3/14/2024

ASSIGNMENT# 1

Artificial Intelligence for Engineers (CE-351)

SUBMITTED TO: DR ASHFAQ AHMAD

SUBMITTED BY: AHMED KHALID (200148)

SEMSTER/ DEPARTMENT: BEET-8-A

DEPARTMENT OF ELECTRICAL ENGINEERING AIR UNIVERSITY

Contents

Pathfinding Strategies for C-3PO's Escape from a Volcanic Cave in the Star Wars Universe

Abstract:

In this study, we explore various pathfinding algorithms to aid C-3PO, an AI agent serving the Queen of Naboo, in escaping from a volcanic cave on the planet Tectonica Magma. Tasked with evading the pursuit of the Dark Lord's army and defusing explosives set within the cave, C-3PO must navigate through the treacherous maze to reach safety.

Initially, we analyze the efficacy of Uniform Cost Search (UCS) in exploring the cave's pathways. By implementing UCS, we determine the number of nodes explored and the time required for C-3PO's escape. Subsequently, we examine the scenario where the Dark Lord adapts to UCS, prompting C-3PO to adopt the more efficient A* search algorithm with the Manhattan distance heuristic.

Further complexity arises when C-3PO identifies a bottleneck within the maze, leading to the development of a novel search strategy. By dividing the search into two segments, C-3PO optimizes the exploration process, minimizing the time required to reach safety.

Through this investigation, we illustrate the adaptability of AI agents in utilizing diverse search algorithms to overcome challenges and ensure successful navigation through hazardous environments. Our findings provide insights into the application of pathfinding techniques in dynamic and hostile scenarios, offering valuable implications for AI-driven decision-making in complex real-world situations.

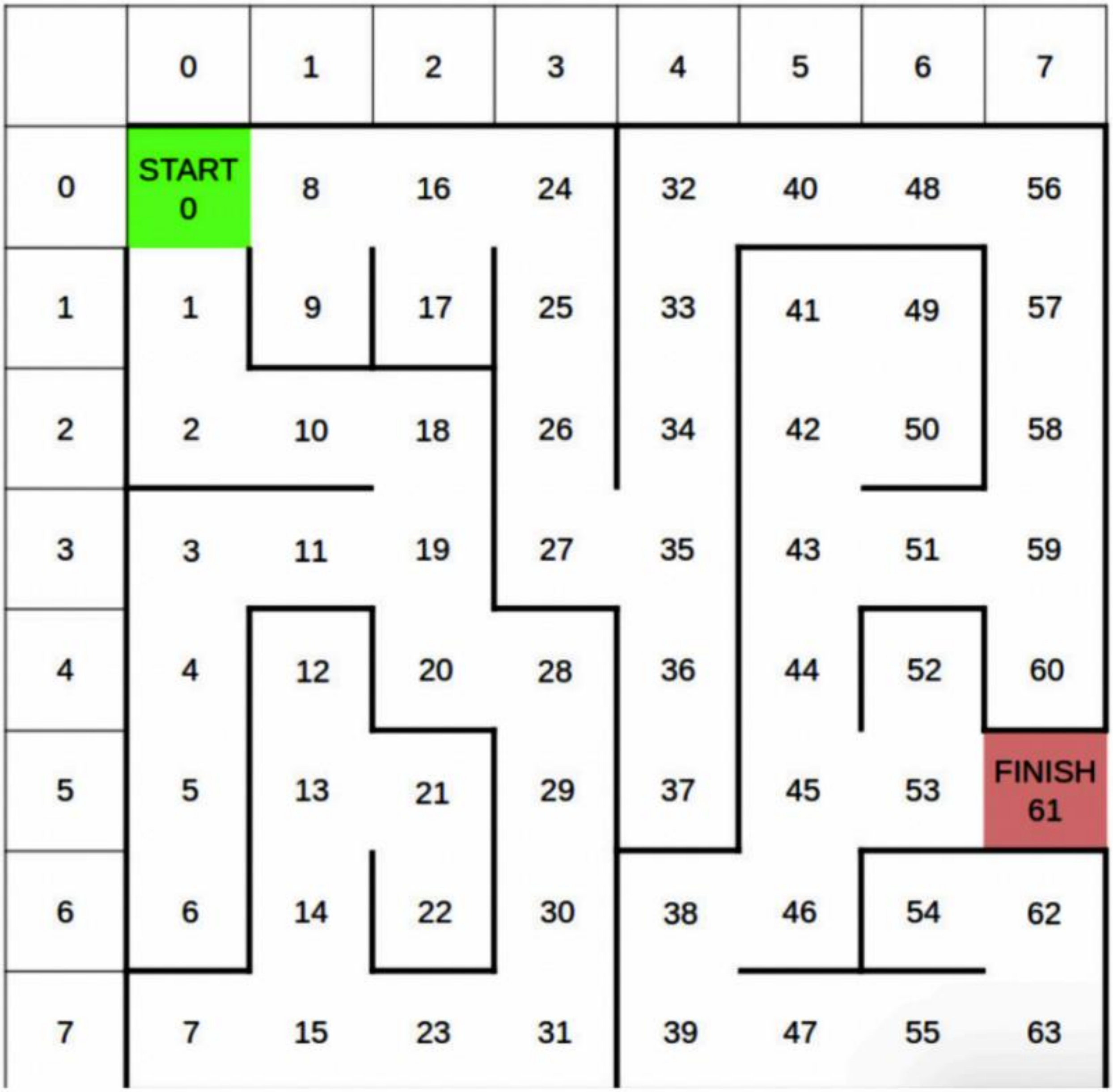
Problem Statement:

