You’re Friend is in Danger and You are Brave Bot: Report

Jan Marzan

March 4, 2024

Abstract

This project explores pathfinding and decision-making strategies for an autonomous agent operating within a dynamic and hostile environment. In a simulated space salvage vessel overrun by aliens, a robotic cleaning unit must rescue the ship’s captain while avoiding detection. Four bot strategies were implemented and evaluated: a baseline shortest-path algorithm, variations incorporating dynamic path replanning and alien avoidance, and a custom strategy that prioritizes safety over progress. Success rates and survivability were analyzed under varying alien densities, providing insights into the efficacy of different approaches to navigation and decision-making in uncertain, adversarial settings.

1 Introduction

Autonomous agents operating in dynamic and potentially dangerous environments require sophisticated navigation and decision-making abilities. This project simulates such a scenario, where a robotic cleaning unit (“the bot”) aboard a space salvage vessel (“the ship”) must locate and rescue the ship’s captain in the presence of hostile aliens. The ship’s layout is procedurally generated for each separate simulation, adding an element of unpredictability to the bot’s task.

The project aims to explore the performance of different bot strategies designed with varying levels of sophistication. Key metrics for evaluation include the bot’s ability to successfully rescue the captain within a set time limit and its survivability when a rescue is not achieved. The analysis will shed light on the strengths and weaknesses of various pathfinding and decision-making approaches in the context of a dynamic, uncertain environment.

1.1 Project Structure

Let us first discuss how the project is structured, the repository for which can be found here. It is built using the Unity Game Engine in C#, as I’ve been wanting to learn game development for a while but never had any major incentive (until now). Nevertheless, the project follows a typical Unity structure, meaning that all relevant code is located in `Assets/Scripts/`. From here on out, when I mention a filename, we assume this is the “root” unless stated otherwise. Here are the files in that directory that pertain to the project:

- `Logic.cs`: Implements the core game loop for the simulation, advancing bots and aliens, determining success/failure, and generating reports.
- `ShipManager.cs`: Manages the ship, including procedural generation, captain and alien placement, and node states. Also provides utility functions for determining states of certain nodes.
- `Alien.cs`: Defines position property and implements random movement. There is a corresponding Alien prefab.
- `Captain.cs`: Defines the position property. There is a corresponding captain prefab.
- `Node.cs`: Defines basic properties (like color, vacancy, and position) and functionality (like opening, closing, and highlighting)

In the `Bots/` directory we have the following files:

- `Bot.cs`: The base class that all bot strategies must inherit.
- `Bot<1...4>`: The implementations for each of the bots, to be discussed later.

Back to the root, there is also the `/Report` directory, with:

- `Analysis/`: This directory contains the data for each of the bot runs and some python scripts computing values about them
- The assignment PDF, the report PDF, and relevant resources, like graph images, used by the report.

2 Methodology

2.1 Ship Generation

To clarify terminology, the ship is a $D \times D$ square grid, where each tile I call a "node". An open node is a node that an alien, captain, or bot can enter, and a closed node one it cannot. The ship generation algorithm adheres closely to the project requirements with very little change, and can be found in `Utils/ParallelShipGen.cs`. The key optimization lies in the name itself - I use parallelization to generate the large number of ship layouts required for the simulations. This optimization was necessary, as I found generating just 100 ships sequentially, and then running the simulations, took around 30-40 seconds. Generating the ships in parallel allows me to generate 10 times as many ships and run all 1000 simulations in the same amount of time.

2.2 Bots

As mentioned before, the bots all inherit a `Bot` class that requires the bot to implement the properties that describes its position and its name, and the method that computes its next step given the state of the ship. The state of the ship is represented by a grid of nodes, each node exposing two properties: occupied (true if an alien is on that node), or closed (as it was generated). This state also includes the location of the captain, and a list of all the aliens and their positions. Furthermore, all of these bots use the A* search algorithm with the Manhattan distance to the captain as a heuristic. I use this heuristic over straight-line distance as it provides a more accurate representation of the amount of "steps" the bot has to make to reach the captain.

