wiegman_lab02

September 11, 2024

TJ Wiegman
ASM 591 AI
Lab 2
2024-09-11

1 The Quest for the Lost Treasure: Navigating Eldoria with Search Algorithms

Welcome, adventurer! In this notebook, you will learn how to navigate through the treacherous lands of Eldoria using various search algorithms. Your goal is to find the **Lost Treasure** hidden deep within the **Cave of Mysteries**. Along the way, you will encounter various challenges, but fear not, as you will have powerful search algorithms at your disposal!

1.1 1. Problem Formulation: Navigating Eldoria

Before you set off on your quest, it's crucial to understand how to formulate the problem. Here's how you can model the land of Eldoria:

- **Initial State:** Your starting point, the village of **Aldoria** at the top-left corner of the grid.
- **Goal State:** The **Cave of Mysteries** where the Lost Treasure is hidden, located at the bottom-right corner of the grid.
- **State Space:** The grid representing Eldoria, where each cell can be a path (you can walk), an obstacle (you cannot walk), or a dangerous area (adds extra cost).
- **Operators/Actions:** You can move up, down, left, or right to adjacent cells.
- **Path Cost:** Each step has a uniform cost unless you enter a dangerous area, which increases the cost.

Let's now explore the tools you'll use to find the Lost Treasure.

[ ]:
```python
# Example setup of the land of Eldoria as a grid
eldoria_grid = [
    ['P', 'P', 'P', 'D', 'P'],
    ['M', 'M', 'P', 'D', 'P'],
    ['P', 'P', 'P', 'P', 'P'],
    ['P', 'M', 'D', 'M', 'P'],
    ['P', 'P', 'P', 'P', 'C']  # 'C' marks the Cave of Mysteries (goal)
]

# Legend:
```

```
# 'P' = Path (can walk)
# 'M' = Mountain (cannot walk)
# 'D' = Dragon's Lair (dangerous area, higher cost)
# 'C' = Cave of Mysteries (goal)
```

#For Loop: In Eldoria, you can move in four directions: up, down, left, and right. Here's how you can use a `for` loop to iterate over these directions.

[ ]:
```
# Directions: (dx, dy)
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

# Current position
x, y = 2, 2  # Middle of the grid

# Explore all possible directions
for dx, dy in directions:
    nx, ny = x + dx, y + dy
    print(f"Checking position ({nx}, {ny})")
```

```
Checking position (1, 2)
Checking position (3, 2)
Checking position (2, 1)
Checking position (2, 3)
```

#If Statements: You can't walk into mountains or out of the grid. Use `if` statements to check whether a move is valid.

[ ]:
```
# Current position
x, y = 2, 2

# Check all directions
for dx, dy in directions:
    nx, ny = x + dx, y + dy
    if 0 <= nx < len(eldoria_grid) and 0 <= ny < len(eldoria_grid[0]):
        if eldoria_grid[nx][ny] != 'M':  # Check if it's not a mountain
            print(f"Moving to ({nx}, {ny}) is allowed")
        else:
            print(f"Cannot move to ({nx}, {ny}) - Mountain ahead!")
    else:
        print(f"Cannot move to ({nx}, {ny}) - Out of bounds!")
```

```
Moving to (1, 2) is allowed
Moving to (3, 2) is allowed
Moving to (2, 1) is allowed
Moving to (2, 3) is allowed
```

#While Loops:Let's use a `while` loop to continue searching until you either find the treasure or run out of places to explore.

```
[ ]: # Example: Simple exploration using while loop
from collections import deque
# Start position
start = (0, 0)
goal = (4, 4)
queue = deque([start])
visited = set()

# While there are places to explore
while queue:
    current = queue.popleft()
    x, y = current

    # Check if we have reached the goal
    if current == goal:
        print("Found the Lost Treasure!")
        break

    visited.add(current)

    # Explore neighbors
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if (nx, ny) not in visited and 0 <= nx < len(eldoria_grid) and 0 <= ny␣
 ↪< len(eldoria_grid[0]):
            if eldoria_grid[nx][ny] != 'M':
                queue.append((nx, ny))
else:
    print("Could not find the Lost Treasure...")
```

```
Found the Lost Treasure!
```

To help you navigate through Eldoria, you'll use various data structures:

- **Queues:** To explore all possible paths level by level.
- **Stacks:** To dive deep into unknown areas first.
- **Priority Queues/Heaps:** To always choose the most promising path based on cost and heuristic.
- **Sets:** To keep track of areas you've already explored.
- **Dictionaries:** To remember your way back (in case you get lost).