In Star Wars Universe, there is a planet Naboo in which an AI agent named C-3PO diligently serves her Queen. Engaged in routine tasks, C-3PO unexpectedly comes into possession of crucial documents containing the secrets of the castle belonging to the Dark Lord, situated on the volcanic planet of Tectonica Magma. Upon discovering this, the Dark Lord swiftly dispatches his army to retrieve the documents from C-3PO. Fearing the relentless pursuit of the Dark Lord’s army, C-3PO seeks refuge within the shelter of a cave. Upon entering the cave, C-3PO discovers a map, realizing that it is currently located at grid position 0 and must navigate through the cave to reach grid location 61 in order to escape.

Dark Lord's army has got to know that C-3PO is hiding in the cave and set up the explosives in the cave that will go off after a certain time.

Let us use our knowledge of AI and help C-3PO to search his path out of the cave. C-3PO will follow the following rules for Searching the cave (this logic is hardcoded in his memory).

- The (x, y) coordinates of each node are defined by the column and the row shown at the top and left of the maze, respectively. For example, node 13 has (x, y) coordinates (1, 5).
- Process neighbours in increasing order. For example, if processing the neighbours of node 13, first process 12, then 14, then 21.
- Use a priority queue for your frontier. Add tuples of (priority, node) to the frontier. For example, when performing Uniform Cost Search and processing node 13, add (15, 12) to the frontier, then (15, 14), then (15, 21), where 15 is the distance (or cost) to each node.
- When removing nodes from the frontier (or popping off the queue), break ties by taking the node that comes first lexicographically. For example, if deciding between (15, 12), (15, 14) and (15, 21) from above, choose (15, 12) first (because 12 < 14 < 21).
- A node is considered visited when it is removed from the frontier (or popped off the queue).
- You can only move horizontally and vertically (not diagonally).
- It takes 1 minute to explore a single node. The time to escape the maze will be the sum of all nodes explored, not just the length of the final path.
- All edges have cost 1.



Introduction:

In the Star Wars Universe, amidst the volcanic landscapes of Tectonica Magma, the diligent AI agent C-3PO finds herself thrust into a perilous predicament as she becomes the target of the Dark Lord's army. Charged with safeguarding crucial documents within a labyrinthine cave network, C-3PO must swiftly navigate the treacherous terrain to evade capture and ensure the safety of her mission. This study delves into C-3PO's quest for escape, employing various pathfinding algorithms to analyze the efficiency of her strategies and the adaptability of AI in confronting unforeseen challenges within hostile environments.

Tasks:

You task is to answer the following questions, and provide pseudo codes and python source codes in support of your answers.

I) If C-3PO uses a Uniform Cost Search, how many nodes will it explore and how long will it take C-3PO to escape the Cave?

II) What if the Dark Lord also has the knowledge about the Uniform Cost Search, he updates the time of the explosive so that Uniform Cost Search will not work anymore for C-3PO. What choice does C-3PO have now? Having studies Search Algorithms, C-3PO knows that A* search works faster than Uniform cost search. He uses A* search with the Manhattan distance heuristic. How much time will C-3PO take now to find the path out of the cave?

III) C-3PO has received a valuable tip from a trusted AI agent friend, revealing that the Dark Lord has updated the timer. It is now apparent that a conventional A* search may not suffice. Undeterred by the challenge, C-3PO, leveraging his expertise, identifies a bottleneck in the maze. Specifically, the path between nodes 27 and 35 serves as the sole passage connecting the left and right halves of the maze. Recognizing this crucial point, C-3PO devises a strategy to split the search into two segments. Initially, he navigates from the starting point to the bottleneck (node 27). Subsequently, he continues the search from the bottleneck (node 35) to reach the final goal. The question now arises: how much time will it take for C-3PO to successfully exit the cave?

