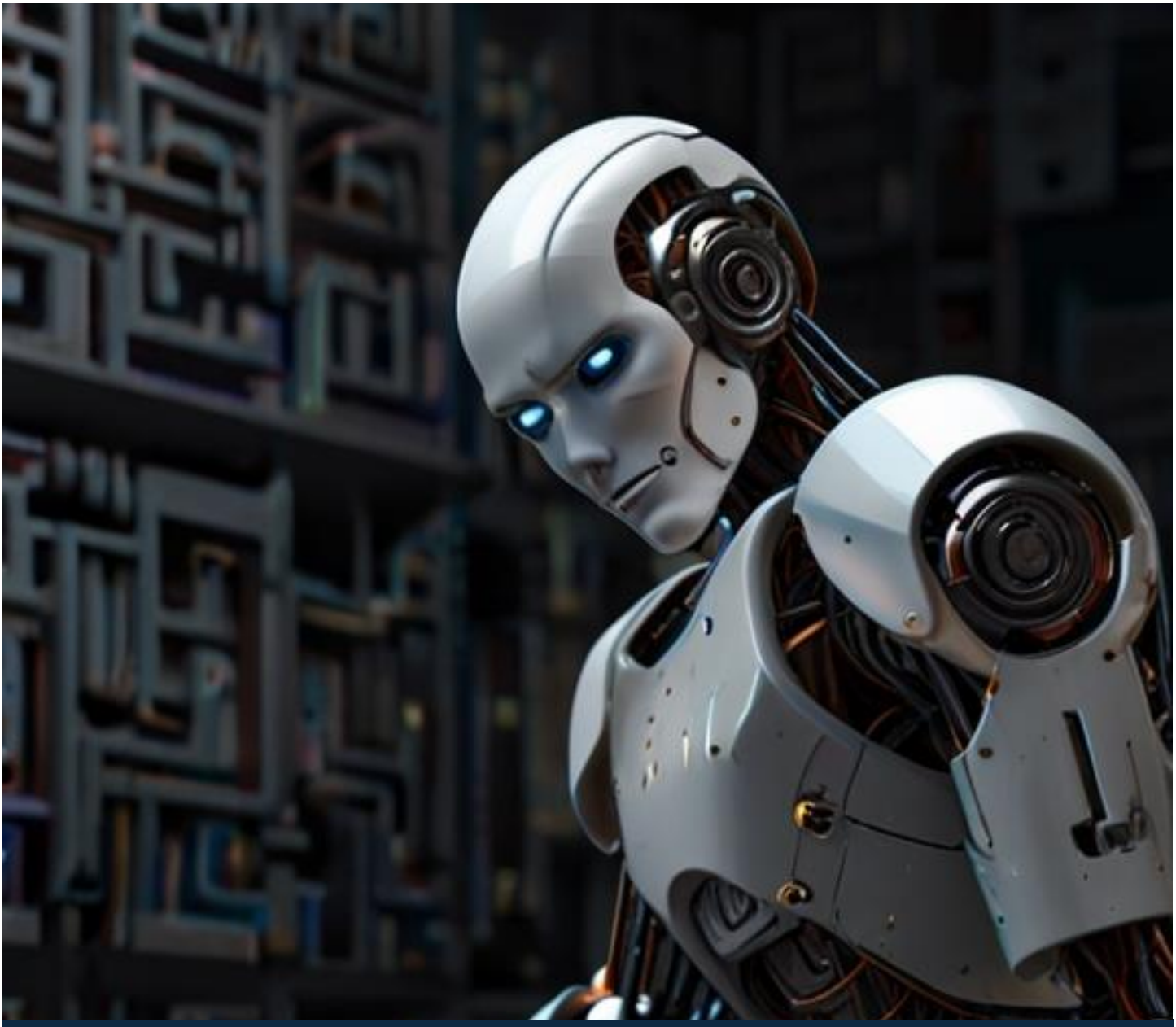


Autonomous Robot Path Planning Program

Documentation

Stephan Dabare

4/10/24

AI and DS

Contents

1. Step-wise Procedure for the Autonomous Robot Path Planning Program.

Implement the grid representation. (Task 01)

- Define a custom grid class with a two-dimensional array of integers.
- Implement methods to check the positions within the bound of the grid and whether it contains an obstacle.

Implement the custom linked list.

- Create a 'Node' class to represent each element.
- Implement methods to check the list is empty, add elements, get the head, and remove the head from the list.

Implement the way to get the robot state. (Task 02)

- Create a custom class to represent the position of the robot on the grid.
- Add a reference to the previous position along with indicators for X and Y coordinates.
- Implement methods to get and set the coordinates and prior position.

Implement the path planning algorithm. (Task 03)

- Use the Breath-First Search algorithm to find the shortest path to the goal. Start the BFS from the starting position of the robot.
- Use the custom linked list to keep track of positions.
- For each position in the linked list, explore its neighboring positions.
- Check if the neighboring positions are within the grid, not obstacles, and not previously visited. (Task 04)
- If the position is valid keep track for further exploration.
- Repeat the exploration process until the list is empty or the goal is found.
- Include a method to reconstruct the path from the goal to the starting position.

Implement a way to test the path planning algorithm. (Task 05)

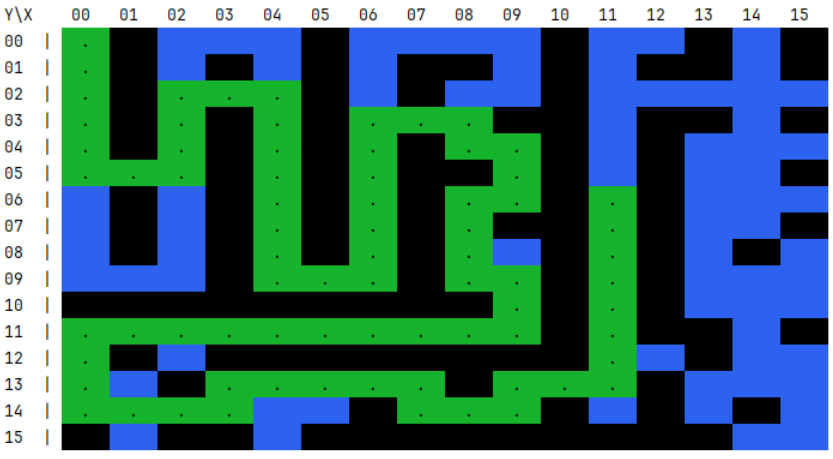
- Implement a method to take user inputs and place the obstacles/goal position in different places inside the grid to test the algorithm.

Create a user interface. (Task 06)

- Create a menu-driven interface that allows the users to either run the program, create a custom grid, or exit.
- Display a grid with the path in green color if the goal position is found, along with the robot's current position.

```
Robot says, "I'm ready to find the path!!"
Please say which option should I need to use
1 <--- Find Path
2 <--- Create a Custom Grid and Find Path
3 <--- Exit
Enter Number: 1
```

```
Enter goal X coordinate: 11

Enter goal Y coordinate: 6

Path found:
```




```
Robot current position:(11,6)
Robot says, "I found the GOAL!!"
. = Path,  = Obstacles,  = Open area
```

2. Justifying the Choice of Data Structures.

Data structures used for task 03 and task 04,

I. Custom linked list

The custom linked list in the path-finding robot program is a singly linked list. In a singly linked list, the sequence is connected linearly, starting from the head node, and ending at the tail node. Each node pointing to the next node. Traversal is done in only one direction.

Why use a singly linked list?

- Required less memory per node because stores only a reference to the next node, not to the previous.

```
3 usages
public void add(RobotPosition data) {  // Add method
    Node newNode = new Node(data);
    if (isEmpty()) {
        head = newNode;
    } else {
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
    }
    size++;
}
```

- Simpler to implement and maintain because have fewer pointers to manage.
- More sufficient for programs that only required forward traversal.

Challenges and strategies for overcoming them.

- Deletion or updating a node from the middle of the list is not directly supported.
  How to overcome → These operations are not involved in a basic path-finding algorithm.

- Revisiting the previous node can be inefficient.
  How to overcome → By storing the previous position while finding the path, allowing efficient backtracking for reconstructing the path. (from goal to starting point)

```
// Get method to get the previous position.
1 usage
public RobotPosition getPrePosition() {
    return prePosition;
}

// Set method to get the previous position.
1 usage
public void setPrePosition(RobotPosition prePosition){
    this.prePosition = prePosition;
}
```

```
nextPosition.setPrePosition(current);  // Set to the previous position for reconstruct the path.
```

II. Two-dimensional Integer Array

The 2D array represents the grid environment with obstacles. Integer 0s represent free spaces and 1s represent obstacles. Each cell in the array corresponds to a node in the linked list.

```
// Initialize 2D integer array called maze, array size depends on user input.
int[][] maze = new int[size][size];
```

III. Two-dimensional Boolean Array

The Boolean 2D array tracks visited positions. The dimensions same as the grid array. If the robot visited a cell, it is stored as true, otherwise false.

```
boolean[][] visited = new boolean[grid.getRows()][grid.getCols()];
// New 2D boolean array with same dimensions of the grid to store visited positions.
```

IV. Positions queue implemented using linked list

Positions need to be visited store in this queue. This structure follows the first-in first-out behavior as a queue. This implementation is done by using the previous custom linked list.

```
// Queue to store positions that need to be visited.
CustomLinkedList positionQueue = new CustomLinkedList();
```

3. Justifications for Algorithms

I. Task 03 Path planning algorithm

Traversal algorithm → Breath-First Search (BFS) Algorithm.

