AI Project 1

Krishan Chaudhary & Zuhayr Rashid

February 2025

Contents

1 Introduction

This project involves designing four different bots to navigate a maze with the goal of reaching a button while avoiding a fire that spreads with various different rates of spreading. We have chosen Python as the programming language for this project because it is high level and it has easy to use in-built data structures and libraries for data structures, plotting, and copying objects.

We are working with 40×40 grids.

For the purposes of visualization:

- Black squares represent closed cells (walls)
- White squares represent open cells (cells the bot can move into)
- Blue square represents the cell the bot occupies
- Green square represents the position of the button

We tested four different approaches, represented with four different bots. The goal is to plan and take a path from the starting position of the bot to the position of the button, before the bot itself or the button catches on fire.

2 Representing the Problem and Global Choices

This section details the design choices we made to represent the problem as efficiently as possible.

We generate a hashmap/dictionary called info to represent information in the ship, it contains the following items

- ship: 2D array representation of ship where each spot is an integer corresponding to open cell, closed cell, bot, button, or fire
- button: tuple containing button position (row, col)
- bot: tuple containing initial bot position (row, col)
- fire: tuple containing initial fire position (row, col)
- fire_q : double-ended queue containing tuples of open cell positions adjacent to initial fire cell

Hashmaps and Sets.

In the maze generation process, we use sets and hashmaps for O(1) access and lookup time for important variables while initializing our 2D grid, such as closed neighbors and deadends.

Fire Progression 3D Array.

We have a separate method to generate a specific fire progression based on an initial ship and q value, which returns a 3D array called fire_prog, where fire_prog[i] = 2D array representing ship with fire spread at time i. The length of the array is determined by the number of time units it takes until the button is consumed. This reduces the amount of fire progression steps we need to compute. This is also helpful for standardizing the data generation and testing process, so bots are fairly competing on not just the same board, but also the same exact fire spread across time.

We also use a double ended queue to keep track at cells that are currently fire adjacent and updating that from time step to time step, which is more efficient than looping through the entire map each time to identify fire adjacent cells. Python's deque (double ended queue) also enables efficient O(1) access from both the front and the back which is useful for simulating the BFS-like fire spread.

NumPy for Vectorized Array Operations.

When it comes to performing operations on 2D and 3D arrays, we use NumPy, which vectorizes for efficiency, resulting in better performance than standard nested arrays. Some of the use cases include generation of the fire progression 3D array and computing a risk map for Bot 4.

3 Bot 1

This section details the design and implementation of Bot 1. We focus on the algorithm, choices made to enhance speed and efficiency, and its shortcomings.

3.1 Design

This bot plans the shortest path to the button, avoiding the initial fire cell, and then executes that plan. The spread of the fire is ignored by the bot.

This bot uses the A* algorithm to calculate the shortest path from the bot's initial position to the goal. The heuristic it uses to compute this is Manhattan Distance:

$$d_{\text{Manhattan}}(P, Q) = |x_2 - x_1| + |y_2 - y_1|$$

where $P = (x_1, y_1)$ and $Q = (x_2, y_2)$.

P is a specific cell.

Q is fixed to be the button cell.

We chose this heuristic, rather than Pythagorean distance because the bot can only move up, down, left or right, making it more applicable to navigating a grid where diagonal movement is not allowed.

3.2 Choices to Improve Speed and Efficiency

This section discusses choices to ensure Bot 1 was efficient.

A* Search Algorithm.

We chose A* over other search algorithms such as Breadth-First-Search, Depth-First-Search, Dijkstra's Algorithm, and Uniform Cost Search because A* combines the benefits of a uniform cost search and greedy search by leveraging a heuristic. This allows A* to prioritize paths that are likely to lead to the goal, improving both efficiency and speed compared to the other algorithms, especially in large search spaces.

Only Add, No Updates.

Within our implementation of A*, we also made the choice to only add new nodes (cells with their associated costs) to the fringe, instead of the traditional add or update approach. We did this because updating the fringe would be an $O(n)$ operation that would require traversing it to find the cell that needs to be updated. Instead, we relied on the min-heap invariant property of `heapq` in Python, which prioritizes smaller values in the heap. There is a tradeoff because the size of the heap will be larger due to our choice to only add, not update, but this avoids the costly operation of reordering elements in the heap during updates.

Order of Elements in Priority Queue Tuple.

We modified the typical order of information within an item on the fringe (priority queue). Normally, when adding elements to the fringe in A* or similar algorithms, we would store each element as a tuple in the form of (node, heuristic(node)). However, if we used this structure, we would need to write extra code to ensure that the heap compares the priority based on the second element (the heuristic value) rather than the first one.

