# Project 1: In Space, No One Can Hear You Shout Fire

**Ship Layout:**

**For the Ship layout (Grid) I have use different symbol for each thing.**

**1. # - For the indicate block cell in ship**

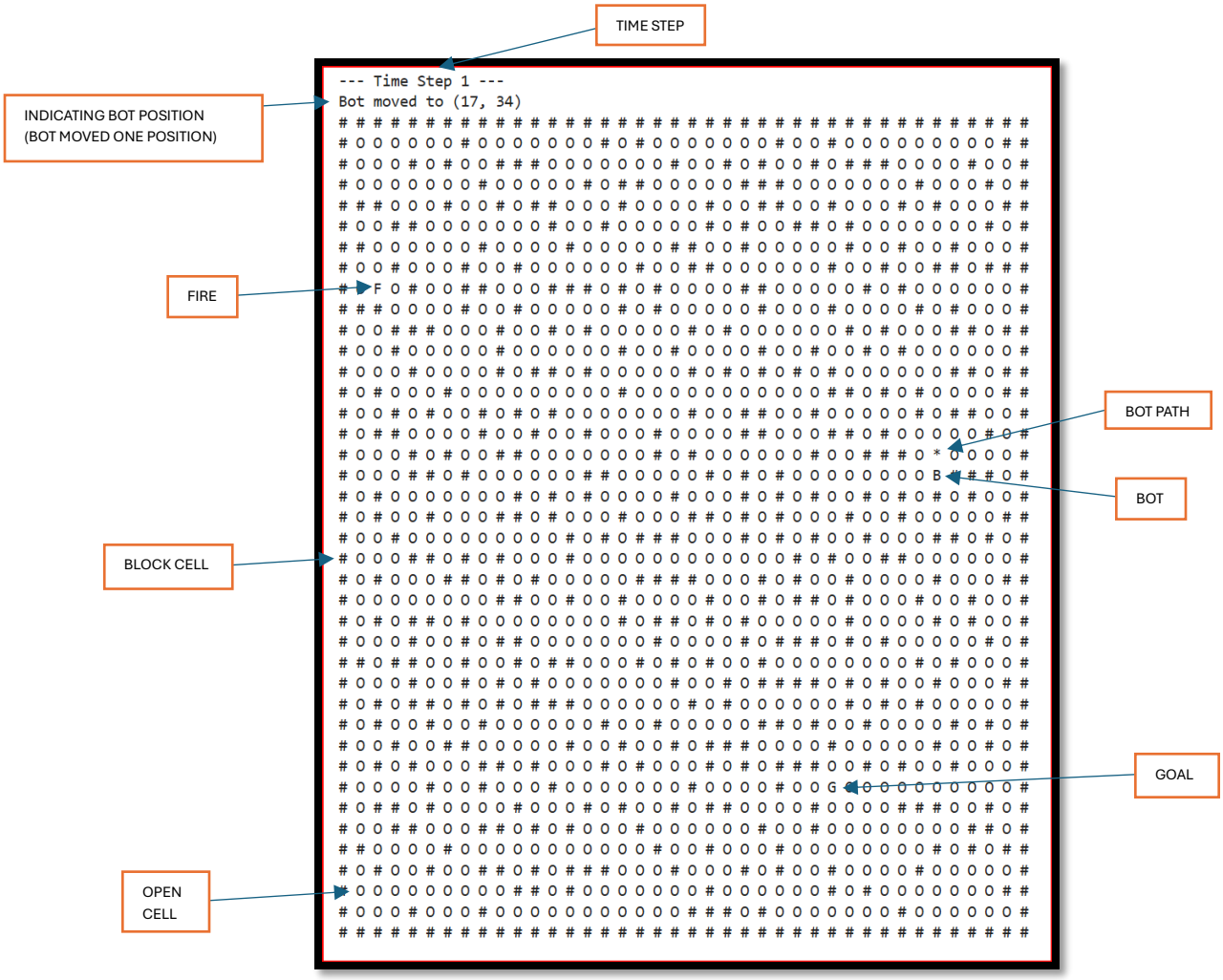
**2. O - For the indicate open cell in ship**

