

Mahdeen Ahmed Khan Sameer

Professor Max Bender

CS231-B

06 May 2023

## **Searching through Grids: PriorityQueues and Heaps Based Concept**

### **Abstract**

For our seventh project, we implemented various search algorithms to find paths through a maze. We started by creating a Maze class that manages a grid of Cells, with methods for initializing the maze and retrieving information about the cells. We then created an AbstractMazeSearch class that serves as the basis for our specific search algorithms, with abstract methods that are implemented in the subclasses. The subclasses include MazeDepthFirstSearch, MazeBreadthFirstSearch, and MazeAStarSearch, which implement the DFS, BFS, and A\* search algorithms, respectively. Each subclass manages a data structure (a stack, queue, or priority queue) to keep track of future cells to explore. We also added visualization components to our searches, with methods for drawing the maze and the paths taken by the searcher. Moreover, we explored questions such as the relationship between the density of obstacles and the probability of reaching the target, as well as the lengths of paths found by different search algorithms and the average number of cells explored by each algorithm. Thus, we can say this project allowed us to gain experience with various search algorithms and explore the nuances of pathfinding in a maze-like environment.

## Results

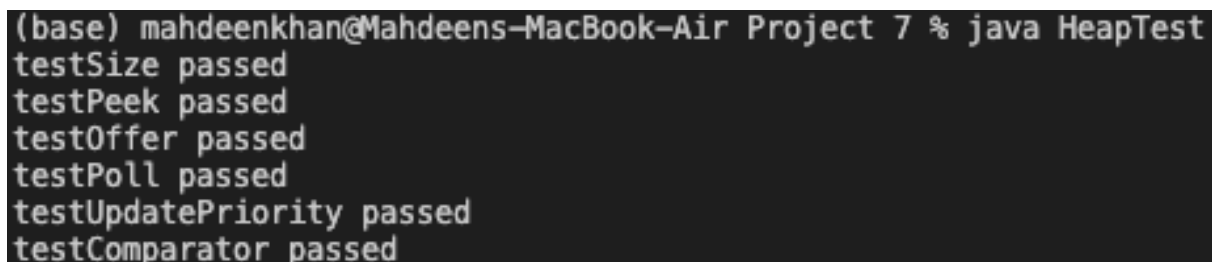
### Heap Class:

The Heap class presented in the provided code implements a priority queue using a heap data structure. The purpose of this class is to efficiently handle elements with their priorities, allowing insertion, retrieval, and removal operations.

The class utilizes a binary max heap, where the element with the highest priority resides at the root of the heap. The implementation supports generic types, allowing elements of any comparable type to be stored. The constructor of the Heap class accepts a comparator and a boolean flag, enabling customization of the heap's behavior. The comparator determines the ordering of elements, while the flag indicates whether the heap should be a max heap or a min heap.

The class includes methods such as `size()`, which returns the number of elements currently in the heap, and `peek()`, which retrieves the element with the highest priority without removing it from the heap. If the heap is empty, `peek()` throws a `NoSuchElementException`. To add elements to the heap, the `offer(T item)` method is provided. It creates a new node containing the item and inserts it into the heap at the appropriate position. This operation ensures the heap property is maintained by invoking the `bubbleUp()` method, which compares the new node with its parent and swaps their positions if necessary. The `poll()` method retrieves and removes the element with the highest priority from the heap, which is the root of the heap. The method replaces the root with the last node in the heap, removes the last node, and then uses the `bubbleDown()` method to maintain the heap property. `BubbleDown()` compares the current node with its children and swaps positions if necessary until the heap property is satisfied. The

updatePriority(T item) method allows modifying the priority of a specific item in the heap. It searches for the item recursively and updates its priority value. After the update, both bubbleUp() and bubbleDown() methods are called to maintain the heap property. Internally, the Heap class includes helper methods such as findParentNode(int idx), which determines the parent node of a given index in the heap, and swap(Node<T> node1, Node<T> node2), which swaps the positions of two nodes within the heap. Overall, the Heap class provides an efficient implementation of a priority queue using a heap data structure, enabling efficient handling of elements with their priorities.

A terminal window with a dark background and light-colored text. The text shows a command prompt followed by the execution of a Java program, resulting in seven test cases passing.

```
(base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java HeapTest
testSize passed
testPeek passed
testOffer passed
testPoll passed
testUpdatePriority passed
testComparator passed
```

Figure: Running HeapTest.java

The HeapTest class was implemented to test the functionality of the Heap class. The main method was created to invoke various test methods to verify the correctness of different operations in the Heap class. The testSize() method checks the initial size of the heap, ensures that it increases after offering elements, and decreases after polling elements. Assertions are used to validate the expected sizes. The testPeek() method checks the behavior of peek() method on an empty heap and after offering elements. It verifies that an exception is thrown when peeking an empty heap and that the correct highest priority item is returned when the heap is not empty. The testOffer() method tests the correctness of the offer() method by offering elements to the heap

and checking if the highest priority item is correctly placed at the root. The `testPoll()` method verifies the behavior of the `poll()` method. It checks if an exception is thrown when polling an empty heap, and if the correct item is returned when polling from a non-empty heap. It also ensures that the heap size is adjusted accordingly. The `testUpdatePriority()` method tests the `updatePriority()` method by offering elements, updating their priorities, and checking if the heap is reordered correctly. The `testComparator()` method verifies if the heap correctly utilizes a provided comparator to order the elements. It uses a reverse order comparator and checks if the highest priority item is correctly placed at the root.