To simplify this, we reversed the order of the tuple to (heuristic(node), node). This allows us to take advantage of Python's default heap behavior, which prioritizes elements based on the first element in the tuple. In this case, the heap will prioritize nodes with lower heuristic values, which aligns with how the algorithm should explore nodes, thus eliminating the need for additional code to handle priority comparisons.

3.3 Analysis of Failure

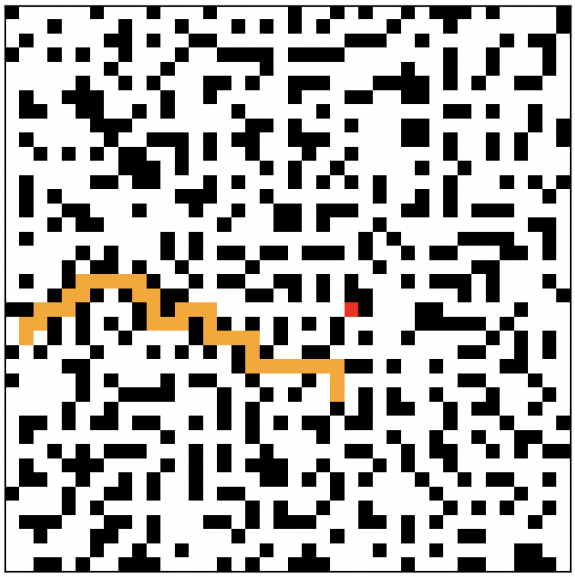
When Bot 1 fails, it is because of a few reasons:

1. **Does not factor in fire**: Since the bot does not factor in the position of the fire into its algorithm, it has no reason to avoid cells that are at a high risk to catch on fire. It has no hesitation taking paths that contain cells that will likely soon be on fire.

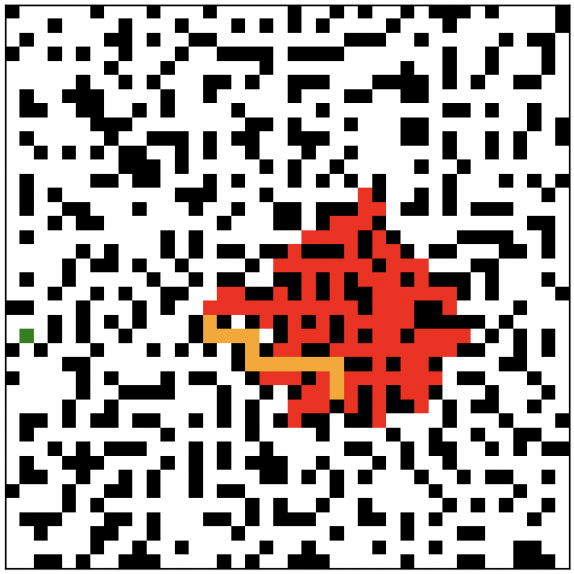
2. **Only prioritizes distance**: Because Bot 1 uses A*, it will optimally find the shortest path, but this will prevent it from considering alternative paths that are slightly longer but less risky based on the fire spread.

3. **Static approach**: Bot 1 does not account for changes to the ship based on actual fire spread after it makes each move.

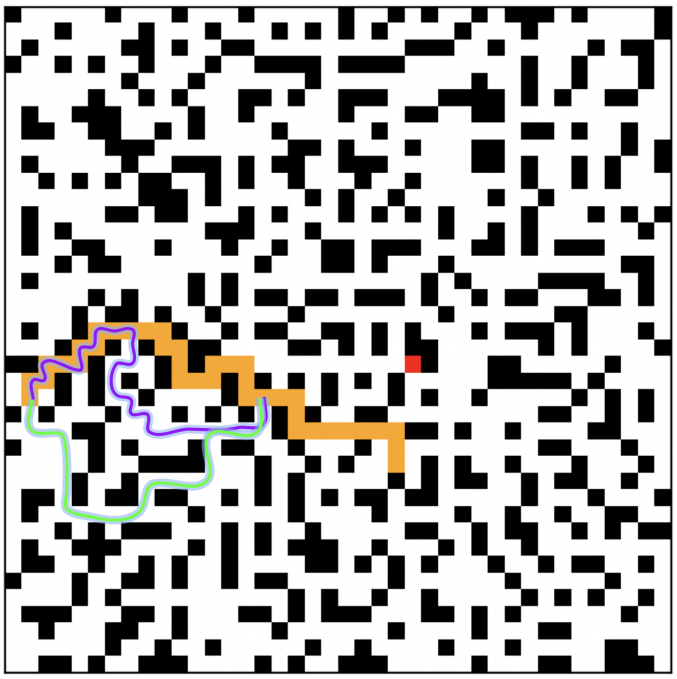
Below is a visual sample that depicts these ideas, with q = 0.65.