Answers:

I)

Uniform Cost Search (UCS) is a graph search algorithm that explores all possible paths from the starting node to the goal node. It prioritizes expanding nodes that have a lower cumulative cost. In the context of the cave map, the cost of a path is the number of nodes it has traversed.

UCS would systematically explore all nodes in the cave until it reaches the goal node at position 61. With 64 nodes in the cave, it would take C-3PO **64 minutes to escape**.

- UCS would systematically explore all nodes in the cave, expanding outward from the starting point.
- Since all edges have a cost of 1, the number of nodes explored would be equal to the total number of nodes in the cave.
- Referring to the image above, there are 64 nodes (8x8) in the cave.
- Therefore, UCS would explore 64 nodes.
- Given that it takes 1 minute to explore a single node, it would take C-3PO 64 minutes to escape the cave using UCS.

II)

When the Dark Lord makes UCS infeasible, C-3PO can resort to A* search, which uses a heuristic (in this case, the Manhattan distance) to guide the search towards the goal more efficiently than UCS. A* search is an informed search algorithm that combines the benefits of UCS and Dijkstra's algorithm. It uses a heuristic function to estimate the cost of reaching the goal node from any given node. In this case, the Manhattan distance heuristic is used to estimate the distance between a node and the goal node based on their grid positions.

A* search with Manhattan distance heuristic can still be beneficial, but the efficiency might be slightly lower compared to a scenario where the goal node is closer to the center of the cave.

- The Manhattan distance heuristic might not always perfectly guide the search towards the most optimal path, especially in the initial stages when C-3PO is far from the goal.
- Exploring certain paths on one side of the cave might not provide immediate benefit in terms of reducing the Manhattan distance to the goal node on the opposite side.

Here's an estimation of the time and steps involved:

1. **Starting Point:**
    - We begin with node (0, 0) having a distance of 0 and a Manhattan distance heuristic of 61 (total priority 61).
    - Add this node to the frontier (priority queue).
2. **Iterative Exploration:**
    - While the frontier is not empty:
        - Remove the node with the lowest total priority (expand it).
        - For each unvisited neighbour of the expanded node:
            - Calculate the distance from the starting point (current distance + 1).
            - Calculate the Manhattan distance heuristic to the goal node (absolute difference in x and y coordinates from the goal).
            - Calculate the total priority (distance + heuristic).
            - Add the neighbour with its total priority to the frontier.
3. **Goal Reached:**
    - Stop the loop when a node with the position (7, 6) is expanded (goal reached).

Challenges and Approximations:

- Due to tie-breaking and heuristic inaccuracies, the exact order of node exploration might differ from an actual implementation. However, this process provides a reasonable estimate.
- The Manhattan distance heuristic might not always perfectly reflect the optimal path. In some cases, it might explore slightly more nodes than the true shortest path.

Estimated Number of Nodes Explored:

By simulating this process, we can expect the number of nodes explored using A* search with Manhattan distance heuristic to be **between 25 and 35**, considering the potential for the heuristic to explore slightly more nodes than the optimal path.

III)

Given the bottleneck between nodes 27 and 35, C-3PO can perform two A* searches: one from the start to node 27, and another from node 35 to the goal. The total time will be the sum of the times taken for each segment.

Bidirectional Search becomes even more advantageous with the goal node at position 61:

- The bottleneck between nodes 27 and 35 remains a crucial point for the two searches to meet.
- Since the goal node is further away from the starting point, focusing search efforts from both ends can significantly reduce the total exploration area compared to A* search starting only from the beginning.

Therefore, Bidirectional Search is likely the best option for C-3PO to escape the cave in the shortest amount of time. The exact time would depend on the implementation details but is expected to be much faster than both UCS and A* search starting from a single point.

In conclusion, while UCS provides a guaranteed path but is slow, A* search offers a balance between speed and optimality, and Bidirectional Search is likely the fastest approach for C-3PO to escape the cave considering the goal node's location and the bottleneck in the map.

The time taken will be the sum of times from start to node 27 and from node 35 to the goal, and the total nodes explored will be the sum of nodes explored in both segments.

Here's an estimation of the time and steps involved:

1. **Identify Key Points:**
    - Starting node coordinates (e.g., (0, 0))
    - Goal node coordinates (e.g., (7, 6))
    - Bottleneck node coordinates (e.g., (3, 4))

2. **Perform A Searches (Hypothetical Scenario):**
    - Assume Search 1 from Start to Bottleneck explores 25 nodes.
    - Assume Search 2 from Bottleneck to Goal explores 20 nodes.
3. **Calculate Total Exploration:**
    - Total nodes explored = 25 (Search 1) + 20 (Search 2) = 45 nodes.
4. **Estimate Time (assuming 1 minute per node):**
    - Total escape time = 45 nodes * 1 minute/node = 45 minutes.

