01/10/2024

Lab-01

1. Program Title: Tic Tac Toe game

Code:

```
import random


def check_win(board, r, c):
    ch = 'X' if board[r - 1][c - 1] == 'X' else 'O'
    # Check the row
    if all(cell == ch for cell in board[r - 1]):
        return True
    # Check the column
    if all(board[i][c - 1] == ch for i in range(3)):
        return True
    # Check main diagonal
    if r == c and all(board[i][i] == ch for i in range(3)):
        return True
    # Check anti-diagonal
    if r + c == 4 and all(board[i][2 - i] == ch for i in range(3)):
        return True
    return False


def display_board(board):
    for row in board:
        print(" | ".join(row))
    print()


def find_block_move(board):
    # Check rows and columns for blocking opportunity
    for i in range(3):
        # Check rows
```

```
        if board[i].count('X') == 2 and board[i].count('-') == 1:
            return i, board[i].index('-')
        # Check columns
        col = [board[0][i], board[1][i], board[2][i]]
        if col.count('X') == 2 and col.count('-') == 1:
            return col.index('-'), i
    # Check diagonals for blocking opportunity
    diag1 = [board[0][0], board[1][1], board[2][2]]
    if diag1.count('X') == 2 and diag1.count('-') == 1:
        idx = diag1.index('-')
        return idx, idx
    diag2 = [board[0][2], board[1][1], board[2][0]]
    if diag2.count('X') == 2 and diag2.count('-') == 1:
        idx = diag2.index('-')
        return idx, 2 - idx
    return None  # No blocking move found

def find_winning_move(board):
    # Check rows and columns for winning opportunity
    for i in range(3):
        # Check rows
        if board[i].count('O') == 2 and board[i].count('-') == 1:
            return i, board[i].index('-')
        # Check columns
        col = [board[0][i], board[1][i], board[2][i]]
        if col.count('O') == 2 and col.count('-') == 1:
            return col.index('-'), i
    # Check diagonals for winning opportunity
    diag1 = [board[0][0], board[1][1], board[2][2]]
    if diag1.count('O') == 2 and diag1.count('-') == 1:
        idx = diag1.index('-')
```

```
        return idx, idx
    diag2 = [board[0][2], board[1][1], board[2][0]]
    if diag2.count('O') == 2 and diag2.count('-') == 1:
        idx = diag2.index('-')
        return idx, 2 - idx
    return None  # No winning move found

def bot_move(board):
    # First, check if there's a move to win
    winning_move = find_winning_move(board)
    if winning_move:
        r, c = winning_move
        board[r][c] = 'O'
        print(f"Bot placed O at winning position: ({r + 1}, {c + 1})")
        display_board(board)
        return r + 1, c + 1

    # Then, check if there's a move to block the human
    block_move = find_block_move(board)
    if block_move:
        r, c = block_move
        board[r][c] = 'O'
        print(f"Bot blocked X at position: ({r + 1}, {c + 1})")
        display_board(board)
        return r + 1, c + 1

    # Otherwise, make a random move
    available_moves = [(r, c) for r in range(3) for c in range(3) if board[r][c] == '-']
    if available_moves:
        move = random.choice(available_moves)
        board[move[0]][move[1]] = 'O'
```

```
        print(f"Bot placed O at position: ({move[0] + 1}, {move[1] + 1})")
        display_board(board)
        return move[0] + 1, move[1] + 1  # Return the move for win check
    return None, None

# Initial board setup
board = [['-', '-', '-'], ['-', '-', '-'], ['-', '-', '-']]
display_board(board)

xo = 1  # 1 for human, 0 for bot
flag = 0  # Flag to check for win or draw

while '-' in board[0] or '-' in board[1] or '-' in board[2]:

    if xo == 1:  # Human's turn (X)
        print("Enter position to place X (row and column between 1-3):")
        x = int(input())
        y = int(input())
        if x > 3 or y > 3 or x < 1 or y < 1:
            print("Invalid position")
            continue
        if board[x - 1][y - 1] == '-':
            board[x - 1][y - 1] = 'X'
            xo = 0  # Switch to bot's turn
            display_board(board)
        else:
            print("Invalid position")
            continue

        if check_win(board, x, y):
            print("X wins!")
```

```
            flag = 1
            break

    else:  # Bot's turn (O)
        print("Bot's turn:")
        x, y = bot_move(board)
        if x and y:  # If bot made a valid move
            xo = 1  # Switch back to human's turn
            if check_win(board, x, y):
                print("O (Bot) wins!")
                flag = 1
                break

if flag == 0:
    print("Draw")
print("Game Over")
```

Algorithm:

01|10|2024

program title : tic tac toe gam

algorithm:

check_win(board, r, c):

step1:

determine the letter placed ('x' (or) 'o')

step2:

check for a win in the row, column and diagonals. return true if a win is found otherwise return false

display_board(board):

step1:

print the board

find_block_move(board):

step1:

look for two 'x's in a row, column (or) diagonal with one empty space ('-')

step2:

return the blocking position (or) none

bot_move(board):

step1:

call find_block_move(board) if found place 'o' there

step2:

if no block is needed choose a random available move

main algorithm

step1:

initialize a 3x3 board with '-'

step2:

set 'x_o' (1 for player 0 for bot) and 'flag' (to check game status)

step3:

while there are empty spots

if players turn (x):