Bot 1 Planned Path



Bot 1: One step before Failure



Potential Alternate Paths

The above figures examine a a specific case that highlights the shortcomings of Bot 1. Once it plans the shortest path, it commits to it even without considering its risk. Because of this, it ends up going down a path that is a lot riskier due to its proximity to the fire. The last image depicts potential alternate paths it could have taken at a specific juncture. Bot 1 chose to go up because that follows the shortest overall path to the goal. However, it would have been better served by going down and choosing longer paths that were further from the fire spread.

4 Bot 2

This section details the design and implementation of Bot 2. We focus on the algorithm, choices made to enhance speed and efficiency, and its shortcomings.

4.1 Design

At every time step, this bot bot re-plans the shortest path to the button, avoiding the current fire cells, and then executes the next step in that plan.

Bot 2 also uses the same A* algorithm used by Bot 1 with the same heuristic and optimizations. The only difference is that Bot 2 calls it after every move, and accounts for all current cells on fire.

4.2 Choices to Improve Speed and Efficiency

The global choices as well as the choices to improve speed and efficiency described in Bot 1 are extended to Bot 2. These can be concisely summed up as:

- leveraging fire progression 3D array to minimize recomputation
- choice of A* and its specific implementation
- early stopping to minimize computation

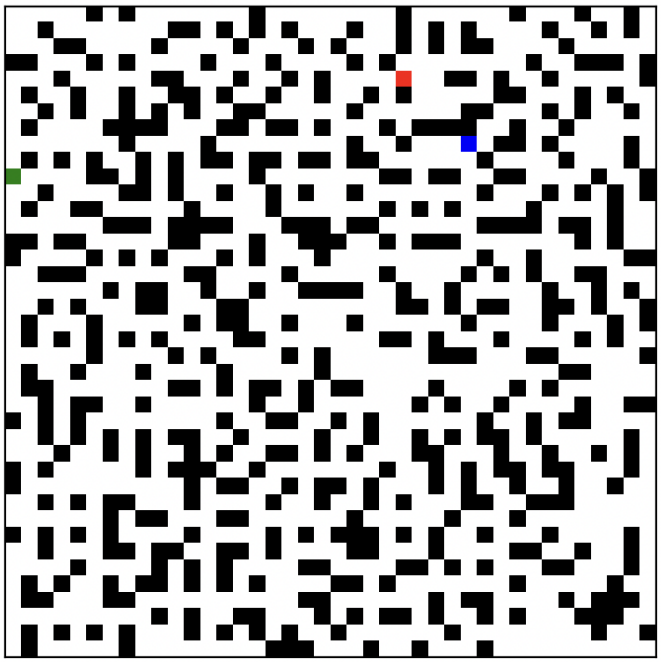
4.3 Analysis of Failure

Below is a visual sample that depicts these ideas, with q = 0.65.

1. **Backtracking is suboptimal**: if a bot recalculates at every step, there are scenarios in which it will need to backtrack in order to go around the fire. In that case, it was more efficient to take that path from the beginning, rather than backtrack once the fire cuts the path off.

2. **Does not incorporate risk of fire in future**: This approach only factors in the current state of the fire, and it does not assign any risk to fire-adjacent open cells, making its planned paths still prone to high risk of fire spreading to them.

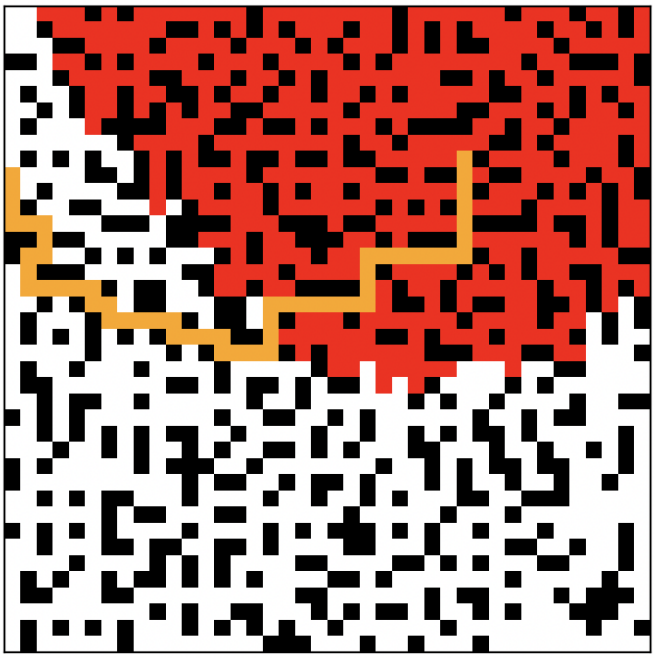
Below is a visual sample that depicts these ideas, with q = 0.65.