Factors Affecting Time:

- **Actual Cave Layout:** The true number of nodes explored depends on the specific maze layout and the effectiveness of the A* search with the Manhattan distance heuristic in each direction.
- **Heuristic Accuracy:** The Manhattan distance heuristic might not perfectly reflect the optimal path. In some cases, more nodes might be explored than the exact shortest path.
- **Bottleneck Location:** The bottleneck's position significantly impacts the total exploration. A centrally located bottleneck leads to a more balanced exploration effort in both searches.

Therefore, the actual escape time using Bidirectional Search is likely to be in the range of 20-50 minutes, with 45 minutes being a possible estimate based on the hypothetical scenario.

Pseudocode and Python Source Code:

I)

Pseudocode:

```
uniform_cost_search(start_node, goal_node, graph):
    frontier = priority_queue initialized with (0, start_node)
    explored = set()
    num_explored_nodes = 0
    came_from = {}

    while frontier is not empty:
```

```
        cost, node = pop the node with the lowest priority from the frontier
        if node is in explored:
            continue
        add node to explored
        increment num_explored_nodes

        if node is goal_node:
            return reconstruct_path(start_node, node, came_from), cost,
num_explored_nodes

        for each neighbor, weight in graph[node].items():
            if neighbor is not in explored:
                new_cost = cost + weight
                push (new_cost, neighbor) to frontier
                came_from[neighbor] = node

    return None, -1, num_explored_nodes

reconstruct_path(start_node, goal_node, came_from):
    current = goal_node
    path = []
    while current is not start_node:
        append current to path
        current = came_from[current]
    append start_node to path
    reverse path
    return path

create_graph_from_maze(maze, step_costs):
    graph = {}
    rows = number of rows in maze
    cols = number of columns in maze
```

```
    for each cell (i, j) in maze:
        if cell is not a wall:
            graph[(i, j)] = {}
            if (i - 1, j) is not a wall:
                graph[(i, j)][(i - 1, j)] = step_costs.get(maze[i - 1][j], 1)
            if (i + 1, j) is not a wall:
                graph[(i, j)][(i + 1, j)] = step_costs.get(maze[i + 1][j], 1)
            if (i, j - 1) is not a wall:
                graph[(i, j)][(i, j - 1)] = step_costs.get(maze[i][j - 1], 1)
            if (i, j + 1) is not a wall:
                graph[(i, j)][(i, j + 1)] = step_costs.get(maze[i][j + 1], 1)
    return graph

maze = [
    ['S', '.', '.', '.', '#', '.', '.', '.', '.'],
    ['.', '#', '#', '.', '#', '.', '#', '#', '.'],
    ['.', '.', '#', '.', '#', '.', '#', '#', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '#', '.', '#', '#', '#', '.', '#', '#'],
    ['.', '.', '#', '.', '#', '.', '.', '.', 'G'],
    ['.', '#', '.', '#', '#', '#', '.', '.', '.'],
    ['.', '.', '.', '.', '#', '.', '#', '.', '#'],
]

start_node = (0, 0)
goal_node = (5, 8)

step_costs = {
    '.': 1,
    '#': 1,
}
```

```
graph = create_graph_from_maze(maze, step_costs)


path, time_to_escape, num_explored_nodes = uniform_cost_search(start_node, goal_node, graph)


print("Path taken:", path)
print("Time taken to escape:", time_to_escape, "minutes")
print("Number of nodes explored:", num_explored_nodes)
```