2.2.1 Static Path (Bot 1)

This bot computes the path to the captain that avoids all current aliens. However, it does so only once when a path becomes available. Then, it executes all the steps of that path regardless of where the aliens move to next.

2.2.2 Adaptive Path (Bot 2)

This bot computes the path to the captain that avoids all current aliens, and executes the next step in that path. Unlike Bot 1, however, we recompute the path at every time step. This adds some sense of adaptability in the bot's pathing relative to the current state of the ship and the movement of the aliens. When a path to the captain cannot be computed, we simply stay in place. I choose this approach over making other movements as it prevents the bot from entering the captain's node if there is an alien on it. Of course, there is still the risk of an alien entering the node the bot is on.

2.2.3 Buffered Adaptive Path (Bot 3)

This bot computes the path to the captain that avoids all current aliens as well as the nodes adjacent to the aliens, and executes the next step in that path. Again, we recompute at each time step. Note, I define adjacent as the nodes which an alien can move to (up, down, left, right) in a single time step. It should be mentioned that it is not necessary to iterate through all aliens to determine their adjacent nodes, which could potentially slow down the search. We can use the fact that if an alien is adjacent to a node, that node is also adjacent to the alien. Thus, when we perform the step in A* that computes

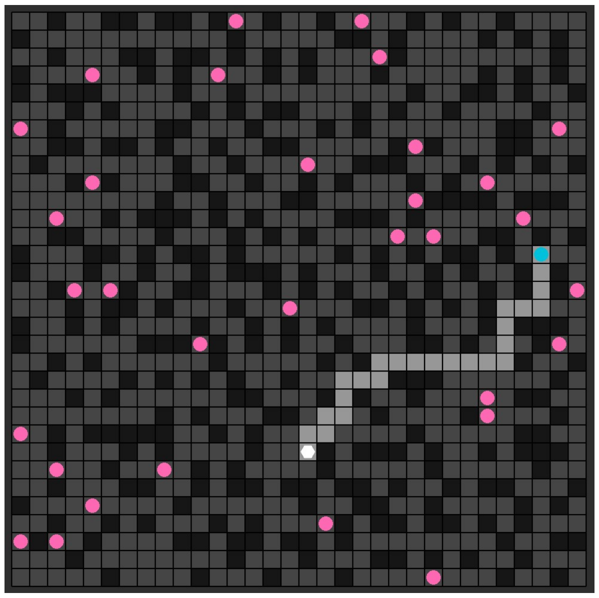
Figure 1: A screenshot of a simulation on a 32×32 ship. Aliens in pink, captain in blue, and the bot in white. The path computed at this particular timestep is highlighted in white

the children of the current node to add to the fringe, we can skip the children who are adjacent to an alien. If there is no path to the captain with this buffer, we compute the path with no buffer.

2.2.4 Safe Pathing (Bot 4)

For this bot, I depart from the previous bots' tendency to relentlessly make progress towards the captain. As the name implies, this bot prioritizes safety over progress. That is, we only move towards the captain if it is "safe" to do so. Otherwise, we take the safest action. I define a node to be "safe" if there is no alien on it or no alien next to it. This bot, at every time step, takes one of five actions:

1-4. Move up, down, left or right.

5. Stay in place.

Then, the bot chooses one these actions depending on the following criteria:

- The action is valid (i.e. the bot does not move onto an alien or into a closed node).
- The action does not place the bot in a node adjacent to an alien.
- The action ideally is on the (buffered) path to the captain.

To explain my implementation of the last criterion, let me call the node on the path to the captain the "ideal" node. The bot only takes the action that brings it to the ideal node if that node is safe, as defined before. If that node is not safe, then the bot takes the action that brings it to a safe node that is closest to the ideal node.

