Applied AI - M1

Artificial Intelligence (M.Tech)

Gaurav Ojha (D015)

Question 1/6 (1 p.)

The Data structure used in standard implementation of Breadth First Search is?
A. Stack
B. Queue
C. Linked List
D. Tree

Answer: Queue (B)

Question 2/6 (1 p.)

A person wants to visit some places. He starts from a vertex and then wants to visit every vertex till it finishes from one vertex, backtracks and then explore other vertex from same vertex. What algorithm he should use?
A. Depth First Search
B. Breadth First Search
C. A* Search
D. Greedy Best First Search

(I feel DFS would be the appropriate algorithm to solve the problem as we want to visit all the vertices. DFS allows us to explore each node to it's full depth and allows us to trackback, in order to explore the unexplored vertices/nodes from before.)

Question 3/6 (1 p.)

What is the equation of Manhattan's Distance?
A. |(X2+X1)| + |(Y2+Y1)|
B. $\sqrt{(X2-X1)^2+(Y2-Y1)^2}$
C. $\sqrt{(X2-X1)^2}+\sqrt{(Y2-Y1)^2}$
D. |(X2-X1)| + |(Y2-Y1)|

Answer: D

Question 4/6 (5 p.)

Explain Adversial Search with pseudocode.

Adversarial search is a search algorithm used in cases where we have an adversary, i.e. 2 or more players playing against each other. Games such as the game of GO, Chess, Monopoly or a simple game of TicTacToe are some of the examples where one might use adversarial search.

Adversarial search comes with optimal move. It has the complete game knowledge. The algorithm assumes that each of the players will make the best possible move such that it causes maximum damage to the opponent. One player's loss is other's advantage.

The algorithm is based on tree search. Where a decision has to be made after each move. The algorithm makes decision by taking into account all possible moves and contrasting it with goal state.

Let's write the pseudocode for Adversarial using min-max algorithm which is the most famous algorithm used in 2 player games. Here we use TicTacToe

Representing a Tic-Tac-Toe AI:

1. S_0: Initial state (in our case, an empty 3X3 board)
2. Players(s): a function that, given a state s, returns which player's turn it is (X or O).
3. Actions(s): a function that, given a state s, return all the legal moves in this state (what spots are free on the board).
4. Result(s, a): a function that, given a state s and action a, returns a new state. This is the board that resulted from performing the action a on state s (making a move in the game).
5. Terminal(s): a function that, given a state s, checks whether this is the last step in the game, i.e. if someone won or there is a tie. Returns True if the game has ended, False otherwise.
6. Utility(s): a function that, given a terminal state s, returns the utility value of the state: -1, 0, or 1.

```
function minimax_decision(state):
    if terminal(state):
        return None  // If the game is over, no action to take

    player = players(state)
    if player is maximizer:
        best_value = -∞
        best_action = None
        for action in actions(state):
            new_state = result(state, action)
            value = min_value(new_state)  // Calculate the minimum value of
the opponents moves
            if value > best_value:
```

```
                best_value = value
                best_action = action
        return best_action
    else:
        best_value = +∞
        best_action = None
        for action in actions(state):
            new_state = result(state, action)
            value = max_value(new_state)  // Calculate the maximum value of
the AI players moves
            if value < best_value:
                best_value = value
                best_action = action
        return best_action

function max_value(state):
    if terminal(state):
        return utility(state)  // Return the utility value for terminal
states
    v = -∞
    for action in actions(state):
        new_state = result(state, action)
        v = max(v, min_value(new_state))  // Get the maximum value among the
opponents moves
    return v

function min_value(state):
    if terminal(state):
        return utility(state)  // Return the utility value for terminal
states
    v = +∞
    for action in actions(state):
        new_state = result(state, action)
        v = min(v, max_value(new_state))  // Get the minimum value among the
AI players moves
    return v

# Define the game-specific functions
function players(state):
    # Returns the player whose turn it is (X or O)

function actions(state):
    # Returns a list of legal moves in the current state
```

```
function result(state, action):
    # Returns the new state that results from applying the action to the
current state

function terminal(state):
    # Returns True if the game is over, False otherwise

function utility(state):
    # Returns the utility value of the terminal state: -1, 0, or 1
```

Question 5/6 (12 p.)

Write a python program implementing any Search strategy to solve the arrange square puzzle (Image below).
Your code should be able to solve the two puzzles attached (one.txt, two.txt)

NOTE – The code should be well formatted and explained with the help of comments wherever required.
Answer the following – (1 mark)

A – Which Search strategy implemented –
B – No. of steps taken –
C – No. of states explored –

Marking scheme for programming question -
3 marks – code
2 mark – above 3 questions
4 mark – formatting & commenting & understanding
3 mark – for applying heuristic

Steps:

Solving using Breadth-First Search (BFS):

