AI: A3

Erick Gonzalez Parada 178145

Searching for treasure

BFS (Breadth First Search)

Pseudocode

```
BFS_Find_Treasure(island, start, treasure):
    rows ← number of rows in island
    cols ← number of columns in island

    directions ← [(up), (right), (down), (left)]  # Possible movements

    queue ← empty list or queue data structure
    enqueue (start_row, start_col, [start]) into queue
    visited ← empty set
    add start to visited

    while queue is not empty:
        (row, col, path) ← dequeue first element from queue
        print "Exploring position:", row, col

        if (row, col) = treasure then:
            return path  # Found the treasure, return the path

        for each (dx, dy) in directions do:
            newRow ← row + dx
            newCol ← col + dy

            if newRow is within bounds AND newCol is within bounds AND
```

```
            island[newRow][newCol] is not an obstacle AND
            (newRow, newCol) is not in visited then:

             add (newRow, newCol) to visited
             newPath ← copy of path with (newRow, newCol) appended
             enqueue (newRow, newCol, newPath) into queue

    return null  # No path found
```

Code

```
from collections import deque

def findTreasureBfs(island, start, treasure):
    rows = len(island)
    cols = len(island[0])

    # Define possible movements (up, right, down, left)
    directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    # Queue will store (row, col, path)
    queue = deque([(start[0], start[1], [start])])

    # Keep track of visited cells
    visited = set([start])

    while queue:
        row, col, path = queue.popleft()
        print(f"Exploring position: ({row}, {col})")

        # Check if we found the treasure
        if (row, col) == treasure:
            return path

        # Try all possible directions
```

```
        for dx, dy in directions:
            newRow, newCol = row + dx, col + dy

            # Check if the new position is valid
            # the first two lines is index bounding
            # then if it it is not an obstacle
            # and finally if not visited
            if (0 <= newRow < rows and
                0 <= newCol < cols and
                island[newRow][newCol] != 1 and
                (newRow, newCol) not in visited):

                visited.add((newRow, newCol))
                newPath = path + [(newRow, newCol)]
                queue.append((newRow, newCol, newPath))

    return None  # No path found

def printPathOnMap(island, path):
    if not path:
        print("No path found!")
        return

    # Create a copy of the island for visualization
    visualMap = []
    for row in island:
        visualMap.append(list(str(cell) for cell in row))

    # Mark the path with '*'
    for i, (row, col) in enumerate(path):
        if i == 0:
            visualMap[row][col] = 'S'  # Start
        elif i == len(path) - 1:
            visualMap[row][col] = 'T'  # Treasure
        else:
            visualMap[row][col] = '*'
```

```
    # Print the map with the path
    for row in visualMap:
        print(' '.join(row))

def BFSAct3TreasureMap():
    island = [
        [0, 0, 1, 0],
        [1, 0, 1, 0],
        [0, 0, 0, 0],
        [0, 1, 1, 1],
        [0, 0, 0, 'T'],
    ]

    start = (0, 0)
    treasure = (4, 3)

    # Find path to treasure
    path = findTreasureBfs(island, start, treasure)

    # Print the result
    if path:
        print("\nPath found! Here's the route:")
        printPathOnMap(island, path)
        print(f"\nSteps to reach treasure: {len(path) - 1}")
    else:
        print("No path to treasure exists!")
```

DFS (Depth First Search)

Pseudocode

```
DFS_Find_Treasure(island, start, treasure):
    rows ← number of rows in island
    cols ← number of columns in island
```

```
directions ← [(up), (right), (down), (left)]  # Possible movements

stack ← empty list or stack data structure
push (start_row, start_col, [start]) onto stack
visited ← empty set
add start to visited

while stack is not empty:
    (row, col, path) ← pop top element from stack
    print "Exploring position:", row, col

    if (row, col) = treasure then:
        return path  # Found the treasure, return the path

    for each (dx, dy) in directions do:
        newRow ← row + dx
        newCol ← col + dy

        if newRow is within bounds AND newCol is within bounds AND
           island[newRow][newCol] is not an obstacle AND
           (newRow, newCol) is not in visited then:

            add (newRow, newCol) to visited
            newPath ← copy of path with (newRow, newCol) appended
            push (newRow, newCol, newPath) onto stack

return null  # No path found
```

