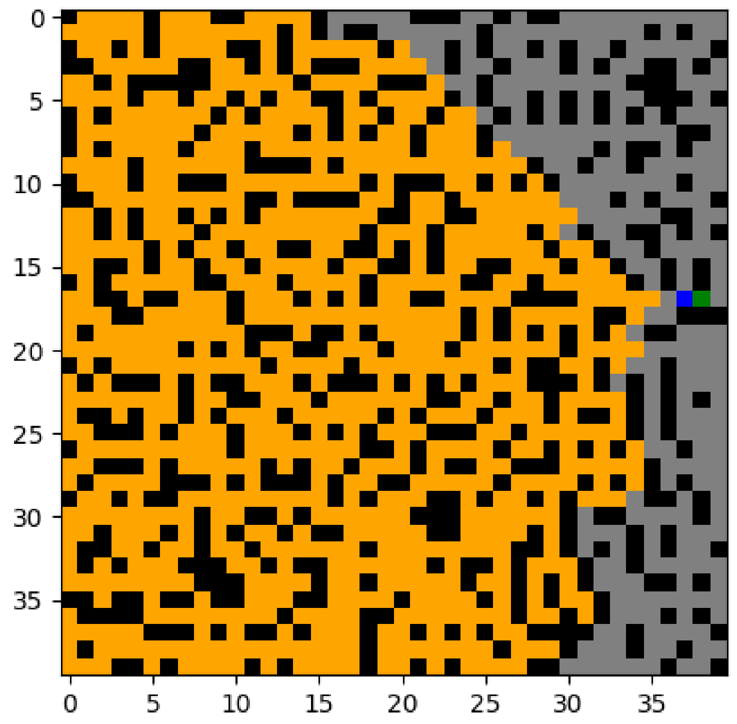


This Bot is on Fire

PROJECT 1 WRITEUP

Abhishu Oza | Intro to AI | 10 March 2025

1) The Code

- **main.py** contains code for simulations.
- **results.csv** stores results of all simulations with timesteps and path of each bot.
- **ship_arrays** folder stores initial configuration as well as the penultimate configuration for all bots as .npy files, for all simulations.
- **visualize_ship.py** visualizes the ship with or without the path taken by bot, saves to **ship_plots.**
- **animate_fire_spread.py** animates the fire spread starting from any initial ship configuration with randomness.

1) Bot 4 Design

For bot 4, I present my best performing bot that I could test reliably. **Bot 4 has an A* algorithm design** with the following steps in sequence -

- **It calculates cost per cell by taking the inverse of its Manhattan distance (dist) to the ORIGINAL fire position (Cost[i, j] = 1 / dist((i,j), fire_pos)).**
- **It calculates the average cost of all cells in the grid (avg_cost).**
- It performs the A* algorithm using a min heap priority queue as the fringe (meaning next cell popped has lowest estimated total cost). A* does the following -
    1. It records the smallest total cost to all cells yet discovered (totalCosts).
    2. It then pops the next element (current) from the priority queue and discovers its valid children (cost = totalCosts[current] + Cost[child[0], child[1]]).
    3. **Estimated total cost for child is found using this heuristic to button -**

       estTotalCost = cost + heuristic(child)

       = cost + avg_cost * dist(child, button_pos)
    4. If this child does not have a recorded totalCost or has the new cost (cost) lower than recorded cost, it -
        a. Pushes the child in the fringe with priority = estTotalCost.
        b. Updates totalCosts[child] = cost.

As shown in the upcoming writeup sections, this algorithm performs better than bot 1 and 2 but not better than bot 3, although it comes close. Perhaps this is because bot 4 does not update information on fires as time passes. However,

- I created the same algorithm that updates the Cost array at each time step (bot 5 in code), but it performed worse than even bot4 and ran much slower.
- I created several candidates for bot 4 with different dynamic costs, but they either performed worse or I could not get them to work (a failure on my part, I did not allocate my project time well).
- The presented bot 4 was best and ran as quickly as bots 1, 2 and 3.