(i) prompt for row and column

(ii) validate and place 'x' check for win

if bots turn (o):

(i) call bot_move(board) check for win

step4:

if no winner print 'draw'

~~step~~

step5:

print 'game over'

output,

```
- | - | -
- | - | -
- | - | -
```

enter position to place x (row and column between 1-3)

1
1

```
x | - | -
- | - | -
- | - | -
```

bot's turn:

bot placed 0 at position (2,3)

```
x | - | -
- | - | 0
- | - | -
```

~~enter position to place x (betwe~~

enter position to place x (row and column between 1-3)

2
2

```
x | - | -
- | x | 0
- | - | -
```

bot's turn:

bot blocked x at position (3,3)

```
x | - | -
- | x | 0
- | - | 0
```

enter position to place x (row and column between 1-3)

2
1

```
x | - | -
x | x | 0
- | - | 0
```

bot's turn:

bot placed 0 at winning position (1,3)

```
x | - | 0
x | x | 0
- | - | 0
```

0(bot) wins

1/10/24

game over

program title : vaccum cleaner

algorithm,

1. vaccum_cleaner_agent(perce

   input : a percept containin

   step1 : extract location and

   step2 :

   if status is "Dirty

   Action: Return

   else if location is "A

   Action : Return

   else if location is "B

   Action : Return

   else :

   Action: Return "

2. main algorithm :

   step 1 : initialize a list of over time

   step 2 : create an empty li

   step3 : for each percept i

   call vaccum-cl

   append the ac corresponding

   step 4 : print the final pe

output,

percept : ['A', 'clean'] , acti

percept : ['A', 'dirty'] , acti

percept : ['B', 'clean'], acti

percept : ['B', 'dirty'] , actic

percept : ['A', 'clean'] , actio

percept : ['A', 'clean'] , acti

percept sequence : [['A', 'clea

['A', 'cle

action sequence : ['right', '

game over

program title : vaccum cleaner

algorithm:

1. vaccum_cleaner_agent(percept):

input : a percept containing the current location and its status (Eg: ['A', 'Dirty'])

step1 : extract location and status from the percept

step2 :

if status is "Dirty" :

Action: Return "suck" (clean the current location)

else if location is "A":

Action : Return "right" (move to location B)

else if location is "B":

Action : Return "left" (move to location A)

else :

Action: Return "No op" (this case should not occur in this simple world)

2. main algorithm :

step 1 : initialize a list of percepts representing the state of the environment over time

step 2 : create an empty list to hold actions

step3 : for each percept in the percept sequence :

call vaccum_cleaner_agent(percept) to determine the action

append the action to the action list and print the percept and corresponding action

step 4 : print the final percept and action sequences

output :

percept : ['A', 'clean'] , action : right

percept : ['A', 'dirty'] , action : suck

percept : ['B', 'clean'], action : left

percept : ['B', 'dirty'] , action : suck

percept : ['A', 'clean'] , action : right

percept : ['A', 'clean'] , action : right

percept sequence : [['A', 'clean'], ['A', 'dirty'], ['B', 'clean'], ['B', 'dirty'], ['A', 'clean'], ['A', 'clean']]

action sequence : ['right', 'suck', 'left', 'suck', 'right', 'right']

Output:

- | - | -

- | - | -

- | - | -

Enter position to place X (row and column between 1-3):

1

1

X | - | -

- | - | -

- | - | -

Bot's turn:

Bot placed O at position: (3, 2)

X | - | -

- | - | -

- | O | -

Enter position to place X (row and column between 1-3):

2

2

X | - | -

- | X | -

- | O | -

Bot's turn:

Bot blocked X at position: (3, 3)

X | - | -

- | X | -

- | O | O

Enter position to place X (row and column between 1-3):

3

1

X | - | -

- | X | -

X | O | O

Bot's turn:

Bot blocked X at position: (2, 1)

X | - | -

O | X | -

X | O | O

Enter position to place X (row and column between 1-3):