**3. F - For the indicate fire cell in ship**

**4. B - For the indicate Bot cell in ship**

**5. G - For the indicate Goal (Button) cell in ship**

**6. * - For the indicate bot path in ship**



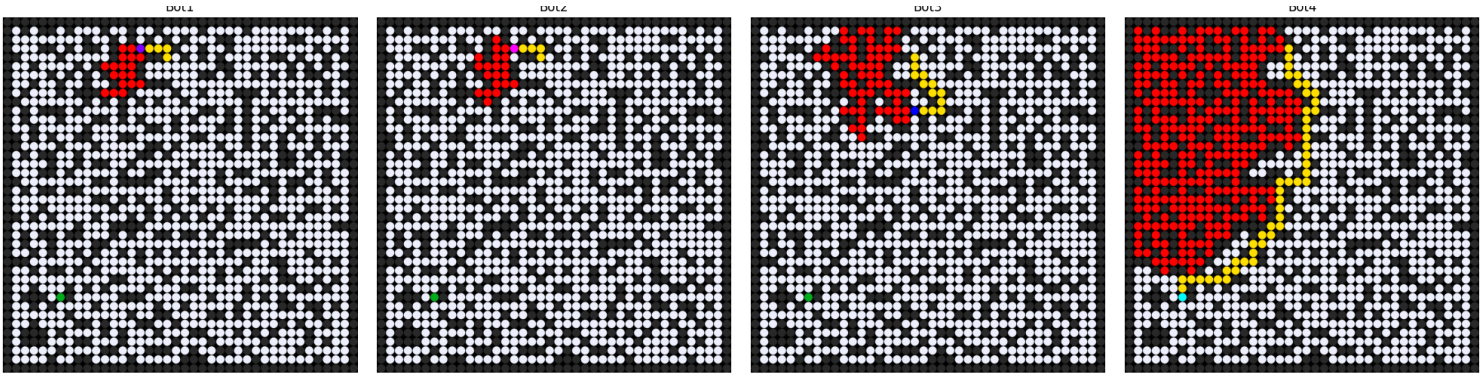
For the Figure I have use matplotlib.pyplot library to draw proper figure of above Ship (Grid).

In this Figure I have given Four Different Color for Bot because by this we can identify bot easily.

BOT 1: Indigo
BOT 2: Magenta
BOT 3: Blue
BOT 4: Cyan

Apart from that I have use other color for represent the other elements of Ship (Grid).

Fire: Red
Goal (Button): Green
Bot Path Tracking: Yellow
Blocked Cell: Black
Open Cell: White



**This Figure is Simulation 92 Q 0.60**

## Analysis Of Bots

## Bot 1:

In Bot 1 I have implemented BFS (Breadth First Search) Algorithm to determine the shortest path from its staring position to the goal in grid-based environment. BFS is an optimal for unweighted graphs, ensuring that the shortest path is always found if one exists. This algo work by exploring nodes level by level, starting from the initial position and visiting all neighboring nodes before moving further outward. This is guarantees that the first-time bot reached to goal, the path taken is the shortest possible route.
The algorithm also uses a queue to storing positions to explore and a visited set to prevent revisiting same nodes, thereby optimizing efficiency. As the bot moves along the identified path, it updated the grid, marking its previous position and checking for potential fire spread. However, This Algo (BFS) does not account for dynamic change like fire spreading which happened with each time step, which can cause the bot to follow a path that may later become inaccessible, because of that Bot 1 failure possibility is higher compared to other boats.

## Bot 2:

In Bot 2 I have implemented two algorithms, one is from our Bot 1 and the other one is A(A star) search algorithm to determine the optimal path to goal in grid environments. BFS is used as a fallback method, while A* is the primary algorithms for navigation.
A* search is a well-informed search algorithm that combines the benefits of two algorithms, BFS and Dijkstra's, by using heuristic function to estimate the cost from the current position to the goal. The heuristic used in bot2 in the Manhattan distance, which calculate the absolute difference the row and column coordinate of the current position of the goal. This heuristic helps prioritize paths thar are more likely to lead to the goal faster, making A* more efficient than BFS in terms of computational time and path optimization.
In Bot 2, A* maintains an open list where nodes are sorted based on their total estimated cost, the cost is determined as the sum of the actual cost from the start position and the heuristic estimate to the goal. Nodes are explored in an order that minimizes this total cost ensuring that an optimal and efficient path.