Given that it uses a non-dynamic cost, bot4 seems to work quite well. **The inverse of distance to the original fire is a good cost because the closer you get to the fire, the larger the difference in cost is locally, thus balancing going away from fire and going towards the button well.**

2) Improving speed and Efficiency

For bots 1, 2 and 3, although the constraints or details are different, the aim at any given timestep is the same - to find the shortest path to the button. To that end, Breadth First Search can be used to implement all of them.

For bot 1, this may work, but for 2 and 3, the path needs to re-plan at every timestep. BFS took too much time for them. Hence, I implemented them using A*.

A* uses costs, but to obtain the shortest path, costs to all cells are set to 1. Manhattan Distance (sum of absolute difference of co-ordinates) could thus be used as a **consistent heuristic** in this setting (actual path is strictly equal or longer as some cells are blocked).

Let C*(cell) be the shortest path from start node s to cell. A consistent heuristic provides helpful properties –

- The first path to a goal node that comes off the fringe under A* will be optimal.
- A* expands no node with $f(n) > C^*(goal)$.
- No other algorithm can expand fewer nodes.

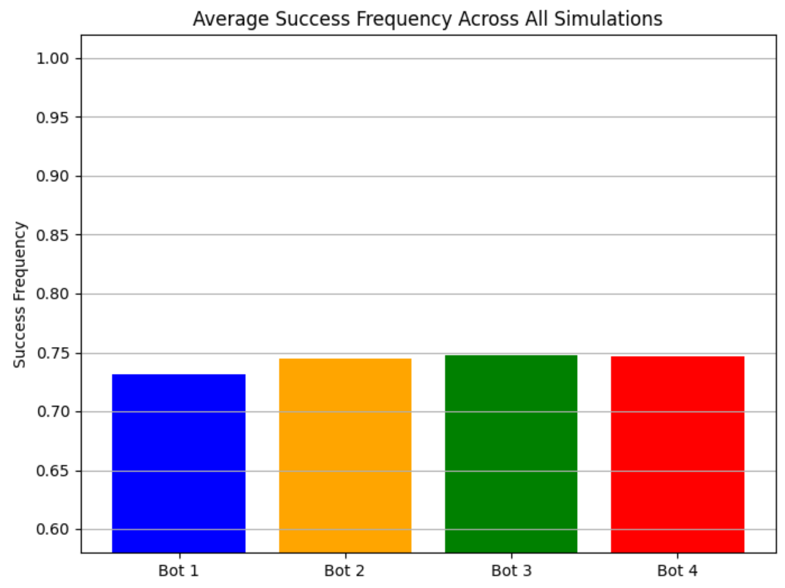
Hence for the strict requirement of shortest path in bots 1, 2 and 3, A* performs the fastest possible job. The heuristic helps make it much more efficient than BFS and now it can be used to re-plan paths. Thanks to this along with a quick bot 4, I was able to run 2000 simulations.