1

3

X | - | X

O | X | -

X | O | O

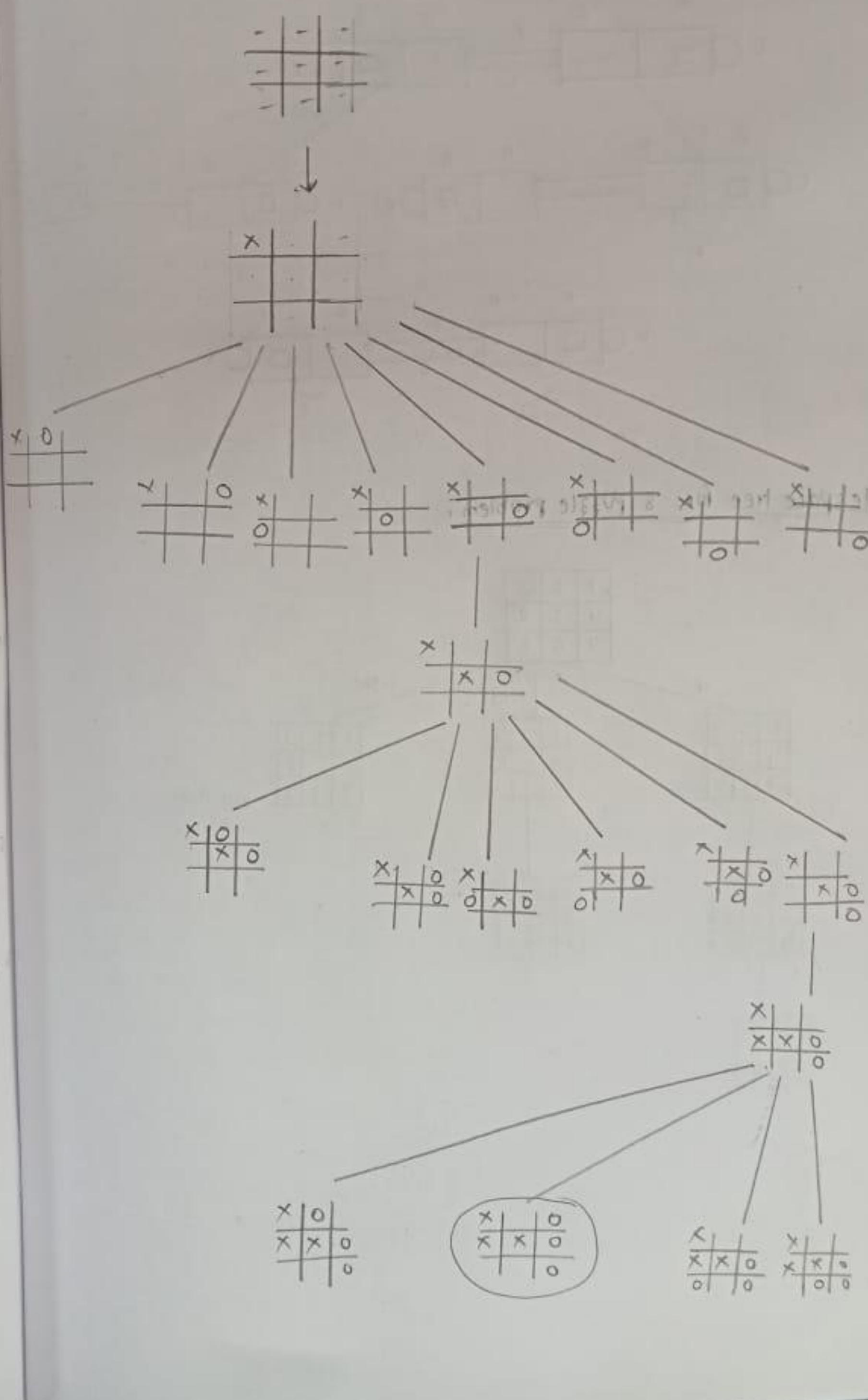
X wins!

Game Over

State space tree:

~~state space tree~~

tic tac toe game state space tree



2. Program Title: vacuum cleaner

Code:

```
def vacuum_cleaner_agent(percept):
  """
  A simple vacuum cleaner agent that operates in a two-location world.

  Args:
    percept: A list containing the current location and whether it is dirty.
          e.g., ['A', 'Dirty']

  Returns:
    The action to be taken by the agent (Left, Right, Suck, NoOp).
  """

  location, status = percept

  if status == 'Dirty':
    return 'Suck'
  elif location == 'A':
    return 'Right'
  elif location == 'B':
    return 'Left'
  else:
    return 'NoOp'  # Should not reach here in this simple world.


# Example percept sequence and action execution
percepts = [['A', 'Clean'], ['A', 'Dirty'], ['B', 'Clean'], ['B', 'Dirty'], ['A', 'Clean'], ['A', 'Clean']]
actions = []

for percept in percepts:
```

```
  action = vacuum_cleaner_agent(percept)
  actions.append(action)
  print(f"Percept: {percept}, Action: {action}")

print("\nPercept Sequence:", percepts)
print("Action Sequence:", actions)
```

Algorithm:

game over

program title : vaccum cleaner

algorithm :

1. vaccum_cleaner_agent(percept):

input : a percept containing the current location and its status (eg: ['A', 'Dirty'])

step1 : extract location and status from the percept

step2 :

if status is "Dirty" :

Action: Return "suck" (clean the current location)

else if location is "A":

Action : Return "right" (move to location B)

else if location is "B":

Action : Return "left" (move to location A)

else :

Action: Return "No op" (this case should not occur in this simple world)

2. main algorithm :

step 1 : initialize a list of percepts representing the state of the environment over time

step 2 : create an empty list to hold actions

step3 : for each percept in the percept sequence :

call vaccum_cleaner_agent(percept) to determine the action

append the action to the action list and print the percept and corresponding action

step4 : print the final percept and action sequences

output :

percept : ['A', 'clean'] , action : right

percept : ['A', 'dirty'] , action : suck

percept : ['B', 'clean'], action : left

percept : ['B', 'dirty'] , action : suck

percept : ['A', 'clean'] , action : right

percept : ['A', 'clean'] , action : right

percept sequence : [['A', 'clean'], ['A', 'dirty'], ['B', 'clean'], ['B', 'dirty'], ['A', 'clean'], ['A', 'clean']]

action sequence : ['right', 'suck', 'left', 'suck', 'right', 'right']