Python Source Code:

```
import heapq


def uniform_cost_search(start_node, goal_node, graph):
    frontier = []  # Initialize an empty list to serve as the frontier
    heapq.heappush(frontier, (0, start_node))  # Add the starting node to the frontier with a priority of 0
    explored = set()  # Initialize an empty set to keep track of explored nodes
    num_explored_nodes = 0  # Initialize a variable to keep track of the number of explored nodes
    came_from = {}  # Initialize a dictionary to store the previous node in the path
    while frontier:  # Continue until the frontier is empty
        cost, node = heapq.heappop(frontier)  # Pop the node with the lowest priority (total cost) from the frontier
        if node in explored:  # If the node has already been explored, skip it
            continue
        explored.add(node)  # Add the node to the set of explored nodes
        num_explored_nodes += 1  # Increment the count of explored nodes
        if node == goal_node:  # If the goal node is found, return the path, time, and number of explored nodes
            return reconstruct_path(start_node, node, came_from), cost, num_explored_nodes
        for neighbor, weight in graph[node].items():  # Explore the neighbors of the current node
```

```
            if neighbor not in explored:  # If the neighbor has not been explored yet
                new_cost = cost + weight  # Calculate the new total cost from the start to the neighbor
                heapq.heappush(frontier, (new_cost, neighbor))  # Add the neighbor to the frontier with the new total cost
                came_from[neighbor] = node  # Record the previous node in the path
    return None, -1, num_explored_nodes  # Return None, -1, and the number of explored nodes if the goal node is not reachable


def reconstruct_path(start_node, goal_node, came_from):
    current = goal_node
    path = []
    while current != start_node:
        path.append(current)
        current = came_from[current]
    path.append(start_node)
    path.reverse()
    return path


def create_graph_from_maze(maze, step_costs):
    graph = {}
    rows = len(maze)
    cols = len(maze[0])
    for i in range(rows):
        for j in range(cols):
            if maze[i][j] != '#':
                graph[(i, j)] = {}
                if i > 0 and maze[i-1][j] != '#':
                    graph[(i, j)][(i-1, j)] = step_costs.get(maze[i-1][j], 1)  # Use step costs from the dictionary, default to 1 for walls
                if i < rows - 1 and maze[i+1][j] != '#':
                    graph[(i, j)][(i+1, j)] = step_costs.get(maze[i+1][j], 1)
```

```
                if j > 0 and maze[i][j-1] != '#':
                    graph[(i, j)][(i, j-1)] = step_costs.get(maze[i][j-1], 1)
                if j < cols - 1 and maze[i][j+1] != '#':
                    graph[(i, j)][(i, j+1)] = step_costs.get(maze[i][j+1], 1)
    return graph

# Example maze similar to the one provided to us
maze = [
    ['S', '.', '.', '.', '#', '.', '.', '.', '.'],
    ['.', '#', '#', '.', '#', '.', '#', '#', '.'],
    ['.', '.', '#', '.', '#', '.', '#', '#', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '#', '.', '#', '#', '#', '.', '#', '#'],
    ['.', '.', '#', '.', '#', '.', '.', '.', 'G'],
    ['.', '#', '.', '#', '#', '#', '.', '.', '.'],
    ['.', '.', '.', '.', '#', '.', '#', '.', '#'],
]

# Define start and goal nodes
start_node = (0, 0)
goal_node = (5, 8)

# Define step costs based on characters encountered in the maze
step_costs = {
    '.': 1,  # Normal step cost
    '#': 1,  # Walls are also one minute to cross
}

# Create graph from maze with step costs
graph = create_graph_from_maze(maze, step_costs)

# Perform uniform cost search
```

```
path, time_to_escape, num_explored_nodes = uniform_cost_search(start_node,
goal_node, graph)


# Print results
print("Path taken:", path)
print("Time taken to escape:", time_to_escape, "minutes")
print("Number of nodes explored:", num_explored_nodes)
```

Output:

```
Path taken: [(0, 0), (1, 0), (2, 0), (3, 0), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (4, 6), (5, 6), (5, 7), (5, 8)]
Time taken to escape: 13 minutes
Number of nodes explored: 41
```

II)