Bot 2 Failure Case

Bot 2 Initial Plan

Bot 2 Revised Plan



Potential Alternate Path

The above figures examine a a specific case that highlights the shortcomings of Bot 2. Even though it recallibrates to a path that avoids the fire, it is too late for it to successfully escape the spread of the fire. This pitfall comes from the fact that advancing down a dangerous path early on and then recalculating only when the fire makes the path absolutely infeasible is still very risky.

5 Bot 3

This section details the design and implementation of Bot 3. We focus on the algorithm, choices made to enhance speed and efficiency, and its shortcomings.

5.1 Design

At every time step, the bot re-plans the shortest path to the button, avoiding the current fire cells and any cells adjacent to current fire cells, if possible, then executes the next step in that plan. If there is no such path, it plans the shortest path based only on current fire cells, then executes the next step in that plan.

Bot 3 also uses the same A* algorithm used by Bot 1 with the same heuristic and optimizations. The main difference in the approach is recalculating a path after each time step and also assuming worst case fire spread while recalculating.

5.2 Choices to Improve Speed and Efficiency

The global choices as well as the choices to improve speed and efficiency described in Bot 1 are extended to Bot 3. These can be concisely summed up as:

- leveraging fire progression 3D array to minimize recomputation
- choice of A* and its specific implementation
- early stopping to minimize computation

5.3 Analysis of Failure

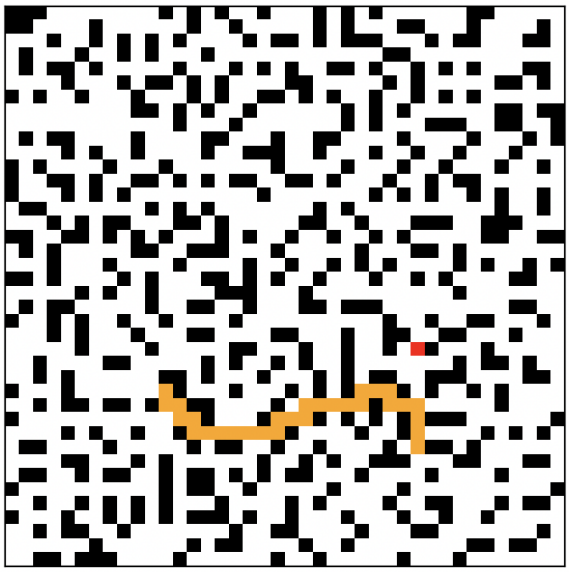
When Bot 3 fails, it is because of the following:

1. **Inflation Map Forces Much Longer Paths**: if a bot's priority is to avoid the inflation map of the current fire spread, it may overestimate the risk level for a certain cell, and rerouting to a significantly longer path that is unsafe in the long run, or causes the fire to reach the button before the bot.

2. **Short sighted**: While this approach looks one time unit into the future, it is possible that many steps on its planned path are infeasible two to three time steps into the future, forcing it to recalculate paths often and backtrack.

   Below is a visual sample that depicts these ideas, with q = 0.15.

Bot 3 Failure Case



Bot 3 Initial Path

The figures illustrate a specific case that highlights the shortcomings of Bot 3. Initially, it uses A* to plan the shortest path to the goal. However, after 23 time units, the inflation map causes it to recalculate its path because one of the cells in its planned route is now adjacent to the current fire.

A more intelligent bot would have recognized that switching to a longer path when the fire is close to the goal (button) is riskier than sticking with a shorter path, even if it passes near the fire. The bot should have weighed the risk of the longer path and taken a calculated risk on the shorter path, especially given the fire's proximity and the low q value.

6 Bot 4

The final bot, Bot 4, is analyzed in this section. We discuss choices made to improve its functionality and its overall approach.

