problem domain}
0: end function=0
```

The implementation of a graph search agent algorithm is described in the pseudocode provided above. The program takes an input problem and uses several data structures to explore the problem's state space in a methodical manner.
In the beginning, the algorithm initializes the required data structures, such as a hash table to monitor visited states, a queue for state exploration, and other variables unique to the problem. This stage methodically guarantees information organization and makes state space exploration more efficient.
The problem's initial state is queued onto the queue after initialization. This is the point at which the exploration process begins, launching the methodical traverse of the state space. The technique simultaneously stores the starting state in the hash table. This hash table acts as a way to maintain track of states that have been visited, minimizing duplication of effort and maximizing search efficiency overall.
After that, the algorithm goes into a loop, keeping searching as long as there are states left in the queue to investigate. The following state is dequeued from the front of the queue to become the active state for exploration throughout each iteration of the loop.The algorithm determines whether the goal state and the current state are compatible. The search is deemed successful and the goal's path is returned if a match is discovered.
The programme makes sure that every successor state that is created by exploring the present state hasn't been visited before. By ensuring that the successor state is present in the hash table, this is accomplished and unnecessary exploration is prevented.The successor state is added to the hash table and placed on the queue for additional investigation if it is not already there. Iteratively examining successors and updating the hash table and queue, this procedure keeps going until the desired state is reached or the queue is empty.
If the objective has not been accomplished, the algorithm ends by either going back to look at more states or stopping if the objective has been successfully attained. This thorough and

goal-oriented search procedure is ensured by this structured approach to state space exploration.

B. collection of functions imitating the environment for Puzzle-8

class Puzzle8Environment represents the environment for the Puzzle-8 problem. The functions include displaying the state, getting the position of the blank tile, generating successor states, and checking if the current state is the goal state. Algorithm/pseudocode of the function:

- Initialize the state of the puzzle using the provided initial state array.
- Define a method called `getBlankPosition` that returns the row and column indices of the blank tile in the current state.
- Define a method called `getSuccessors` that generates all possible successor states by moving the blank tile in all four possible directions (up, down, left, right).
- Define a method called `swap` that swaps the positions of two tiles in the current state.
- Define a method called `isGoalState` that checks if the current state is the goal state.
- Define a main method that demonstrates the usage of the Puzzle8Environment class.

Function `displayState`():

- For each row in the state, print the row elements separated by a space.

Function `getBlankPosition`():

- Initialize `blankIndex` to -1.
- For each tile in the state:
  - If the tile is 0 (the blank tile):
    - Set `blankIndex` to the current index.
    - Break the loop.
- Calculate the row and column indices of the blank tile.
- Return the row and column indices as an array.

Function `getSuccessors`():

- Initialize an empty list of successors.
- Get the row and column indices of the blank tile.
- If the blank tile is not at the top row, add the state obtained by swapping the blank tile with the tile above it to the list of successors.
- If the blank tile is not at the bottom row, add the state obtained by swapping the blank tile with the tile below it to the list of successors.
- If the blank tile is not at the leftmost column, add the state obtained by swapping the blank tile with the tile to the left of it to the list of successors.
- If the blank tile is not at the rightmost column, add the state obtained by swapping the blank tile with the tile to the right of it to the list of successors.
- Return the list of successors.

Function `swap(row1, col1, row2, col2)`:

- Create a new copy of the current state.
- Swap the positions of the two specified tiles in the new state.
- Return the new state.

Function `isGoalState(goalState)`:

- Check if the current state is equal to the provided goal state.
- Return the result of the check.

Main Method:

- Create a puzzle environment with the initial state.
- Display the current state.
- Generate and display all possible successor states.
- Check if the current state is the goal state.
- Print the result of the goal state check.

The environment is initialised with the puzzle's initial state, which is accepted as an array in the function codes. The class contains methods for creating successor states by exchanging the blank tile with its neighbors (getSuccessors), obtaining the position of the blank tile (0) within the puzzle (getBlankPosition), and determining whether the current state corresponds to a predefined goal state (isGoalState).
The puzzle's current state is printed in a grid format by the displayState method, which facilitates visualisation. To make later actions easier, the getBlankPosition method returns the row and column of the puzzle's blank tile.
The getSuccessors function checks for potential movements for the puzzle's blank tile to provide a list of successor states. To obtain the successors, exchange the blank tile with its adjacent tiles in four directions: up, down, left, and right.
The puzzle's two tiles can be switched to generate a new state internally using a helper method called swap. This technique helps to create successor states.
The isGoalState function determines whether the puzzle has reached the intended configuration by comparing its present state to a predefined goal state.

C. Iterative Deepening Search

The search method known as Iterative Deepening Search (IDS) avoids the limitations of both depth-first search (DFS) and breadth-first search (BFS) by combining their advantages. Through a series of depth-limited searches, the depth limit is gradually increased in an attempt to discover the shallowest goal state in a tree or graph. Let's use scientific and mathematical language to explain the Iterative Deepening Search and provide an example:

- State Space: S represents the state space of the problem.
- State: s is an individual state in the state space.
- Successor Function:succ(s) provides the set of successor states for a given state s.
- Goal Test: GoalTest(s) checks if a state s is a goal state.
- Depth Limit: d denotes the depth limit for a particular iteration.

Explanation:
The process begins with an initialization step where the depth

limit (d) is set to zero. The algorithm then enters an iterative loop, performing a depth-limited search (DLS) with the current depth limit. If a goal state is found, the solution is returned. If the goal is not reached within the depth limit, the algorithm increments the depth limit and repeats the process.
The depth-limited search explores the state space up to a specified depth limit (d) using a depth-first search strategy within that limit. If a goal state is encountered during this limited search, the solution is returned. If the depth limit is reached and no goal state is found, the algorithm proceeds to the next iteration with an increased depth limit.
Iterative Deepening Search offers several advantages. It is a complete algorithm, ensuring it will find a solution if one exists. Additionally, when applied to uniform-cost problems, IDS guarantees an optimal solution.

D. Uniform Cost function backtracking Goal to Initial State

The Node class should have a state, cost, and a reference to its parent. The uniform cost function function is assumed to return the goal node. Function code on the right side:
So, the above algorithm is explained below:

Algorithm 2 Uniform Cost Search

