Artificial Intelligence Assignment 4

By Himanshu Sardana

Question 1

If the initial and final states are as below and $H(n)$: number of misplaced tiles in the current state n as compared to the goal node need to be considered as the heuristic function. You need to use Best First Search algorithm

2		3
1	8	4
7	6	5

1	2	3
8		4
7	6	5

```
import copy

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

def h_score(state):
    score = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != final[i][j]:
                score += 1
    return score

def find_zero(matrix):
    for i in range(3):
        for j in range(3):
            if matrix[i][j] == 0:
                return i, j

def move_up(matrix):
    i, j = find_zero(matrix)
    if i == 0:
        return None
    else:
        new_matrix = copy.deepcopy(matrix)
        new_matrix[i][j], new_matrix[i-1][j] = new_matrix[i-1][j],
↪   new_matrix[i][j]
        return new_matrix
```

```
def move_down(matrix):
    i, j = find_zero(matrix)
    if i == 2:
        return None
    else:
        new_matrix = copy.deepcopy(matrix)
        new_matrix[i][j], new_matrix[i+1][j] = new_matrix[i+1][j],
↪   new_matrix[i][j]
        return new_matrix

def move_left(matrix):
    i, j = find_zero(matrix)
    if j == 0:
        return None
    else:
        new_matrix = copy.deepcopy(matrix)
        new_matrix[i][j], new_matrix[i][j-1] = new_matrix[i][j-1],
↪   new_matrix[i][j]
        return new_matrix

def move_right(matrix):
    i, j = find_zero(matrix)
    if j == 2:
        return None
    else:
        new_matrix = copy.deepcopy(matrix)
        new_matrix[i][j], new_matrix[i][j+1] = new_matrix[i][j+1],
↪   new_matrix[i][j]
        return new_matrix

def is_goal(matrix):
    return matrix == final

def print_matrix(matrix):
    for i in matrix:
        for j in i:
            if j == 0:
                print(' ', end=' ')
            else:
                print(j, end=' ')
        print("")
    print()

def best_first_search(matrix):
    moves = [move_up, move_down, move_left, move_right]
    open_set = [(h_score(matrix), matrix, [])]  # (heuristic, state, path)
    visited = set()

    while open_set:
        open_set.sort()
        _, current, path = open_set.pop(0)
```

```
        if is_goal(current):
            return path

        matrix_tuple = tuple(map(tuple, current))
        if matrix_tuple in visited:
            continue
        visited.add(matrix_tuple)

        for move in moves:
            new_matrix = move(current)
            if new_matrix and tuple(map(tuple, new_matrix)) not in visited:
                open_set.append((h_score(new_matrix), new_matrix, path +
↪   [new_matrix]))

    return None

result = best_first_search(initial)

if result:
    print("Solution found:")
    for step in result:
        print_matrix(step)
else:
    print("No solution found.")
```

Question 2

If the initial and final states have been changed as below and the approach you need to use is Hill Climbing searching algorithm.
$H(n) : number of misplaced tiles in the current state as compared to goal node$ **as the heuristic function for the following states**

2	8	3
1	5	4
7	6	

1	2	3
8		4
7	6	5

```
import copy

initial = [[2, 8, 3], [1, 5, 4], [7, 6, 0]]
final = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
```

```
def h_score(state):
    score = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != final[i][j]:
                score += 1
    return score

def find_zero(matrix):
    for i in range(3):
        for j in range(3):
            if matrix[i][j] == 0:
                return i, j

def move_up(matrix):
    i, j = find_zero(matrix)
    if i == 0:
        return None
    else:
        new_matrix = copy.deepcopy(matrix)
        new_matrix[i][j], new_matrix[i-1][j] = new_matrix[i-1][j],
 ↪  new_matrix[i][j]
        return new_matrix

def move_down(matrix):
    i, j = find_zero(matrix)
    if i == 2:
        return None
    else:
        new_matrix = copy.deepcopy(matrix)
        new_matrix[i][j], new_matrix[i+1][j] = new_matrix[i+1][j],
 ↪  new_matrix[i][j]
        return new_matrix

def move_left(matrix):
    i, j = find_zero(matrix)
    if j == 0:
        return None
    else:
        new_matrix = copy.deepcopy(matrix)
        new_matrix[i][j], new_matrix[i][j-1] = new_matrix[i][j-1],
 ↪  new_matrix[i][j]
        return new_matrix

def move_right(matrix):
    i, j = find_zero(matrix)
    if j == 2:
        return None
    else:
        new_matrix = copy.deepcopy(matrix)
```

```
        new_matrix[i][j], new_matrix[i][j+1] = new_matrix[i][j+1],
↪   new_matrix[i][j]
        return new_matrix

def is_goal(matrix):
    return matrix == final

def print_matrix(matrix):
    for i in matrix:
        for j in i:
            if j == 0:
                print(' ', end=' ')
            else:
                print(j, end=' ')
        print("")
    print()

def hill_climbing_search(matrix):
    moves = [move_up, move_down, move_left, move_right]
    current = matrix
    path = [current]

    while True:
        best_move = None
        best_h = h_score(current)

        for move in moves:
            new_matrix = move(current)
            if new_matrix:
                new_h = h_score(new_matrix)
                if new_h < best_h:
                    best_h = new_h
                    best_move = new_matrix

        if best_move is None or is_goal(current):
            break

        current = best_move
        path.append(current)

    return path if is_goal(current) else None

result = hill_climbing_search(initial)

if result:
    print("Solution found:")
    for step in result:
        print_matrix(step)
else:
    print("No solution found.")
```