6.1 Design

Bot 4 is an improvement upon the other bots, primarily building off of the idea in Bot 2. In addition to Manhattan distance heuristic, the bot calculates risk, which is added to the Manhattan distance to create total cost. Risk is defined as $\frac{10 \cdot (1-q)}{\text{Manhattan distance to nearest fire cell}}$ in order to properly scale. We tested with different constant coefficients for the numerator and 10 worked the best.

We create a 40 x 40 risk map by assigning each cell in the grid a risk value using the following formula:

$$\text{risk}(r,c) = \begin{cases} \infty, & \text{if ship}(r,c) = \text{wall or ship}(r,c) = \text{fire} \\ \frac{10 \cdot (1-q)}{d_{\min}(r,c)}, & \text{otherwise} \end{cases}$$

where $d_{\min}(r,c)$ is the Manhattan distance to the nearest fire cell, with

$$d_{\min}(r,c) = \min_{(r_f,c_f) \in F} |r - r_f| + |c - c_f|$$

where F is the set of all tuples representing fire cell positions.

The intuition behind this is that the closer a cell is to a fire cell, the higher its risk value is. The reasoning behind the $(1-q)$ component is that the bot should take longer safer paths for low q because it can afford to do so, but for high q, it should take gambles because longer paths will be dramatically riskier.

$d_{Manhattan}(cell, button)$ uses the formula explained in Bot 1.

The revised A* method incorporates this information into its heuristic as follows:

$$Cost(cell) = d_{Manhattan}(cell, button) + risk(cell)$$

Bot 4 plans out its path using an A* implementation that incorporates this modified heuristic. At each step, it reruns A* based on the current state of the ship, and readjusts its best path accordingly.

At every time step, this bot runs this A*, creates a path to the button, and executes the next step in that plan.

Bot 2 also uses the same A* algorithm used by Bot 1 with the same heuristic and optimizations. The only difference is that Bot 2 calls it after every move, and accounts for all current cells on fire.

6.2 Choices to Improve Speed and Efficiency

The global choices as well as the choices to improve speed and efficiency described in Bot 1 are extended to Bot 4. These can be concisely summed up as:

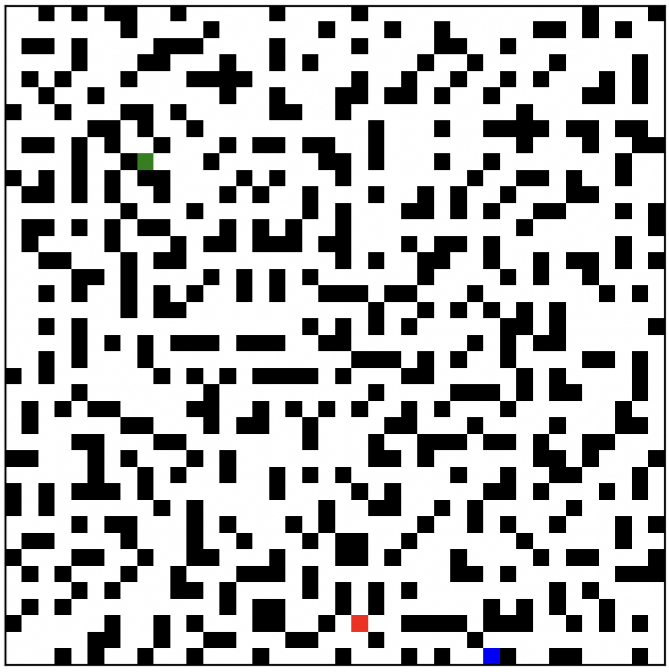
- leveraging fire progression 3D array to minimize recomputation
- choice of A* and its specific implementation
- early stopping to minimize computation
- using NumPy arrays for memory efficiency and time efficiency

6.3 Analysis of Failure

When Bot 4 fails, it is because of the following:

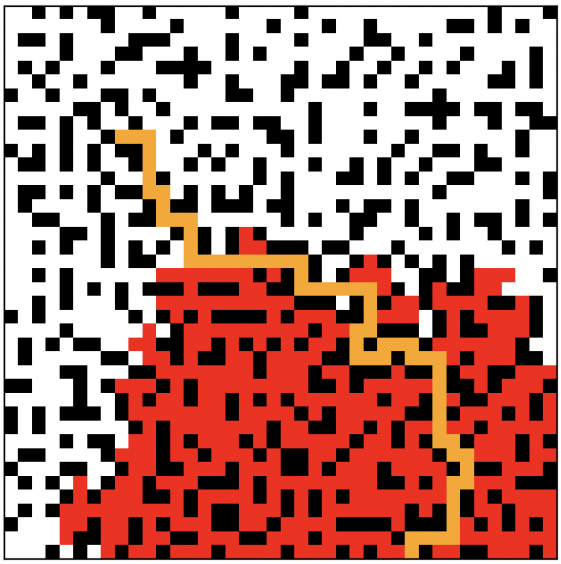
1. **Has to Backtrack if its Gambles do not pay off**: Bot 4 might think that a certain path closer to the fire is worth it because it is shorter, so it will take on the risk of passing closer to the fire for the reward of reaching to the button with a shorter path. However, if its gamble does not pay off it is forced to backtrack on its path and pursue a different path. It would have been better served to take the safer path originally instead of taking a risk and then reversing its actions to escape inevitable danger.