1. **Puzzle Representation**: The 8-puzzle problem is represented using a 3x3 grid where each cell contains a number from 1 to 8 and one empty space.
2. **Puzzle State Class**: A `PuzzleState` class is defined to represent a state of the puzzle. It includes the current state of the puzzle, the parent state that led to this state, and the action (move) taken to reach this state.
3. **Generating Successors**: The `generate_successors` method of the `PuzzleState` class generates all possible successor states by swapping the blank tile (0) with its neighboring tiles (up, down, left, right). These successors represent the valid moves from the current state.
4. **Breadth-First Search (BFS)**: BFS is implemented using a queue to explore states in a breadth-first manner. The algorithm starts with the initial state and iteratively explores its successors. Each state is added to the queue after being explored, and the algorithm continues until a solution state (goal state) is found or the queue becomes empty.
5. **Tracking Visited States**: To avoid revisiting states, a set called `visited` is used to keep track of already explored states.
6. **Solution Path**: When the goal state is found, the solution path is constructed by backtracking from the goal state to the initial state using the `parent` information of each state.
7. **Input Reading**: The initial state and goal state are read from an input file.
8. **Main Execution**: The main execution block reads the input filename from the command line arguments and reads the initial and goal states. It then applies the BFS algorithm to find a solution path.
9. **Output**: If a solution is found, the algorithm prints each step of the solution path, including the state at each step. If no solution is found, a message indicating that is printed.
10. **Total Explored States**: The total number of states explored during the BFS algorithm is counted and printed at the end.

BFS explores states layer by layer, starting from the initial state and moving outward in a breadth-first manner. It guarantees the shortest solution path in terms of the number of moves, but it might explore a large number of states depending on the complexity of the puzzle and the specific arrangement of the initial and goal states.