Code

```
from collections import deque

def findTreasureDfs(island, start, treasure):
    rows = len(island)
```

```
    cols = len(island[0])

    directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    queue = deque([(start[0], start[1], [start])])

    visited = set([start])

    while queue:
        row, col, path = queue.pop()
        print(f"Exploring position: ({row}, {col})")

        if (row, col) == treasure:
            return path

        for dx, dy in directions:
            newRow, newCol = row + dx, col + dy

            if (0 <= newRow < rows and
                0 <= newCol < cols and
                island[newRow][newCol] != 1 and
                (newRow, newCol) not in visited):

                visited.add((newRow, newCol))
                newPath = path + [(newRow, newCol)]
                queue.append((newRow, newCol, newPath))

    return None  # No path found


def printPathOnMap(island, path):
    if not path:
        print("No path found!")
        return

    # Create a copy of the island for visualization
```

```
    visualMap = []
    for row in island:
        visualMap.append(list(str(cell) for cell in row))

    # Mark the path with '*'
    for i, (row, col) in enumerate(path):
        if i == 0:
            visualMap[row][col] = 'S'  # Start
        elif i == len(path) - 1:
            visualMap[row][col] = 'T'  # Treasure
        else:
            visualMap[row][col] = '*'

    # Print the map with the path
    for row in visualMap:
        print(' '.join(row))

def DFSAct3TreasureMap():
    island = [
        [0, 0, 1, 0],
        [1, 0, 1, 0],
        [0, 0, 0, 0],
        [0, 1, 1, 1],
        [0, 0, 0, 'T'],
    ]

    start = (0, 0)
    treasure = (4, 3)

    # Find path to treasure
    path = findTreasureDfs(island, start, treasure)

    # Print the result
    if path:
        print("\nPath found! Here's the route:")
        printPathOnMap(island, path)
```

```
        print(f"\nSteps to reach treasure: {len(path) - 1}")
    else:
        print("No path to treasure exists!")
```

Cities problem

Greedy Search

Pseudocode

```
FUNCTION greedySearchCities():
    // Define the graph representing cities and connections
    graph = {
        'A': [('B', 1), ('C', 4)],
        'B': [('D', 5), ('E', 2)],
        'C': [('F', 3), ('G', 4)],
        'D': [('H', 3)],
        'E': [('H', 6)],
        'F': [('I', 4)],
        'G': [('J', 2)],
        'H': [('I', 1)],
        'I': [('J', 2)],
        'J': []
    }

    // Define heuristic values for each city
    heuristic = {
        'A': 7, 'B': 6, 'C': 3, 'D': 5, 'E': 4, 'F': 2, 'G': 1, 'H': 3, 'I': 1, 'J': 0
    }

    // Define the Greedy Search function
    FUNCTION greedySearch(graph, start, goal, heuristic):
        // Initialize the frontier with the start node, its path, and initial cost
        frontier = []
```

```
    ADD (start, [start], 0) TO frontier  // (current_node, path, total_cost)

    // Loop until the frontier is empty
    WHILE frontier is not empty:
        // Sort the frontier by the heuristic value of the current node
        SORT frontier BY heuristic value of the first element in each tuple
        current_node, path, total_cost = REMOVE FIRST ELEMENT FROM frontier

        // Check if the goal is reached
        IF current_node == goal:
            RETURN path, total_cost

        // Explore neighbors of the current node
        FOR each neighbor, cost IN graph[current_node]:
            IF neighbor is not in path:
                new_path = COPY path
                ADD neighbor TO new_path
                new_cost = total_cost + cost
                ADD (neighbor, new_path, new_cost) TO frontier

    // If no path is found, return failure
    RETURN None, infinity

// Run Greedy Search from start to goal
start = 'A'
goal = 'J'
path, cost = greedySearch(graph, start, goal, heuristic)

// Print the result
PRINT "Greedy Search Path: " + path
PRINT "Total Cost: " + cost
```