Output:

Percept: ['A', 'Clean'], Action: Right

Percept: ['A', 'Dirty'], Action: Suck

Percept: ['B', 'Clean'], Action: Left

Percept: ['B', 'Dirty'], Action: Suck

Percept: ['A', 'Clean'], Action: Right

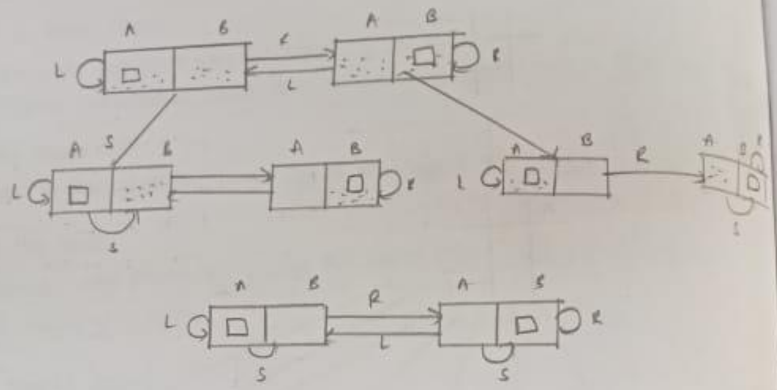
Percept: ['A', 'Clean'], Action: Right

Percept Sequence: [['A', 'Clean'], ['A', 'Dirty'], ['B', 'Clean'], ['B', 'Dirty'], ['A', 'Clean'], ['A', 'Clean']]

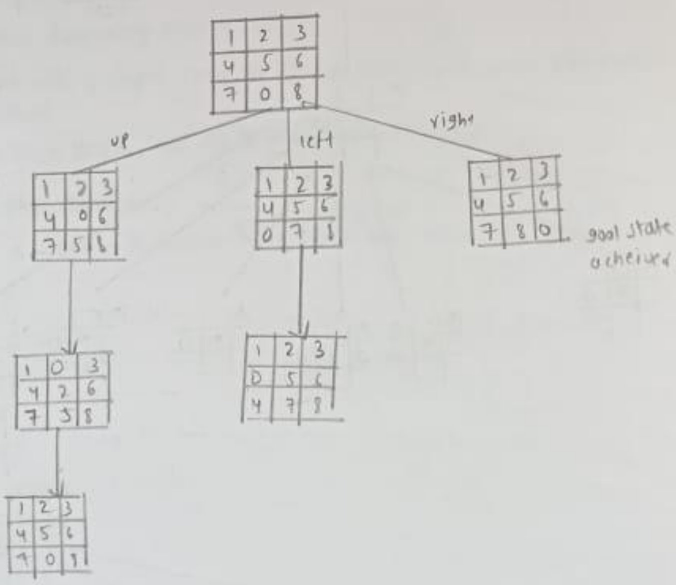
Action Sequence: ['Right', 'Suck', 'Left', 'Suck', 'Right', 'Right']

State space tree:

state space tree for vaccum cleaner agent :



statespace tree for 8 puzzle problem :



08/10/2024

Lab-02

3. Program title: 8 puzzle problem

Code:

```
import copy

# Directions for movement: up, down, left, right
moves = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}

# Check if a state is the goal state
def is_goal(state, goal_state):
    return state == goal_state

# Get the position of the empty space (0)
def get_empty_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

# Move the empty space in a specified direction if possible
def move_tile(state, direction):
    new_state = copy.deepcopy(state)
    empty_i, empty_j = get_empty_position(state)
    di, dj = moves[direction]
    new_i, new_j = empty_i + di, empty_j + dj
    if 0 <= new_i < 3 and 0 <= new_j < 3:
        new_state[empty_i][empty_j], new_state[new_i][new_j] = new_state[new_i][new_j],
new_state[empty_i][empty_j]
        return new_state
    return None
```

```
# Depth-limited search
def depth_limited_search(state, goal_state, depth_limit, path):
    if is_goal(state, goal_state):
        return state, path

    if depth_limit == 0:
        return None, []

    empty_i, empty_j = get_empty_position(state)
    for direction in moves:
        new_state = move_tile(state, direction)
        if new_state is not None and new_state not in path:  # Avoid loops
            result, new_path = depth_limited_search(new_state, goal_state, depth_limit - 1, path +
[new_state])
            if result:
                return result, new_path

    return None, []

# Iterative deepening search
def iterative_deepening_search(initial_state, goal_state):
    depth = 0
    while True:
        result, path = depth_limited_search(initial_state, goal_state, depth, [initial_state])
        if result is not None:
            return path, depth
        depth += 1

# Print the state of the puzzle
def print_state(state):
```

```
    for row in state:
        print(row)
    print()

# Test the 8-puzzle
initial_state = [
    [1, 2, 3],
    [4, 0, 5],
    [6, 7, 8]
]

goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

# Solve the puzzle using iterative deepening search
solution_path, depth = iterative_deepening_search(initial_state, goal_state)

# Output the steps
print(f"Solution found in {depth} steps.\n")
print("Steps to reach the goal:")

for i, state in enumerate(solution_path):
    print(f"Step {i}:")
    print_state(state)
```

Algorithm:

08|10|2024

program title : 8 puzzle problem

algorithm :

1. is_goal(state, goal_state) :

input :

state : current state of the puzzle

~~goa~~

goal_state : target configuration

Process:

check if state is equal to goal_state

output :

return true if equal, otherwise return false.