2.3 Simulation Setup

The simulation setup establishes a structured framework to compare the effectiveness of different bot strategies across various alien densities. The core parameters are as follows:

- **Ship Dimension:** A fixed 32×32 grid is used to provide a consistent environment.
- **Alien Count:** The number of aliens, K, is systematically increased from 1 until a quantity K_N. If we let $s(n)$ denote the average success rate of simulations with n aliens, then K_N is defined as:

$$K_N = \min\{N \in \mathbf{N} \mid \sum_{i=N+1}^{m} \frac{s(i)}{m-N} < 1\% \text{ for all } m > N\}$$

In other words, K_N is the minimum number of aliens resulting in an average success rate below 1% for all higher alien counts.

- **Simulations Per Configuration:** For each bot strategy and alien count combination, 800 simulations are executed. This ensures sufficient data points to analyze success rate and trends while running reasonably quick.

2.3.1 On K

I redefine K_N from the project requirements as I found the original definition to be somewhat incompatible with this setup. Using a single number as the failure point can be misleading due to the randomness inherent within the simulation, which is exacerbated by the fact that we run many simulations per alien count. For example, it is possible for a bot strategy to succeed in one particular simulation just by spawning right next to the captain. So, by defining failure based on a consistently low average success rate across multiple alien counts, I establish a more robust and reliable metric for determining the point at which the bots become ineffective.

However, as I cannot run simulations across an infinite number of alien amounts, I use the original definition for K_N as a stopping point. The reasoning being that if a certain strategy fails across all 800 simulations for a certain number of aliens, it is extremely likely that the rate of failures after this amount are also extremely low, possibly less than 1%.

I would like to note that varying this success rate threshold, which I have fixed to be 1% in the above definition for K_N, can lead to interesting results. So, we may view $K_N(x)$ as

$$K_N(x) = \min\{N \in \mathbf{N} \mid \sum_{i=N+1}^{m} \frac{s(i)}{m-N} < x\% \text{ for all } m > N\}$$

We will discuss more on this in a later section. Until then, let us assume that $K_N = K_N(1)$.

3 Results and Analysis

This section delves into the performance of the different bot strategies as the number of aliens within the ship increases. The primary results are visualized through line graphs, where success rate percentages (in blue) and the average survivability on failure (in red) are plotted against the number of aliens. These graphs showcase how each bot's effectiveness changes with increasing hostile density.

For a focused analysis, I use the the metric K_N as defined before: the minimum number of aliens resulting in an average success rate below 1% for all higher alien counts.

Let us now examine each bot individually, discussing trends observed in their graphs, how they compare to the other bots' performances, and what insights we can obtain regarding the strengths and weaknesses of different decision-making strategies.

3.1 Bot 1

Figure 2 gives the graph for Bot 1.

Bot 1 exhibits poor overall performance due to the the static strategy it employs within a dynamic environment. Its success rate declines rapidly with increasing alien density, exemplified by the graph's immediate convex nature. While the A* algorithm it uses finds the shortest valid path on the first available time step, it does not replan as aliens move. This is the core issue, the bot assumes a static environment. The presence of dynamic obstacles (aliens) invalidates the pre-computed shortest path. Without adaptive path replanning, Bot 1 finds itself running into aliens easily, leading to early failures. Nonetheless, we find Bot 1's K_N to be 257.

Interestingly, the average steps on failure show a slight upward trend at around 100 aliens, before