## Bot 3:

In Bot 3 I have implemented an additional safety mechanism, the ability to avoid fire adjacent cell when computing the path. It retains the A* search algorithm and BFS but adds an enhanced safety features that help to prevent the bot from moving near fire whenever possible. This makes Bot 3 more resilient in dynamic environment where the fire spreads unpredictable with time step.
Same as Bot 2, in Bot 3 I have prioritizes A* for path finding due to its efficiency in finding the optimal route. However, it Introduces a new **is_safe** function that evaluate whether a given cell is safe for moving or not. This function ensure that the bot not only avoid fire but also avoid dangerous cells adjacent to fire unless absolutely necessary, By doing so, Bot 3 significantly reduces the risk of getting trapped in mid route due to fire expansion.

## Bot 4:

Most importantly, the Initialization (__init__) component is required. As I initialize the bot, I provide it with a certain starting position, a destination point, and the ship grid, which is the map it will utilize for navigation. The bot further includes a risk_tolerance parameter that merely indicates the acceptable risk under fire. In case the estimated risk is greater than this value, the bot will not take a particular path. It's like, "I'll take a little risk, but not too much."

Next, the Heuristic Function (or the heuristic) determines the most advantageous path for the bot. It considers two significant elements: the distance to the goal, determined by the Manhattan distance, which is determined by summing up the row and column differences, and the respective accompanying fire hazard at the cell. The algorithm uses these two parameters together to find the optimum cell to move. The algorithm may avoid a cell that is close to the destination but has a high fire risk if the risk is determined to be too great.

And finally, we have the Fire Risk Calculation (calculate_fire_risk). Here, the bot calculates how risky a cell is because of fire. The system looks at all the surrounding cells within a specific radius and calculates how many active fires are there. Quite cleverly, it assigns more risk to fires that are nearer and less risk to fires that are farther away. Thus, the bot employs a "risk decay" system to calculate the level of danger that each fire represents. For example, a fire right next to the cell is super risky, but a fire three cells away is only a little risky.

Then there is the Multi-Step Lookahead (multi_step_lookahead) mechanism. This method works similarly to the mental process of foresight demonstrated by the bot. It does not just evaluate the very next action; instead, it considers several possible actions ahead to determine the long-term safety of the path.

For example, when the bot moves into a specific cell, can it get trapped in fire at a later point? This function solves this question by scanning nearby cells systematically and evaluating their corresponding fire risks. If the path seems safe, the bot gives itself a high "survivability score."

The main brain of the bot is the A Resilient Pathfinding (a_star_resilient). This is where the bot uses the A algorithm (one of the most popular pathfinding algorithms) to find the best path to the goal. But it's not just about finding the shortest path—it's about finding the safest path. The bot uses the heuristic function to guide its search and the multi-step lookahead to make sure the path won't lead to fire in the future. The algorithm continues to investigate cells with the lowest f_score, a combination of the cost incurred to reach the cell and the approximated cost to reach the goal, until it arrives at the target. Once it has found the target, the bot employs the Path Reconstruction (reconstruct_path) function to construct the path from the destination to the source. Essentially, it reverses its move through the cells it has visited and inverts the order of the sequence to obtain the final path. Lastly, the Move Function (move) is where the bot navigates the grid. It continually determines the safest route with a_star_resilient, updates its position accordingly, and determines whether it has reached the target or if something has gone.Successfully reaching the goal is success for the bot; failure to reach the goal sees it react to the failure and shut down.

## **Pseudocode** version of My Bot 4

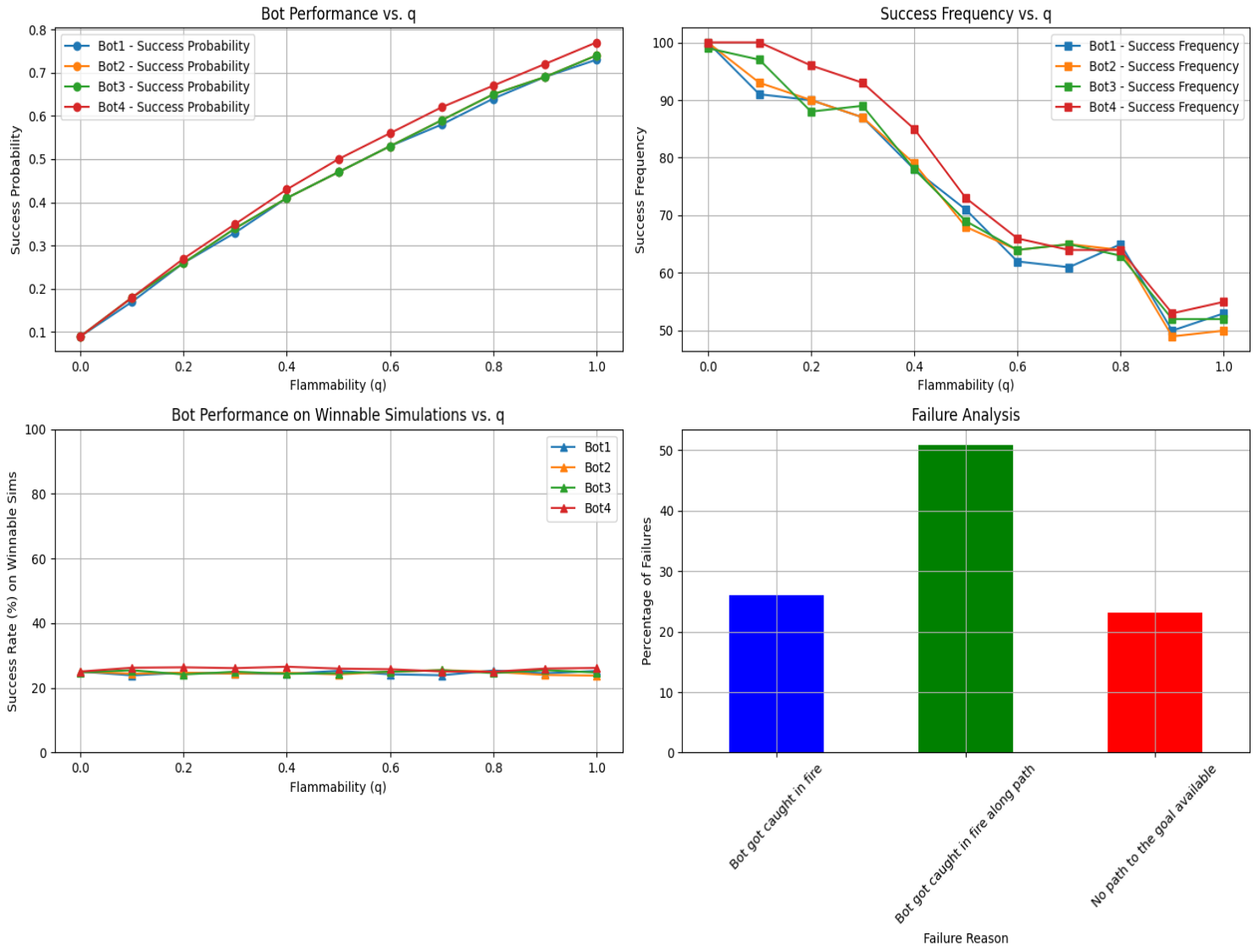
```
Class Bot4:
  Initialize(start_position, goal_position, ship):
    Set the bot's current position to start_position
    Set the bot's goal position to goal_position
    Set the ship grid (ship)
    Set the risk_tolerance to 0.2
  Function heuristic(cell):
    Calculate Manhattan distance from the current cell to goal position
    Calculate fire risk at the current cell
    Return the sum of the distance and fire risk scaled by the risk_tolerance
  Function calculate_fire_risk(cell, max_radius = 3):
    Initialize risk_decay for different distance levels (1: 1.0, 2: 0.8, 3: 0.6)
    Initialize fire_count to 0
    For each cell within the range of max_radius around the current cell:
      If the neighboring cell is within bounds and contains fire ('F'):
        Increase fire_count based on the risk_decay values
    Return the total fire risk
  Function multi_step_lookahead(cell, depth = 3):
    If depth is 0 or the current cell contains fire ('F'):
      Return 0 (no survivability)
    Initialize survivability to 1
    For each neighboring cell of the current cell:
      If the neighboring cell is open ('O'), goal ('G'), or previously visited ('*'):
        Calculate fire risk of the neighbor
        If fire risk is within the acceptable limit (less than risk_tolerance):
          Recursively call multi_step_lookahead on the neighbor and reduce depth by 1
          Add survivability of the future steps
```

```
    Return survivability

  Function a_star_resilient():

    Initialize open_list as an empty priority queue

    Add the start position to the open_list with a priority of 0

    Initialize dictionaries for came_from, g_score, and f_score

    Set the g_score of the start position to 0

    Set the f_score of the start position to the heuristic value of the start position

    While open_list is not empty:

      Pop the cell with the lowest f_score from open_list

      If the current cell is the goal:

        Return the reconstructed path from the start to the goal

      For each neighbor of the current cell:

        If the neighbor is not on fire ('F') and is open ('O'), goal ('G'), or previously visited ('*'):

          Calculate tentative g_score (cost to move to neighbor)

          Calculate the survivability using multi-step lookahead

          If survivability is within the acceptable limit:

            If the tentative g_score is better than any previous score for the neighbor:

              Update g_score and f_score for the neighbor

              Add the neighbor to open_list with the updated priority

    Return failure (if no path is found)

  Function move():

    Initialize time_step to 0

    Initialize an empty list to track the traversed path

    While true:

      Call a_star_resilient to get the next path

      If no path is found:

        Print "No safe path to the goal!"

        Return failure

      Get the next step from the path
```

```
        Mark the bot's old position as revisitable ('*')
        Move the bot to the new position ('B')
        Increment time_step
        Print the new position and time_step
        Display the updated grid
        If the bot reaches the goal:
            Print "The bot reached the goal!"
            Return success
        Spread fire on the grid and check if the goal is on fire
        If the goal is on fire:
            Print "The goal is on fire!"
            Return failure
        Check if the bot's current position has caught fire
        If the bot is on fire:
            Print "The bot got caught in the fire!"
            Return failure
Function reconstruct_path(came_from, current):
    Initialize an empty list for path
    While current is not None:
        Add current to the path
        Set current to came_from[current]
    Reverse and return the path
```