3) Performance Evaluation

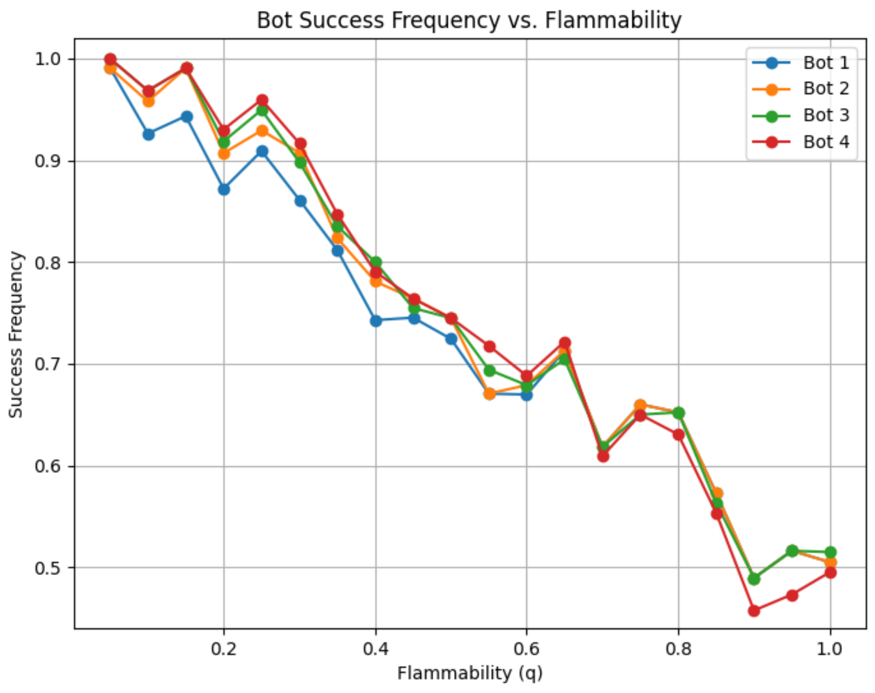
Q is chosen from among 20 linspaced values between 0 and 1 (0.05, 0.1, 0.15 …). All bots are run on the same fire spread, and 1 simulation runs and records outcomes of all 4 bots on a randomly selected q. 2000 simulations are recorded in results.csv.

Simulations won –

Bot 1	**– 1462, 73.1%**
Bot 2	**– 1490, 74.5%**
Bot 3	**– 1496, 74.8%**
Bot 4	**– 1493, 74.65%**



Observation – The bots have a close number of successful performances despite running lots of simulations. It indicates that a bound of winnable simulations may be close.



Observations –

- Values are according to expectation – a steady decrease of success rate as q increases.
- Bot 1 is worse than 2 and 3, especially for lower values of q when there is more time to course-correct before the fire approaches.
- Bot 4 does slightly better than bot 3 at moderate q values, but does worse on higher q.

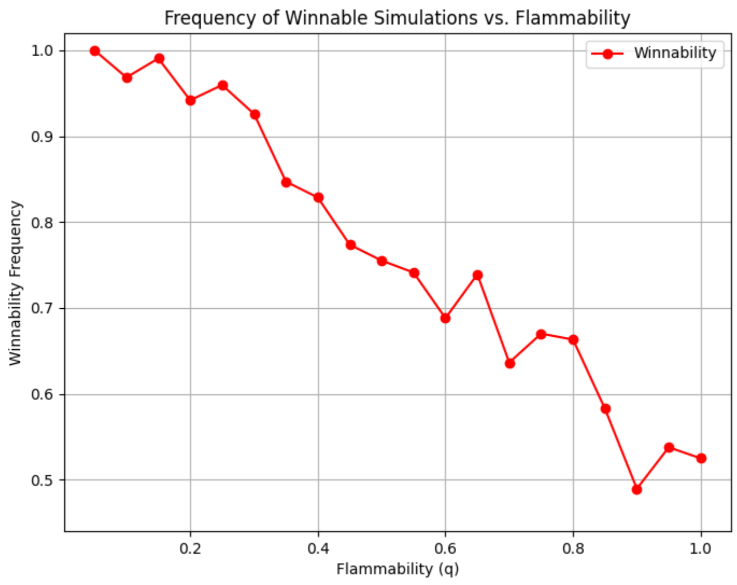
4) Winnability

The close values for different bot strategies indicate that they may be near to optimality. I write a test for winnability (function winnable()) by -

1) **Recording the entire fire spread** till the button cell is on fire – creating a 3D array with each level as a timestep
2) Running search through time from the first level to find if there is a path to the button **on any level**.

The function winnable() is just an A* search through the 3D array, but with only adjacent cells exactly one timestep below as legal neighbors of any cell, because exactly 1 timestep must pass per move. A* was again used here with cost of each move as 1 and Manhattan Distance to the button on its own level as the heuristic (can be thought of as Manhattan Distance through time to the closest reachable button cell - it is again consistent).