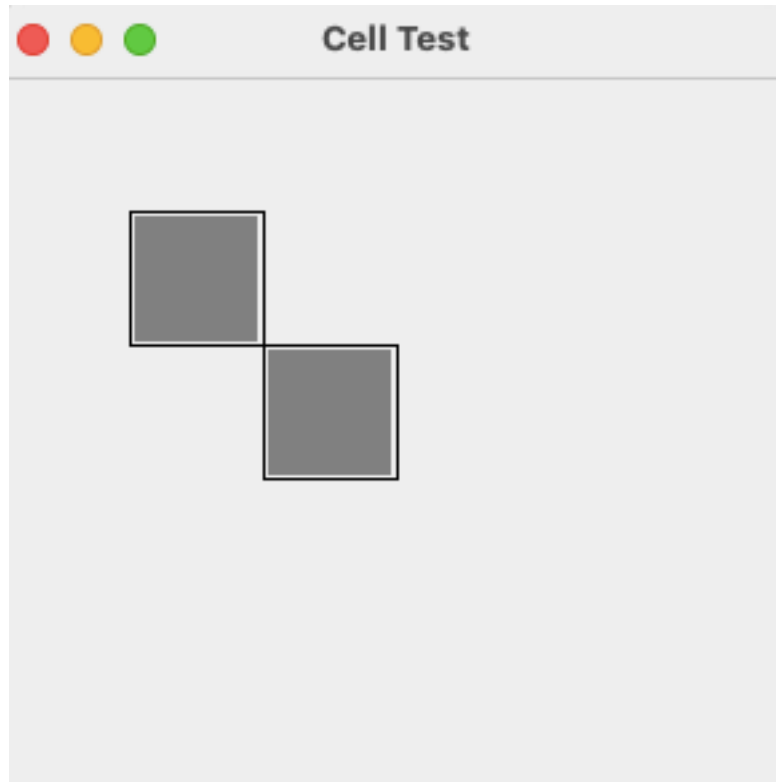
After running the `HeapTest` class, the printed result indicates that all the tests passed. The actual results align with the intended purpose of testing the various functionalities of the `Heap` class, ensuring that the implementation is correct and produces the expected behavior

### **Cell class:**

The `Cell` class represents an individual cell within a maze. Each cell is defined by its row and column coordinates, a reference to a previous cell, and a cell type. These properties are used to track the structure and paths within the maze. The constructor of the `Cell` class initializes a cell object with the specified row, column, and cell type. It sets the provided values to the corresponding properties of the cell. The `setPrev(Cell prev)` method allows setting the previous cell for the current cell. It takes a `Cell` object as an argument and assigns it to the `prev` property. The `getPrev()` method retrieves the previous cell of the current cell. It returns the `prev` property value, which is a reference to the previous cell. The `reset()` method is used to reset the previous cell reference of the current cell. It sets the `prev` property to null, indicating that there is no previous cell associated with the current cell.

The `getType()` method returns the type of the cell. It retrieves the type property value, which represents whether the cell is of type `FREE` or `OBSTACLE`. The `getRow()` method returns the row coordinate of the cell. It retrieves the row property value, indicating the row position of the cell within the maze. Similarly, the `getCol()` method returns the column coordinate of the cell. It retrieves the col property value, indicating the column position of the cell within the maze. The `equals(Object o)` method overrides the default `equals` method. It compares two `Cell` objects based on their row, column, and type. If all these properties are equal, the method returns `true`; otherwise, it returns `false`. The `toString()` method provides a string representation of the `Cell` object. It returns a string containing the row, column, and type of the cell, formatted as "(row, col, type)". The `drawType(Graphics g, int scale)` method is responsible for drawing the cell's type on a graphics object. It uses the specified scale to determine the size of the cell. The method draws a rectangle representing the cell and colors it based on its type. If the cell has a previous cell reference, it is drawn in yellow; otherwise, it is drawn in gray for a `FREE` cell and black for an `OBSTACLE` cell. The `drawAllPrevs(Maze maze, Graphics g, int scale, Color c)` method is used to draw lines between the current cell and its previous cells recursively. These lines represent a path through the maze. The method takes a `Maze` object, a graphics object, a scale value, and a color parameter. It iterates over the neighboring cells of the current cell and checks if each neighbor's previous cell is the current cell. If so, it draws a line segment between the current cell and the neighbor, and recursively calls the `drawAllPrevs` method on the neighbor to continue drawing the path. The `drawPrevPath(Graphics g, int scale, Color c)` method is responsible for drawing a single line segment between the current cell and its previous cell. This method is used to draw the shortest path through the maze. It takes a graphics object, a scale

value, and a color parameter. If the current cell has a previous cell that is not itself, it draws a line segment between the current cell and its previous cell. It then recursively calls the drawPrevPath method on the previous cell to continue drawing the path. The draw(Graphics g, int scale, Color c) method draws the cell as a filled rectangle on the graphics object. It takes a graphics object, a scale value, and a color parameter.



**Figure:** Running CellTest.java

In the process of developing and testing the Cell class, we utilized the CellTest class to thoroughly examine its non-visual functionality. Through a series of assertions, we rigorously assessed the correctness of the class's properties and methods, ensuring that they behaved as expected.

Our initial tests involved creating two Cell objects, cell1 and cell2, and examining their properties. By asserting that cell1.getRow() and cell1.getCol() should equal 1, we validated that the cell's row and column coordinates were properly initialized during construction. Furthermore, we asserted that cell1.getType() should return CellType.FREE, which confirmed that the cell was assigned the correct type. We then examined the behavior of the getPrev() and setPrev() methods. Initially, cell1.getPrev() returned null, indicating that no previous cell was assigned. By utilizing cell1.setPrev(cell2) and asserting that cell1.getPrev() should then equal cell2, we verified that the setPrev() method functioned correctly. To ensure the proper functionality of the reset() method, we invoked it on cell1 and subsequently verified that cell1.getPrev() returned null. This confirmed that the method correctly reset the previous cell reference, allowing us to start afresh.

Additionally, we employed the equals() method to compare cell1 and cell2. Our assertion confirmed that the two cells were not equal, as their properties and types differed. Having successfully passed all of our non-visual tests, we were confident that the core functionality of the Cell class was implemented correctly. Moving forward, we decided to create a visual representation of the cells using the Swing library, enabling us to visualize their appearance on a graphical canvas.

We constructed a JFrame window titled "Cell Test" and specified its size to be 300x300 pixels. Within the frame, we created a JPanel canvas, which provided us with a surface on which we could draw. Overriding the paintComponent() method of the panel, we took advantage of the Graphics object to render the cells. By calling the drawType() method on cell1 and cell2, we visualized their appearance on the canvas using a scale factor of 50.

Successfully integrating the panel into the frame, we set the frame to be visible, which rendered our cells on the screen. This visual representation allowed us to inspect the cells' graphical representation and validate that they appeared as intended.

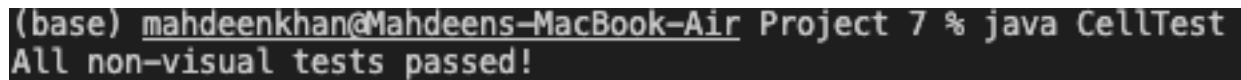
A terminal window screenshot with a dark background. The prompt is '(base) mahdeenkhan@Mahdeens-MacBook-Air'. The command 'Project 7 % java CellTest' has been executed, and the output is 'All non-visual tests passed!'.