Breath-First Search is a graph traversal algorithm that investigates nodes level by level in a certain order. It focuses on visiting neighbor nodes at the current level.

How it works

1. Start with a queue data structure that is implemented using a linked list to keep track of nodes that need to be traveled.
2. Remove the first element and mark it as visited. This position is going to be explored in the current iteration. Check whether this position is the goal.

```
while (!positionQueue.isEmpty()) {
    // Loop continue until position queue is empty. Until head = null

    RobotPosition current = positionQueue.removeFirst();
    // Mark the current position and remove it, this position will explore in the iteration.

    visited[current.getX()][current.getY()] = true;
    // Mark the current position as visited. Prevent infinite loop.

    // Check if the current position is the goal.
    if (current.getX() == goal.getX() && current.getY() == goal.getY()) {
        return reconstructPath(current);
        // If goal found reconstruct the path from start to goal.
    }
```

3. Explore all the neighboring positions if they are not visited, and in the bound of the grid, and not an obstacle.

```
// Checks if the new position (newX, newY) is within the bounds of the grid, not visited before, and is not an obstacle.
if (grid.isValidPosition(newX, newY) && !visited[newX][newY] && !grid.isObstacle(newX, newY)) {
    RobotPosition nextPosition = new RobotPosition(newX, newY);
    // Create a new position instance with neighboring position.
    nextPosition.setPrePosition(current);  // Set to the previous position for reconstruct the path.
    positionQueue.add(nextPosition);   // Add neighbor to the position queue.
    visited[newX][newY] = true;        // Mark the neighbor as visited avoiding revisiting.
}
```

4. Repeat steps 2 and 3 until the queue is empty or found the goal.

Why use the BFS algorithm

- BFS explores nodes level by level, and it guarantees that it finds the shortest path between the starting position and the goal.
- In an unweighted graph, since all the edges have the same weight, BFS travels without considering edge costs. This ensures a faster traversal process.
- Simplicity of implementation.

Challenges and strategies for overcoming them

BFS travels through all possible paths from the start node to the end node, level by level. This method is not efficient for larger grids. However, we considering a smaller grid, so BFS remains a reasonable choice.

Time complexity analysis

In the worst-case scenario, BFS explores all cells in the grid. Therefore, the time complexity of BFS is O(V + E). V is the number of vertices(cells), and E is the number of edges (connections between cells).

In BFS each cell has up to 4 neighbors, therefore,

$$\text{Number of edges} \propto \text{Number of vertices}$$

Hence, the time complexity simplifies to O(V).

Result → Time complexity is linearly proportional to the number of cells in the grid.

Asymptotic analysis

Best Case (Ω): When the goal position is assigned to the starting position. Since, BFS travel through each cell once, even in the best case, time complexity remains O(V).

Average Case (Θ): When the goal position is in between the starting and goal position. The time complexity remains O(V).

Worst Case (O): When the goal position is assigned at the farthest position or out of the bound grid. Time complexity is O(V) since it travels through every cell.

II. Task 04 Obstacle avoidance

Mechanism

Obstacle detection and avoidance are integrated into the breath-first search algorithm. BFS checks neighboring positions to ensure there are no obstacles. If found an obstacle skip over the detected node in the list.

How it works

1. Define the directions. Directions 2D array holds values for moving 4 directions. They are represented using 2 integers, where the first element is for X coordinates and the second for Y coordinates.

```
// Exploring neighboring positions.
int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
// 2D array with directions. (deltaX, deltaY)   (left, right, down, up)

for (int[] dir : directions) {  // loop iterate through each direction in the array.
    // Add both current position plus change of the coordinate. 0 for X
    int newX = current.getX() + dir[0];
    int newY = current.getY() + dir[1];  // 1 for Y
```

2. The loop iterates through each direction and checks if the new position is within the bound of the grid, not visited before, and not an obstacle.
3. If the new position satisfies the conditions, the robot successfully avoids the obstacles in the grid.

