```
1 def water_jug_dfs(capacity_x, capacity_y, target):
    stack = [(0, 0, [])]  # (x, y, path)
    visited_states = set()
    while stack:
        x, y, path = stack.pop()
        if (x, y) in visited_states:
            continue
        visited_states.add((x, y))
        if x == target or y == target:
            return path + [(x, y)]
        operations = [
            ("fill_x", capacity_x, y),
            ("fill_y", x, capacity_y),
            ("empty_x", 0, y),
            ("empty_y", x, 0),
            ("pour_x_to_y",
             max(0, x - (capacity_y - y)),
             min(capacity_y, y + x)),
            ("pour_y_to_x",
             min(capacity_x, x + y),
             max(0, y - (capacity_x - x))),
        ]
        for operation, new_x, new_y in operations:
            if 0 <= new_x <= capacity_x and 0 <= new_y <= capacity_y:
                stack.append((new_x, new_y, path + [(x, y, operation)]))
    return None
capacity_x = 4
capacity_y = 3
target = 2
solution_path = water_jug_dfs(capacity_x, capacity_y, target)
if solution_path:
    print("Solution found:")
    for state in solution_path:
        print(f"({state[0]}, {state[1]})")
```

```
else:
    print("No solution found.")
```

output-

```
Solution found:
(0, 0)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
```

2 from tracemalloc import start

```
tree = {
    1: [2, 9, 10],
    2: [3, 4],
    3: [],
    4: [5, 6, 7],
    5: [8],
    6: [],
    7: [],
    8: [],
    9: [],
    10: []
}
def breadth_first_search(tree, start):
    q = [start]
    visited = []
    while q:
        print("before", q)
        node = q.pop(0)
        visited.append(node)

        for child in tree[node]:
```

```
            if child not in visited and child not in q:
                q.append(child)

        print("after", q)
    return visited
result = breadth_first_search(tree, 1)
print(result)
```

output-

```
before [1]
after [2, 9, 10]
before [2, 9, 10]
after [9, 10, 3, 4]
before [9, 10, 3, 4]
after [10, 3, 4]
before [10, 3, 4]
after [3, 4]
before [3, 4]
after [4]
before [4]
after [5, 6, 7]
before [5, 6, 7]
after [6, 7, 8]
before [6, 7, 8]
after [7, 8]
before [7, 8]
after [8]
before [8]
after []
[1, 2, 9, 10, 3, 4, 5, 6, 7, 8]
```

3 import heapq

```
road_graph = {
```

```
    'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
    'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},
    'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
    'Drobeta': {'Mehadia': 75, 'Craiova': 120},
    'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
    'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
    'Bucharest': {'Fagaras': 211, 'Pitesti': 101}
}
heuristic_cost = {
    "Arad": {"Bucharest": 366},
    "Bucharest": {"Bucharest": 0},
    "Craiova": {"Bucharest": 160},
    "Drobeta": {"Bucharest": 242},  # fixed spelling
    "Fagaras": {"Bucharest": 176},
    "Lugoj": {"Bucharest": 244},
    "Mehadia": {"Bucharest": 241},
    "Oradea": {"Bucharest": 380},
    "Pitesti": {"Bucharest": 100},
    "Rimnicu Vilcea": {"Bucharest": 193},
    "Sibiu": {"Bucharest": 253},
    "Timisoara": {"Bucharest": 329},
    "Zerind": {"Bucharest": 374}
}
def heuristic_cost_estimate(node, goal):
    return heuristic_cost[node][goal]
```

```
def a_star(graph, start, goal):
    open_set = [(0, start)]
    came_from = {}
    g_score = {city: float('inf') for city in graph}
    g_score[start] = 0
    while open_set:
        current_cost, current_city = heapq.heappop(open_set)
        if current_city == goal:
            return reconstruct_path(came_from, goal)
        for neighbor, cost in graph[current_city].items():
            tentative_g_score = g_score[current_city] + cost
            if tentative_g_score < g_score[neighbor]:
                g_score[neighbor] = tentative_g_score
                f_score = tentative_g_score + heuristic_cost_estimate(neighbor, goal)
                heapq.heappush(open_set, (f_score, neighbor))
                came_from[neighbor] = current_city
    return None  # No path found
def reconstruct_path(came_from, current_city):
    path = [current_city]
    while current_city in came_from:
        current_city = came_from[current_city]
        path.insert(0, current_city)
    return path
def calculate_distance(graph, path):
    total_distance = 0
    for i in range(len(path) - 1):
        current_city = path[i]
        next_city = path[i + 1]
        total_distance += graph[current_city][next_city]
    return total_distance
start_city = 'Arad'
```

```
goal_city = 'Bucharest'

path = a_star(road_graph, start_city, goal_city)

if path is None:

    print("No path found")

else:

    distance = calculate_distance(road_graph, path)

    print("Shortest Path from {} to {}: {}".format(start_city, goal_city, path))

    print("Total distance: {}".format(distance))
```

output-

Shortest Path from Arad to Bucharest: ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest']

Total distance: 418

4

```
def ao_star(node, graph, heurisitc, solved):

    if node in solved:

        return heurisitc[node], [node]

    if not graph[node]:

        solved.add(node)

        return heurisitc[node], [node]

    min_cost = float('inf')

    best_path = []

    children = graph[node]

    i = 0

    while i < len(children):

        child, relation = children[i]

        if relation == 'OR':

            cost1, path1 = ao_star(child, graph, heurisitc, solved)

            total_cost = cost1

            total_path = [node] + path1

            if total_cost < min_cost:

                min_cost = total_cost

                best_path = total_path

            i += 1
```

```
        elif relation == 'AND':
            group = [child]
            j = i + 1
            while j < len(children) and children[j][1] == 'AND':
                group.append(children[j][0])
                j += 1
            total_cost = 0
            total_path = [node]
            for g_child in group:
                c, p = ao_star(g_child, graph, heurisitc, solved)
                total_cost += c
                total_path += p
            if total_cost <= min_cost:
                min_cost = total_cost
                best_path = total_path
            i = j
        else:
            i += 1
    heurisitc[node] = min_cost
    solved.add(node)
    return min_cost, best_path
graphh = {
    'A': [('B', 'OR'), ('C', 'AND'), ('D', 'AND')],
    'B': [('E', 'OR'), ('F', 'OR')],
    'C': [('G', 'OR'), ('H', 'AND'), ('I', 'AND')],
    'D': [('J', 'OR')],
    'E': [], 'F': [], 'G': [], 'H': [], 'I': [], 'J': []
}
heurisitc = {
    'A': 0, 'B': 6, 'C': 4, 'D': 5,
    'E': 13, 'F': 10, 'G': 12, 'H': 7, 'I': 8, 'J': 0
```

```
}
solved_nodes = set()
cost, optimal_path = ao_star('A', graphh, heurisitc, solved_nodes)
print("Optimal Path:", " -> ".join(optimal_path))
print("Cost:", cost)
```

output-

Optimal Path: A -> B -> F

Cost: 10