Below is a visual sample that depicts these ideas, with q = 0.35.



Bot 4 Failure Case

Attempted Path



Potential Alternate Path

These figures highlight a specific shortcoming of Bot 4. In the attempted path, we have circled instances where Bot 4 took a risk by trying to go on a shorter path to the button, but because of the fire spread when it started exploring those shorter paths, it ran into paths completely blocked off by the fire. It was then forced to backtrack and go back on its path, which eventually caused it to get consumed by the fire. If it had hypothetically gone on a safer route to begin with, it could have succeeded in this simulation. A smarter bot would somehow have an even more complex model or better model for weighing risk.

7 Data

We discuss data generation in this section, specifically the choices we made for generating it, and the actual results from the experiments performed.

7.1 Data Generation - Choices

Consistent Fire Progressions.

As mentioned in **Representing the Problem and Global Choices**, simulating a fire progression array was an important choice made to standardize the difficulty of the simulations. This way, the bots are competing on a fair playing field, and the results are not skewed due to variations in the way a fire spreads for a particular ship.

Text Files.

In order to save the results locally after running the tests on all the bots, we wrote the results to txt files for each bot in our test script, test.py. We then had a separate Python script, plot.py, that would read the results from the txt files and display graphs using Matplotlib.

Random Seed.

We used a constant random seed for reproducibility of the tests.

Early Termination.

In order to speed up the testing process, we added checks inside all the bots that would quit the path progression of the bot, if one of the following occurred:

- **Bot reaches goal** $\implies$ success
- **Bot catches on fire** $\implies$ failure
- **Button catches on fire** $\implies$ failure

We also only ran the bots only if the specific ship and fire progression based on q was winnable. This saved time by not running the bots on ships with unwinnable fire progressions.

7.2 Data Generation - Procedure

We create 20 distinct ships.

For each ship, we simulate 5 fire progressions at fixed increments of q:

q = 0.00, 0.05, 0.10, ... 1.00

So, Ship 1 has 5 specific, different fire progressions for q = 0.00.

And, so on.

It has 5 specific, different fire progressions for q = 0.05 to ensure our results are not skewed by unlikely fire progressions due to variance.

This is repeated for each of the 20 ships, and then we store the success rate of each bot for each q value. Overall, we have $20 \cdot 5 = 100$ experiments for each bot per q value.

$$\text{Success Rate of Bot } x \text{ at } q = \frac{\sum_{i=1}^{100} S_x(i,q)}{100}$$

$$\text{where } S_x(i,q) = \begin{cases} 1 & \text{if Bot } x \text{ succeeds on Ship } i \text{ at } q, \\ 0 & \text{if it fails.} \end{cases}$$

We also compute winnability of a ship at a specific q based on whether or not the bot could have succeeded given the specific fire progression. The winnability function leverages BFS and factors in the actual fire progression to determine if any path is feasible.

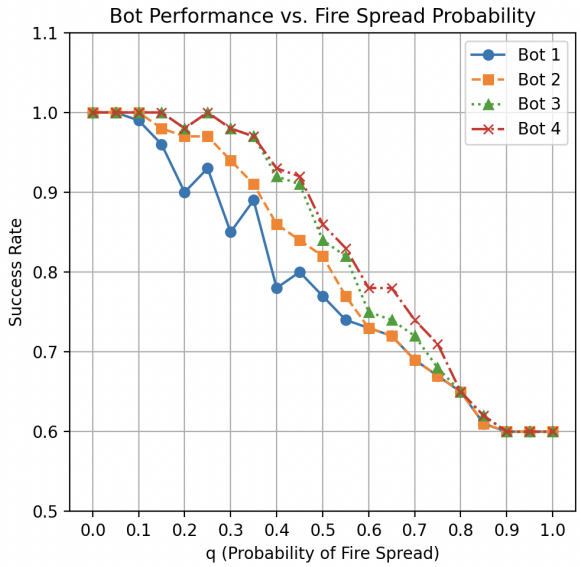
Let $w(q)$ = number of winnable ships for q.

$$\text{Adjusted Success Rate of Bot } x \text{ at } q = \frac{\sum_{i=1}^{100} S_x(i,q)}{w(q)}$$

$$\text{where } S_x(i,q) = \begin{cases} 1 & \text{if Bot } x \text{ succeeds on Ship } i \text{ at } q, \\ 0 & \text{if it fails.} \end{cases}$$

The numerator stays the same because the unwinnable simulations will never be won by the bot.