Why use this obstacle avoidance method

- This method operates systematically if an obstacle is found.
- At the same time the program avoids the obstacles and finds the free path. This mechanism is more efficient than using a separate method to avoid obstacles.

4. Test Plan

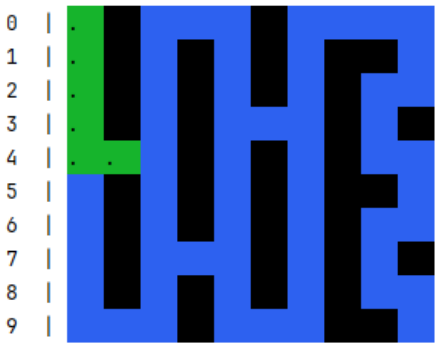
```
Robot says, "I'm ready to find the path!!"
Please say which option should I need to use
1 <--- Find Path
2 <--- Create a Custom Grid and Find Path
3 <--- Exit
1

Robot says, "I'm in (0,0) position. Now say where should I need to go!"
Robot current position (0,0)
Y\X  0  1  2  3  4  5  6  7  8  9
0  | 0  1  0  0  0  1  0  0  0  0
1  | 0  1  0  1  0  1  0  1  1  0
2  | 0  1  0  1  0  1  0  1  0  0
3  | 0  1  0  1  0  0  0  1  0  1
4  | 0  0  0  1  0  1  0  1  0  0
5  | 0  1  0  1  0  1  0  1  1  0
6  | 0  1  0  1  0  1  0  1  0  0
7  | 0  1  0  0  0  1  0  1  0  1
8  | 0  1  0  1  0  1  0  1  0  0
9  | 0  0  0  1  0  0  0  1  1  0
```

I. The starting point and the goal position on the same place. (Best case)

```
Enter goal X coordinate: 0

Enter goal Y coordinate: 0

Path found:
Y\X  0  1  2  3  4  5  6  7  8  9
0  | .
1  |
2  |
3  |
4  |
5  |
6  |
7  |
8  |
9  |

Robot current position:(0,0)
Robot says, "I found the GOAL!!"
. = Path,  = Obstacles,  = Open area
```

II. The starting point is close to the goal.

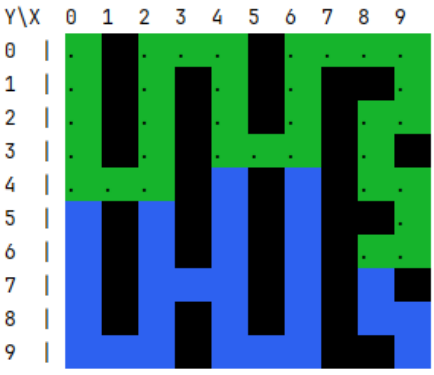
```
Enter goal X coordinate: 1

Enter goal Y coordinate: 4

Path found:
```




```
Robot current position:(1,4)
Robot says, "I found the GOAL!!"
. = Path,  = Obstacles,  = Open area
```

III. The starting point and goal position are far apart.

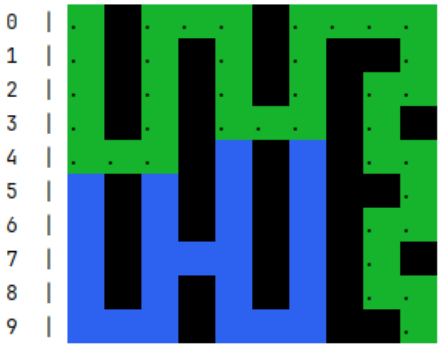
```
Enter goal X coordinate: 8

Enter goal Y coordinate: 6

Path found:
```




```
Robot current position:(8,6)
Robot says, "I found the GOAL!!"
. = Path,  = Obstacles,  = Open area
```

IV. The goal is in the farthest position in the grid. (Worst case)

```
Enter goal X coordinate: 9

Enter goal Y coordinate: 9

Path found:
```




```
Robot current position:(9,9)
Robot says, "I found the GOAL!!"
. = Path,  = Obstacles,  = Open area
```

V. The goal on an obstacle.

```
Enter goal X coordinate: 3

Enter goal Y coordinate: 2
Robot says, "Sorry! Path not found :\"
```

VI. The goal is out of the bounds of the grid.

```
Enter goal X coordinate: 20

Enter goal Y coordinate: 20
Robot says, "Sorry! Path not found :\"
```