2. get_empty_position(state) :

input :

state: current state of the puzzle

process:

iterate through the 3x3 grid to find the position of 0 (empty space)

output :

return the coordinates (i,j) of the empty space

3. move_tile(state, direction):

input:

state : current state of the puzzle

direction: one of ['up', 'down', 'left', 'right']

process :

~~Create~~

create a deep copy of state

get the position of the empty space

calculate the new position based on direction

if valid, swap the empty space with the adjacent tile

output :

return the new state if valid. otherwise return none

4. depth_limited_search(state, goal_state, depth_limit, path):

input :

state, goal_state, depth_limit, path: current path taken

process:

if state is the goal return state and path

if depth_limit is 0 return none and an empty list

get the empty sp

for each direction

if the new

- search rec

if a solution

output :

return none and an

5. iterative_deeping_searc

input :

initial_state, goal_

process :

initialize depth to 0

loop:

call depth_limited

if a solution is fou

increment depth

output :

return the solution

6. print_state (state):

input :

state : current state

process :

print each row of th

output :

display the current

test 8 puzzle :

~~1- define~~

1. define initial_state a

2. call iterative_deepin

3. print the solution pa

output.

Solution found in 14 steps

Steps to reach the goal:

Step 0 :

get the empty space position
for each direction, attemp to move:
if the new state is valid and not visited, call depth-limited-search recursively
if a solution is found, return it

output:
return none and an empty list if no solution is found

5. iterative_deeping_search(initial_state, goal_state):

input:
initial_state, goal_state.

process:
initialize depth to 0
loop:
call depth_limited_search with current depth
if a solution is found return the path and depth
increment depth

output:
return the solution path and depth when found

6. print_state(state):

input:
state: current state of the puzzle

process:
print each row of the state

output:
display the current configuration of the puzzle

test 8_puzzle:

~~1-define~~.

1. define initial_state and goal_state
2. call iterative_deeping_search with the initial_state and goal_state
3. Print the solution path and the number of steps taken to reach the goal.

output.

Solution found in 14 steps
Steps to reach the goal:
Step0:

[1,2,3]
[4,0,5]
[6,7,8]

1 Step 1:
[1,2,3]
[4,5,0]
[6,7,8]

Step 2:
[1,2,3]
[4,5,8]
[6,7,0]

Step 3:
[1,2,3]
[4,5,8]
2 [6,0,7]

Step 4:
[1,2,3]
[4,5,8]
[0,6,7]

Step 5:
[1,2,3]
[0,5,8]
[4,6,7]

3 Step 6:
[1,2,3]
[5,0,8]
[4,6,7]

Step 7:
[1,2,3]
[5,6,8]
[4,0,7]

Step 8:
[1,2,3]
[5,6,0]
[4,7,8]

Step 9:
[1,2,3]
[5,6,0]
4 [4,7,8]

