Artificial Intelligence Assignment 3

By Himanshu Sardana

Question 1

Solve the following blocks world problem using Depth First Search.


```
initial = {
    'A': 'Table',
    'B': 'Table',
    'C': 'B',
}

final = {
    'A': 'Table',
    'B': 'A',
    'C': 'B'
}

def generate_cases(state):
    movable_blocks = set(state.keys())
    blocked_blocks = set(state.values()) - {'Table'}
    top_blocks = movable_blocks - blocked_blocks

    cases = []
    for block in top_blocks:
        for target in movable_blocks | {'Table'}:
            if block != target and state[block] != target:
                cases.append((block, target))
    return cases

def move(state, case):
    block, target = case
    new_state = state.copy()

    for key, value in new_state.items():
        if value == block:
            new_state[key] = 'Table'

    new_state[block] = target
    return new_state
```

```
def is_final(state):
    return state == final

def dfs(state, path, visited):
    if is_final(state):
        return path

    state_tuple = tuple(sorted(state.items()))
    if state_tuple in visited:
        return None
    visited.add(state_tuple)

    for case in generate_cases(state):
        new_state = move(state, case)
        new_path = path + [case]
        result = dfs(new_state, new_path, visited)
        if result:
            return result

    return None

result = dfs(initial, [], set())

for block, target in result:
    print(f'Move block {block} to {target}')
```

Question 2

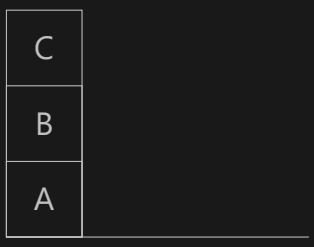
Solve the following blocks world problem using Breadth First Search.


```
initial = {
    'A': 'Table',
    'B': 'Table',
    'C': 'B',
}

final = {
    'A': 'Table',
    'B': 'A',
    'C': 'B'
```

```
}

def generate_cases(state):
    movable_blocks = set(state.keys())
    blocked_blocks = set(state.values()) - {'Table'}
    top_blocks = movable_blocks - blocked_blocks

    cases = []
    for block in top_blocks:
        for target in movable_blocks | {'Table'}:
            if block != target and state[block] != target:
                cases.append((block, target))
    return cases

def move(state, case):
    block, target = case
    new_state = state.copy()

    for key, value in new_state.items():
        if value == block:
            new_state[key] = 'Table'

    new_state[block] = target
    return new_state

def is_final(state):
    return state == final

def bfs():
    queue = [(initial, [])]

    while queue:
        state, path = queue.pop(0)

        if is_final(state):
            return path

        for case in generate_cases(state):
            new_state = move(state, case)
            new_path = path + [case]
            queue.append((new_state, new_path))

    return None

result = bfs()
for block, target in result:
    print(f'Move block {block} to {target}')
```

Question 3

Write a python program to solve the follow blocks world problem using Depth Limited Search (D=1). Check if it is complete or incomplete for depth = 1.

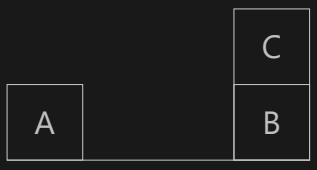


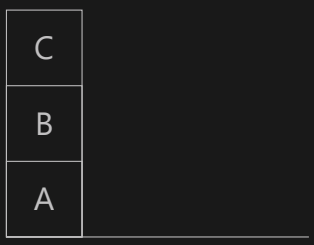

```
initial = {
    'A': 'Table',
    'B': 'Table',
    'C': 'A',
}

final = {
    'A': 'Table',
    'B': 'A',
    'C': 'B'
}

def generate_cases(state):
    movable_blocks = set(state.keys())
    blocked_blocks = set(state.values()) - {'Table'}
    top_blocks = movable_blocks - blocked_blocks

    cases = []
    for block in top_blocks:
        for target in movable_blocks | {'Table'}:
            if block != target and state[block] != target:
                cases.append((block, target))
    return cases

def move(state, case):
    block, target = case
    new_state = state.copy()

    for key, value in new_state.items():
        if value == block:
            new_state[key] = 'Table'

    new_state[block] = target
    return new_state

def is_final(state):
```

```
    return state == final

def depth_limited_search(state, path, depth):
    if is_final(state):
        return path

    if depth == 0:
        return None

    for case in generate_cases(state):
        new_state = move(state, case)
        new_path = path + [case]
        result = depth_limited_search(new_state, new_path, depth - 1)
        if result:
            return result

    return None

D = 1
result = depth_limited_search(initial, [], D)

print(f"Solution at depth {D}:", result)
if result:
    print("Complete")
else:
    print("Incomplete")
```

Question 4

Find the depth at which the goal is achieved using Iterative Deepening for the following problem.





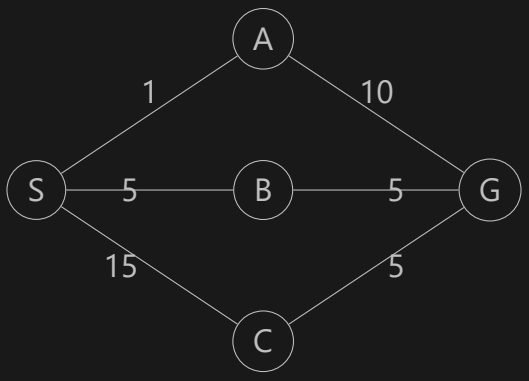

```
initial = {
    'A': 'Table',
    'B': 'Table',
    'C': 'B',
}

final = {
    'A': 'Table',
    'B': 'A',
    'C': 'B'
}
```

```
def generate_cases(state):
    movable_blocks = set(state.keys())
    blocked_blocks = set(state.values()) - {'Table'}
    top_blocks = movable_blocks - blocked_blocks

    cases = []
    for block in top_blocks:
        for target in movable_blocks | {'Table'}:
            if block != target and state[block] != target:
                cases.append((block, target))
    return cases

def move(state, case):
    block, target = case
    new_state = state.copy()

    for key, value in new_state.items():
        if value == block:
            new_state[key] = 'Table'

    new_state[block] = target
    return new_state

def is_final(state):
    return state == final

def depth_limited_search(state, path, depth):
    if is_final(state):
        return path

    if depth == 0:
        return None

    for case in generate_cases(state):
        new_state = move(state, case)
        new_path = path + [case]
        result = depth_limited_search(new_state, new_path, depth - 1)
        if result:
            return result

    return None

def iterative_deepening_search(initial_state):
    depth = 0
    while True:
        result = depth_limited_search(initial_state, [], depth)
        if result:
            return depth  # Goal found at this depth
        depth += 1
```

```
depth_of_solution = iterative_deepening_search(initial)

print("Goal achieved at depth:", depth_of_solution)
```

Question 5

Solve this given problem using Uniform Cost Search.




```
def uniform_cost_search(graph, start, goal):
    queue = [(0, start, [])]  # (cost, node, path)
    visited = set()

    while queue:
        queue.sort()
        cost, node, path = queue.pop(0)

        if node in visited:
            continue

        path = path + [node]
        visited.add(node)

        if node == goal:
            return path, cost

        for neighbor, weight in graph.get(node, []):
            if neighbor not in visited:
                queue.append((cost + weight, neighbor, path))

    return None, float('inf')

graph = {
    'A': [('S', 1), ('G', 10)],
    'S': [('B', 5), ('C', 15)],
    'B': [('G', 5)],
    'C': [('G', 5)],
    'G': []
}
```

```
path, cost = uniform_cost_search(graph, 'A', 'C')
print("Shortest Path:", path)
print("Total Cost:", cost)
```