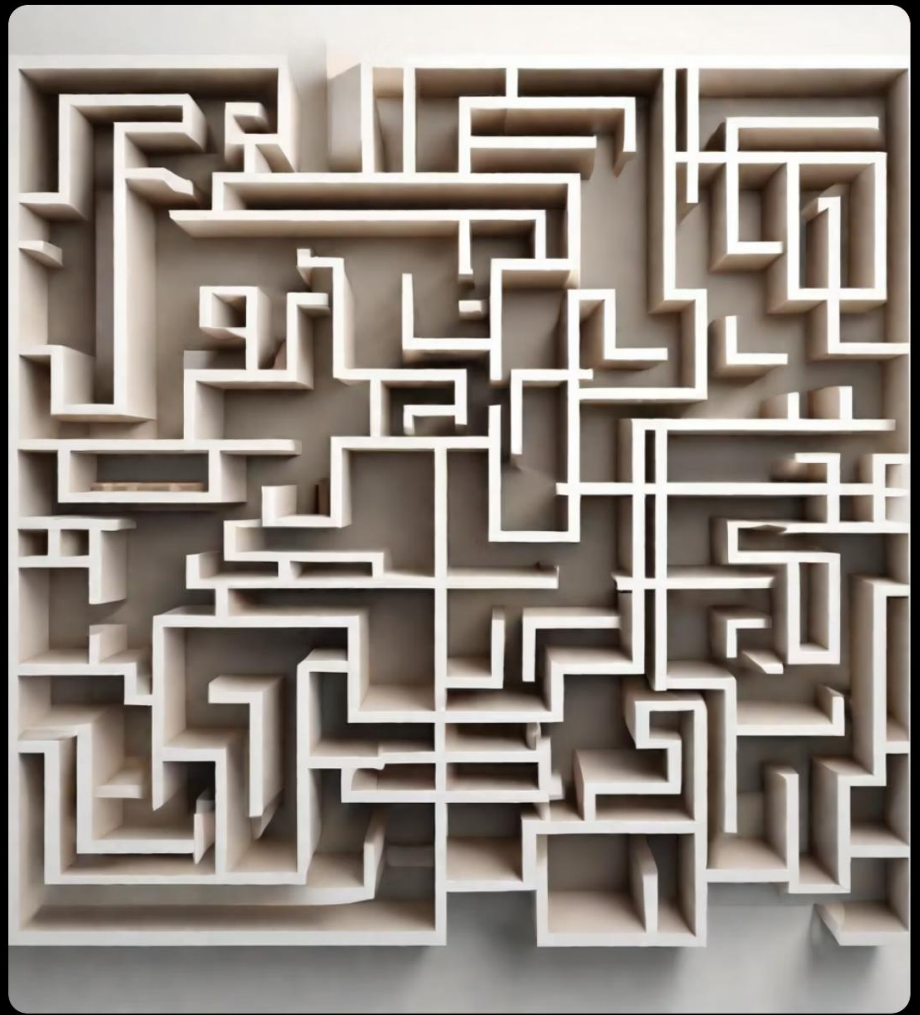
Maze Solving Intelligent Agent



Definition of the Problem

Problem of Maze Solving

The problem of maze solving involves finding a path from a starting point to a goal point in a maze.

A maze is a complex network of interconnected paths and walls, and the goal is to navigate through the maze while avoiding obstacles and reaching the goal in the most efficient way possible.