#Queues: A **queue** is a data structure that follows the **First In, First Out (FIFO)** principle. Imagine a line of people waiting to get into a treasure vault. The first person in line is the first one to enter, and as more people join the line, they wait at the back.

In search algorithms like Breadth-First Search (BFS), we use a queue to explore all possible paths at the current level before moving on to deeper levels. This ensures that we explore paths in the order of their proximity to the starting point.

1.1.1 Example: Using a Queue to Explore Eldoria

[ ]:
```python
from collections import deque

# Initialize a queue with the starting position (0, 0)
queue = deque([(0, 0)])  # Start at Aldoria
visited = set()  # Keep track of visited positions

while queue:
    current_position = queue.popleft()  # Get the position at the front of the␣
 ↪queue
    x, y = current_position
    visited.add(current_position)  # Mark this position as visited

    print(f"Exploring position: {current_position}")

    # Explore the neighbors of the current position
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        neighbor_position = (nx, ny)

        if (0 <= nx < len(eldoria_grid) and 0 <= ny < len(eldoria_grid[0]) and
            neighbor_position not in visited and eldoria_grid[nx][ny] != 'M'):
            queue.append(neighbor_position)  # Add valid neighbors to the back␣
 ↪of the queue
```

```
Exploring position: (0, 0)
Exploring position: (0, 1)
Exploring position: (0, 2)
Exploring position: (1, 2)
Exploring position: (0, 3)
Exploring position: (2, 2)
Exploring position: (1, 3)
Exploring position: (1, 3)
Exploring position: (0, 4)
Exploring position: (3, 2)
Exploring position: (2, 1)
Exploring position: (2, 3)
Exploring position: (2, 3)
Exploring position: (1, 4)
Exploring position: (2, 3)
Exploring position: (1, 4)
Exploring position: (1, 4)
Exploring position: (4, 2)
Exploring position: (2, 0)
Exploring position: (2, 4)
Exploring position: (2, 4)
Exploring position: (2, 4)
Exploring position: (2, 4)
Exploring position: (2, 4)
Exploring position: (2, 4)
Exploring position: (4, 1)
Exploring position: (4, 3)
Exploring position: (3, 0)
Exploring position: (3, 4)
Exploring position: (3, 4)
Exploring position: (3, 4)
Exploring position: (3, 4)
Exploring position: (3, 4)
Exploring position: (3, 4)
Exploring position: (4, 0)
Exploring position: (4, 4)
Exploring position: (4, 0)
Exploring position: (4, 4)
Exploring position: (4, 4)
Exploring position: (4, 4)
Exploring position: (4, 4)
Exploring position: (4, 4)
Exploring position: (4, 4)
```

1.1.2 Key Points:

- **popleft()** removes and returns the first element in the queue, representing the next position to explore.
- **append()** adds elements to the end of the queue, representing new positions to explore in the future.
- This structure ensures that we explore positions in the order they were discovered, which is ideal for finding the shortest path in unweighted scenarios.

#Stacks: A **stack** is a data structure that follows the **Last In, First Out (LIFO)** principle. Imagine a stack of books where you can only take the top book off first. The last book you put on the stack is the first one you take off.

In search algorithms like Depth-First Search (DFS), we use a stack to dive deep into one path before backtracking and trying another. This approach is useful when you want to explore as far as possible along a path before considering alternatives.

1.1.3 Example: Using a Stack to Explore Eldoria

[ ]:
```python
# Initialize a stack with the starting position (0, 0)
stack = [(0, 0)]  # Start at Aldoria
visited_stack = set()  # Keep track of visited positions

while stack:
    current_position = stack.pop()  # Get the position at the top of the stack
    x, y = current_position
    visited_stack.add(current_position)  # Mark this position as visited

    print(f"Diving into position: {current_position}")

    # Explore the neighbors of the current position
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        neighbor_position = (nx, ny)

        if (0 <= nx < len(eldoria_grid) and 0 <= ny < len(eldoria_grid[0]) and
            neighbor_position not in visited_stack and eldoria_grid[nx][ny] !=␣
↪'M'):
            stack.append(neighbor_position)  # Add valid neighbors to the top␣
↪of the stack
```

```
Diving into position: (0, 0)
Diving into position: (0, 1)
Diving into position: (0, 2)
Diving into position: (0, 3)
Diving into position: (0, 4)
Diving into position: (1, 4)
Diving into position: (1, 3)
Diving into position: (1, 2)
Diving into position: (2, 2)
Diving into position: (2, 3)
Diving into position: (2, 4)
Diving into position: (3, 4)
Diving into position: (4, 4)
Diving into position: (4, 3)
Diving into position: (4, 2)
Diving into position: (4, 1)
Diving into position: (4, 0)
Diving into position: (3, 0)
Diving into position: (2, 0)
Diving into position: (2, 1)
Diving into position: (3, 2)
Diving into position: (2, 1)
Diving into position: (3, 2)
Diving into position: (2, 3)
Diving into position: (2, 4)
Diving into position: (1, 3)
Diving into position: (1, 2)
```