```
0: function UNIFORMCOSTSEARCH(initialState, goalState)
0:   stack ← empty Stack
0:   visited ← empty Set
0:   push Node(initialState, 0, null) onto stack
0:   while stack is not empty do
0:     currentNode ← pop from stack
0:     if currentNode.state equals goalState then
0:       return currentNode {Goal reached}
0:     end if
0:     if currentNode.state is not in visited then
0:       add currentNode.state to visited
0:       successors ← generateSuccessors(currentNode.state)
0:       for all successorState in successors do
0:         successorNode ← Node(successorState, currentNode.cost + 1, currentNode)
0:         push successorNode onto stack
0:       end for
0:     end if
0:   end while
0:   return null {Goal not found}
0: end function=0
```

- Initialization:An empty stack is created to store the nodes yet to be explored.Another empty set is used to store the visited nodes to avoid revisiting them.The starting state is pushed onto the stack.
- Looping:The loop continues as long as the stack is not empty. In each iteration, the node with the least cost is explored first (which is achieved by using a priority queue like a stack).If the current node is the goal state, the algorithm terminates successfully and returns the current node.
- Pushing successors onto the stack:Each newly created successor node is pushed onto the stack.
- Termination:If the loop completes and the stack becomes empty, it means the goal state was not found and the algorithm terminates unsuccessfully.

Algorithm 3 Generate Successors