VII. For grid structure created by the user.

```
Robot says, "Let's create a grid with obstacles."
Enter the size of the grid (n x n) n = 6
Enter number of obstacles (<= 36): 8
Enter X coordinate: 1
Enter Y coordinate: 4
Enter X coordinate: 2
Enter Y coordinate: 4
Enter X coordinate: 3
Enter Y coordinate: 4
Enter X coordinate: 5
Enter Y coordinate: 4
Enter X coordinate: 2
Enter Y coordinate: 6
Y coordinate out of bounds. Please enter a valid Y coordinate:
2
Enter X coordinate: 7
X coordinate out of bounds. Please enter a valid X coordinate:
2
Enter Y coordinate: 1
Enter X coordinate: 0
Enter Y coordinate: 1
Enter X coordinate: 0
Enter Y coordinate: 5
```

```
Robot says, "I'm in (0,0) position. Now say where should I need to go!"
Robot current position (0,0)
Y\X   00   01   02   03   04   05
00  | 0    1    0    0    0    1
01  | 0    0    0    0    1    0
02  | 0    1    1    0    1    0
03  | 0    0    0    0    1    0
04  | 0    0    0    0    0    0
05  | 0    0    0    0    1    0
```

```
Enter goal X coordinate: 5

Enter goal Y coordinate: 5

Path found:
```


```
Robot current position:(5,5)
Robot says, "I found the GOAL!!"
. = Path,  = Obstacles,  = Open area
------------------------------------------------------------
```

VIII. The starting point is covered by the obstacles.

```
Robot says, "I'm in (0,0) position. Now say where should I need to go!"
Robot current position (0,0)
Y\X  0  1  2  3  4
0  | 0  0  1  1  0
1  | 0  0  1  1  1
2  | 1  0  1  0  1
3  | 0  1  0  1  1
4  | 1  0  1  0  1

Enter goal X coordinate: 3

Enter goal Y coordinate: 4
Robot says, "Sorry! Path not found :\"
```

IX. A complex obstacle arrangement.

```
Robot says, "I'm in (0,0) position. Now say where should I need to go!"
0 = Open positions, 1 = Obstacles
Robot current position (0,0)
Y\X   00   01   02   03   04   05   06   07   08   09   10   11   12   13   14   15
00  | 0    1    0    0    0    1    0    0    0    0    1    0    0    1    0    1
01  | 0    1    0    1    0    1    0    1    1    0    1    0    1    1    0    1
02  | 0    1    0    0    0    1    0    1    0    0    1    0    0    0    0    0
03  | 0    1    0    1    0    1    0    0    0    1    1    0    1    1    0    1
04  | 0    1    0    1    0    1    0    1    0    0    1    0    1    0    0    0
05  | 0    0    0    1    0    1    0    1    1    0    1    0    1    0    0    1
06  | 0    1    0    1    0    1    0    1    0    0    1    0    1    0    0    0
07  | 0    1    0    1    0    1    0    1    0    1    1    0    1    0    0    1
08  | 0    1    0    1    0    1    0    1    0    0    1    0    1    0    1    0
09  | 0    0    0    1    0    0    0    1    0    0    1    0    1    0    0    0
10  | 1    1    1    1    1    1    1    1    1    0    1    0    1    0    0    0
11  | 0    0    0    0    0    0    0    0    0    0    1    0    1    1    0    1
12  | 0    1    0    1    1    1    1    1    1    1    1    0    0    1    0    0
13  | 0    0    1    0    0    0    0    0    1    0    0    0    1    0    0    0
14  | 0    0    0    0    0    0    1    0    0    0    1    0    1    0    1    0
15  | 1    0    1    1    0    1    1    1    1    1    1    1    1    1    0    0
```

```
Enter goal X coordinate: 15

Enter goal Y coordinate: 15

Path found:
Y\X   100   01   02   03   04   05   06   07   08   09   10   11   12   13   14   15
00  |
01  |
02  |
03  |
04  |
05  |
06  |
07  |
08  |
09  |
10  |
11  |
12  |
13  |
14  |
15  |

Robot current position:(15,15)
Robot says, "I found the GOAL!!"
. = Path,  = Obstacles,  = Open area
```

5. References

GeeksforGeeks, 2013. *GeeksforGeeks.* [Online]
Available at: https://www.geeksforgeeks.org/what-is-linked-list/
[Accessed February 2024].

Goodrich, M. T., 2014. *Data structures and Algorithms.* 6 ed. Hoboken: Wiley.

Jain, D., 2024. *baeldung.com.* [Online]
Available at: https://www.baeldung.com/java-solve-maze
[Accessed February 2024].