The simulation store data in CSV files: “Bot Results.csv” which records bot success probabilities and average successes, while “Bot Failure Reasons.csv” store bot failure reasons, time steps, and simulation conditions. After storing the data, I have generated four graphs,

1. Bot Performance vs. Flammability (q)
2. Success Frequency vs. q
3. Bot Performance on Winnable Simulations vs. q
4. Failure Analysis




The Provide figure is a comparative performance analysis of my four different bots under different conditions of flammability (q). In first two graphs we can show that the Bot 4 performance is better than other bots, as we have given more check and reconstruct path, so Bot 4 winnable situation is higher compared to other Bots.

Although some taking predictive measures and keeping track of alternative decisions might have delayed failure in some cases, the inherent randomness and speed of fire spread in case of the higher value of q often make it impossible for a bot to avoid the fire completely. That's why a lower tolerance to risk, i.e., a bot that is a less risky taker, performs better as instead of moving towards the shortest or optimal cells or path; it chooses the one that is less risky. However, even with such improvements, certain scenarios with a higher value of q still result in failure due to unpredictable fire patterns that close off all viable paths.

## The Ideal Bot

Obviously, neither of our Bot algorithms are ideal. An ideal bot would be something that can integrate both real-time awareness of the available paths and has the predictive capability of the fire spread. A way to significantly enhance the bot's performance, making it almost ideal, would be if the bot could save information on all the paths it could take and all the possible ways it can take, taking into consideration what it would do in case each of the combinations of cells was on fire. However, it would be computationally impossible and costly as it requires the model to learn almost all the possible paths and combinations. Another way to construct an ideal bot would be to use reinforcement learning. A bot that has learned from experience and improved over time is more likely to make the best decisions. Using algorithms like deep Q-learning, the bot could approximate the best action in any given state by using trained neural networks to predict the rewards of different paths. Additionally, training the bot in a simulated environment with diverse ship layouts and fire spread patterns would allow it to generalize across various scenarios, increasing its adaptability. Combining this approach with predictive modeling, the bot could incorporate not only the current fire conditions, but also probabilistic forecasts of fire spread, making informed decisions in real-time and adjusting its strategies based on what it has learned, making it an ideal bot.