```
0: function GENERATESUCCESSORS(state)
0:   successors ← empty List
0:   add (state + 1) to successors
0:   add (state - 1) to successors
0:   return successors
0: end function
0: function BACKTRACKPATH(node)
0:   path ← empty List
0:   while node is not null do
0:     insert node.state at the beginning of path
0:     node ← node.parent
0:   end while
0:   return path
0: end function=0
```

generateSuccessors creates successor states for a given state. It does the following:

- Takes a state as input: The function expects a state as input, which could represent a position in a maze, a configuration of a puzzle, or any other problem you're trying to solve.
- Creates an empty list of successors: It initializes an empty list to store the successor states.
- Adds successor states: It adds two new states to the list of successors:
  One state is created by incrementing the input state by 1. Another state is created by decrementing the input state by 1.
- Returns the list of successors: The function returns the list containing the two successor states (obtained by adding 1 and subtracting 1 from the input state).

E. Puzzle-8 instances with the goal state

One can start with a random initial state and make moves to go to the appropriate depth in order to generate Puzzle-8 instances with the goal state at depth "d." The objective of the Puzzle-8 problem is to arrive at a predetermined configuration using an 8-puzzle board with numbered tiles and an empty space. The algorithm is as follows:

Algorithm 4 Generate Puzzle-8

```
0: function GENERATEPUZZLE8(depth)
0:     puzzle ← GETGOALSTATE
0:     blankPosition ← {2, 2}
0:     for $i$ from 1 to depth do
0:         possibleMoves ← GETPOSSIBLEMOVES(blankPosition)
0:         SHUFFLE(possibleMoves)
0:         move ← possibleMoves[0]
0:         puzzle ← PERFORMMOVE(puzzle, blankPosition, move)
0:         blankPosition ← move
0:     end for
0:     return puzzle
0: end function=0
```

GENERATEPUZZLES creates a puzzle by taking a list of possible moves. It works by iterating through a certain depth and for each iteration it gets possible moves from the current blank position, shuffles them, and performs the first move on the puzzle. Then it updates the blank position based on the move. This process repeats for a set depth. Overall, the function generates a sequence of moves representing a possible solution to a puzzle.

Algorithm 5 Get Goal State

```
0: function GETGOALSTATE
0:     return 2D array representing the goal state:
0:         $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$
0: end function=0
```

GETGOALSTATE. It simply returns a 2D array representing the goal state, which in this case is laid out as a 3x3 grid with numbers 1 through 8 filling the cells, and a blank space at position [1, 2]. This is likely the target configuration for a puzzle solving algorithm.

Algorithm 6 Get Possible Moves

```
0: function GETPOSSIBLEMOVES(blankPosition)
0:     row ← blankPosition[0]
0:     col ← blankPosition[1]
0:     return List of possible moves relative to blankPosition:
0:         Up: {row - 1, col}
0:         Down: {row + 1, col}
0:         Left: {row, col - 1}
0:         Right: {row, col + 1}
0: end function=0
```

GETPOSSIBLMOVES generates a list of possible moves relative to a given blank position on a grid. It takes the blank position (blankPosition) as input, likely a 2D array representing coordinates.
It returns a list of possible moves for moving the blank space up, down, left, or right on the grid.Up, down, left, and right are represented as relative changes to the row and column indices of the blank position.

Algorithm 7 Perform Move

```
0: function PERFORMMOVE(puzzle, blankPosition, move)
0:     temp ← puzzle[blankPosition[0]][blankPosition[1]]
0:     puzzle[blankPosition[0]][blankPosition[1]] ← puzzle[move[0]][move[1]]
0:     puzzle[move[0]][move[1]] ← temp
0:     return puzzle
0: end function=0
```

PERFORMMOVE. It takes three inputs: the puzzle itself, the current blank position, and a desired move.
Gets the element at new position: It calculates the new blank position based on the current position and the desired move. Then, it retrieves the element at that new position in the puzzle.
Swaps elements: It swaps the element at the new blank position with the element currently at the blank position.
Updates blank position: Finally, it updates the blank position in the puzzle to reflect the move.

F. *Memory and time requirements to solve Puzzle-8 instances*

The table below presents the memory and time requirements for solving Puzzle-8 instances at various depths using a graph search agent.
The space requirement for iterative deepening search agent is O(bd)
The time requirement for iterative deepening search agent is O(b^d)
b - branch factor
d - depth factor
So time will be exponential for this and the space will go linearly.

REFERENCES

[1] S. Russell and P. Norvig, "Artificial Intelligence: a Modern Approach," 4th ed.,Pearson, pp. 36-106
[2] Deepak Khemani, A first course in Artificial Intelligence, 2nd ed., McGraw Hill, pp.53-118.
[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.