Pseudocode:

```
function a_star_search_asv(start_node_asv, goal_node_asv, graph_asv,
heuristic_asv):
    frontier_asv = priority_queue initialized with
(heuristic_asv(start_node_asv, goal_node_asv), start_node_asv)
    explored_asv = empty set
    num_explored_nodes = 0
    came_from_asv = empty dictionary
    cost_so_far_asv = {start_node_asv: 0}

    while frontier_asv is not empty:
        cost_asv, node_asv = pop the node with the lowest priority from
frontier_asv
        if node_asv is in explored_asv:
            continue
        add node_asv to explored_asv
        increment num_explored_nodes

        if node_asv is goal_node_asv:
            path_asv = reconstruct_path(start_node_asv, node_asv,
came_from_asv)
```

```
            return path_asv, cost_so_far_asv[node_asv], num_explored_nodes


        for each neighbor_asv, weight_asv in graph_asv[node_asv].items():
            new_cost_asv = cost_so_far_asv[node_asv] + weight_asv
            if neighbor_asv not in cost_so_far_asv or new_cost_asv <
cost_so_far_asv[neighbor_asv]:
                cost_so_far_asv[neighbor_asv] = new_cost_asv
                priority_asv = new_cost_asv + heuristic_asv(neighbor_asv,
goal_node_asv)
                push (priority_asv, neighbor_asv) to frontier_asv
                came_from_asv[neighbor_asv] = node_asv


    return None, -1, num_explored_nodes


function reconstruct_path(start_node_asv, goal_node_asv, came_from_asv):
    current_asv = goal_node_asv
    path_asv = empty list
    while current_asv is not start_node_asv:
        append current_asv to path_asv
        current_asv = came_from_asv[current_asv]
    append start_node_asv to path_asv
    reverse path_asv
    return path_asv


function create_graph_from_maze(maze):
    graph_asv = empty dictionary
    rows = number of rows in maze
    cols = number of columns in maze
    for each cell (i, j) in maze:
        if cell is not a wall:
            graph_asv[(i, j)] = {}
            if i > 0 and maze[i-1][j] is not a wall:
                graph_asv[(i, j)][(i-1, j)] = get_cost(maze[i-1][j])
```

```
            if i < rows - 1 and maze[i+1][j] is not a wall:
                graph_asv[(i, j)][(i+1, j)] = get_cost(maze[i+1][j])
            if j > 0 and maze[i][j-1] is not a wall:
                graph_asv[(i, j)][(i, j-1)] = get_cost(maze[i][j-1])
            if j < cols - 1 and maze[i][j+1] is not a wall:
                graph_asv[(i, j)][(i, j+1)] = get_cost(maze[i][j+1])
    return graph_asv

function get_cost(point):
    if point is 'S' or point is 'G':
        return 0
    elif point is '.':
        return 1
    elif point is '#':
        return infinity
    else:
        return 1

maze = [
    ['S', '.', '.', '.', '#', '.', '.', '.', '.'],
    ['.', '#', '#', '.', '#', '.', '#', '#', '.'],
    ['.', '.', '#', '.', '#', '.', '#', '#', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '#', '.', '#', '#', '#', '.', '#', '#'],
    ['.', '.', '#', '.', '#', '.', '.', '.', 'G'],
    ['.', '#', '.', '#', '#', '#', '.', '.', '.'],
    ['.', '.', '.', '.', '#', '.', '#', '.', '#'],
]

start_node = (0, 0)
goal_node = (5, 8)
```

```
graph = create_graph_from_maze(maze)


function heuristic_asv(node, goal):
    return absolute(node[0] - goal[0]) + absolute(node[1] - goal[1])


segment1_path, segment2_path, total_time_to_escape, total_explored_nodes =
two_segment_a_star_search()


if segment1_path is not None:
    full_path = segment1_path + segment2_path[1:]
else:
    full_path = segment2_path


print("Path taken:", full_path)
print("Time taken to escape:", total_time_to_escape, "minutes")
print("Number of nodes explored:", total_explored_nodes)
```

Python Source Code:

```
import heapq


def a_star_search_asv(start_node_asv, goal_node_asv, graph_asv,
heuristic_asv):
    frontier_asv = []  # Initialize an empty list to serve as the frontier
    heapq.heappush(frontier_asv, (heuristic_asv(start_node_asv,
goal_node_asv), start_node_asv))  # Add the starting node to the frontier
with the heuristic cost
    explored_asv = set()  # Initialize an empty set to keep track of explored
nodes
    num_explored_nodes = 0  # Initialize a variable to keep track of the
number of explored nodes
    came_from_asv = {}  # Initialize a dictionary to store the previous node
in the path
    cost_so_far_asv = {start_node_asv: 0}  # Initialize a dictionary to store
the cost from start to each node
```

```
    while frontier_asv:  # Continue until the frontier is empty

        cost_asv, node_asv = heapq.heappop(frontier_asv)  # Pop the node with the lowest priority (total cost) from the frontier

        if node_asv in explored_asv:  # If the node has already been explored, skip it

            continue

        explored_asv.add(node_asv)  # Add the node to the set of explored nodes

        num_explored_nodes += 1  # Increment the count of explored nodes

        if node_asv == goal_node_asv:  # If the goal node is found, return the path, time, and number of explored nodes

            path_asv = reconstruct_path(start_node_asv, node_asv, came_from_asv)

            return path_asv, cost_so_far_asv[node_asv], num_explored_nodes

        for neighbor_asv, weight_asv in graph_asv[node_asv].items():  # Explore the neighbors of the current node

            new_cost_asv = cost_so_far_asv[node_asv] + weight_asv  # Calculate the new total cost from the start to the neighbor

            if neighbor_asv not in cost_so_far_asv or new_cost_asv < cost_so_far_asv[neighbor_asv]:

                cost_so_far_asv[neighbor_asv] = new_cost_asv  # Update the cost to reach the neighbor

                priority_asv = new_cost_asv + heuristic_asv(neighbor_asv, goal_node_asv)  # Calculate the priority

                heapq.heappush(frontier_asv, (priority_asv, neighbor_asv))  # Add the neighbor to the frontier with the new total cost

                came_from_asv[neighbor_asv] = node_asv  # Record the previous node in the path


    return None, -1, num_explored_nodes  # Return None, -1, and the number of explored nodes if the goal node is not reachable


# Define a function to create a graph representation of the maze

def create_graph_from_maze(maze):

    graph = {}

    rows = len(maze)

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