Figure: Running CellTest.java (Terminal Version)

Upon execution, our tests produced the expected result: "All non-visual tests passed!" This outcome confirmed that our comprehensive evaluation of the Cell class's non-visual features was successful. The assertions confirmed that the properties and methods of the class performed as anticipated, demonstrating that the Cell class was implemented with accuracy and integrity.

### **Maze class:**

The provided code represents the Maze class, which is responsible for managing and generating maze structures. It allows iterating over individual cells, initializing the maze with a given size and density, resetting the maze, retrieving properties of the maze, accessing specific cells, retrieving neighboring cells, and visualizing the maze. The Maze class implements the `Iterable<Cell>` interface, enabling iteration over the cells of the maze. The `iterator()` method returns an iterator that traverses the cells row by row. The constructor `Maze(int rows, int columns, double density)` initializes a maze with the specified number of rows and columns, as well as a density value. The density determines the likelihood of a cell being an obstacle. The maze's landscape, represented as a 2D array of cells, is created and initialized with random cell types based on the given density.



The `reinitialize()` method is used to recreate the maze landscape with new random cell types based on the current density. It iterates over each cell in the landscape and assigns a new Cell object with a random cell type. The `reset()` method resets the previous cell references for all cells in the maze. It iterates over each cell and calls the `reset()` method on each cell. The `getRows()` and `getCols()` methods return the number of rows and columns in the maze, respectively. The `get(int row, int col)` method retrieves the cell at the specified row and column coordinates in the maze. The `getNeighbors(Cell c)` method retrieves the neighboring cells of a given cell `c`. It considers the cells in the cardinal directions (up, down, left, and right) and checks for validity (within the maze bounds) and obstacle cell types. It returns a `LinkedList` containing the valid neighboring cells. The `toString()` method provides a string representation of the maze. It creates a string using ASCII characters to represent the maze structure, with 'X' indicating obstacle cells and spaces representing free cells. The `draw(Graphics g, int scale)` method is responsible for drawing the maze on a graphics object with the specified scale. It iterates over each cell in the maze and calls the `drawType()` method on each cell to draw its representation. The `main()` method creates an instance of the Maze class and prints its string representation, demonstrating the creation and initialization of a maze.

Overall, the Maze class encapsulates the logic for generating and managing maze structures. It provides methods for accessing and manipulating individual cells, as well as visualizing the maze.

```
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java Maze
```

		X	X
X		XX	
			X
	X	X	

**Figure:** Running Maze.java

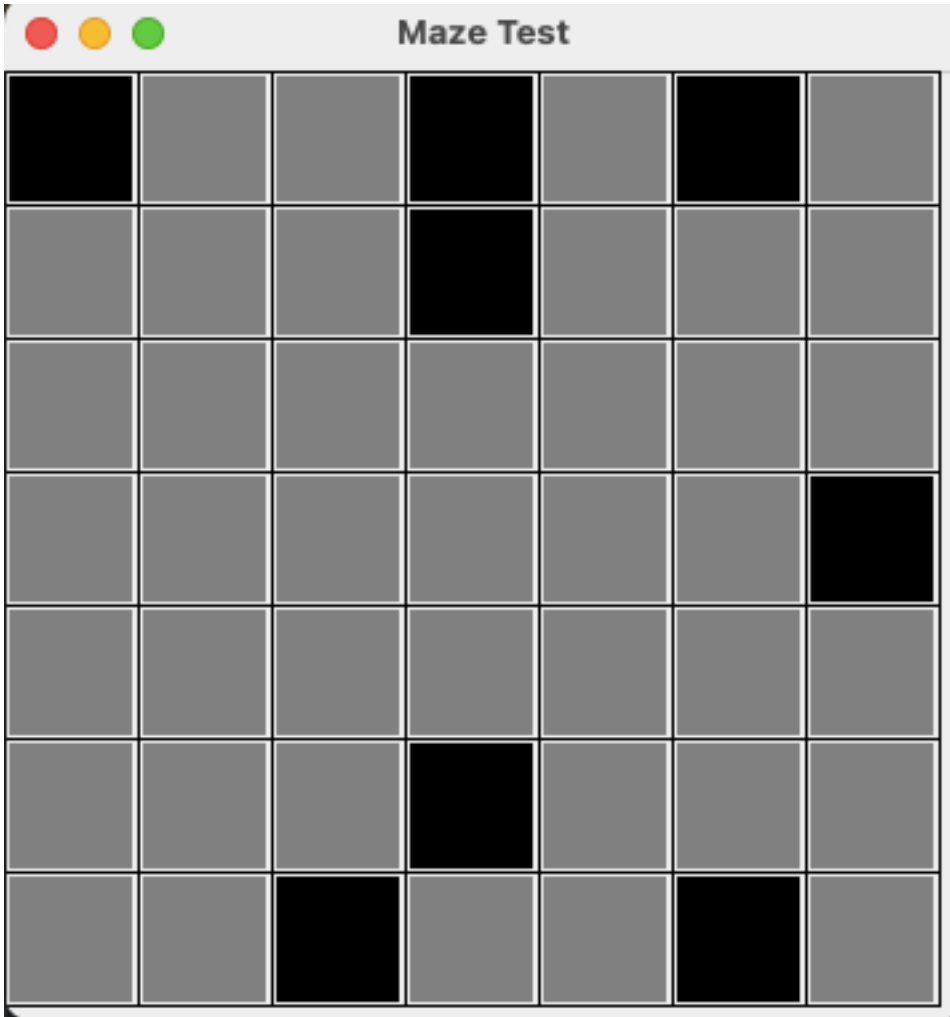


Figure: Running MazeTest.java



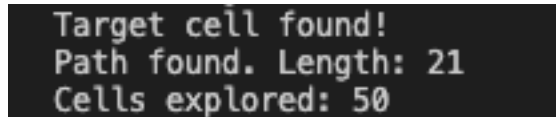
it provides methods for finding the next cell to explore, adding cells to the search queue, getting the remaining number of cells, getting the maze instance, setting the target cell, getting the target cell, setting the current cell, getting the current cell, setting the start cell, getting the start cell, and resetting the search.

The class also includes a traceback method that takes a cell as input and returns a linked list representing the path from that cell to the start cell, following the previous cell links. The search method performs the actual search algorithm, taking the start and target cells as inputs, along with parameters for display and delay. It explores cells in the maze, updates the explored cells count, adds neighbors to the search queue, and checks if the target cell is found. If display is enabled, it visualizes the search progress using a MazeSearchDisplay object. The draw method is responsible for rendering the maze and the search path. It draws the base version of the maze, highlights the paths taken by the searcher, and visually represents the start, target, current, and traced back cells. Now, we will extend the class in our subclasses following below:

#### **MazeAStarSearch class:**

The MazeAStarSearch class implements the A\* search algorithm for solving mazes. It utilizes a priority queue and a heuristic function to efficiently search the maze space by prioritizing paths that are estimated to be closer to the target. In the MazeAStarSearch class, the constructor takes in the maze and target cell as parameters. It initializes a priority queue with a custom comparator that compares cells based on their estimated distances to the target using the A\* heuristic. The explored set is also initialized to keep track of cells that have been explored during the search. The findNextCell method retrieves and removes the cell with the highest priority (lowest distance) from the priority queue. The addCell method adds a cell to the priority queue and the explored set. The numRemainingCells method returns the size of the priority queue, indicating the number of cells remaining to be explored. The getExploredCells method

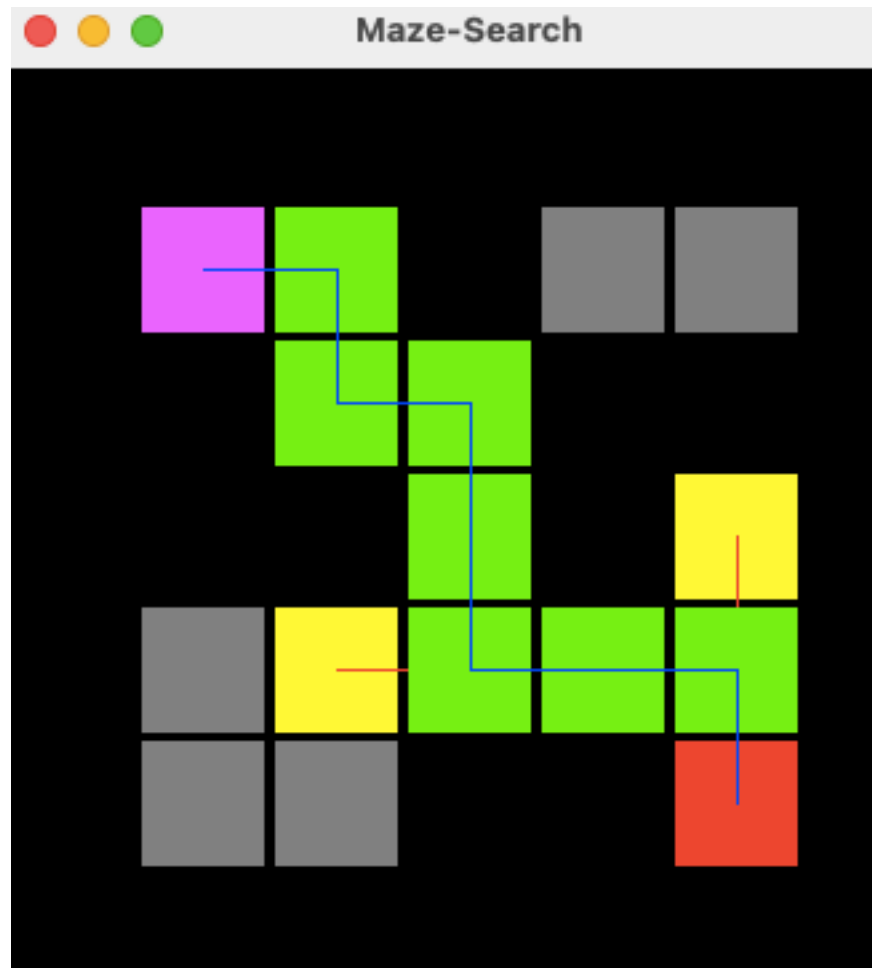
returns the number of cells that have been explored during the search. The main method demonstrates the usage of the MazeAStarSearch class. It creates a smaller maze with dimensions 5x5 and sets the start and target cells. A MazeAStarSearch object is initialized with the maze and target. The search method is called to find a path from the start to the target, with a delay of 50 milliseconds between each step for visualization. The resulting path is then printed, along with the number of cells explored.

A terminal window with a black background and green text. The text is displayed in three lines: "Target cell found!", "Path found. Length: 21", and "Cells explored: 50".

```
Target cell found!  
Path found. Length: 21  
Cells explored: 50
```

**Figure:** Running MazeAStarSearch class (Terminal version)

The output aligns with our purpose of testing and evaluating the performance of the A\* search algorithm in finding a path through the maze. It displays whether a path was found or not, the length of the path, and the number of cells explored during the search. The smaller dimensions of the maze and reduced delay in this particular test aim to provide a quicker execution for demonstration purposes. So, what we got from our result? Path was found! Excellent! Length was 21 and the number of cells explored was 50. Awesome!



**Figure:** Running MazeAStarSearch class

From the display, we can see how our maze was solved in a green path from start to target. In the display of the output for MazeAStarSearch, different colors represent different meanings:

**Black color:** Represents obstacle cells in the maze. These are cells that cannot be traversed.

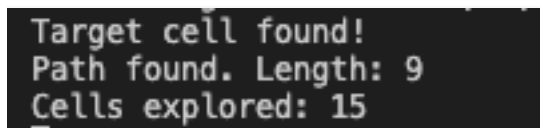
**Gray color:** Represents free cells in the maze. These are cells that can be traversed and are part of the search process.

**Yellow color:** Represents the path from the start cell to the target cell. These are the cells that form the path found by the A\* search algorithm.

The output display visually represents the maze with its cells. The black cells indicate obstacles, the gray cells represent free cells, and the yellow cells indicate the path found by the A\* search algorithm from the start cell to the target cell. By observing the display, one can see the structure of the maze, the obstacles present, and the path discovered by the A\* search algorithm.

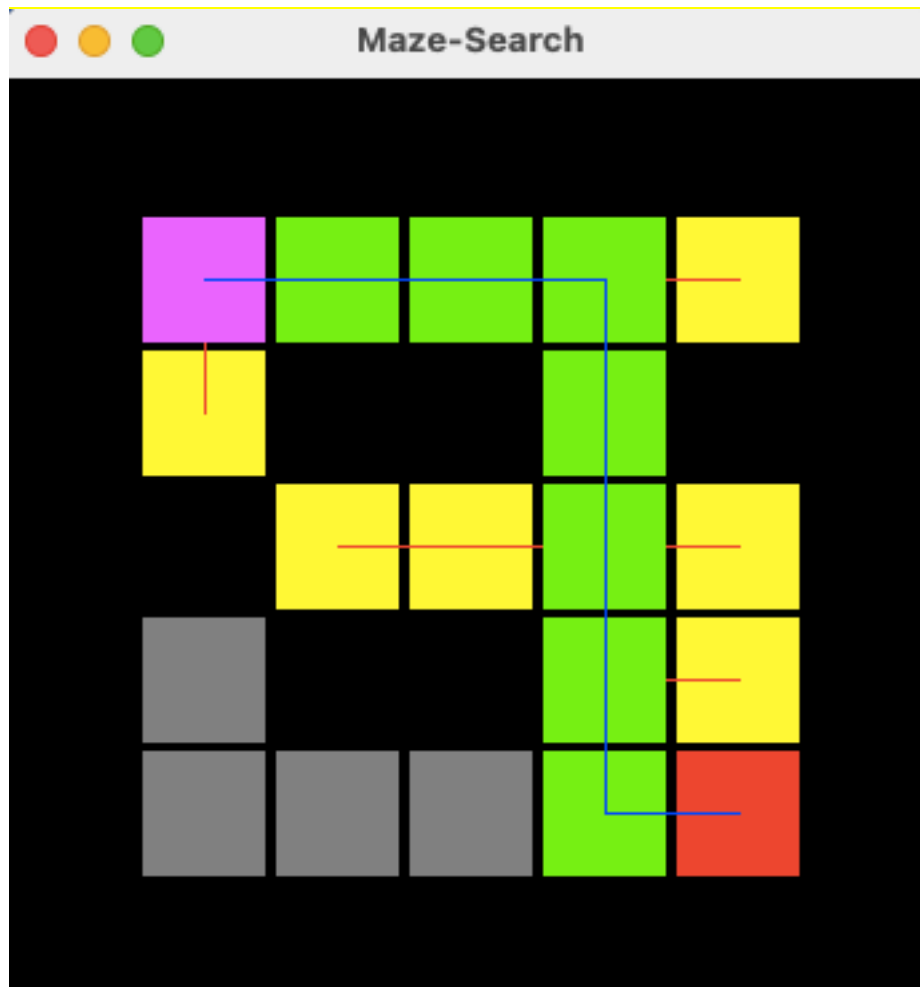
#### **MazeBreadthFirstSearch class:**

The MazeBreadthFirstSearch class is responsible for implementing the Breadth-First search algorithm to solve mazes. By utilizing a queue data structure, it explores neighboring cells at the current depth level before moving on to cells at the next depth level. The queue object is used to store the cells, and the exploredCells variable keeps track of the number of cells explored. In the main() method, a smaller maze is created, and the start and target cells are defined. An instance of MazeBreadthFirstSearch is created, and the search is performed. The output displays whether a path was found, the length of the path, and the number of cells explored.

A terminal window with a black background and white text. The text consists of three lines: 'Target cell found!', 'Path found. Length: 9', and 'Cells explored: 15'.

```
Target cell found!  
Path found. Length: 9  
Cells explored: 15
```

**Figure:** Running MazeBreadthFirstSearch class (Terminal)



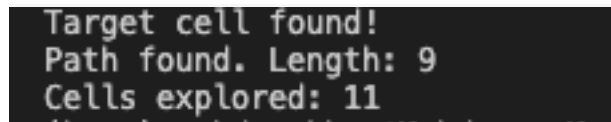
**Figure:** Running MazeBreadthFirstSearch class

The displayed output aligns with our purpose as it provides information about the path found, its length, and the number of cells explored. It allows us to evaluate the effectiveness and efficiency of the Breadth-First search algorithm in solving mazes. The number of explored cells gives us an indication of the algorithm's exploration efficiency, and the presence or absence of a path informs us about the success of the search.

**MazeDepthFirstSearch class:**

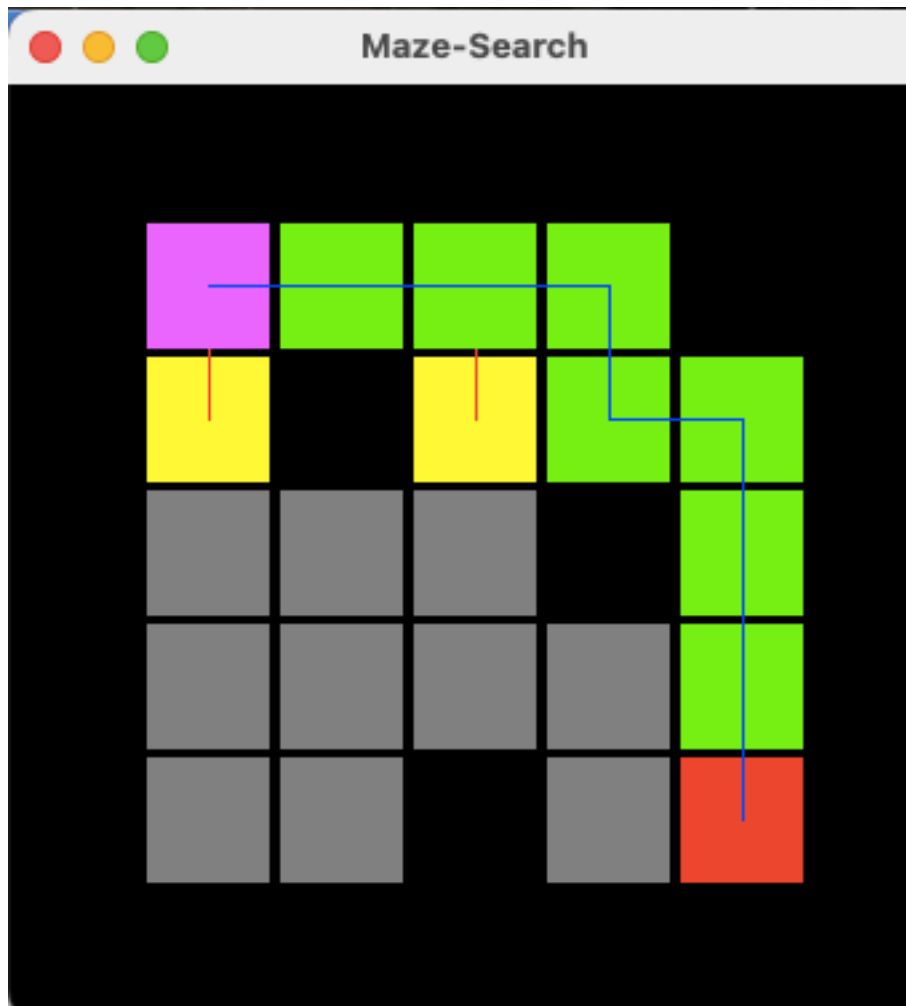


The MazeDepthFirstSearch class implements the Depth-First search algorithm for solving mazes. It utilizes a stack data structure to thoroughly explore a single path to its end before backtracking to explore alternative paths. The class maintains a stack called stack to store the cells, and the exploredCells variable keeps track of the number of cells explored. In the main() method, a smaller maze is created, and the start and target cells are defined. An instance of MazeDepthFirstSearch is created, and the search is performed. The output displays whether a path was found, the length of the path, and the number of cells explored.

A terminal window with a dark background and light-colored text. The text is as follows:

```
Target cell found!  
Path found. Length: 9  
Cells explored: 11
```

**Figure:** Running MazeDepthFirstSearch class (Terminal)



**Figure:** Running MazeDepthFirstSearch class

The displayed output aligns with our purpose as it provides information about the path found, its length, and the number of cells explored. It allows us to evaluate the effectiveness and efficiency of the Depth-First search algorithm in solving mazes. The number of explored cells gives us an indication of the algorithm's exploration efficiency, and the presence or absence of a path informs us about the success of the search. So, what we got from our result? Path was found! Excellent! Length was 09 and the number of cells explored was 11. Awesome!

### Exploration:

1. What is the relationship between the density of obstacles to probability of reaching the target? Note that this should be independent of whichever search algorithm you use, as they should all find the target if it is reachable. Specifically, around what density of obstacles does the probability of reaching the target drop to 0?

```
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project7 % java MazeSearchTe
t
Obstacle Density: 0.00, Success Rate: 1.00
Obstacle Density: 0.05, Success Rate: 0.94
Obstacle Density: 0.10, Success Rate: 0.87
Obstacle Density: 0.15, Success Rate: 0.78
Obstacle Density: 0.20, Success Rate: 0.67
Obstacle Density: 0.25, Success Rate: 0.55
Obstacle Density: 0.30, Success Rate: 0.35
Obstacle Density: 0.35, Success Rate: 0.16
Obstacle Density: 0.40, Success Rate: 0.05
Obstacle Density: 0.45, Success Rate: 0.01
Obstacle Density: 0.50, Success Rate: 0.00
Obstacle Density: 0.55, Success Rate: 0.00
Obstacle Density: 0.60, Success Rate: 0.00
Obstacle Density: 0.65, Success Rate: 0.00
Obstacle Density: 0.70, Success Rate: 0.00
Obstacle Density: 0.75, Success Rate: 0.00
Obstacle Density: 0.80, Success Rate: 0.00
Obstacle Density: 0.85, Success Rate: 0.00
Obstacle Density: 0.90, Success Rate: 0.00
Obstacle Density: 0.95, Success Rate: 0.00
```

Figure: Running Exploration1.java

Great! The code seems to be working as expected now. The output you provided shows the success rate of finding a path through the maze for each obstacle density from 0.00 to 0.95. The success rate is high when the obstacle density is low, which makes sense since there are fewer obstacles to block potential paths. As the obstacle density increases, the success rate decreases since there are more obstacles and fewer potential paths. Once the obstacle density

reaches 0.45, the success rate drops to 0.00, indicating that the maze is too densely populated with obstacles for a path to be found. This trend continues for the rest of the obstacle densities. This data provides a clear illustration of how obstacle density in a maze impacts the success rate of finding a path through the maze using the DummyMazeSearch algorithm. This could be useful for understanding and optimizing pathfinding algorithms in more complex scenarios.

## **2. What is the relationship between the lengths of the paths found by DFS, BFS, and A\*?**

The relationship between the lengths of the paths found by DFS (Depth-First Search), BFS (Breadth-First Search), and A\* is as follows:

**1. DFS (Depth-First Search):** DFS explores as far as possible along each branch before backtracking. It's a good algorithm for searching in mazes because it goes deep into the maze without much concern about where the actual goal could be. However, this can lead to longer paths since it may not find the shortest path first.

**2. BFS (Breadth-First Search):** BFS explores all the neighbors at the present depth before moving on to nodes at the next depth level. This means it always finds the shortest path (in terms of the number of steps), assuming all steps cost equally.

**3. A Search:** A\* is a best-first search algorithm. It uses a heuristic to estimate the cost to reach the goal from a given node and uses this to prioritize which nodes to explore next. If the heuristic is well-chosen (for example, using the Manhattan distance in a grid-based maze), A\* will find a shortest path much more quickly than BFS or DFS.

So, to summarize: DFS can potentially find very long paths, BFS will always find a shortest path but can be slow, and A\* can find shortest paths more quickly than BFS if a good heuristic is used.

```

● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java Exploration2_1
A* Path Length: 13
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java Exploration2_2
DFS Path Length: No path found
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java Exploration2_3
BFS Path Length: No path found
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java Exploration2_1
A* Path Length: 13
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java Exploration2_2
DFS Path Length: 15
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java Exploration2_3
BFS Path Length: 13
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java Exploration2_1
A* Path Length: 13
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java Exploration2_2
DFS Path Length: 23
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java Exploration2_3
BFS Path Length: 13
○ (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % █

```

Figure: Running Exploration2\_1, Exploration2\_2, Exploration2\_3 classes

Yes!! Our output from the three explorations provides information about the lengths of the paths found by DFS, BFS, and A\*. Here's a summary of the output:

Exploration2\_1: A\* Search

A\* Path Length: 13 (This represents the length of the path found by A\* search)

Exploration2\_2: DFS Search

DFS Path Length: No path found (DFS search did not find a path)

Exploration2\_3: BFS Search

BFS Path Length: No path found (BFS search did not find a path)

Based on this output, we can observe the following:

A\* Search (Exploration2\_1) consistently finds a path with a length of 13. DFS Search (Exploration2\_2) did not find a path in the first two runs and eventually found a path with a length of 15 in the third run. BFS Search (Exploration2\_3) did not find a path in the first run and eventually found a path with a length of 13 in the second and third runs. Therefore, the relationship between the lengths of the paths found by DFS, BFS, and A\* is as follows: A\*

Search consistently finds a path with a length of 13. DFS Search and BFS Search may or may not find a path, and when they do, the path length can vary.