1.1.4 Key Points:

- **pop()** removes and returns the last element in the stack, representing the next position to explore.

- **append()** adds elements to the top of the stack, representing new positions to explore immediately.
- This structure leads to exploring one path as deeply as possible before backtracking, which can be efficient in specific scenarios.

#Priority Queue: A **priority queue** is a data structure where each element has a priority, and elements with higher priority are processed before those with lower priority. In Python, we often implement priority queues using a **heap**. A heap is a binary tree that maintains the heap property: the parent node is always less than or equal to its child nodes, making it easy to retrieve the smallest element.

In search algorithms like A* and Best-First Search, we use a priority queue to always explore the most promising path first, based on a combination of cost and heuristic.

1.1.5 Example: Using a Priority Queue to Explore Eldoria

[ ]:
```
import heapq

# Initialize a priority queue with the starting position (0, 0) and priority 0
priority_queue = []
heapq.heappush(priority_queue, (0, (0, 0)))  # (priority, position)
visited_pq = set()  # Keep track of visited positions

while priority_queue:
    current_priority, current_position = heapq.heappop(priority_queue)  # Get␣
↪the position with the highest priority
    x, y = current_position
    visited_pq.add(current_position)  # Mark this position as visited

    print(f"Exploring position with priority {current_priority}:␣
↪{current_position}")

    # Explore the neighbors of the current position
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        neighbor_position = (nx, ny)

        if (0 <= nx < len(eldoria_grid) and 0 <= ny < len(eldoria_grid[0]) and
            neighbor_position not in visited_pq and eldoria_grid[nx][ny] !=␣
↪'M'):
            new_priority = current_priority + 1  # Example: priority is the␣
↪path cost
            heapq.heappush(priority_queue, (new_priority, neighbor_position)) ␣
↪# Add neighbor with new priority
```

```
Exploring position with priority 0: (0, 0)
Exploring position with priority 1: (0, 1)
Exploring position with priority 2: (0, 2)
Exploring position with priority 3: (0, 3)
Exploring position with priority 3: (1, 2)
Exploring position with priority 4: (0, 4)
Exploring position with priority 4: (1, 3)
Exploring position with priority 4: (1, 3)
Exploring position with priority 4: (2, 2)
Exploring position with priority 5: (1, 4)
Exploring position with priority 5: (1, 4)
Exploring position with priority 5: (1, 4)
Exploring position with priority 5: (2, 1)
Exploring position with priority 5: (2, 3)
Exploring position with priority 5: (2, 3)
Exploring position with priority 5: (2, 3)
Exploring position with priority 5: (3, 2)
Exploring position with priority 6: (2, 0)
Exploring position with priority 6: (2, 4)
Exploring position with priority 6: (2, 4)
Exploring position with priority 6: (2, 4)
Exploring position with priority 6: (2, 4)
Exploring position with priority 6: (2, 4)
Exploring position with priority 6: (2, 4)
Exploring position with priority 6: (4, 2)
Exploring position with priority 7: (3, 0)
Exploring position with priority 7: (3, 4)
Exploring position with priority 7: (3, 4)
Exploring position with priority 7: (3, 4)
Exploring position with priority 7: (3, 4)
Exploring position with priority 7: (3, 4)
Exploring position with priority 7: (3, 4)
Exploring position with priority 7: (4, 1)
Exploring position with priority 7: (4, 3)
Exploring position with priority 8: (4, 0)
Exploring position with priority 8: (4, 0)
Exploring position with priority 8: (4, 4)
Exploring position with priority 8: (4, 4)
Exploring position with priority 8: (4, 4)
Exploring position with priority 8: (4, 4)
Exploring position with priority 8: (4, 4)
Exploring position with priority 8: (4, 4)
Exploring position with priority 8: (4, 4)
```

1.1.6 Key Points:

- **heappop()** removes and returns the element with the highest priority (smallest value) from the priority queue.
- **heappush()** adds an element to the priority queue with a specific priority, which determines the order of exploration.
- This structure allows for more informed exploration, making it ideal for finding the shortest

path when path costs vary.