Significance of Maze Solving

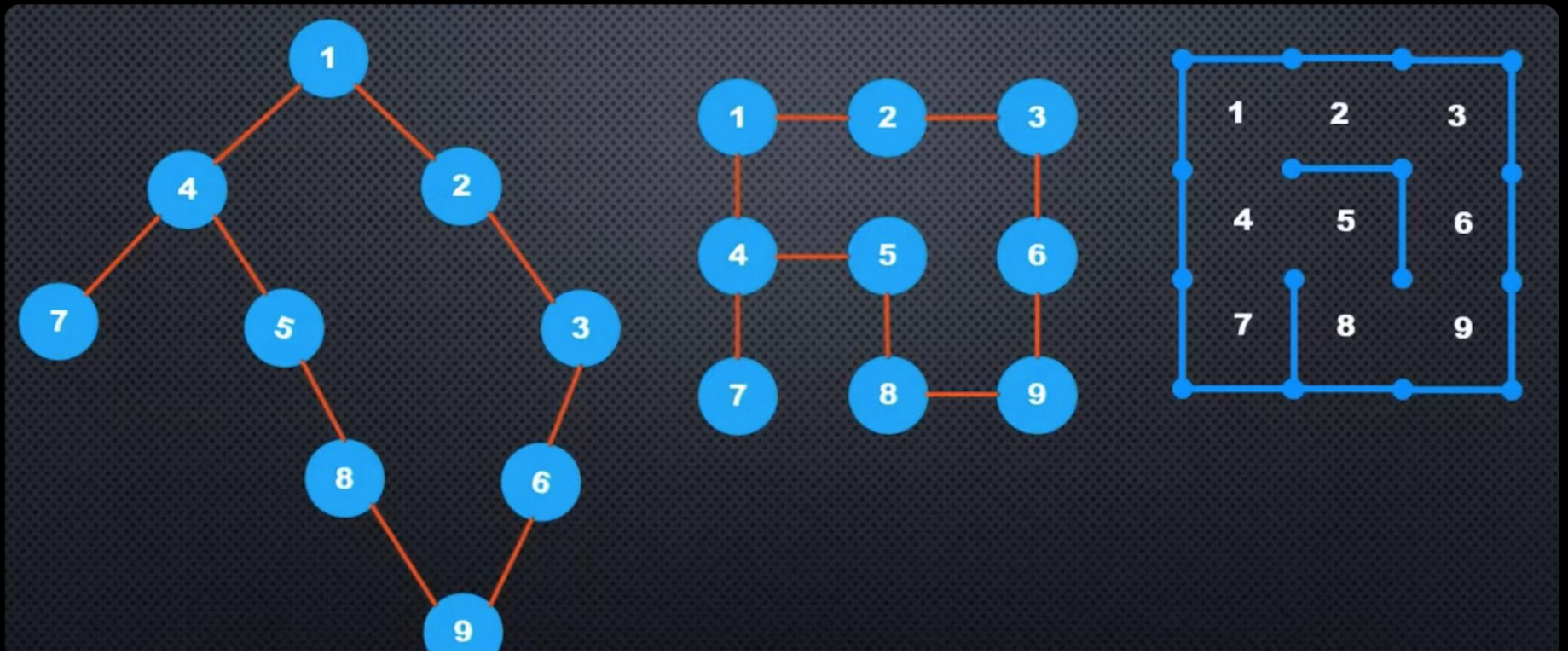
Maze solving is a fundamental problem in the field of artificial intelligence and has various real-world applications.

It is used in robotics for path planning and navigation, in video games for character movement, and in logistics for optimizing routes.

By solving mazes, intelligent agents can demonstrate their ability to make decisions, analyze complex environments, and find optimal solutions.

Why The Problem of Maze Solving is a Searching Problem

The problem of maze solving can be framed as a searching problem because it involves finding a path from a starting point to a goal point in a maze. The maze can be represented as a graph, where each cell in the maze is a node and the connections between adjacent cells are edges. By treating the maze as a graph, we can apply various searching algorithms to find the optimal path.



Solving the Maze Problem Using Searching Algorithms

Dijkstra's Algorithm

- Dijkstra's algorithm is a popular searching algorithm used to find the shortest path between nodes in a graph. It can also be applied to solve the maze problem by treating the maze as a graph.
- In the context of the Dijkstra's algorithms, the shortest path doesn't strictly mean the path with the minimum number of steps, but the path with the *lease cost.*
- The algorithm starts at the entrance of the maze and explores the neighboring nodes, assigning a cost to each cell based on the distance from the entrance. It then selects the node with the lowest cost and continues exploring its neighbors. This process continues until the algorithm reaches the goal node.
- By keeping track of the path with the lowest cost, Dijkstra's algorithm guarantees that the shortest path from the entrance to the exit of the maze is found.

A* Algorithm

- The A* algorithm is another searching algorithm commonly used to solve the maze problem. It is an extension of Dijkstra's algorithm that incorporates *heuristics* to guide the search towards the goal more efficiently.
- Similar to Dijkstra's algorithm, A* starts at the entrance of the maze and explores neighboring cells. However, it also takes into account the estimated distance from each cell to the goal using some heuristic function. This heuristic function helps prioritize the exploration of cells that are more likely to lead to the goal.
- By combining the cost of reaching a cell from the entrance with the estimated distance to the goal, A* algorithm can find the shortest path from the entrance to the exit of the maze more efficiently than Dijkstra's algorithm.

Choice of Algorithms

Dijkstra's Algorithm

Dijkstra's algorithm is a popular choice for solving the maze problem due to its ability to find the shortest path between two points in a weighted graph. In the context of maze solving, each cell in the maze can be represented as a node in the graph, and the edges between adjacent cells have weights representing the distance between them. By applying Dijkstra's algorithm to the maze, we can find the shortest path from the start cell to the goal cell, taking into account the distances between cells.

A* Algorithm

The A* algorithm is another suitable choice for solving the maze problem, especially when we want to prioritize finding the shortest path efficiently. A* algorithm combines the advantages of both Dijkstra's algorithm and a heuristic function that estimates the cost from the current cell to the goal cell. By considering both the actual cost from the start cell to the current cell and the estimated cost from the current cell to the goal cell, A* algorithm can guide the search towards the most promising path, leading to faster and more efficient maze solving.

Dijkstra's Algorithm

Introduction

Dijkstra's algorithm is a graph search algorithm that is used to find the shortest path between two nodes in a weighted graph. It is named after its inventor, Edsger Dijkstra, and is commonly used in various applications such as routing and network optimization.

Algorithm Steps

- Initialize the algorithm by setting the start node as the current node and assigning a distance of 0 to it.
- Mark all other nodes as unvisited and assign a distance of infinity to them.
- While there are unvisited nodes or until the goal is reached, select the node with the smallest distance and mark it as visited.
- For each neighbor of the current node, calculate the distance from the start node through the current node.
- If the calculated distance is smaller than the previously assigned distance, update the distance and set the current node as the previous node for the neighbor.

Implementation

1. Initialize the algorithm by setting the start node as the current node and assigning a distance of 0 to it.
2. Mark all other nodes as unexplored and assign a distance of infinity to them.

```python
def dijkstra(graph, *weights):
    initial_state = (graph.rows, graph.cols)
    goal = (1, 1)
    unexplored = {node: np.inf for node in graph.grid}
    unexplored[initial_state] = 0
    explored = {}
    node_weights = [(i.position, i.cost) for i in weights]
    directions = ['E', 'S', 'N', 'W']

    # searching space
    searching_path = [initial_state]
    reversed_path = {}
    shortest_path = {}
```

3. While there are unexplored nodes or until the goal is reached, select the node with the smallest distance and mark
. it as explored.

```
while len(unexplored) > 0:
    current_node = min(unexplored, key=unexplored.get)
    explored[current_node] = unexplored[current_node]
    searching_path.append(current_node)
    if current_node == goal:
        break
```

4. For each neighbor of the current node, calculate the distance from the start node through the current node.

5. If the calculated distance is smaller than the previously assigned distance, update the distance and set the current node as the previous node for the neighbor and pop the current node from the unexplored list.

```python
for direction in directions:
    if graph.maze_map[current_node][direction] == 1:
        if direction == 'E':
            next_node = (current_node[0], current_node[1]+1)
        elif direction == 'N':
            next_node = (current_node[0]-1, current_node[1])
        elif direction == 'S':
            next_node = (current_node[0]+1, current_node[1])
        elif direction == 'W':
            next_node = (current_node[0], current_node[1]-1)

        if next_node in explored:
            continue

        next_node_cost = unexplored[current_node] + 1

        for W in node_weights:
            if W[0] == current_node:
                next_node_cost += W[1]

        if next_node_cost < unexplored[next_node]:
            unexplored[next_node] = next_node_cost
            reversed_path[next_node] = current_node

unexplored.pop(current_node)
```

Pros and Cons of Dijkstra's Algorithm

- Dijkstra's algorithm is a popular and widely used algorithm for solving mazes due to its simplicity and effectiveness.
- One of the main advantages of Dijkstra's algorithm is that it guarantees finding the shortest path in a maze, as it explores all possible paths.
- Dijkstra's algorithm can be easily implemented and understood, making it accessible for beginners and those new to maze solving algorithms.
- The algorithm is versatile and can be applied to different types of mazes, including both weighted and unweighted mazes.
- Dijkstra's algorithm is deterministic, meaning it will always produce the same output given the same input, making it reliable and predictable.
- However, one of the main drawbacks of Dijkstra's algorithm is its inefficiency when dealing with large mazes or mazes with many obstacles.
- Another disadvantage of Dijkstra's algorithm is that it explores all possible paths, which can be time-consuming and computationally expensive in complex mazes.

A* Algorithm

Overview

The A* algorithm is a popular pathfinding algorithm used in computer science and artificial intelligence. It is an extension of Dijkstra's algorithm and is particularly effective in solving maze problems. A* stands for 'A star' and it is named after the star symbol used to represent the algorithm in literature.

How It Works

The A* algorithm uses a combination of two metrics to determine the best path: the actual cost of reaching a node (known as 'g') and the estimated cost of reaching the goal from that node (known as 'h'). The algorithm maintains a priority queue of nodes to explore, with the lowest total cost (g + h) being the highest priority. It iteratively selects the node with the lowest total cost and explores its neighbors, updating their costs and predecessors as necessary.

- **Implementation**

```python
def a_star_m(maze, start: tuple=None) -> dict:
    maze_nodes = maze.grid
    initial_state = (maze.rows, maze.cols)
    goal = (1, 1)
    search_path = [initial_state]
    reverse_path = {}
    path = {}
    directions = ['E', 'S', 'N', 'W']
    g_values = {node: np.inf for node in maze_nodes}
    g_values[initial_state] = 0
    h_values = {node: np.inf for node in maze_nodes}
    h_values[initial_state] = h_m(initial_state, goal)
    f_values = {node: np.inf for node in maze_nodes}
    f_values[initial_state] = g_values[initial_state] + h_m(initial_state, goal)
    fringe = PriorityQueue()
    fringe.put((f_values[initial_state], h_values[initial_state], initial_state))

    while not (fringe.empty()):
        current_node = fringe.get()[2]
        search_path.append(current_node)
        if current_node == goal:
            break

```

```
        for direction in directions:
            if maze.maze_map[current_node][direction] == 1:
                if direction == 'E':
                    child_node = (current_node[0], current_node[1]+1)
                elif direction == 'S':
                    child_node = (current_node[0]+1, current_node[1])
                elif direction == 'N':
                    child_node = (current_node[0]-1, current_node[1])
                elif direction == 'W':
                    child_node = (current_node[0], current_node[1]-1)

                new_g_value = g_values[current_node] + 1
                new_f_value = new_g_value + h_m(child_node, goal)

                if new_f_value < f_values[child_node]:
                    g_values[child_node] = new_g_value
                    f_values[child_node] = new_f_value
                    fringe.put((f_values[child_node], h_m(child_node, goal), child_node))

                    reverse_path[child_node] = current_node

    trace_node = goal
    while trace_node != initial_state:
        path[reverse_path[trace_node]] = trace_node
        trace_node = reverse_path[trace_node]

    return search_path, reverse_path, path
```

Pros and Cons of A* Algorithm

The A* algorithm is a popular choice for solving mazes due to its efficiency and ability to find optimal paths. However, it also has some limitations that should be considered.

Pros

- Efficiency: The A* algorithm is generally more efficient than other search algorithms, such as breadth-first search or depth-first search. It uses heuristics to prioritize nodes and explore the most promising paths first, leading to faster solutions.
- Optimality: A* guarantees finding the optimal path if certain conditions are met. It uses a combination of the actual cost from the start node and the estimated cost to the goal node, ensuring the shortest path is found.
- Flexibility: A* can be customized by choosing different heuristic functions to guide the search. This allows for adaptability to different maze layouts and problem domains.

Cons

- Heuristic Accuracy: The effectiveness of the A* algorithm heavily relies on the accuracy of the heuristic function. If the heuristic function is not well-designed or doesn't accurately estimate the cost to the goal, the algorithm may not find the optimal path.
- Memory Usage: A* algorithm requires storing and updating information for each visited node, which can consume significant memory resources, especially for large and complex mazes.
- Path Selection: A* may not always find the shortest path if there are multiple paths with the same cost. The algorithm may favor one path over the other, leading to suboptimal solutions.

Why A* Algorithm over Dijkstra's

When it comes to solving the maze problem, the choice between A* algorithm and Dijkstra's algorithm depends on the specific requirements and constraints of the problem. While both algorithms are effective in finding the shortest path in a graph, A* algorithm offers several advantages over Dijkstra's algorithm.

Comparison of A* Algorithm and Dijkstra's Algorithm

Criteria	A* Algorithm	Dijkstra's Algorithm
Heuristic Function	Uses a heuristic function to guide the search towards the goal node, resulting in faster convergence.	Does not use a heuristic function, leading to a more exhaustive search of the entire graph.
Time Complexity	Generally faster than Dijkstra's algorithm due to the use of heuristics.	Slower than A* algorithm as it explores all possible paths in the graph.
Memory Usage	Requires more memory due to the need to store additional information for the heuristic function.	Requires less memory compared to A* algorithm.
Optimality	Provides an optimal solution if the heuristic function is admissible and consistent.	Provides an optimal solution.
Applications	Well-suited for pathfinding problems where the goal is known and the graph is large.	Suitable for general shortest path problems without specific constraints.

Conclusion

Key Points

- The AI maze solving intelligent agent uses Dijkstra's algorithm and A* algorithm to solve the maze.
- Dijkstra's algorithm guarantees finding the shortest path between two points in the maze.
- A* algorithm improves upon Dijkstra's algorithm by using heuristics to prioritize paths and find the optimal solution more efficiently.

Benefits of A* Algorithm

- Faster solution finding compared to Dijkstra's algorithm.
- Optimal path finding by using heuristics to guide the search.
- Efficiently handles large and complex mazes.