Code

```
from collections import defaultdict, deque

def greedySearchCities():
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'D': 5, 'E': 2},
        'C': {'F': 3, 'G': 4},
        'D': {'H': 3},
        'E': {'H': 6},
        'F': {'I': 4},
        'G': {'J': 2},
        'H': {'I': 1},
        'I': {'J': 2},
        'J': {}
    }

    # Heuristic values
    heuristic = {
        'A': 7, 'B': 6, 'C': 3, 'D': 5, 'E': 4, 'F': 2, 'G': 1, 'H': 3, 'I': 1, 'J': 0
    }

    def greedySearch(graph, start, goal, heuristic):
        # Priority queue to store nodes to be explored, sorted by heuristic value
        frontier = deque()
        frontier.append((start, [start], 0))  # (current_node, path, total_cost)

        while frontier:
            # Sort the frontier by heuristic value of the last node in the path
            frontier = deque(sorted(frontier, key=lambda x: heuristic[x[0]]))
            current_node, path, total_cost = frontier.popleft()

            # Check if the goal is reached
            if current_node == goal:
                return path, total_cost
```

```
        # Explore neighbors
        for neighbor, cost in graph[current_node].items():
            if neighbor not in path:
                new_path = list(path)
                new_path.append(neighbor)
                new_cost = total_cost + cost
                frontier.append((neighbor, new_path, new_cost))

    return None, float('inf')  # If no path is found

# Run Greedy Search
start = 'A'
goal = 'J'
path, cost = greedySearch(graph, start, goal, heuristic)

print(f"Greedy Search Path: {path}")
print(f"Total Cost: {cost}")
```

Uniform Cost Search

Pseudocode

```
FUNCTION uniformCostSearchCities():
    // Define the graph representing cities and their connections
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'D': 5, 'E': 2},
        'C': {'F': 3, 'G': 4},
        'D': {'H': 3},
        'E': {'H': 6},
        'F': {'I': 4},
        'G': {'J': 2},
        'H': {'I': 1},
        'I': {'J': 2},
        'J': {}
```

```
}

// Define the Uniform Cost Search function
FUNCTION uniformCostSearch(graph, start, goal):
    // Initialize the frontier with the start node, its path, and initial cost
    frontier = []
    ADD (start, [start], 0) TO frontier  // (currentNode, path, totalCost)

    // Set to keep track of visited nodes
    visited = SET()

    // Loop until the frontier is empty
    WHILE frontier is not empty:
        // Sort the frontier by total cost
        SORT frontier BY totalCost (third element in each tuple)
        currentNode, path, totalCost = REMOVE FIRST ELEMENT FROM frontier

        // Check if the goal is reached
        IF currentNode == goal:
            RETURN path, totalCost

        // Mark the current node as visited
        ADD currentNode TO visited

        // Explore neighbors of the current node
        FOR each neighbor, cost IN graph[currentNode]:
            IF neighbor is not in visited:
                newPath = COPY path
                ADD neighbor TO newPath
                newCost = totalCost + cost
                ADD (neighbor, newPath, newCost) TO frontier

    // If no path is found, return failure
    RETURN None, infinity

// Run Uniform Cost Search from start to goal
```

```
    startNode = 'A'
    goalNode = 'J'
    path, cost = uniformCostSearch(graph, startNode, goalNode)

    // Print the result
    PRINT "Uniform Cost Search Path: " + path
    PRINT "Total Cost: " + cost
```

Code

```
from collections import deque

def uniformCostSearchCities():
    # Graph representation
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'D': 5, 'E': 2},
        'C': {'F': 3, 'G': 4},
        'D': {'H': 3},
        'E': {'H': 6},
        'F': {'I': 4},
        'G': {'J': 2},
        'H': {'I': 1},
        'I': {'J': 2},
        'J': {}
    }

    def uniformCostSearch(graph, start, goal):
        # Priority queue to store nodes to be explored, sorted by total cost
        frontier = deque()
        frontier.append((start, [start], 0))  # (currentNode, path, totalCost)

        # Set to keep track of visited nodes
        visited = set()
```

```
    while frontier:
        # Sort the frontier by total cost
        frontier = deque(sorted(frontier, key=lambda x: x[2]))
        currentNode, path, totalCost = frontier.popleft()

        # Check if the goal is reached
        if currentNode == goal:
            return path, totalCost

        # Mark the current node as visited
        visited.add(currentNode)

        # Explore neighbors
        for neighbor, cost in graph[currentNode].items():
            if neighbor not in visited:
                newPath = list(path)
                newPath.append(neighbor)
                newCost = totalCost + cost
                frontier.append((neighbor, newPath, newCost))

    return None, float('inf')  # If no path is found

# Run Uniform Cost Search
startNode = 'A'
goalNode = 'J'
path, cost = uniformCostSearch(graph, startNode, goalNode)

print(f"Uniform Cost Search Path: {path}")
print(f"Total Cost: {cost}")
```