Out of 2000, 1528 simulations are winnable (76.4%). Frequency of a simulation being winnable is plotted below –

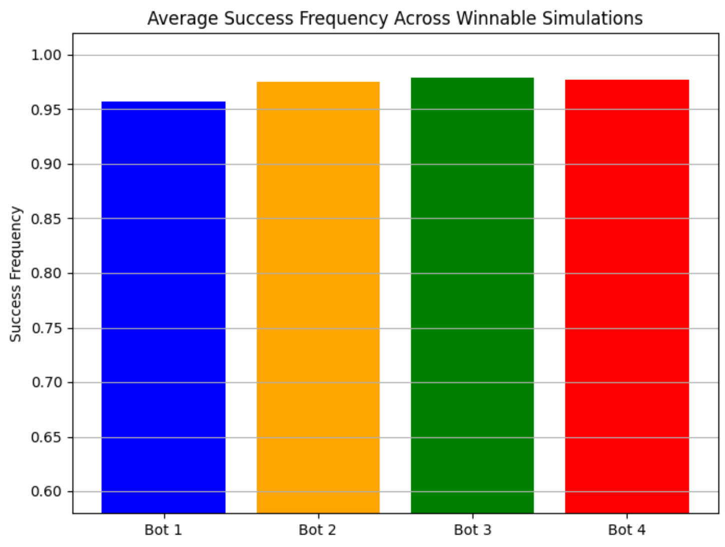


Observations –

- Values are again according to expectation – steady decrease in winnability with q.
- Also shows that hiccups in data of bot success come from randomization of simulations – they are not a function of q itself.
- With full flammability (q=1), 50% of simulations are winnable.

5) Performance for Winnable Simulations

We now plot success frequency among winnable simulations –



Out of 1528 Winnable simulations, % won –

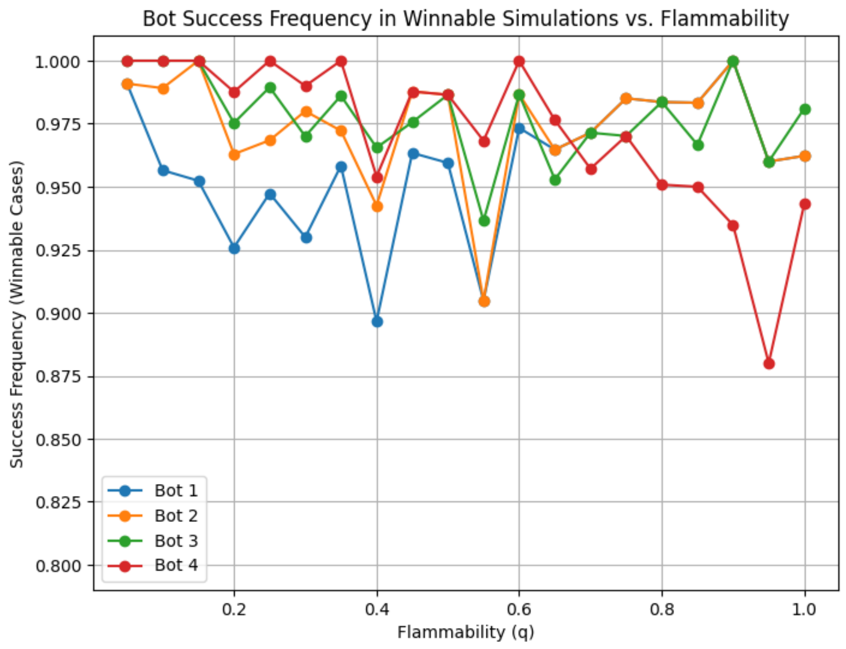
Bot 1 – 95.7%

Bot 2 – 97.5%

Bot 3 – 97.9%

Bot 4 – 97.7%

Observation – Closer to optimal than expected!