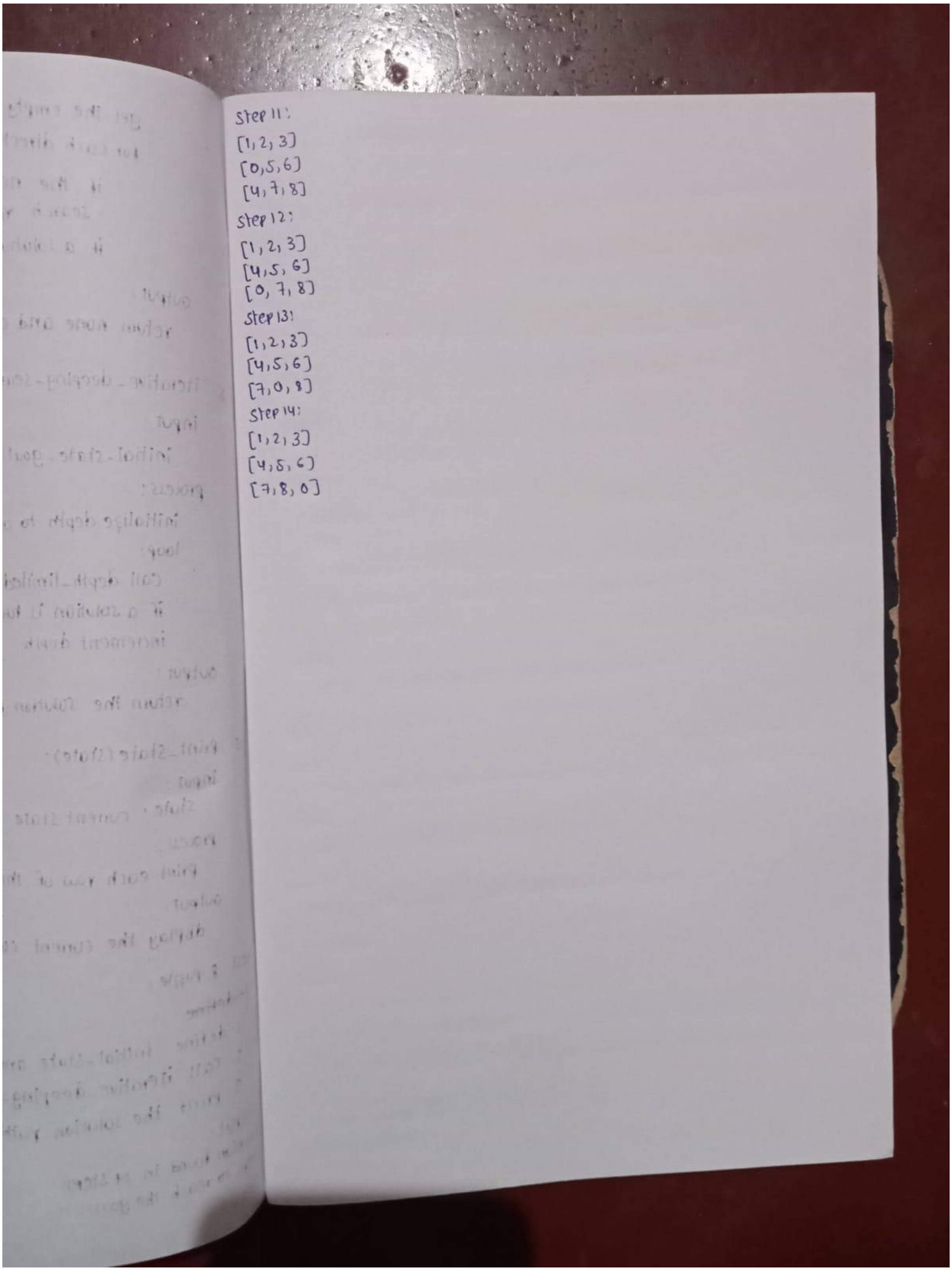
Step 10:
[1,2,3]
[5,0,6]
[4,7,8]

Step 11:
[1,2,3]
[0,5,6]
[4,7,8]

Step 12:
[1,2,3]
[4,5,6]
[0,7,8]

Step 13:
[1,2,3]
[4,5,6]
[7,0,8]

Step 14:
[1,2,3]
[4,5,6]
[7,8,0]



step 11:
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
step 12:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
step 13:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
step 14:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Output:

Solution found in 14 steps.

Steps to reach the goal:

Step 0:

[1, 2, 3]

[4, 0, 5]

[6, 7, 8]

Step 1:

[1, 2, 3]

[4, 5, 0]

[6, 7, 8]

Step 2:

[1, 2, 3]

[4, 5, 8]

[6, 7, 0]

Step 3:

[1, 2, 3]

[4, 5, 8]

[6, 0, 7]

Step 4:

[1, 2, 3]

[4, 5, 8]

[0, 6, 7]

Step 5:

[1, 2, 3]

[0, 5, 8]

[4, 6, 7]

Step 6:

[1, 2, 3]

[5, 0, 8]

[4, 6, 7]

Step 7:

[1, 2, 3]

[5, 6, 8]

[4, 0, 7]

Step 8:

[1, 2, 3]

[5, 6, 8]

[4, 7, 0]

Step 9:

[1, 2, 3]

[5, 6, 0]

[4, 7, 8]

Step 10:

[1, 2, 3]

[5, 0, 6]

[4, 7, 8]

Step 11:

[1, 2, 3]

[0, 5, 6]

[4, 7, 8]

Step 12:

[1, 2, 3]

[4, 5, 6]

[0, 7, 8]

Step 13:

[1, 2, 3]

[4, 5, 6]

[7, 0, 8]