Now that you've seen how queues, stacks, and priority queues work, let's summarize the key differences and how they affect search algorithms:

Structure	**Principle**	**Search Algorithm**	**Use Case**	**Advantages**	**Disadvantages**
Queue	First In, First Out (FIFO)	Breadth-First Search (BFS)	Exploring all paths at the same level	Guarantees the shortest path in terms of steps (unweighted)	Can be memory-intensive due to large frontier
Stack	Last In, First Out (LIFO)	Depth-First Search (DFS)	Exploring as deep as possible before backtracking	Requires less memory, can find solutions quickly	Might not find the shortest path, can get stuck in deep paths
Priority Queue	Elements prioritized by cost	A*, Best-First Search	Exploring the most promising paths first	Finds optimal path when using appropriate heuristics	Requires careful heuristic design, more complex to implement

1.1.7 Key Takeaways:

1. **Queues** (used in BFS) are ideal when you want to explore paths in an orderly fashion, ensuring that you find the shortest unweighted path.
2. **Stacks** (used in DFS) are useful when deep exploration is needed, though they may not always find the most optimal path.
3. **Priority Queues** (used in A* and Best-First Search) provide the best of both worlds by considering both path cost and heuristic, making them powerful tools for finding optimal paths in weighted scenarios.

[ ]:
```
from collections import deque
import heapq

# Queue implementation for exploring paths level by level
queue = deque()
queue.append((0, 0))  # Start at Aldoria (0, 0)
visited = set()

# Stack implementation for diving deep into unknown paths
stack = []
stack.append((0, 0))  # Start at Aldoria (0, 0)
visited_stack = set()

# Priority queue for choosing the best path
priority_queue = []
heapq.heappush(priority_queue, (0, (0, 0)))  # (cost, position)
visited_pq = set()
```

1.1.8 Implementing BFS to Find the Lost Treasure

Breadth-First Search (BFS) explores all possible paths level by level, ensuring you find the shortest path to the Cave of Mysteries (if one exists). Let's implement BFS to help you find the Lost Treasure!

[ ]:
```
from collections import deque

def bfs(start, goal, grid):
    queue = deque([start])
    visited = set()
    parent = {}

    visited.add(start)
    parent[start] = None

    while queue:
        current = queue.popleft()

        if current == goal:
            return reconstruct_path(parent, start, goal)

        for neighbor in get_neighbors(current, grid):
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = current
                queue.append(neighbor)

    return None  # If no path is found

def get_neighbors(pos, grid):
    neighbors = []
    x, y = pos
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]):
            if grid[nx][ny] != 'M':  # Can't walk on mountains
                neighbors.append((nx, ny))
    return neighbors

def reconstruct_path(parent, start, goal):
    path = []
    current = goal
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()
    return path

# Test BFS to find the Lost Treasure
start = (0, 0)  # Aldoria
goal = (4, 4)   # Cave of Mysteries
path = bfs(start, goal, eldoria_grid)
print("Path to the Lost Treasure:", path)
```

```
Path to the Lost Treasure: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (4,
2), (4, 3), (4, 4)]
```

1.1.9 Implementing A* Search to Avoid Danger

A* Search combines the advantages of BFS with a heuristic to avoid dangerous areas (like the Dragon's Lair) while searching for the Lost Treasure. This algorithm will help you find a safe and efficient path.

[ ]:
```
import heapq

def a_star(start, goal, grid, heuristic):
    priority_queue = []
    heapq.heappush(priority_queue, (0, start))
    g_costs = {start: 0}
    parent = {start: None}

    while priority_queue:
        _, current = heapq.heappop(priority_queue)

        if current == goal:
            return reconstruct_path(parent, start, goal)

        for neighbor in get_neighbors(current, grid):
            tentative_g_cost = g_costs[current] + get_cost(neighbor, grid)
            if neighbor not in g_costs or tentative_g_cost < g_costs[neighbor]:
                g_costs[neighbor] = tentative_g_cost
                f_cost = tentative_g_cost + heuristic(neighbor, goal)
                heapq.heappush(priority_queue, (f_cost, neighbor))
                parent[neighbor] = current

    return None  # If no path is found

def get_cost(pos, grid):
    x, y = pos
    if grid[x][y] == 'D':
        return 5  # High cost to enter Dragon's Lair
    return 1  # Normal cost

def heuristic(node, goal):
    # Example heuristic: Manhattan distance
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

# Test A* to find a safe path to the Lost Treasure
path = a_star(start, goal, eldoria_grid, heuristic)
print("Safe path to the Lost Treasure:", path)
```

```
Safe path to the Lost Treasure: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3),
(2, 4), (3, 4), (4, 4)]
```

