Muhammad usman

01-134202-115

In [2]:

```python
def dfs_iterative(node, graph):
    visited = set()
    if node not in graph:
        return
    stack = []
    cost = 0
    stack.append(node)
    while stack:
        current = stack.pop()
        if current not in visited:
            visited.add(current)
            for neighbor, weight in graph[current].items():
                cost += weight
                stack.append(neighbor)
    return visited, cost

def iddfs(root, goal):
    for depth in range(0, float('inf')):
        found = dls(root, depth, goal)
        if found is not None:
            return found

def dls(node, depth, goal):
    if depth == 0 and node == goal:
        return node
    if depth > 0:
        for neighbor in node:
            found = dls(neighbor, depth - 1, goal)
            if found is not None:
                return found
    return None

graph = {
 'start': {'Room A': 0, 'Room B': 2, 'Room C': 6},
 'Room A': {'start': 0},
 'Room B': {'start': 0},
 'Room C': {'start': 0}
}

path, cost = dfs_iterative('Room A', graph)
print(path)
print("Total cost for cleaning all rooms :", cost)
```

```
{'Room C', 'Room A', 'Room B', 'start'}
Total cost for cleaning all rooms : 8
```

In [3]:

```python
import heapq

def get_state_hash(state):
    return ''.join(sorted([''.join(sorted(str(s))) for s in state]))
def goal_test(state):
    return len(state[0]) == 0 and len(state[1]) == 4
def heuristic(state):
    left_bank = state[0]
    if len(left_bank) < 2:
        return 0
    else:
        return max(left_bank[:-1])

def get_actions(state):
    left_bank = state[0]
    if len(left_bank) < 2:
        return []
    actions = []
    for i in range(len(left_bank)):
        for j in range(i+1, len(left_bank)):
            tourists = [left_bank[i], left_bank[j]]
            actions.append(('AB', tourists))
    return actions

def successor_fn(state, action):
    left_bank, right_bank = state
    tourists = action[1]
    new_left_bank = [t for t in left_bank if t not in tourists]
    new_right_bank = sorted(right_bank + tourists)
    new_state = (new_left_bank, new_right_bank)
    step_cost = max(tourists)
    return new_state, action[0], step_cost

def a_star_search(start_state, heuristic_fn, successor_fn, goal_fn):
    visited_states = set()
    frontier = [(0, start_state, [])]
    while frontier:
        _, state, actions = heapq.heappop(frontier)
        if goal_fn(state):
            return actions
        if get_state_hash(state) in visited_states:
            continue
        visited_states.add(get_state_hash(state))
        for action in get_actions(state):
            new_state, action_type, step_cost = successor_fn(state, action)
            new_actions = actions + [(action_type, tourists) for tourists in action[1]]
            f = len(new_actions) + heuristic_fn(new_state)
            heapq.heappush(frontier, (f, new_state, new_actions))
    return None

times = []
for i in range(4):
    time = int(input(f"Enter time taken by tourist {i+1}: "))
    times.append(time)
start_state = (sorted(times), [])

actions = a_star_search(start_state, heuristic, successor_fn, goal_test)

if actions is None:
 print("No solution found.")
else:
 total_time = sum([step_cost for _, step_cost in actions])
 print(f"Actions: {actions}")
 print(f"Total time taken: {total_time}")
```

```
Enter time taken by tourist 1: 4
Enter time taken by tourist 2: 5
Enter time taken by tourist 3: 2
Enter time taken by tourist 4: 7
Actions: [('AB', 5), ('AB', 7), ('AB', 2), ('AB', 4)]
Total time taken: 18
```

In [8]:

```python
class Graph:
    def __init__(self, nodes=None, edges=None):
        self.nodes, self.adj = [], {}
        if nodes != None:
            self.add_nodes_from(nodes)
        if edges != None:
            self.add_edges_from(edges)

    def length(self):
        return len(self.nodes)

    def traverse(self):
        return 'V: %s\nE: %s' % (self.nodes, self.adj)

    def add_node(self, n):
        if n not in self.nodes:
            self.nodes.append(n)
            self.adj[n] = []

    def add_edge(self, u, v):
        self.adj[u] = self.adj.get(u, []) + [v]
        self.adj[v] = self.adj.get(v, []) + [u]

    def number_of_nodes(self):
        return len(self.nodes)

    def number_of_edges(self):
        return sum(len(l) for _, l in self.adj.items())

class DGraph(Graph):
    def add_edge(self, u, v):
        self.adj[u] = self.adj.get(u, []) + [v]

    def A_star(self, start, goal, f=0):
        adjacent = self.adj[start]
        if start == goal:
            return f
        minimum = []
        for i in range(0, len(adjacent)):
            minimum.append(hur(adjacent[i]))
        smallest = minimum.index(min(minimum))
        f = f + minimum[smallest] + 1
        p.append(adjacent[smallest])
        return self.A_star(adjacent[smallest], 'ABC', f)

class WGraph(Graph):
    def __init__(self, nodes=None, edges=None):
        self.nodes, self.adj, self.weight = [], {}, {}
        if nodes != None:
            self.add_nodes_from(nodes)
        if edges != None:
            self.add_edges_from(edges)

    def add_edge(self, u, v, w):
        self.adj[u] = self.adj.get(u, []) + [v]
        self.adj[v] = self.adj.get(v, []) + [u]
        self.weight[(u,v)] = w
        self.weight[(v,u)] = w

    def get_weight(self, u, v):
        return self.weight[(u,v)]

class DWGraph(WGraph):
    def add_edge(self, u, v, w):
        self.adj[u] = self.adj.get(u, []) + [v]
        self.weight[(u,v)] = w

    def find_path(self, start, end, path=[], cost=0):
        pass

p=[]

def hur(ar):
    goalstate='ABC'
    hc=0
    for x in range(0, len(ar)):
        if ar[x] is not goalstate[x]:
            hc = hc+1
    return hc

G=DGraph()
G.add_node('ACB')
G.add_node('aCB')
G.add_node('BAC')
G.add_node('CBA')
G.add_node('BCA')
G.add_node('CAb')
G.add_node('abC')
G.add_node('aBC')
G.add_node('ABC')
G.add_node('CAB')
G.add_node('ABc')
G.add_node('bAC')
G.add_node('BAC')
G.add_edge('ACB', 'aCB')
G.add_edge('BAc', 'CBA')
G.add_edge('aCB', 'CAb')
G.add_edge('aCB', 'abC')
G.add_edge('BAc', 'abC')
G.add_edge('BAc', 'aBC')
G.add_edge('BCA', 'CAb')
G.add_edge('CAb', 'abC')
G.add_edge('abC', 'aBC')
G.add_edge('aBC', 'ABC')
G.add_edge('abC', 'ABc')
G.add_edge('abC', 'bAC')
G.add_edge('CAB', 'ABc')
G.add_edge('ABc', 'bAC')
G.add_edge('bAC', 'BAC')
G.A_star('CAb','ABC',0)
print(p)
```

```
['abC', 'aBC', 'ABC']
```

In [ ]: