CS6053 Artificial Intelligence

Assignment: Problem Solving by State Space Search

Zana Simanel 20045259

30 04 2024

Table Of Contents

1. Introduction

The maze challenge involves navigating from a start point to an endpoint in a grid-based labyrinth, with each grid cell representing a node. The objective is to find the shortest route through the maze, which requires overcoming various barriers and constraints.

To achieve this, the A* algorithm is utilised for its efficiency in pathfinding. It combines shortest-path principles with a heuristic that prioritises closer proximity to the goal, ensuring optimal navigation.

The implementation leverages the matplotlib library for visualising the maze and the algorithm's progress. Matplotlib is used for plotting and provides tools for drawing maze cells, enhancing the visualisation of the algorithm's path and node exploration. This setup not only demonstrates the A* algorithm's capabilities but also provides an interactive graphical representation of the maze navigation process.

1.1 Description Of the Maze

A maze is a complex and intricate puzzle composed of interconnected paths, corridors, and walls. It is designed to challenge and confuse individuals trying to navigate through it. The walls of the maze create dead ends and force individuals to find the correct path to reach the desired destination.

The paths within a maze can twist, turn, and intersect in unpredictable ways. Some paths may lead to dead ends, requiring individuals to backtrack and find alternative routes. The goal of navigating a maze is often to reach a specific endpoint or find a hidden treasure within its confines. They require individuals to analyse patterns, make strategic decisions, and use trial and error to find the correct path. Maze provides a unique and engaging experience that captivates individuals of all ages and encourages them to think creatively and critically.

1.2 Problem Definition

The task for this assignment includes finding the shortest path through a maze represented as a 5x6 grid. The initial point is positioned at the top-left corner of the grid, which means position (0,0) and the goal is to reach the end node (4,5) that is positioned at the bottom-right corner. The grid includes passable and impassable cells, the impassable cells are marked in red to show the obstacles. The grid is walled, which makes it impossible to make any move outside the grid's build.

The movement of the grid is restricted to four directions: up, down, left, and right. Each action allows the navigator to move to a different cell if it is not blocked by the wall or an obstacle. The main test of this task is to go through the start to an end node by making tactical movements that will avoid all the obstacles. So, the tasks involve both, the fastest route and avoiding boundaries and obstacles. The procedure must justify the complexity of the maze design, which will need to make sure that there is an intellectual investigation of possible pathways and the ability to decide which one will be the most efficient. The problem of this task is not only the ability to find working algorithms but also makes it effectively create a maze setting and constraints in a way that helps in the successful navigation of the maze.

1.3 Key Objectives

The main goal and objective of the project is to create a working solution to create a way through a maze that is created in the 5x6 grid by using algorithms. The algorithm must have an efficient

approach to solve the problem and find the quickest paths with the lowest cost to move from the starting point (0,0) to the endpoint (4,5) and be aware of the obstacles and the frame of the maze.

The key objectives are:

1. Algorithm development: creating a design and developing a state space searching algorithm that will move around the possible ways (up, down, left, and right) within the given environment. This includes creating an A* algorithm, adapting to maze features such as boundaries, and creating the code in Python language.
2. Integration of the Algorithm: The integration of the algorithm should be with Python data structures and handle the following grid (5,6), pathway tracking of the search, and node evaluation.
3. Problem-solving formula: Using problem-solving skills to solve the problems and challenges created by the maze. This will include establishing how to deal with the obstacles and create better solutions to reach the goal by effective management.
4. Proper Testing of Validation of the Algorithm: Detailed testing to make sure that the algorithm works as expected thoroughly different scenarios and possibilities within the maze, guarantee that it will always find the most efficient path and operate within the success criteria.
5. Documentation and Reporting: Creating documentation explaining all the processes is crucial when it comes to the problem-solving procedure and the algorithm mechanics, integration with Python, and its presentation. The report will include a clear view of the design of the algorithm, creating a deep understanding of the techniques.
6. The objective includes the guides in the creation of the algorithm and also helps to transfer the theoretical knowledge into the practical, helping to understand artificial intelligence.

1.4 Success Criteria

To represent the maze nodes, success criteria conditions must be met by choosing accurate structure capture of the maze. The representation should facilitate easy access to nodes, handle obstacles effectively, and support pathfinding algorithms without unnecessary complexity.

1. Correctness of Pathfinding: The algorithm must correctly identify a path from the starting node (0, 0) to the end node (4, 5) if one exists. The identified path must not pass through any obstacles (represented by 1s in the grid) and must adhere to the movement rules.
2. Optimality of Path: The path found by the A* algorithm should be the shortest possible path, taking into consideration the layout of the maze and the location of obstacles.
3. Efficiency of the Heuristic: The heuristic function should never overestimate the distance to the goal; it should provide a cost estimate that assists the A* algorithm in finding the shortest path quickly. The number of nodes evaluated by the algorithm should be minimized, indicating that the heuristic is guiding the search efficiently.
4. Performance: The algorithm should complete the search within a reasonable time frame, demonstrating good performance even as the complexity of the maze increases. The implementation should be able to handle larger mazes with the same efficiency, assuming adequate computational resources.

5. Robustness: The pathfinding process must be able to handle different maze configurations without requiring modifications to the algorithm. The algorithm should be able to return a result (or a failure if no path exists) regardless of the maze's complexity.
6. Visualization: The visualization should accurately represent the state of the maze at each step, including the exploration process, the final path, and any heuristic cost annotations. Changes to the maze or the path should be reflected immediately in the visualization, providing clear feedback on the algorithm's process.

2. Primary Task

2.1 Maze representation

To begin the implementation of maze-solving algorithms in Python, we first need to establish a suitable representation for mazes. We will adopt a grid-based representation, where each cell in the maze is represented as a node in a graph. The maze structure will be stored as a 2D array, with walls denoted by obstacles and open pathways denoted by empty cells. This representation will enable us to apply graph traversal algorithms for navigating through the maze.

To solve the maze problem using state space search, we can represent the maze nodes using a two dimensional list in Python. Each element in the list represents a node in the maze.

In Figure 1 representation, 0 represents an empty node where movement is allowed, and 1 represents an obstacle node where movement is blocked. Start Point: Denoted by the coordinates (0, 0), the start points marks the initial position within the maze grid from where the traversal begins. End point is in the coordinates (4, 5), the end point marks the destination within the maze grid, representing the goal to be reached. Movement are allowed horizontally and vertically within the maze, indicating that traversal can occur along rows of the grid.

```
maze = [
    [0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0, 0],
    [0, 1, 0, 0, 1, 0],
    [0, 0, 0, 0, 1, 0]
]

start = (0, 0) #row, col
end = (4, 5)   #row, col


# Define possible movements
movements = [(-1, 0), (1, 0), (0, -1), (0, 1)]
```

Figure 1 Maze Representation

2.2 A* Search Algorithm

Figure 2 represents used algorithm in the code example, where A* search algorithm starts by prioritizing the queue and starting the node in the beginning (0,0). The initial node is also added to the" visited nodes" so the node will not visit the same box again. In the next step, it successfully deleted the node with the highest priority from the queue for processing. The node's placement is compared to the goal destination of the maze, and if they match, then the Algorithm successfully finds the most efficient pathway and returns the path with the list of the nodes that have been examined.

With each node that has been processed, the algorithm checks all the potential moves that could be up, down, left or right. For each movement, it searches for the consequential position and checks if its new position is in the maze, not blocked by any obstacle or was not visited before. If all the previous requirements are met, the technique creates a new pathway that includes this movement and analyses

all the costs to reach the current location. The sum includes the way from the starting node (0,0) to the destination (4,5) and the cost provided by the heuristic method. (Stuart Russel, 2020, p. 89)

The heuristic function is important for the efficiency of the A* search algorithm, it shows the distance left to the target, creating a guideline towards the most efficient pathways. The paths that require lower costs of the movements are prioritized in the created queue and the process does not stop until the path is found, or the queue is finished and has no more options. If such a situation happens, we will get a 'None' return with the list of the movements that were checked. This approach makes sure that the algorithm finds the best way by intellectually finding the quickest ways to find the target.

```
# A* algorithm to find the shortest path
def astar(maze, start, end):
    print("Running A* algorithm...")
    queue = PriorityQueue()
    queue.put((0, start, [start]))
    visited = set([start])
    attempts = []  # To track all the nodes that the algorithm examines

    while not queue.empty():
        _, position, path = queue.get()
        attempts.append(position)  # Record the node as an attempt

        if position == end:
            return path, attempts

        for move in movements:
            next_pos = (position[0] + move[0], position[1] + move[1])

            if (0 <= next_pos[0] < len(maze)) and (0 <= next_pos[1] < len(maze[0])) and (maze[next_pos[0]][next_pos[1]] == 0) and (next_pos not in visited):
                new_path = path + [next_pos]
                cost = calculate_cost(new_path) + heuristic(next_pos, end)
                queue.put((cost, next_pos, new_path))
                visited.add(next_pos)

    return None, attempts
```

Figure 2 A Search Algorithm*

2.3 Heuristic Function

Figure 3 depicts implemented heuristic function to A* algorithm. The heuristic function is commonly used in A* and other possible pathfinding algorithms to calculate the cost of movements between the initial node (0,0) to the goal node (4,5). This function uses the Manhattan distance as its system of measurement by calculating the sum of the differences between x-coordinates which are rows, and y-coordinates which are columns. This method estimates how many steps down, right, top, or right to get from the first position to the end position without moving.

After the calculation, the value is an immediate output and creates an understanding of the function's path at each step of the way, which is crucial to successfully debug or trace the algorithm's implementation. The method also shows the graphic component by the "highlight_react" call, which will create a visual indication of the value of the maze. The conception helps to understand how the A* algorithm will evaluate and prioritize nodes through the cost of aiming the last stop, this reduces the exploration zone and enlightens performance while finding the quickest pathway.

In return, the shortest distance is returned, showing the crucial component that A* algorithm decision-making development. By using the heuristic function with the pathway cost, the algorithm can successfully prioritize ways that are shorter and more effective to improve the performance of the task. (Stuart Russel, 2020, p. 91)

```
def heuristic(position, end):
    distance = abs(position[0] - end[0]) + abs(position[1] - end[1])
    print("Heuristic value at position", position, ":", distance)
    #set_rectangle_label(position[1], position[0], str(distance))
    highlight_rect(position[1], position[0], str(distance))
    return distance
```

Figure 3 Heuristic Function

2.4 Cost Calculation

The calculate cost function computes the total cost associated with traversing the path in the maze, representing the number of steps taken from the start to the end point. In input the function accepts a path, provided as a list of node coordinates from the start to the end point.

The total cost f(n) for each node n in the A* algorithm is indeed the sum of two key components:

g(n): The cost of the path from the start node to the current node n. This is known as the path cost or "actual" cost and is calculated by adding up all the costs incurred from the beginning to the current node n.

h(n): The heuristic estimate of the cost from node n to the goal. The heuristic is a way to inform the algorithm about the direction to the goal without knowing the exact path. It's like an educated guess of the remaining cost to reach the end. (Games, Introduction to the A* Algorithm, 2014)

By output It returns an integer value representing the total cost of the path. If a valid path exists, the cost corresponds to the length of the path (i.e., the number of steps). Conversely, if no valid path is found, it returns -1. In method function calculates the cost by determining the length of the path. This length reflects the number of steps taken from the start to the end point. If the path is deemed invalid (None), the function returns -1. Otherwise, it returns the length of the path as the cost. In summary, the calculate cost function offers a straightforward means to quantify the efficiency of a path in terms of step count within the maze that can be seen in Figure 4.

```
# Define cost calculation function
def calculate_cost(path):
    # In this simple maze problem, each step has the same cost (1)
    return len(path) if (path != None) else -1
```

Figure 4 Cost Calculation Function

2.5 Draw Path Function

The draw path function code example is represented in Figure 5 and serves to visually depict the shortest path derived from the A* algorithm on the maze plot, . Its primary objective is to highlight this path, given as a list of node coordinates from the start to the end point. By iterating through the path, it systematically adjusts the face color of each corresponding rectangle patch on the plot to blue, indicating its inclusion in the shortest path. Through this process, the function enhances user comprehension by providing a clear and animated representation of the solution. (Pathfinding, n.d.)

```
def draw_path(path):
    print("Drawing path...")
    if path:
        for node in path:
            ax.patches[ get_patches_index(node[1],node[0]) ].set_facecolor('blue')
            plt.pause(.3)
```

Figure 5 Draw Path Function

3. Output

1. Heuristic Computation: The output begins with the A* algorithm's execution, which involves computing heuristic values for various positions within the maze displayed in Table 1. Each heuristic value indicates the estimated distance from the respective position to the goal (end) position. The positions are highlighted on the maze plot, and their corresponding heuristic values are annotated.