Question 3

Apply A* searching algorithm by taking $H(n) : number of correctly placed tiles in the current state n as compared to the goal node$ as the heuristic function

2		3
1	8	4
7	6	5

1	2	3
8		4
7	6	5

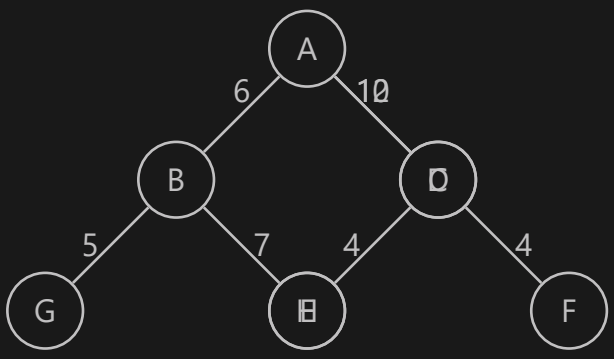
```
import copy

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

def h_score(state):
    score = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] == final[i][j]:
                score += 1
    return score

def find_zero(matrix):
    for i in range(3):
        for j in range(3):
            if matrix[i][j] == 0:
                return i, j

def move_up(matrix):
    i, j = find_zero(matrix)
    if i == 0:
        return None
    else:
        new_matrix = copy.deepcopy(matrix)
        new_matrix[i][j], new_matrix[i-1][j] = new_matrix[i-1][j],
↪   new_matrix[i][j]
        return new_matrix

def move_down(matrix):
    i, j = find_zero(matrix)
    if i == 2:
        return None
```

```
    else:
        new_matrix = copy.deepcopy(matrix)
        new_matrix[i][j], new_matrix[i+1][j] = new_matrix[i+1][j],
↪   new_matrix[i][j]
        return new_matrix

def move_left(matrix):
    i, j = find_zero(matrix)
    if j == 0:
        return None
    else:
        new_matrix = copy.deepcopy(matrix)
        new_matrix[i][j], new_matrix[i][j-1] = new_matrix[i][j-1],
↪   new_matrix[i][j]
        return new_matrix

def move_right(matrix):
    i, j = find_zero(matrix)
    if j == 2:
        return None
    else:
        new_matrix = copy.deepcopy(matrix)
        new_matrix[i][j], new_matrix[i][j+1] = new_matrix[i][j+1],
↪   new_matrix[i][j]
        return new_matrix

def is_goal(matrix):
    return matrix == final

def print_matrix(matrix):
    for i in matrix:
        for j in i:
            if j == 0:
                print(' ', end=' ')
            else:
                print(j, end=' ')
        print("")
    print()

def a_star(matrix):
    moves = [move_up, move_down, move_left, move_right]
    open_set = [(h_score(matrix), 0, matrix, [])]
    visited = set()

    while open_set:
        open_set.sort(reverse=True)
        _, cost, current, path = open_set.pop(0)

        if is_goal(current):
            return path
```

```
        matrix_tuple = tuple(map(tuple, current))
        if matrix_tuple in visited:
            continue
        visited.add(matrix_tuple)

        for move in moves:
            new_matrix = move(current)
            if new_matrix and tuple(map(tuple, new_matrix)) not in visited:
                open_set.append((cost + 1 + h_score(new_matrix), cost + 1,
↪   new_matrix, path + [new_matrix]))

    return None

result = a_star(initial)

if result:
    print("Solution found:")
    for step in result:
        print_matrix(step)
else:
    print("No solution found.")
```

Question 4

Apply AO* searching algorithm for the following search tree




```
class Node:
    def __init__(self, name, heuristic):
        self.name = name
        self.heuristic = heuristic
        self.children = []
        self.cost = heuristic
        self.best_path = None

    def add_children(self, children):
        self.children = children

def ao_star(node, edge_cost=1):
    if not node.children:
        return node.heuristic

    min_cost = float('inf')
```

```
    best_subpath = None

    for group in node.children:
        total_cost = sum(ao_star(child, edge_cost) + edge_cost for child
 ↪  in group)

        if total_cost < min_cost:
            min_cost = total_cost
            best_subpath = group

    node.cost = min_cost
    node.best_path = best_subpath
    return node.cost

def print_solution(node):
    if not node.best_path:
        print(node.name, end=" ")
        return

    print(node.name, "→", end=" ")
    for child in node.best_path:
        print_solution(child)

A = Node('A', 0)
B = Node('B', 6)
C = Node('C', 12)
D = Node('D', 10)
G = Node('G', 5)
H = Node('H', 7)
E = Node('E', 4)
F = Node('F', 4)

B.add_children([[G, H]])
D.add_children([[E, F]])
A.add_children([[B], [C], [D]])

ao_star(A)

print("\nOptimal Path:")
print_solution(A)
```