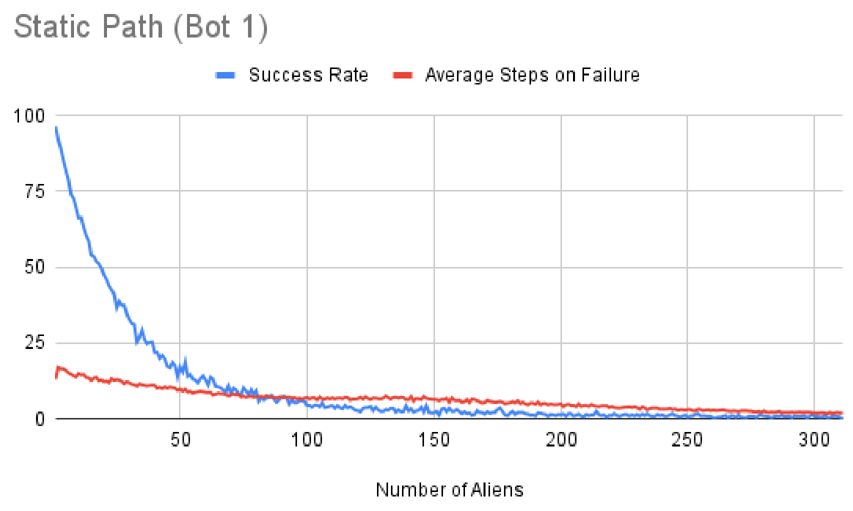

Figure 2: Success Rate and Average Steps on Failure to Number of Aliens Graph for Bot 1

decreasing again at around 150. This could suggest that in environments around this particular alien density, the bot occasionally benefits from the sheer chaos. Aliens might unintentionally obstruct each other, creating temporary openings that the bot unknowingly exploits, allowing it to persist for a few more steps. Of course, as the amount of aliens continues to increase, these openings become less and less common.

3.2 Bot 2

Figure 3 gives the graph for Bot 2.

Bot 2, employing dynamic path replanning, exhibits significantly improved performance compared to Bot 1. Its success rate declines more gradually with increasing alien density, demonstrating a more constant decline until around 80 aliens. We also see an overall higher success rate than Bot 1 across all alien amounts up until around 150 aliens. We see a similar case for the average steps on failure, where Bot 2 is consistently higher up until around 200 aliens. Of course, Bot 2's advantage lies in its ability to recalculate paths as the aliens move.

However, A*, and all other pathfinding algorithms as a matter of fact, still relies on the *existence* of a valid path to the goal. In highly crowded environments no such path may exist, causing Bot 2 to stay in place (as I've intended) without any response to alien movements. For this reason, Bot 2 may find itself trapped, leading to eventual failure despite its adaptive capabilities.

For Bot 2, K_N is 259, which is an almost negligible increase over the value found for Bot 1. This may indicate that both of these bots consistently fail in environments with a similar range of alien densities. This is reasonable, as in highly chaotic scenarios with many aliens, even frequent replanning may not be enough to find a route to the captain.

3.3 Bot 3

Figure 4 gives the graph for Bot 3.

Bot 3 demonstrates a similar performance to Bot 2, which is not surprising as both use very similar strategies. The main difference being that Bot 3 plans a path that avoids not only current alien positions, but also potential areas the aliens could occupy in the next time step, i.e. a "buffer". This anticipatory evasion allows Bot 3 a marginally higher success rate on average than Bot 2 by just 0.0657%. The average difference of the the average steps taken on failure between both bots are also

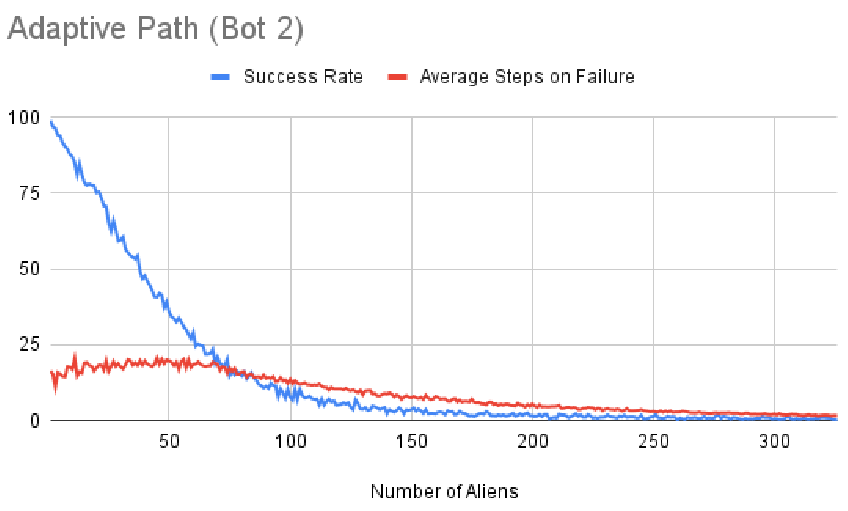

Figure 3: Success Rate and Average Steps on Failure to Number of Aliens Graph for Bot 2