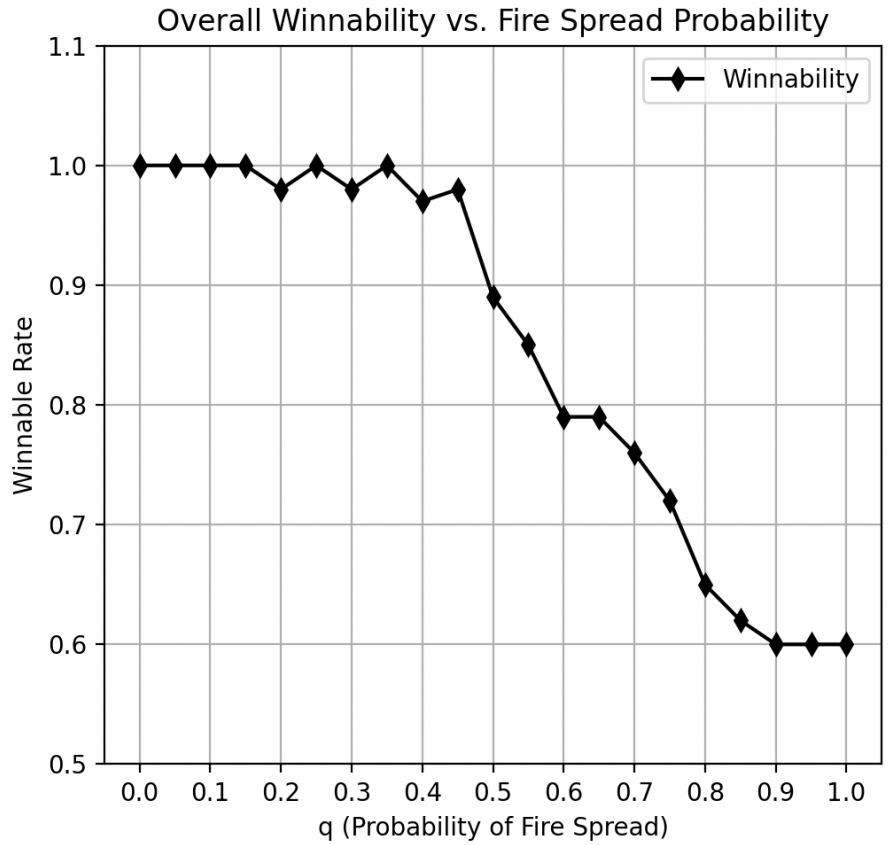
7.3 Data Generation - Results



This looks reasonable. All bots do well in cases when fire spread probability is low - $q \leq 0.15$, which makes sense. Planning shortest paths and executing them is very likely to succeed if the fire is not spreading fast enough to hinder the safety of those paths.

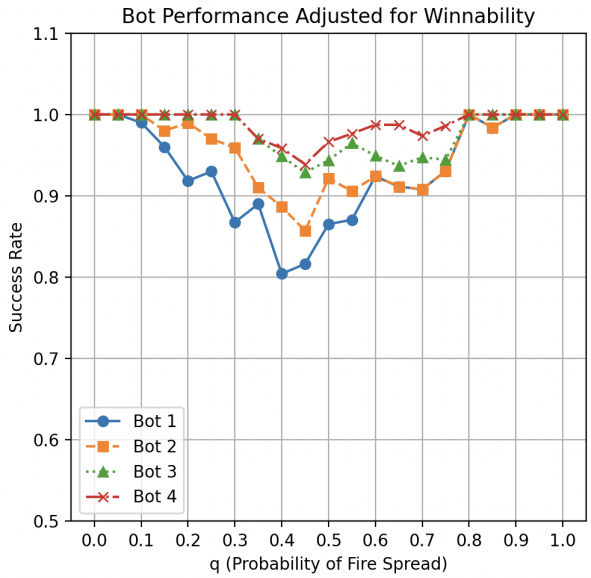
We see the greatest variability in success rates of the bots for medium q values - $0.15 \leq q \leq 0.75$. Bot 1's success rate is significantly lower than that of all the other bots. This makes sense, since ignoring the fire when the fire is likely to spread would likely cause the fire to interfere with the path. Bot 2 does slightly better than Bot 1 for these q values, which also makes sense because it is considering the fire somewhat and trying to avoid it after each turn. However Bot 2 still struggles because it only considers cells that are actively on fire and associates no risk with fire adjacent cells, which is suboptimal for medium q. Bot 3 does better than the other 2 bots because it does factor in risk 1 move into the future, and Bot 4 does even better than all three other bots because it implicitly factors in fire proximity into its A* heuristic and recalculates after each step.