#Problem Sets

1.2 Problem 1: Dive Deep with Depth-First Search (DFS)

You need to explore a dangerous cave network. Use DFS to help you dive deep into the cave and find the Lost Treasure!

1.2.1 Instructions:

- Complete the DFS function to search for the treasure.
- The function should return the path from the start to the goal.

[ ]:
```
def dfs(start, goal, grid):
    stack = [start]
    visited = set()
    parent = {}

    visited.add(start)
    parent[start] = None

    while stack:
        current = stack.pop()

        if current == goal:
            return reconstruct_path(parent, start, goal)

        for neighbor in get_neighbors(current, grid):
            if neighbor not in visited:
                parent[neighbor] = current
                stack.append(neighbor)
                visited.add(neighbor)

    return None  # If no path is found

# Test DFS to find the Lost Treasure
path = dfs(start, goal, eldoria_grid)
print("DFS Path to the Lost Treasure:", path)
```

```
DFS Path to the Lost Treasure: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4),
(2, 4), (2, 3), (2, 2), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3),
(4, 4)]
```

1.3 Problem 2: Lead with Caution using Best-First Search

The path to the treasure is filled with traps! Implement Best-First Search, where you only consider the heuristic to avoid danger.

1.3.1 Instructions:

- Complete the Best-First Search function using a heuristic.
- The function should return the safest path to the treasure.

[ ]:
```python
def best_first_search(start, goal, grid, heuristic):
    priority_queue = []
    heapq.heappush(priority_queue, (0, start))
    parent = {start: None}
    visited = set()
    visited.add(start)

    while priority_queue:
        _, current = heapq.heappop(priority_queue)

        if current == goal:
            return reconstruct_path(parent, start, goal)

        for neighbor in get_neighbors(current, grid):
            if neighbor not in visited:
                parent[neighbor] = current
                visited.add(neighbor)

                # Because I didn't realize we were GIVEN a heur function ...
                # xx, yy = neighbor
                # pos = eldoria_grid[xx][yy]
                # if pos == 'C': weight = 0   #cave is best
                # elif pos == 'P': weight = 1 #paths are safe
                # elif pos == 'D': weight = 2 #dragons are less safe
                # else: weight = 8 #mountains are impossible
                # ...........................................................

                weight = heuristic(neighbor, goal)
                heapq.heappush(priority_queue, (weight, neighbor))

    return None  # If no path is found

# Test Best-First Search to find a safe path to the Lost Treasure
path = best_first_search(start, goal, eldoria_grid, heuristic)
print("Best-First Search Path to the Lost Treasure:", path)
```

```
Best-First Search Path to the Lost Treasure: [(0, 0), (0, 1), (0, 2), (0, 3),
(0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]
```

1.4 Problem 3: Compare Breadth-First Search and Depth-First Search

Sometimes, you need to weigh the benefits of different strategies. Compare the paths found by BFS and DFS to see which one leads you to the treasure faster.

1.4.1 Instructions:

- Implement a function that compares the paths and their costs.
- Print the paths and costs for BFS and DFS.

[ ]:
```
def compare_bfs_dfs(start, goal, grid):
    bfs_path = bfs(start, goal, grid)
    dfs_path = dfs(start, goal, grid)

    print(f"BFS Path: {bfs_path}")
    print(f"DFS Path: {dfs_path}", end="\n\n")

    # assuming cost equals number of nodes on path
    print(f"BFS Path Cost: {len(bfs_path)}")
    print(f"DFS Path Cost: {len(dfs_path)}")

# Compare BFS and DFS paths and costs
compare_bfs_dfs(start, goal, eldoria_grid)
```

```
BFS Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (4, 3), (4,
4)]
DFS Path: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (2, 3), (2,
2), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]

BFS Path Cost: 9
DFS Path Cost: 17
```

1.5 Problem 4: Improve A* with Euclidean Distance

The landscape of Eldoria is vast. Modify A* to use Euclidean distance as the heuristic, making your path more direct and efficient.