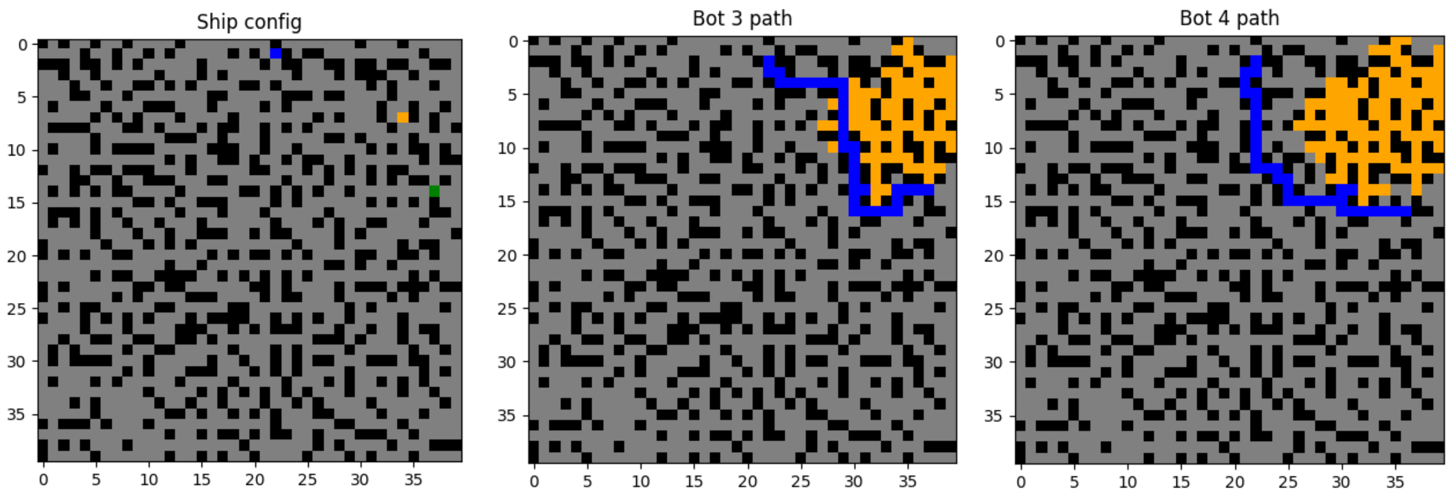
Observations -

- Due to a decrease in overall variance, bot performance can be studied more clearly.
- Bot 4 wins at lower q but performs much worse at high q levels.
- Bot 2 slightly outperforms bot 3 at high q levels, maybe because bot needs to be riskier when near to the button for high q.

6) Failure Analysis

The noise in winnability plots indicate that running *even more* simulations could help clear the picture. But from the generated results, there is a clear weakness in bot 4 – high q values. It indicates that bot 4 may be underestimating the fire cost, especially toward the end of the path.

We consider the simulation test = 1946 which had q = 0.9 (blue is bot, orange is fire, green is button). Bot3 succeeds, but bot4 fails.



Here, bot 3 prioritizes distance to goal correctly. For bot 4, this indicates that at high q, there is an overestimation of fire cost for cells in the beginning!

To address the 2 failures – adapting to q and to time – bot 4 could be improved by tweaking the cost function to **incorporate time as a factor in proportion to q value**. That is, the cost is initially lower and increases with time but also increases *faster* with a higher q value.

A good mathematical formulation that smartly adjusts the heuristic rather than taking the average of all costs could also help.

7) Ideal bot Speculation

The ideal bot perfectly balances distance to goal and risk of fire for all ship configurations. Bots 1, 2, 3 err on the side of distance to goal by working on shortest path. Bot 4 errs on the side of risk of fire. An ideal bot is somewhere in between – one having a sharper cost decrease with increasing distance to fire.

A limitation comes from the randomness of fire spread. For dynamically updating cost with time, a function of q and time should be able to approximate the radius of fire spread. Alternatively, running simulations of fire spread inside the bot can be used to calculate the probabilities of cells catching on fire.

According to my speculation, a good radius approximation/fire probability calculation, combined with the right cost decrease curve should create the ideal bot.