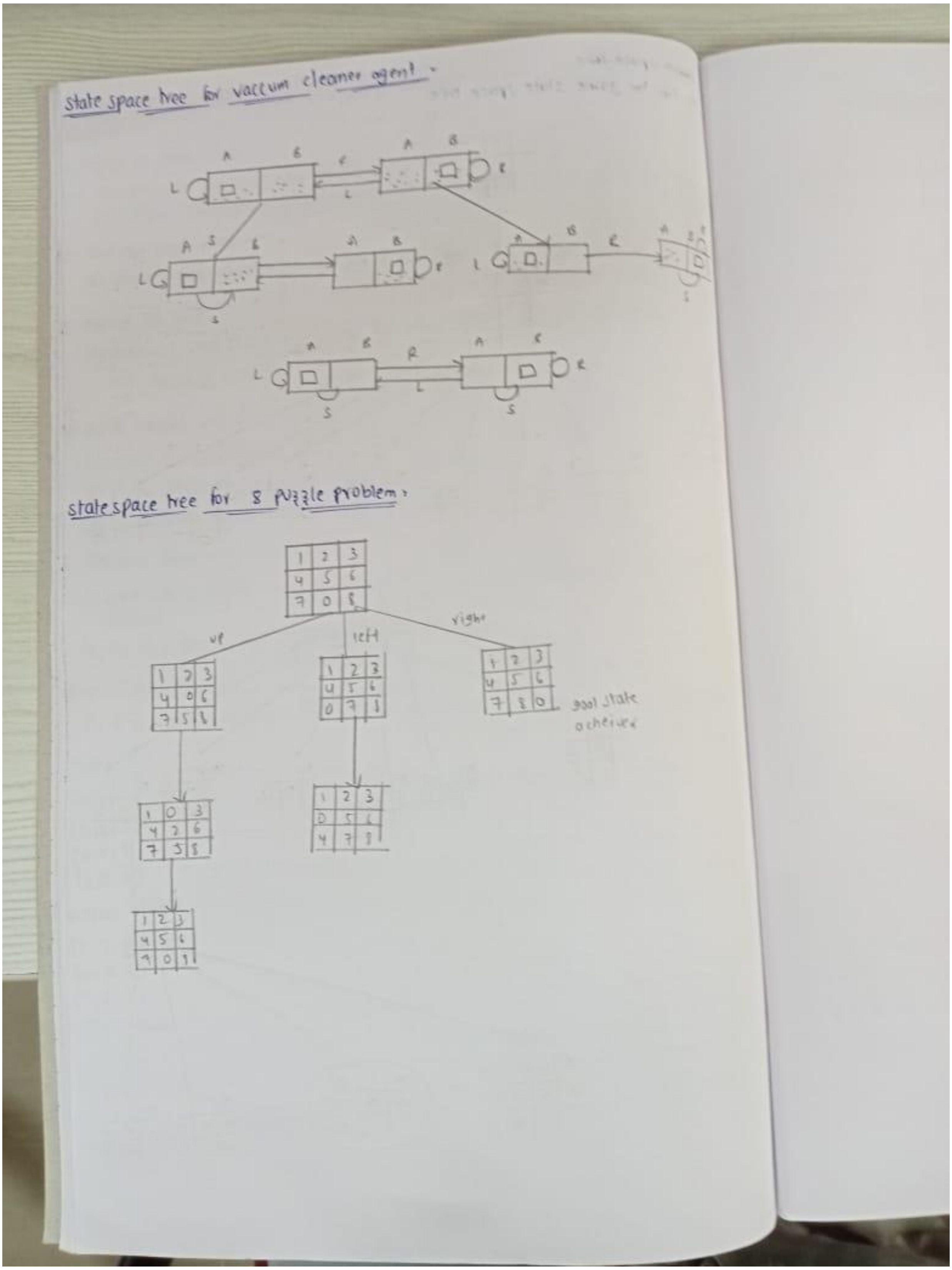
Step 14:

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

State space tree:



4. Program title: Implement Iterative deepening search algorithm.

Code:

```
import copy


class Node:
    def __init__(self, state, parent=None, action=None, depth=0):
```