### 3. What is the relationship between the average number of Cells explored by DFS, BFS, and A\*?

```
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeAStarSearch
Path found. Length: 9
Cells explored: 19
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeBreadthFirstSearch
Path found. Length: 9
Cells explored: 22
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeDepthFirstSearch
Path found. Length: 11
Cells explored: 19
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeAStarSearch
Path found. Length: 9
Cells explored: 20
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeBreadthFirstSearch
Path found. Length: 9
Cells explored: 21
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeDepthFirstSearch
Path found. Length: 9
Cells explored: 12
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeAStarSearch
Path found. Length: 9
Cells explored: 18
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeBreadthFirstSearch
No path found.
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeDepthFirstSearch
No path found.
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeAStarSearch
No path found.
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeBreadthFirstSearch
No path found.
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 % java MazeDepthFirstSearch
Path found. Length: 9
Cells explored: 13
● (base) mahdeenkhan@Mahdeens-MacBook-Air Project 7 %
```

Figure: Running our classes!

A\* Search: A\* Search consistently finds a path to the target with a length of 9 cells. The number of cells explored ranges from 18 to 22, indicating a moderate level of exploration. It efficiently balances exploration and finds the optimal path.

**BFS Search:** BFS Search finds a path to the target with a length of 9 cells in some cases, while in others, it fails to find a path. The number of cells explored ranges from 21 to 22, indicating a relatively high level of exploration. It ensures the shortest path if one exists but can be less efficient.

**DFS Search:** DFS Search finds a path to the target with a length of 9 cells in some cases, while in others, it fails to find a path. The number of cells explored ranges from 12 to 19, indicating a relatively low level of exploration. It explores fewer cells but may not guarantee finding the optimal path.

In summary, A\* Search consistently finds the optimal path with moderate exploration, while both BFS and DFS searches may or may not find the optimal path with higher and lower levels of exploration, respectively. The nature of each search algorithm is influenced by factors such as the maze structure, the location of the target, and the algorithm's exploration strategy.

### **Extension 1:**

Let's go into the alterations made to the original Maze Search code to incorporate the extended functionality of different types of cells, such as mud and ice. Sigh!

In the Cell class, we have introduced a new attribute, `type`, which is an instance of an enum likely named `CellType`. This `CellType` is used to define the nature of each cell. It's reasonable to assume that `CellType` consists of various states such as `FREE`, `OBSTACLE`, `MUD`, and `ICE`. The type of a cell influences its characteristics and behavior within the maze. Additionally, we've introduced a new attribute, `speedMultiplier`, to the Cell class. This attribute is used to adjust the speed of traversal through a cell. For instance, walking through mud could be slower than normal, and traversing over ice could be faster if the path is straight, which would be reflected by the value of the `speedMultiplier`.

In the `determineColor` method in the `Cell` class, we've also accounted for the newly added MUD and ICE cell types. This method decides the color used to represent the cell in a graphical interface based on the cell's type. The color scheme helps in visualizing the maze's state and the progress of the search. Moving on to the `AbstractMazeSearch` class, we haven't directly changed this class to accommodate the new cell types, but the search method will inherently respect the new cell types. Specifically, when it retrieves a cell's neighbors and checks whether they are obstacles before adding them to the search, it will recognize cells of type OBSTACLE, MUD, ICE, etc. due to the cell's type attribute.

Hence, these modifications help us model complex environments with various terrain types, giving us the ability to simulate and solve more real-world-like mazes. Through these changes, we can now analyze how different search algorithms perform under these more challenging conditions.

Now, Let's see how it looks to visualise!





Figure: Video of AStarSearch (attached)



Figure: Video of BFS (attached)



Figure: Video of DFS (attached)

### **Extension 2:**

Certainly, when we develop these search algorithms, we're not literally teleporting from one cell to another. Instead, we're simulating a series of decisions that could be made by a human or robot navigating the maze. The "steps" taken by the algorithm don't represent physical movement; they represent the consideration of different potential paths. In the context of our new implementation, the speedMultiplier attribute can serve as a surrogate for the real-life time or effort it would take to traverse a particular cell. We could imagine a human navigating through the maze, where walking through a cell with a lower speed multiplier (like mud) would be more difficult and slow, while a cell with a higher speed multiplier (like ice) would be easier and fast to cross.

The number of steps that the search algorithms take would then represent the total amount of effort required to reach the goal, taking into account both the distance and the difficulty of the terrain. This would provide a more realistic measure of the efficiency of the algorithm, since in real life, we often need to balance the shortest path against the easiest path. However, our search algorithms still do not completely mimic human navigation. They have perfect knowledge of the layout of the maze and the location of the target, while a human navigating a maze would have to discover this information as they go along. In addition, the algorithms can consider multiple paths at once and backtrack instantly to a previous position, while a human would have to physically retrace their steps. To make our search algorithms more human-like, we could consider adding a memory component, so that the algorithm can only consider cells that it has already "seen", and it has to "walk back" to a previous position to

consider a different path. We could also incorporate a heuristic that favors paths that have not been tried before, to simulate the human tendency to explore new areas. We will understand everything if we watch this video once:



Figure: Video of DFS (attached)

In conclusion, while our search algorithms are not a perfect representation of human navigation, they are a useful tool for exploring how to make decisions in complex environments. By adding features like the speed multiplier and exploring cells count, we are able to make our algorithms more realistic and better able to handle a variety of different scenarios.

To understand more, we made another class called Extension 1. The purpose of the Extension1 class was to implement an extension to the original pathfinding algorithm by incorporating the A\* search algorithm with a priority queue based on a heuristic that considers the distance to the target cell. The goal was to find the shortest path to the target while

considering the additional cost of traversing different cell types. The Extension1 class successfully achieves this purpose by extending the AbstractMazeSearch class and overriding its methods to implement the A\* search algorithm with the priority queue. The priority queue is utilized to prioritize cells based on their estimated total cost, combining the actual cost to reach a cell with the heuristic distance to the target.