For high q - $q \geq 0.75$, the bot success for all bots falls and levels out to around 0.6. This is reasonable since when the fire spread is almost certain, it is much more difficult for the bots to find a path that is safe enough and short enough to get to the button. Either the bot is lucky based on a ship set up where it is close to the button and the fire is far away, in which case it succeeds, or the fire is close enough that it will always be very difficult, if not impossible for the bot to win.



We computed winnability by defining a function that takes in the specific fire progression and the ship, and runs a BFS to check if any possible path from the start to the finish that does not involve backtracking on itself could have succeeded.

This looks reasonable. It makes sense that winnability is relatively high for low q values since there exist many safe paths for the bots to find. As q increases, it also makes sense that winnability significantly decreases since the number of possible safe paths decrease. As q approaches 1.0, the winnability rate levels out at 0.6. This is reflective of even less possible safe paths, and the idea that on average, for high q, only approximately 60% of maps are winnable. The other 40% have such difficult initial relative positions of button, bot, and fire that there is no way to win those simulations.



This looks reasonable. This graph changes our perception of the bots in the sense that all of them are actually doing quite well - consistently winning more than 80% of winnable simulations at various q values. It makes sense that all the bots succeed for low values of q for essentially all winnable maps because the fire does not really play a role. It barely interferes with planned paths.

It also makes sense that all the bots succeed for high values of q for essentially all winnable maps. This is because for really high q values, the failure of the bots is not due to poor path planning, but rather the fact that the fire spreads so fast that it is impossible to come up with a safe and short path before either the button or bot is consumed, regardless of strategy. If a map is winnable for high q values, it is probably because the position of the initial map has the bot and button close to each other and the fire is far away. In these winnable cases, all the bots will do well because they will take the shortest path to the button, and the fire is too far to play a role.

It makes sense that the success on winnbable maps is most variable for values of q that are neither high nor low. These maps have the most variance in terms of what the map could look like n turns into the future. Because of this variance, the differences between the success rates of the bots is most stark. Bot 1 does the most poorly because it ignores the fire, and the fire does have a reasonable chance of impeding its original one-shot path. Bot 2 does slightly better because it accounts for the current state of the fire and recalculates, but fails to consider risk of fire adjacent cells. Bot 3 does even better because it extrapolates risk 1 time unit into the future, and Bot 4 does the best out of the bots because it incorporates a risk factor into its heuristic instead of a binary inflation map, which can sometimes cause inefficient paths due to fear of fire spread.

8 Speculation on Ideal Bot

Here, we discuss some potential ideas for an ideal bot.

Ideal Bot Idea 1: Simulation Bot

This ideal bot could run 100s of simulations on a specific ship in a specific state. It could then extrapolate which cells are riskier in the future and how to plan a path to best avoid those. It would redo this type of simulation at each stage to develop a strong understanding of how to plan a path in accordance with the risk of each cell.

Ideal Bot Idea 2: Bot 4 Pro Max:

This ideal bot would be an improved version of Bot 4 that has a modified heuristic that uses a more complex model than the sum of factors like Manhattan distance to goal and manhattan distance to nearest fire cell multiplied by a constant to better assess risk of paths. It could be a model involving more than just linear functions of the variables. It would be something more complicated that better models risk.

Ideal Bot Idea 3: Simulation Path Bot

This ideal bot would do the following at each time step:

1. Examine every possible path from the bot to the goal.
2. For each path, simulate 100s of fire progressions starting with the actual fire progression at that time step and going into the future until success or failure.
3. Determine in what proportion of the simulations that path succeeds.
4. For each of the possible open spaces that the bot can move to determine which adjacent cell has the most corresponding paths that won the simulations, weighted by how likely each path is to succeed.
5. Move in the best direction based on the metric computed in Step 4.