A* Tree Search

Pseudocode

```
FUNCTION AStarTreeSearchCities():
    // Define the graph representing cities and their connections
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'D': 5, 'E': 2},
        'C': {'F': 3, 'G': 4},
        'D': {'H': 3},
        'E': {'H': 6},
        'F': {'I': 4},
        'G': {'J': 2},
        'H': {'I': 1},
        'I': {'J': 2},
        'J': {}
    }

    // Define heuristic values for each city
    heuristic = {
        'A': 7, 'B': 6, 'C': 3, 'D': 5, 'E': 4, 'F': 2, 'G': 1, 'H': 3, 'I': 1, 'J': 0
    }

    // Define the A* Tree Search function
    FUNCTION aStarTreeSearch(graph, start, goal, heuristic):
        // Initialize the frontier with the start node, its path, and initial cost (g(n))
        frontier = []
        ADD (start, [start], 0) TO frontier  // (currentNode, path, gCost)

        // Loop until the frontier is empty
        WHILE frontier is not empty:
            // Sort the frontier by f(n) = g(n) + h(n)
            SORT frontier BY (gCost + heuristic[currentNode])
            currentNode, path, gCost = REMOVE FIRST ELEMENT FROM frontier

            // Check if the goal is reached
            IF currentNode == goal:
                RETURN path, gCost
```

```
        // Explore neighbors of the current node
        FOR each neighbor, cost IN graph[currentNode]:
            newPath = COPY path
            ADD neighbor TO newPath
            newGCost = gCost + cost
            ADD (neighbor, newPath, newGCost) TO frontier

    // If no path is found, return failure
    RETURN None, infinity

// Run A* Tree Search from start to goal
startNode = 'A'
goalNode = 'J'
path, cost = aStarTreeSearch(graph, startNode, goalNode, heuristic)

// Print the result
PRINT "A* Tree Search Path: " + path
PRINT "Total Cost: " + cost
```

Code

```
from collections import deque

def AStarTreeSearchCities():

    # Graph representation
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'D': 5, 'E': 2},
        'C': {'F': 3, 'G': 4},
        'D': {'H': 3},
        'E': {'H': 6},
        'F': {'I': 4},
        'G': {'J': 2},
```

```
    'H': {'I': 1},
    'I': {'J': 2},
    'J': {}
}

# Heuristic values
heuristic = {
    'A': 7, 'B': 6, 'C': 3, 'D': 5, 'E': 4, 'F': 2, 'G': 1, 'H': 3, 'I': 1, 'J': 0
}

def aStarTreeSearch(graph, start, goal, heuristic):
    # Priority queue to store nodes to be explored, sorted by f(n) = g(n) + h(n)
    frontier = deque()
    frontier.append((start, [start], 0))  # (currentNode, path, g(n))

    while frontier:
        # Sort the frontier by f(n) = g(n) + h(n)
        frontier = deque(sorted(frontier, key=lambda x: x[2] + heuristic[x[0]]))
        currentNode, path, gCost = frontier.popleft()

        # Check if the goal is reached
        if currentNode == goal:
            return path, gCost

        # Explore neighbors
        for neighbor, cost in graph[currentNode].items():
            newPath = list(path)
            newPath.append(neighbor)
            newGCost = gCost + cost
            frontier.append((neighbor, newPath, newGCost))

    return None, float('inf')  # If no path is found

# Run A* Tree Search
startNode = 'A'
goalNode = 'J'
```

```
    path, cost = aStarTreeSearch(graph, startNode, goalNode, heuristic)

    print(f"A* Tree Search Path: {path}")
    print(f"Total Cost: {cost}")
```