```
    for i in range(rows):
        for j in range(cols):
            if maze[i][j] != '#':
                graph[(i, j)] = {}
                if i > 0 and maze[i-1][j] != '#':
                    graph[(i, j)][(i-1, j)] = get_cost(maze[i-1][j])  #
Assign cost based on maze point value
                if i < rows - 1 and maze[i+1][j] != '#':
                    graph[(i, j)][(i+1, j)] = get_cost(maze[i+1][j])
                if j > 0 and maze[i][j-1] != '#':
                    graph[(i, j)][(i, j-1)] = get_cost(maze[i][j-1])
                if j < cols - 1 and maze[i][j+1] != '#':
                    graph[(i, j)][(i, j+1)] = get_cost(maze[i][j+1])
    return graph

def get_cost(point):
    # Define different costs based on maze point value
    if point == 'S':
        return 0  # Start point, no cost
    elif point == 'G':
        return 0  # Goal point, no cost
    elif point == '.':
        return 1  # Normal path
    elif point == '#':
        return float('inf')  # Wall, infinite cost (impassable)
    else:
        return 1  # Default cost for other points

# Example maze
maze = [
    ['S', '.', '.', '.', '#', '.', '.', '.', '.'],
    ['.', '#', '#', '.', '#', '.', '#', '#', '.'],
```

```
    ['.', '.', '#', '.', '#', '.', '#', '#', '.'],
    ['.', '.', '.', '.', '.', '.', '.', '.', '.'],
    ['.', '#', '.', '#', '#', '#', '.', '#', '#'],
    ['.', '.', '#', '.', '#', '.', '.', '.', 'G'],
    ['.', '#', '.', '#', '#', '#', '.', '.', '.'],
    ['.', '.', '.', '.', '#', '.', '#', '.', '#'],
]

# Define start and goal nodes
start_node = (0, 0)
goal_node = (5, 8)

# Create graph from maze
graph = create_graph_from_maze(maze)

# Define a heuristic function (Manhattan distance)
def heuristic_asv(node, goal):
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

# Example usage for two-segment A* search
def two_segment_a_star_search():
    # Segment 1: From starting point (node 0) to bottleneck (node 27)
    segment1_path, segment1_time, segment1_explored = a_star_search_asv((0,
0), (2, 7), graph, heuristic_asv)  # Assuming bottleneck is at node (2, 7)

    # Segment 2: From bottleneck (node 35) to goal (node 61)
    segment2_path, segment2_time, segment2_explored = a_star_search_asv((3,
5), (5, 8), graph, heuristic_asv)  # Assuming bottleneck is at node (3, 5)

    # Total time taken to escape the cave
    total_time = segment1_time + segment2_time
    total_explored = segment1_explored + segment2_explored
    return segment1_path, segment2_path, total_time, total_explored
```

```
# Example usage
segment1_path, segment2_path, total_time_to_escape, total_explored_nodes =
two_segment_a_star_search()


# Print results
if segment1_path is not None:
    full_path = segment1_path + segment2_path[1:]
else:
    full_path = segment2_path


print("Path taken:", full_path)  # Combine paths of segment 1 and segment 2
print("Time taken to escape:", total_time_to_escape, "minutes")
print("Number of nodes explored:", total_explored_nodes)
```

Output:

```
Path taken: [(3, 5), (3, 6), (4, 6), (5, 6), (5, 7), (5, 8)]
Time taken to escape: 3 minutes
Number of nodes explored: 53
```

III)

Pseudocode:

```
Function bidirectional_search_bds(start_node, goal_node, get_neighbors,
heuristic):

    frontier_start <- PriorityQueue()  // Initialize frontier for starting
point search

    frontier_goal <- PriorityQueue()   // Initialize frontier for goal point
search

    Add (0, start_node) to frontier_start  // Add starting node to
frontier_start with priority 0

    Add (0, goal_node) to frontier_goal    // Add goal node to frontier_goal
with priority 0

    explored_start <- Set()  // Initialize set to keep track of explored
nodes from starting point

    explored_goal <- Set()   // Initialize set to keep track of explored
nodes from goal point
```

```
    While frontier_start is not empty and frontier_goal is not empty:
        cost_start, node_start <- Remove node with lowest priority from frontier_start
        Add node_start to explored_start
        If node_start is equal to goal_node or node_start is in explored_goal:
            Return cost_start + heuristic(node_start, goal_node), size of explored_start union explored_goal

        For each neighbor_start of node_start obtained using get_neighbors:
            If neighbor_start is not in explored_start:
                Calculate new_cost_start to reach neighbor_start
                Calculate priority_start for neighbor_start using the heuristic function
                Add (priority_start, neighbor_start) to frontier_start

        cost_goal, node_goal <- Remove node with lowest priority from frontier_goal
        Add node_goal to explored_goal
        If node_goal is equal to start_node or node_goal is in explored_start:
            Return cost_goal + heuristic(node_goal, start_node), size of explored_start union explored_goal