1.5.1 Instructions:

- Implement the Euclidean distance heuristic.
- Use it in the A* search function to find the treasure.

[ ]:
```
import math

def euclidean_heuristic(node, goal):
    x0,y0 = node
    x1,y1 = goal
    def sq(a): return a*a
    return math.sqrt(sq(x1-x0) + sq(y1-y0))

# Test A* with Euclidean heuristic to find the Lost Treasure
path = a_star(start, goal, eldoria_grid, euclidean_heuristic)
print("A* Path with Euclidean Heuristic to the Lost Treasure:", path)
```

```
A* Path with Euclidean Heuristic to the Lost Treasure: [(0, 0), (0, 1), (0, 2),
(1, 2), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4)]
```

Answers

1.6 Problem 1 Solution: Depth-First Search (DFS)

DFS dives deep into the cave network first, exploring as far as possible before backtracking. Here's the completed implementation.

```
[ ]: def dfs(start, goal, grid):
    stack = [start]
    visited = set()
    parent = {}

    visited.add(start)
    parent[start] = None

    while stack:
        current = stack.pop()

        if current == goal:
            return reconstruct_path(parent, start, goal)

        for neighbor in get_neighbors(current, grid):
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = current
                stack.append(neighbor)

    return None  # If no path is found

# Test DFS to find the Lost Treasure
path = dfs((0, 0), (4, 4), eldoria_grid)
print("DFS Path to the Lost Treasure:", path)
```

```
DFS Path to the Lost Treasure: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4),
(2, 4), (2, 3), (2, 2), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3),
(4, 4)]
```

1.7 Problem 2 Solution: Best-First Search

Best-First Search uses the heuristic alone to guide the search, trying to lead you directly to the treasure.

[ ]:
```python
def best_first_search(start, goal, grid, heuristic):
    priority_queue = []
    heapq.heappush(priority_queue, (0, start))
    parent = {start: None}
    visited = set()
    visited.add(start) # otherwise reconstruct_path gets stuck in a loop :)

    while priority_queue:
        _, current = heapq.heappop(priority_queue)

        if current == goal:
            return reconstruct_path(parent, start, goal)

        for neighbor in get_neighbors(current, grid):
            if neighbor not in visited:
                visited.add(neighbor)
                f_cost = heuristic(neighbor, goal)
                heapq.heappush(priority_queue, (f_cost, neighbor))
                parent[neighbor] = current

    return None  # If no path is found

# Test Best-First Search to find a safe path to the Lost Treasure
path = best_first_search((0, 0), (4, 4), eldoria_grid, heuristic)
print("Best-First Search Path to the Lost Treasure:", path)
```

```
Best-First Search Path to the Lost Treasure: [(0, 0), (0, 1), (0, 2), (0, 3),
(0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]
```

1.8 Problem 3 Solution: Compare BFS and DFS

This solution compares the paths and costs between BFS and DFS. BFS typically finds the shortest path in terms of the number of steps, while DFS might find a path quicker but not necessarily the shortest.

[ ]:
```python
def compare_bfs_dfs(start, goal, grid):
    bfs_path = bfs(start, goal, grid)
    dfs_path = dfs(start, goal, grid)

    print("BFS Path:", bfs_path)
    print("DFS Path:", dfs_path)

    print("BFS Path Cost:", len(bfs_path) - 1)
    print("DFS Path Cost:", len(dfs_path) - 1)
```

```
# Compare BFS and DFS paths and costs
compare_bfs_dfs((0, 0), (4, 4), eldoria_grid)
```

```
BFS Path: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (4, 3), (4,
4)]
DFS Path: [(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (2, 3), (2,
2), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]
BFS Path Cost: 8
DFS Path Cost: 16
```

1.9 Problem 4 Solution: A* with Euclidean Distance

A* with Euclidean distance tends to produce more direct paths, especially in open spaces like Eldoria.

[ ]:
```
import math

def euclidean_heuristic(node, goal):
    return math.sqrt((node[0] - goal[0])**2 + (node[1] - goal[1])**2)

# Test A* with Euclidean heuristic to find the Lost Treasure
path = a_star((0, 0), (4, 4), eldoria_grid, euclidean_heuristic)
print("A* Path with Euclidean Heuristic to the Lost Treasure:", path)
```

```
A* Path with Euclidean Heuristic to the Lost Treasure: [(0, 0), (0, 1), (0, 2),
(1, 2), (2, 2), (2, 3), (2, 4), (3, 4), (4, 4)]
```