Guru Gobind Singh Indraprastha University
University School of Automation & Robotics

Artificial Intelligence Lab (ARI251)

Practical File

Name	Sujal Singh
Enrollment Number	04119051723
Batch	IIOT–B1–23

Index

No.	Question	Remarks
1.	Water Jug Problem.	
2.	8 – Tile Problem.	
3.	Graph coloring problem.	
4.		
5.		
6.		
7.		
8.		
9.		
10.		
11.		
12.		
13.		
14.		
15.		

1. Given two jugs with capacities 4 liters and 3 liters respectively, measure exactly 1 liter in the second jug, by only performing the following operations: fill either jug to full capacity, empty either jug or transfer water from one jug to another until either first jug is empty or the second jug is full (or vice versa). Find the sequence of operations that reaches the target state with the minimum amount of moves.

Program:

```python
from collections import deque

INITIAL, TARGET,J1_CAPACITY, J2_CAPACITY = (0, 0), (0, 1), 4, 3

def possible_moves(current_state):
    moves = {current_state}
    a, b = current_state
    # Empty
    moves.add((0, b))
    moves.add((a, 0))
    # Fill
    moves.add((J1_CAPACITY, b))
    moves.add((a, J2_CAPACITY))
    # Transfer
    moves.add((a - (transfer := min(a, J2_CAPACITY - b)), b + transfer))
    moves.add((a + (transfer := min(b, J1_CAPACITY - a)), b - transfer))
    moves.remove(current_state)
    return moves

def search(initial_state, target_state):
    queue = deque([(initial_state, [])])
    visited = set()

    while queue:
        current, path = queue.popleft()
        if current == target_state:
            return path + [current]
        visited.add(current)

        for child in possible_moves(current):
            if child not in visited:
                queue.append((child, path + [current]))
    return []

result = search(INITIAL, TARGET)
print(f"Moves: {len(result) - 1 if result else 'Unreachable Target'}\nPath: ", end="")
print(*result, sep=" -> ")
```

Output:

```
Moves: 4
Path: (0, 0) -> (4, 0) -> (1, 3) -> (1, 0) -> (0, 1)
```

2. 8 Tile Problem.

Program:

```python
from copy import deepcopy
from heapq import heappush, heappop

SIZE = 3
INITIAL = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]]
TARGET = [
    [2, 8, 1],
    [0, 4, 3],
    [7, 6, 5]]
BLANK_POS = [1, 1]
ALLOWED_MOVES = ((1, 0), (-1, 0), (0, 1), (0, -1))


def heuristic(state):
    # Increases cost for every misplaced tile.
    h = 0
    for x in range(SIZE):
        for y in range(SIZE):
            if state[x][y] != TARGET[x][y]:
                h += 1
    return h


def possible_moves(state, blank_pos):
    x, y = blank_pos
    moves = []
    for i, j in ALLOWED_MOVES:
        p, q = x + i, y + j
        if not ((0 <= p <= 2) and (0 <= q <= 2)):
            continue
        next_state = deepcopy(state)
        next_state[x][y], next_state[p][q] = next_state[p][q],
        ↪ next_state[x][y]
        moves.append((next_state, (p, q)))
    return moves


def search():
    pq = []
    visited = set()
    heappush(pq, (heuristic(INITIAL), 0, INITIAL, BLANK_POS))

    while pq:
        h, g, current, current_blank_pos = heappop(pq)
```

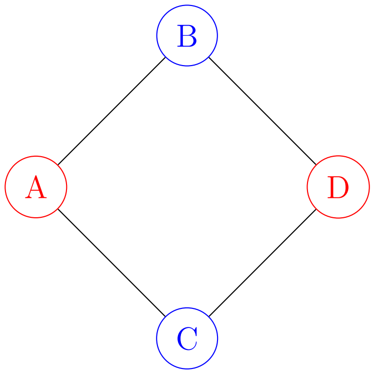
```
        print(current)
        if current == TARGET:
            return
        visited.add(tuple(map(tuple, current)))

        for next_state, next_blank_pos in possible_moves(current, current_blank_pos):
            if tuple(map(tuple, next_state)) in visited:
                continue
            new_g = g + 1
            new_h = new_g + heuristic(next_state)
            heappush(pq, (new_h, new_g, next_state, next_blank_pos))


search()
```

Output:

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

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

3. Color nodes such that no adjacent nodes have the same color with the minimum number of colors.



Program:

```python
graph = {
    "A": ["B", "C"],
    "B": ["A", "D"],
    "C": ["A", "D"],
    "D": ["B", "C"],
}

color = {
    "A": 0,
    "B": 0,
    "C": 0,
    "D": 0
}

for parent, children in graph.items():
    for child in children:
        if color[child] == color[parent]:
            color[child] += 1

print(color)
print(len(set(color.values())))
```

Output:

```
{'A': 0, 'B': 1, 'C': 1, 'D': 0}
2
```