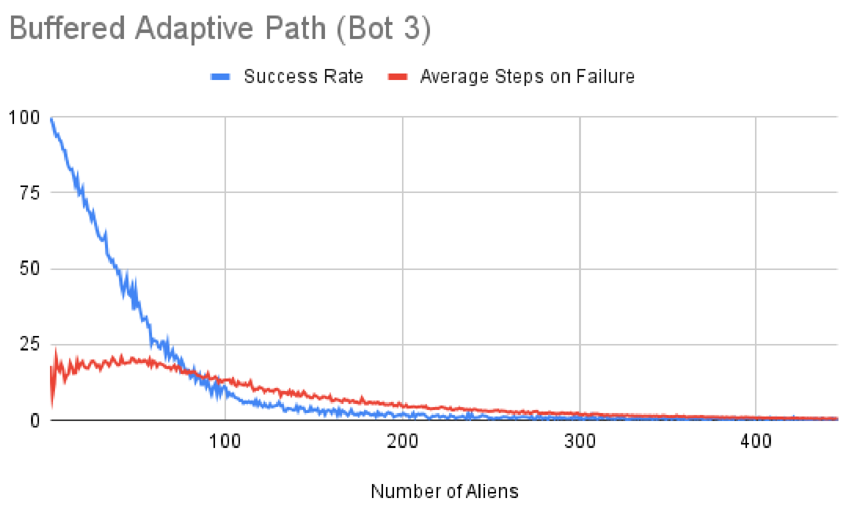

Figure 4: Success Rate and Average Steps on Failure to Number of Aliens Graph for Bot 3

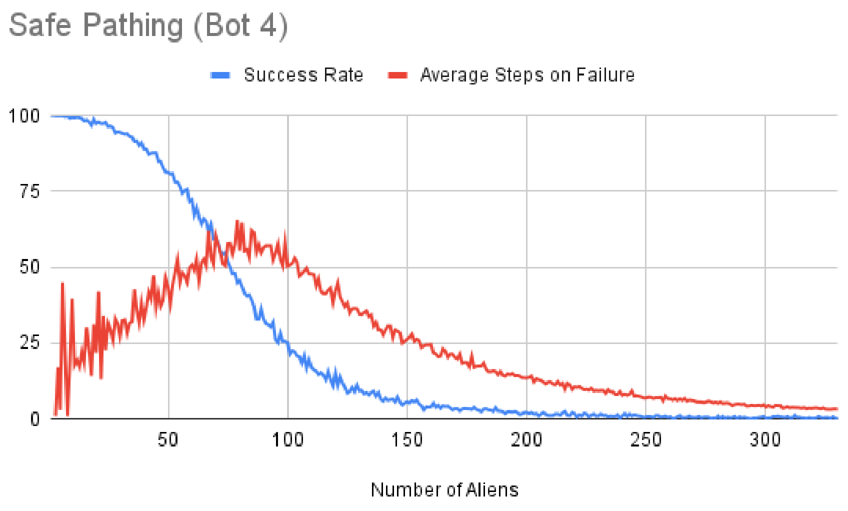

Figure 5: Success Rate and Average Steps on Failure to Number of Aliens Graph for Bot 4

very close, with Bot 3 on top by a mere 0.0319 steps. Though, the difference between K_N values are a little more varied, with 248 for Bot 3 and 259 for Bot 2.