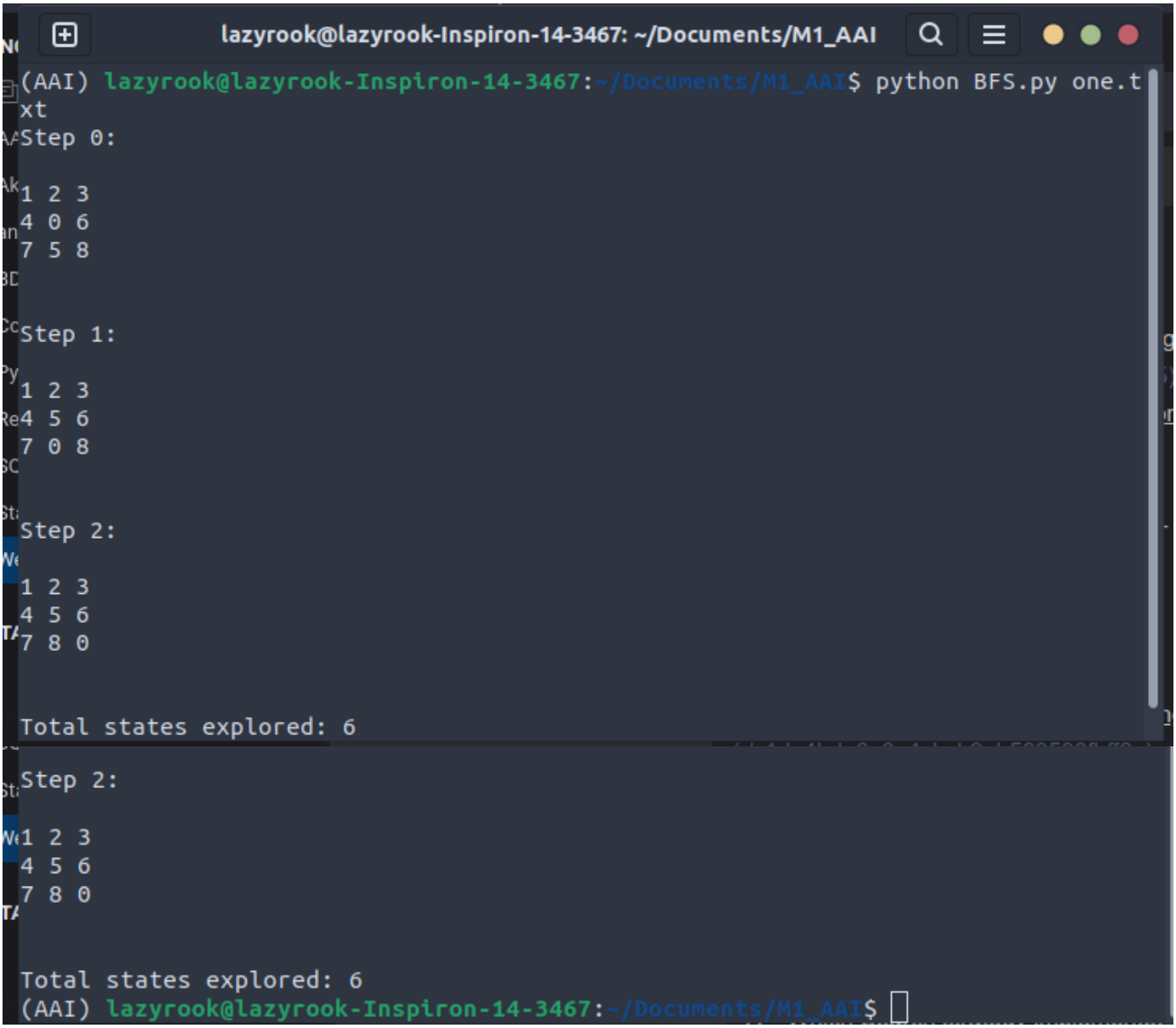
```
from collections import deque  # Import the 'deque' class for implementing a
queue

# Define a class to represent a state of the puzzle
class PuzzleState:
    def __init__(self, state, parent=None, action=None):
        self.state = state  # The current state of the puzzle
        self.parent = parent  # The parent state that led to this state
        self.action = action  # The action (row, column) taken to reach this
state
```

```
    def __eq__(self, other):
        return self.state == other.state  # Compare the states of two PuzzleState objects

    def __hash__(self):
        return hash(str(self.state))  # Calculate the hash value based on the string representation of the state

    def get_blank_position(self):
        for i, row in enumerate(self.state):  # Iterate through each row of the state
            if 0 in row:  # If 0 (blank tile) is found in the row
                return i, row.index(0)  # Return the row and column indices of the blank tile

    def generate_successors(self):
        successors = []  # List to store generated successor states
        blank_row, blank_col = self.get_blank_position()  # Get the current position of the blank tile

        moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # Define possible moves: (down, up, right, left)

        for dr, dc in moves:
            new_row, new_col = blank_row + dr, blank_col + dc  # Calculate the new position after the move

            if 0 <= new_row < 3 and 0 <= new_col < 3:  # Check if the new position is within the puzzle boundaries
                new_state = [row[:] for row in self.state]  # Create a copy of the current state
                new_state[blank_row][blank_col], new_state[new_row][new_col] = new_state[new_row][new_col], new_state[blank_row][blank_col]  # Perform the swap
                successors.append(PuzzleState(new_state, self, (blank_row, blank_col)))  # Append the generated successor state

        return successors  # Return the list of generated successor states


# Declare global variable explored_states
explored_states = 0
```

```
# Define the Breadth-First Search (BFS) function
def bfs(initial_state, goal_state):
    global explored_states
    visited = set()  # Set to store visited states
    queue = deque([PuzzleState(initial_state)])  # Queue to store states to
be explored

    while queue:
        current_state = queue.popleft()  # Get the current state from the
front of the queue
        visited.add(current_state)  # Mark the current state as visited
        explored_states += 1  # Increment the counter for explored states

        if current_state.state == goal_state:  # If the current state
matches the goal state
            path = []
            while current_state:
                path.append(current_state.state)
                current_state = current_state.parent
            return list(reversed(path))  # Return the reversed path from
initial state to goal state

        successors = current_state.generate_successors()  # Generate
successor states
        for successor in successors:
            if successor not in visited:  # If the successor state hasn't
been visited
                queue.append(successor)  # Add the successor to the queue
for exploration

    return None  # Return None if no solution is found


# Define a function to read the input from a file
def read_input(filename):
    with open(filename, 'r') as file:
        lines = file.readlines()  # Read all lines from the file

    initial_state = [[int(num) if num != ' ' else 0 for num in line.strip()]
for line in lines[:3]]  # Extract the initial state from the first 3 lines
of the file
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]  # Define the goal state
```

```
    return initial_state, goal_state  # Return the initial and goal states as a tuple


if __name__ == '__main__':
    import sys

    if len(sys.argv) != 2:
        print("Usage: python m1.py filename.txt")  # Print usage instructions if the command line argument is missing or incorrect
        sys.exit(1)  # Exit with an error code

    input_filename = sys.argv[1]  # Get the input filename from the command line argument
    initial_state, goal_state = read_input(input_filename)  # Read the initial and goal states from the input file

    solution = bfs(initial_state, goal_state)  # Solve the puzzle using BFS
    if solution:
        for step, state in enumerate(solution):
            print(f"Step {step}:\n")
            for row in state:
                print(" ".join(map(str, row)))  # Print each step of the solution
            print("\n")
    else:
        print("No solution found.")  # Print a message if no solution is found

    print("Total states explored:", explored_states)  # Print the total number of explored states
```

one.txt



A – Which Search strategy implemented – DFS

B – No. of steps taken – 3

C – No. of states explored – 6

two.txt

```
Total states explored: 6
(AAI) lazyrook@lazyrook-Inspiron-14-3467:~/Documents/M1_AAI$ python BFS.py two.txt
Step 0:

6 8 1
4 7 3
5 0 2


Step 1:

6 8 1
4 0 3
5 7 2


Step 2:

6 0 1
4 8 3
5 7 2


Step 3:

6 1 0
4 8 3
5 7 2


Step 4:
0 1 3
4 2 6
7 5 8


Step 22:

1 0 3
4 2 6
7 5 8


Step 23:

1 2 3
4 0 6
7 5 8


Step 24:

1 2 3
4 5 6
7 0 8


Step 25:

1 2 3
4 5 6
7 8 0


Total states explored: 310387
(AAI) lazyrook@lazyrook-Inspiron-14-3467:~/Documents/M1_AAI$
```

A – Which Search strategy implemented – DFS

B – No. of steps taken – 26

C – No. of states explored – 310387