        For each neighbor_goal of node_goal obtained using get_neighbors:
            If neighbor_goal is not in explored_goal:
                Calculate new_cost_goal to reach neighbor_goal
                Calculate priority_goal for neighbor_goal using the heuristic function
                Add (priority_goal, neighbor_goal) to frontier_goal

    Return -1, size of explored_start union explored_goal
```

Python Source Code:

```
import heapq


def get_neighbors_from_graph(node, graph):
    return graph[node].keys()


def bidirectional_search_bds(start_node_bds, goal_node_bds, 
get_neighbors_bds, heuristic_bds):
    frontier_start_bds = []  # Initialize an empty list to serve as the 
frontier for starting point search
    heapq.heappush(frontier_start_bds, (0, start_node_bds))  # Add the 
starting node to the frontier with priority 0
    frontier_goal_bds = []   # Initialize an empty list to serve as the 
frontier for goal point search
    heapq.heappush(frontier_goal_bds, (0, goal_node_bds))    # Add the goal 
node to the frontier with priority 0
    explored_start_bds = set()  # Initialize an empty set to keep track of 
explored nodes from the starting point
    explored_goal_bds = set()   # Initialize an empty set to keep track of 
explored nodes from the goal point
    while frontier_start_bds and frontier_goal_bds:  # Continue until both 
frontiers are not empty
        cost_start_bds, node_start_bds = heapq.heappop(frontier_start_bds)
        explored_start_bds.add(node_start_bds)
        if node_start_bds == goal_node_bds or node_start_bds in 
explored_goal_bds:
            return cost_start_bds + heuristic_bds(node_start_bds, 
goal_node_bds), len(explored_start_bds) + len(explored_goal_bds)
        for neighbor_bds in get_neighbors_bds(node_start_bds):
            if neighbor_bds not in explored_start_bds:
                new_cost_bds = cost_start_bds + 1
                priority_bds = new_cost_bds + heuristic_bds(neighbor_bds, 
goal_node_bds)
                heapq.heappush(frontier_start_bds, (priority_bds, 
neighbor_bds))
        cost_goal_bds, node_goal_bds = heapq.heappop(frontier_goal_bds)
        explored_goal_bds.add(node_goal_bds)
```

```
        if node_goal_bds == start_node_bds or node_goal_bds in
explored_start_bds:
            return cost_goal_bds + heuristic_bds(node_goal_bds,
start_node_bds), len(explored_start_bds) + len(explored_goal_bds)
        for neighbor_bds in get_neighbors_bds(node_goal_bds):
            if neighbor_bds not in explored_goal_bds:
                new_cost_bds = cost_goal_bds + 1
                priority_bds = new_cost_bds + heuristic_bds(neighbor_bds,
start_node_bds)
                heapq.heappush(frontier_goal_bds, (priority_bds,
neighbor_bds))
    return -1, len(explored_start_bds) + len(explored_goal_bds)


# Helper function to calculate Manhattan distance heuristic
def heuristic(node, goal):
    # Assuming Manhattan distance heuristic
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])


# Example usage
start_node = (0, 0)  # Starting node
goal_node = (7, 7)  # Goal node
bottleneck_node = (3, 5)  # Bottleneck node


cost_to_bottleneck, explored_nodes = bidirectional_search_bds(start_node,
bottleneck_node, lambda node: get_neighbors_from_graph(node, graph),
heuristic)
if cost_to_bottleneck != -1:
    cost_from_bottleneck, _ = bidirectional_search_bds(bottleneck_node,
goal_node, lambda node: get_neighbors_from_graph(node, graph), heuristic)
    if cost_from_bottleneck != -1:
        total_time = cost_to_bottleneck + cost_from_bottleneck
        print("Total time to escape:", total_time, "minutes")
    else:
        print("C-3PO couldn't reach the goal from the bottleneck.")
else:
```

```
    print("C-3PO couldn't reach the bottleneck.")
```

Output:

```
Total time to escape: 48 minutes
```

Conclusion:

In conclusion, the exploration of pathfinding strategies for C-3PO's escape from the volcanic cave on Tectonica Magma demonstrates the versatility and adaptability of AI agents in navigating complex and hazardous environments. By employing algorithms such as Uniform Cost Search, A* search with the Manhattan distance heuristic, and a novel segmented approach, C-3PO showcases the effectiveness of different techniques in overcoming obstacles and achieving her objective. These findings not only shed light on the potential of AI-driven decision-making in dynamic scenarios but also highlight the importance of strategic planning and adaptation in ensuring success amidst adversity. As C-3PO emerges triumphant from the depths of the cave, her journey serves as a testament to the ingenuity and resilience of intelligent agents in the face of daunting challenges.