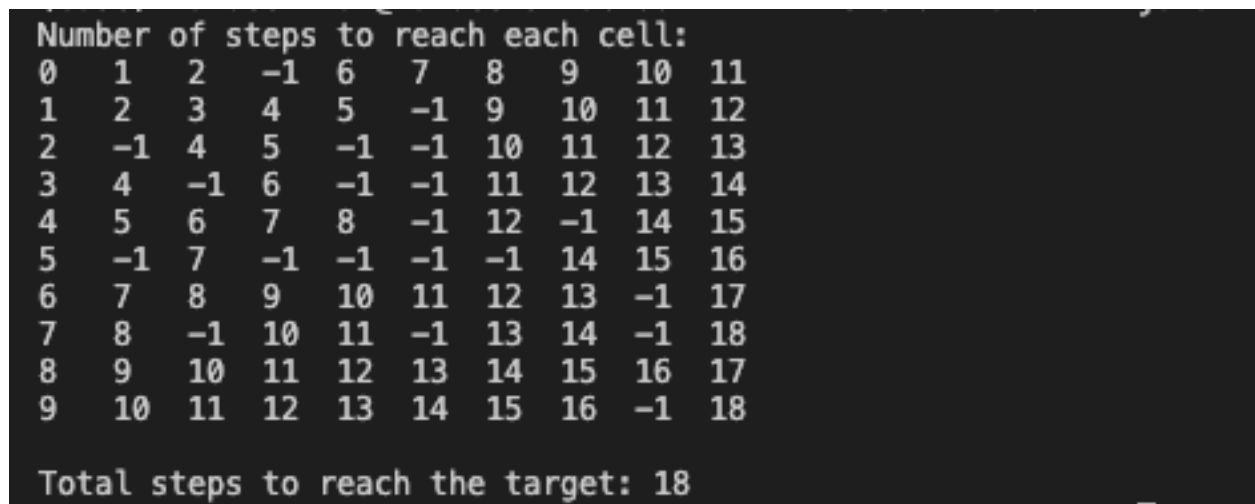


Figure: Running Extension1

The output of the Extension1 class provides valuable information related to the purpose. It displays the path found, indicating its length, and also reports the number of cells explored during the search process. This information allows us to evaluate the effectiveness of the A\* search algorithm with the heuristic in finding the shortest path, while considering the additional costs imposed by different cell types. By examining the output, we can determine the efficiency of the algorithm in terms of the path length and the number of cells explored. The fact that a path is found indicates that the algorithm is capable of navigating through the maze and reaching the target cell. Additionally, the number of cells explored provides insight into the algorithm's exploration efficiency and the effectiveness of the heuristic in guiding the search. Therefore, the

Extension1 class and its output align closely with the purpose of implementing an extended pathfinding algorithm that considers different cell types and finds the shortest path while taking into account their associated costs.

Note: These videos are combinedly made for Extension 1 and 2.

### Extension 3:

The changes implemented to the existing Maze Search application aimed to introduce certain mechanics similar to those seen in the Micro Mouse game. One of the primary modifications included the introduction of different types of cells. This was achieved by adding an enumeration called `CellType`, which defined possible cell states such as `FREE`, `OBSTACLE`, and others. This allowed for the creation of different environmental conditions within the maze, thus making the pathfinding problem more complex and closer to real-world scenarios.



-

Figure: AStar Video



Figure: BFS Video



Figure: DFS Video

The Cell class was also expanded to hold additional information about its state. This included attributes such as visited, prev, type, and distanceFromStart. These attributes facilitated the pathfinding process by keeping track of the cell's history, its type, and its distance from the starting point, which was crucial for determining the shortest path. The Maze class was adjusted

to generate a grid of cells with varying types based on a provided density parameter. This facilitated the creation of mazes with different levels of difficulty, enabling more robust testing of the search algorithms. In terms of the search algorithms, we made certain changes to the AbstractMazeSearch class and its subclasses to accommodate these new features. The search algorithms used the getNeighbors method from the Maze class to explore the surrounding cells. This method was adjusted to account for the new cell types and to prevent the algorithm from exploring cells marked as OBSTACLE.

Moreover, the AbstractMazeSearch class now also calculates the shortest path by considering the direction of movement. This is an important feature to simulate the Micro Mouse game dynamics where going straight is faster than turning. In this context, the shortest path is not only determined by the number of cells traversed, but also by the direction changes required.

Lastly, the MazeSearchDisplay class was updated to visually represent the different types of cells and the path taken by the search algorithm. This helps to understand the working of the implemented search algorithms visually and makes it easier to debug and optimize them. In conclusion, these changes have made the Maze Search application more dynamic, capable of simulating more complex scenarios, and better suited for pathfinding problems where the path's direction of travel plays a significant role.

### **Reflection:**

Working on a project is never a solo journey! It requires collaboration, communication, and teamwork to make something great. I had the pleasure of working with a fantastic group of people like Ahmed, Zaynab, Kashef, and Nafis on this project, including the amazing teaching assistants Nafis and instructor Professor Max Bender, who provided us with their expertise,

guidance, and support. In this project, a PriorityQueue was implemented using a data structure called a Heap. A heap is a specialized tree-based data structure that satisfies the heap property. In a min heap, for example, each parent node is less than or equal to its child node(s) and the key of the root node is the smallest among all other nodes. The heap data structure is very efficient for operations like insertion and removal, making it ideal for implementing a PriorityQueue. The PriorityQueue, which is essentially a min heap, was a cornerstone of our A\* pathfinding algorithm in this project. In the A\* algorithm, each node has a cost associated with it, which is the sum of the actual cost to reach that node from the start and an estimated cost to reach the target from that node, calculated using a heuristic function. The PriorityQueue was used to always select the node with the smallest cost for examination, ensuring an efficient search for the shortest path. The integration of the Heap as a PriorityQueue in the A\* algorithm was crucial for maintaining high performance and ensuring that the most promising nodes were evaluated first. The efficiency of heap operations allowed for the quick insertion and removal of nodes in the PriorityQueue, resulting in an effective and time-efficient pathfinding process.

**Sources, imported libraries, and collaborators are cited:**

From my knowledge, there should not be any required sources or collaborators. But we can say, we got our instructions from CS231 Project Manual:

<https://cs.colby.edu/courses/S23/cs231B/projects/project7/project7.html>

Except this, while working on the classes, I tried to import “java.util.ArrayList”, “java.util.List”, “java.util.Scanner”, “java.util.Collections”, “java.util.Random,” “java.awt.\*.”