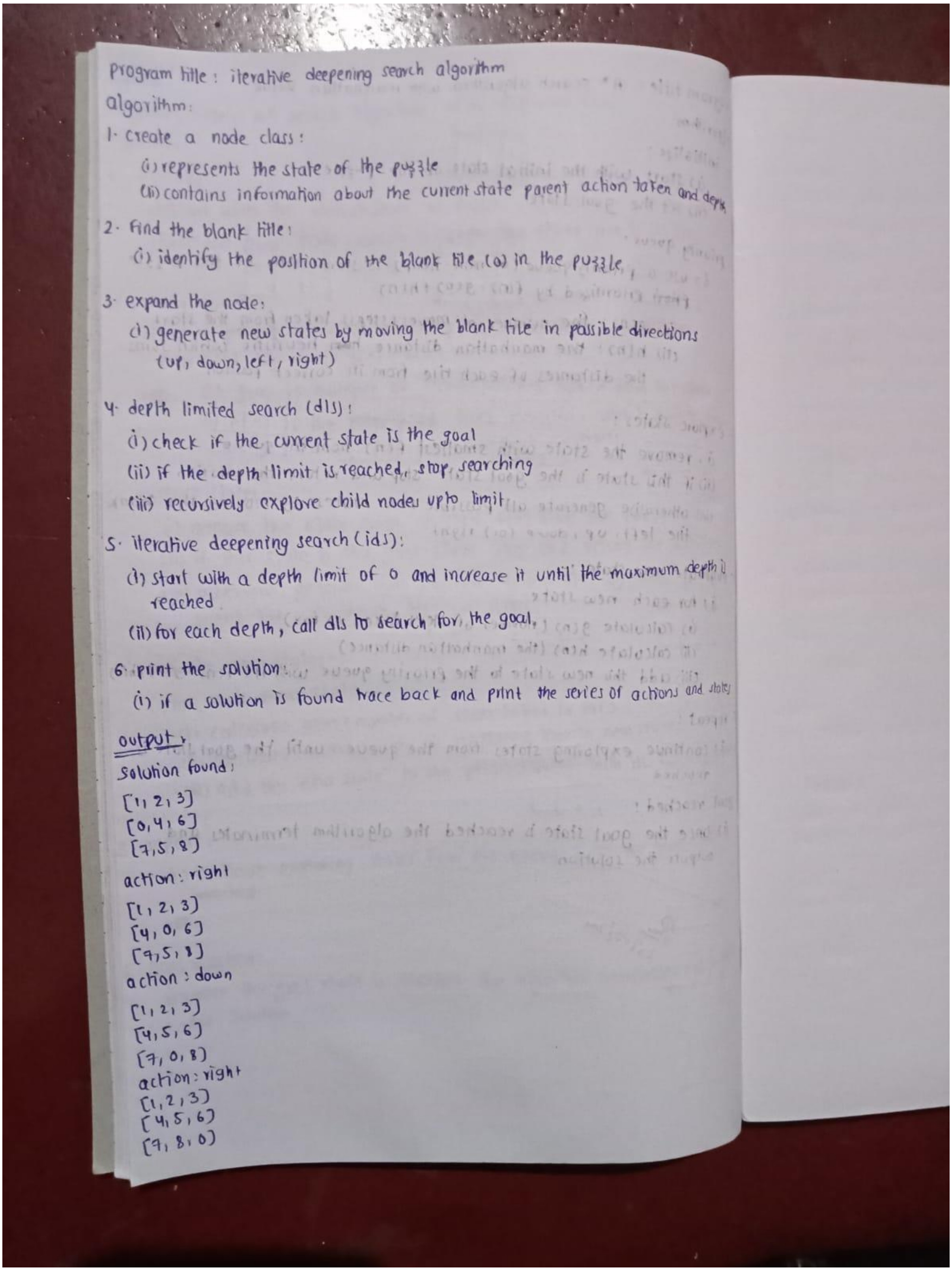
```
    self.state = state
    self.parent = parent
    self.action = action
    self.depth = depth

def __lt__(self, other):
    return self.depth < other.depth

def expand(self):
    children = []
    row, col = self.find_blank()
    possible_actions = []
```

```
        if row > 0:  # Can move the blank tile up
            possible_actions.append('Up')
        if row < 2:  # Can move the blank tile down
            possible_actions.append('Down')
        if col > 0:  # Can move the blank tile left
            possible_actions.append('Left')
        if col < 2:  # Can move the blank tile right
            possible_actions.append('Right')

        for action in possible_actions:
            new_state = copy.deepcopy(self.state)
            if action == 'Up':
                new_state[row][col], new_state[row - 1][col] = new_state[row - 1][col], new_state[row][col]
            elif action == 'Down':
                new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col],
new_state[row][col]
            elif action == 'Left':
                new_state[row][col], new_state[row][col - 1] = new_state[row][col - 1], new_state[row][col]
            elif action == 'Right':
                new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1],
new_state[row][col]
            children.append(Node(new_state, self, action, self.depth + 1))
        return children

    def find_blank(self):
        for row in range(3):
            for col in range(3):
                if self.state[row][col] == 0:
                    return row, col

def depth_limited_search(node, goal_state, limit):
    if node.state == goal_state:
```

```
        return node
    if node.depth >= limit:
        return None
    for child in node.expand():
        result = depth_limited_search(child, goal_state, limit)
        if result is not None:
            return result
    return None

def iterative_deepening_search(initial_state, goal_state, max_depth):
    for depth in range(max_depth):
        result = depth_limited_search(Node(initial_state), goal_state, depth)
        if result is not None:
            return result
    return None

def print_solution(node):
    path = []
    while node is not None:
        path.append((node.action, node.state))
        node = node.parent
    path.reverse()
    for action, state in path:
        if action:
            print(f"Action: {action}")
        for row in state:
            print(row)
        print()

# Example usage
initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]
```

```
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
max_depth = 20
solution = iterative_deepening_search(initial_state, goal_state, max_depth)

if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("Solution not found.")
```

Algorithm:


Program title: iterative deepening search algorithm

algorithm:

1. create a node class:
   (i) represents the state of the puzzle
   (ii) contains information about the current state parent action taken and depth

2. Find the blank tile:
   (i) identify the position of the blank tile (0) in the puzzle,

3. expand the node:
   (i) generate new states by moving the blank tile in possible directions (up, down, left, right)

4. depth limited search (dls):
   (i) check if the current state is the goal
   (ii) if the depth limit is reached, stop searching
   (iii) recursively explore child nodes upto limit

5. iterative deepening search (ids):
   (i) start with a depth limit of 0 and increase it until the maximum depth is reached
   (ii) for each depth, call dls to search for the goal,

6. print the solution:
   (i) if a solution is found trace back and print the series of actions and states

output:

Solution found:

[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

action: right

[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

action: down

[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

action: right

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Output:

Solution found:

[1, 2, 3]

[0, 4, 6]

[7, 5, 8]

Action: Right

[1, 2, 3]

[4, 0, 6]

[7, 5, 8]

Action: Down

[1, 2, 3]

[4, 5, 6]

[7, 0, 8]

Action: Right

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]