Heuristic value at position (1, 0) : 8
Highlighting rectangle (0, 1) with label '8'
Setting label for rectangle (0, 1) to '8'
Heuristic value at position (2, 0) : 7
Highlighting rectangle (0, 2) with label '7'
Setting label for rectangle (0, 2) to '7'
Heuristic value at position (1, 1) : 7
Highlighting rectangle (1, 1) with label '7'
Setting label for rectangle (1, 1) to '7'
Heuristic value at position (1, 2) : 6
Highlighting rectangle (2, 1) with label '6'
Setting label for rectangle (2, 1) to '6'
Heuristic value at position (0, 2) : 7
Highlighting rectangle (2, 0) with label '7'
Setting label for rectangle (2, 0) to '7'
Heuristic value at position (2, 2) : 5
Highlighting rectangle (2, 2) with label '5'
Setting label for rectangle (2, 2) to '5'
Heuristic value at position (1, 3) : 5
Highlighting rectangle (3, 1) with label '5'
Setting label for rectangle (3, 1) to '5'
Heuristic value at position (0, 3) : 6
Highlighting rectangle (3, 0) with label '6'
Setting label for rectangle (3, 0) to '6'
Heuristic value at position (1, 4) : 4
Highlighting rectangle (4, 1) with label '4'
Setting label for rectangle (4, 1) to '4'
Heuristic value at position (0, 4) : 5
Highlighting rectangle (4, 0) with label '5'
Setting label for rectangle (4, 0) to '5'
Heuristic value at position (2, 4) : 3
Highlighting rectangle (4, 2) with label '3'
Setting label for rectangle (4, 2) to '3'
Heuristic value at position (1, 5) : 3
Highlighting rectangle (5, 1) with label '3'
Setting label for rectangle (5, 1) to '3'
Heuristic value at position (0, 5) : 4
Highlighting rectangle (5, 0) with label '4'
Setting label for rectangle (5, 0) to '4'
Heuristic value at position (2, 5) : 2
Highlighting rectangle (5, 2) with label '2'
Setting label for rectangle (5, 2) to '2'
Heuristic value at position (3, 0) : 6
Highlighting rectangle (0, 3) with label '6'
Setting label for rectangle (0, 3) to '6'
Heuristic value at position (3, 2) : 4
Highlighting rectangle (2, 3) with label '4'
Setting label for rectangle (2, 3) to '4'
Heuristic value at position (3, 5) : 1
Highlighting rectangle (5, 3) with label '1'
Setting label for rectangle (5, 3) to '1'
Heuristic value at position (4, 0) : 5
Highlighting rectangle (0, 4) with label '5'
Setting label for rectangle (0, 4) to '5'
Heuristic value at position (4, 2) : 3
Highlighting rectangle (2, 4) with label '3'
Setting label for rectangle (2, 4) to '3'
Heuristic value at position (3, 3) : 3
Highlighting rectangle (3, 3) with label '3'
Setting label for rectangle (3, 3) to '3'
Heuristic value at position (4, 3) : 2
Highlighting rectangle (3, 4) with label '2'
Setting label for rectangle (3, 4) to '2'
Heuristic value at position (4, 5) : 0
Highlighting rectangle (5, 4) with label '0'
Setting label for rectangle (5, 4) to '0'
Heuristic value at position (4, 1) : 4
Highlighting rectangle (1, 4) with label '4'
Setting label for rectangle (1, 4) to '4'

Table 1 Heuristic Computation Output

2. The title of the maze plot is updated to include the cost of the shortest path. The cost is computed using the calculate cost function and appended to the existing title.

>>> ax.set_title(ax_title_text)

Text(0.5, 1.0, 'Maze Solver\nCost of the shortest path:10')

3. Shortest Path Visualization: After the A* algorithm completes its execution represented in Figure 6, the shortest path found by the algorithm is visualized on the maze plot. The draw path function highlights the nodes along the shortest path by adjusting the color of the corresponding rectangles to blue.