A* Graph Search

Pseudocode

```
FUNCTION AStarGraphSearchCities():
    // Define the graph representing cities and their connections
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'D': 5, 'E': 2},
        'C': {'F': 3, 'G': 4},
        'D': {'H': 3},
        'E': {'H': 6},
        'F': {'I': 4},
        'G': {'J': 2},
        'H': {'I': 1},
        'I': {'J': 2},
        'J': {}
    }

    // Define heuristic values for each city
    heuristic = {
        'A': 7, 'B': 6, 'C': 3, 'D': 5, 'E': 4, 'F': 2, 'G': 1, 'H': 3, 'I': 1, 'J': 0
    }

    // Define the A* Graph Search function
    FUNCTION aStarGraphSearch(graph, start, goal, heuristic):
        // Initialize the frontier with the start node, its path, and initial cost (g(n))
        frontier = []
        ADD (start, [start], 0) TO frontier  // (currentNode, path, gCost)
```

```
    // Set to keep track of visited nodes
    visited = SET()

    // Loop until the frontier is empty
    WHILE frontier is not empty:
        // Sort the frontier by f(n) = g(n) + h(n)
        SORT frontier BY (gCost + heuristic[currentNode])
        currentNode, path, gCost = REMOVE FIRST ELEMENT FROM frontier

        // Check if the goal is reached
        IF currentNode == goal:
            RETURN path, gCost

        // Mark the current node as visited
        ADD currentNode TO visited

        // Explore neighbors of the current node
        FOR each neighbor, cost IN graph[currentNode]:
            IF neighbor is not in visited:
                newPath = COPY path
                ADD neighbor TO newPath
                newGCost = gCost + cost
                ADD (neighbor, newPath, newGCost) TO frontier

    // If no path is found, return failure
    RETURN None, infinity

// Run A* Graph Search from start to goal
startNode = 'A'
goalNode = 'J'
path, cost = aStarGraphSearch(graph, startNode, goalNode, heuristic)

// Print the result
PRINT "A* Graph Search Path: " + path
PRINT "Total Cost: " + cost
```

Code

```
from collections import deque

def AStarGraphSearchCities():
    # Graph representation
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'D': 5, 'E': 2},
        'C': {'F': 3, 'G': 4},
        'D': {'H': 3},
        'E': {'H': 6},
        'F': {'I': 4},
        'G': {'J': 2},
        'H': {'I': 1},
        'I': {'J': 2},
        'J': {}
    }

    # Heuristic values
    heuristic = {
        'A': 7, 'B': 6, 'C': 3, 'D': 5, 'E': 4, 'F': 2, 'G': 1, 'H': 3, 'I': 1, 'J': 0
    }

    def aStarGraphSearch(graph, start, goal, heuristic):
        # Priority queue to store nodes to be explored, sorted by f(n) = g(n) + h(n)
        frontier = deque()
        frontier.append((start, [start], 0))  # (currentNode, path, g(n))

        # Set to keep track of visited nodes
        visited = set()

        while frontier:
            # Sort the frontier by f(n) = g(n) + h(n)
            frontier = deque(sorted(frontier, key=lambda x: x[2] + heuristic[x[0]]))
            currentNode, path, gCost = frontier.popleft()
```

```
        # Check if the goal is reached
        if currentNode == goal:
            return path, gCost

        # Mark the current node as visited
        visited.add(currentNode)

        # Explore neighbors
        for neighbor, cost in graph[currentNode].items():
            if neighbor not in visited:
                newPath = list(path)
                newPath.append(neighbor)
                newGCost = gCost + cost
                frontier.append((neighbor, newPath, newGCost))

    return None, float('inf')  # If no path is found

# Run A* Graph Search
startNode = 'A'
goalNode = 'J'
path, cost = aStarGraphSearch(graph, startNode, goalNode, heuristic)

print(f"A* Graph Search Path: {path}")
print(f"Total Cost: {cost}")
```

References

GeeksforGeeks. (n.d.). GeeksforGeeks. Retrieved February 4, 2025, from https://www.geeksforgeeks.org/