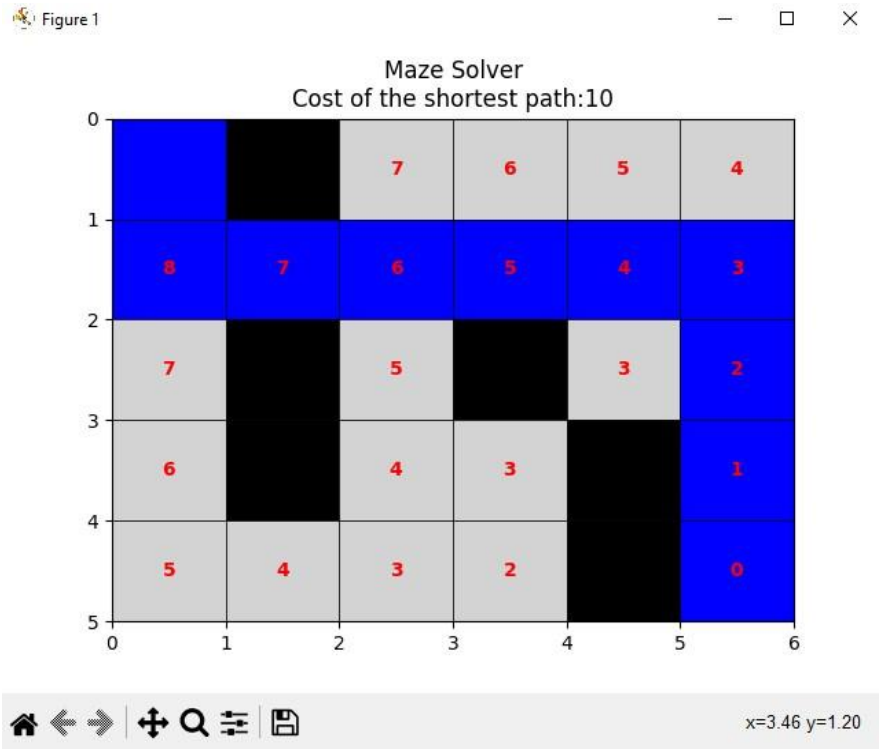

Figure 6 Maze Visualization Output

4. The coordinates of the nodes in the shortest path are printed to the console, providing a detailed representation of the path.

Shortest path: [(0, 0), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 5), (3, 5), (4, 5)]

5. Total Attempts: Lastly, the total number of attempts made by the A* algorithm during its execution is printed. Each attempt represents a position evaluated by the algorithm in its search for the shortest path.

   >>> print ("Total attempts:", len(attempts))

   Total attempts: 20

4. Conclusion

Our project's journey through algorithm development, integration, and problem-solving has been enlightening and rewarding. We successfully designed and implemented the A* algorithm within a Python environment, addressing the intricacies of maze navigation. The algorithm navigating the 5x6 grid by efficiently determining the shortest path while overcoming hurdles represented by obstacles and calculates heuristic values to guide our pathfinding, ensuring that the algorithm remained both accurate and optimal. We adopted the Manhattan distance, a heuristic befitting our grid-based maze, to estimate the cost of reaching the goal from any given node. This choice was pivotal, allowing for swift and effective traversal while minimizing computational overhead.

Integration of the algorithm with Python data structures was a dynamic and responsive solution. The draw_path function provided an essential visual aid, mapping out the algorithm's decisions time, and offering a clear visual validation of the pathfinding process. We documented every step, resulting in a comprehensive report that not only details the algorithm's design and integration but also provides insights into the underlying mechanics and decision-making processes. Our group have met our success criteria, demonstrating the algorithm's resilience and reliability.

Furthermore, our project contributed to a deeper understanding of artificial intelligence applications, particularly in heuristic-based search algorithms. It served as a bridge, transferring theoretical knowledge into practical, real-world problem-solving skills.

In summary, this project did not just fulfil its objectives; it paved the way for further exploration and innovation in the realm of intelligent algorithm design. As we reflect on our accomplishments, we are reminded that the journey through the maze is akin to the process of learning—filled with challenges, requiring strategic thinking, and ultimately leading to the rewarding destination of growth and knowledge.

5. References

Games, R. B. (2014, 05 26). *Introduction to the A* Algorithm*. Retrieved from https://www.redblobgames.com/pathfinding/a-star/introduction.html

Pathfinding, A. T. (n.d.). *Heuristics*. Retrieved from https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html

Stuart Russel, P. N. (2020). *Artificial Intelligence A Modern Aproach, Fourth Edition.* Pearson Series in Artifical Intelligence).