The most obvious reason for the similarity in the results is in how Bot 3 deals with the case where it cannot find a buffered path to the captain. When no such path to the captain exists, it just removes the buffer from its pathfinding algorithm, thus falling back to same exact logic as Bot 2. This could explain the slight increase over success rate and average steps on failure, as this edge is just enough to make it marginally better than Bot 2. As for the K_N values, while it's lower for Bot 3, they are still very similar. We can draw the same conclusion as before, that these bots perform similarly under dense conditions.

But, something's still amiss. Bot 2 had its first complete failure (always dying or failing to reach the captain) at $K = 326$. However, it took over 100 more aliens for Bot 3 to witness its first complete failure at $K = 446$! So, what gives? My best guess is that when a buffered path to the captain exists, any position on this path is inherently "safer" than any position on the unbuffered path to the captain. In highly dense scenarios, where the only chance of success is spawning right next to the captain, this advantageous positioning may be enough to give the bot the opportunity for a straight shot to the captain.

3.4 Bot 4

Figure 5 gives the graph for Bot 4.

Bot 4 demonstrates an immense level of improvement over all bots, with a 10% higher success rate on average and surviving for about 15 time steps longer when failing than both Bot 2 and Bot 3. We may attribute this success to the radically different approach the bot utilizes over previous strategies. Rather than relentlessly pursuing the captain (or remaining stationary when no path to the captain exists), Bot 4 prioritizes self-preservation, implementing a "safety over progress" paradigm.

When analyzing Figure 5, we see Bot 4's success rate exhibiting a concave decline up until around 70-80 aliens, whereas the other bots demonstrated linear or convex patterns in this range. This means that the initial rate of decline of the success rate for low alien counts was rather slow, indicating that the bot does exceptionally well within this range of alien densities. A possible reason for this is the greater freedom of movement available within this particular range. There are not that many aliens, so there is more space to move around. Bot 4's strategy places an emphasis on identifying and moving towards "safe" nodes, so the increased availability of safe nodes allows Bot 4 to navigate the ship effectively while minimizing risks. In contrast, other bots' focus on reaching the captain might occasionally lead them into encounters with the few scattered aliens. After this range, however, we begin seeing a sharp drop in the success rate as the bot suffocates under these crowded scenarios, a trend present with the other bots as well.

Another thing to note is the massive improvement in the average amount of steps taken on failure. At its peak, we're seeing averages of around 60 steps! The other bots pale in comparison, not even seeing a failing trial over 25 steps. Bot 4's prioritization of safety directly prolongs its operational lifespan. In chaotic scenarios, it actively seeks out safe nodes and avoids unnecessary confrontations, allowing it to persist longer than other bots that might be eliminated quickly due to risky maneuvers. What's more interesting is that we see the peak in average steps on failure occur at around 70-80 aliens, coincidentally right after the success rate transitions from concave to convex. This suggests that at these amounts of aliens, there is still ample room to avoid any nearby danger, but perhaps not enough to create a safe path to the captain. Thus, we see the bot to survive for so long without reaching the captain. I would also like to make a note of the large fluctuations we see at the beginning of the graph. This makes sense, as low failure rates means less data for this metric, and less data means more variability. We see the fluctuations decrease as the success rate decreases.

As for K_N, it is surprisingly low in comparison to the other bots at 247. One might expect for this bot to fail less consistently than other bots due to its preference to avoid aliens. But, I argue that it is for this exact reason why we see a marginally lower number than the other bots. Rather than making risky maneuvers that might allow it to reach the captain, it prefers safer maneuvers that may force the bot away from the captain. Under crowded conditions, this significantly decreases the bot's chance to reach the captain. We can see this in the data, since between 240 and 260 aliens Bot 4 survives on average 2.33 times longer than Bot 3 when failing, despite failing approximately the same amount.

Overall, Bot 4's exceptional performance underscores the importance of integrating safety considerations alongside mission objectives in dynamic, adversarial environments. The data demonstrates that a carefully calibrated emphasis on self-preservation within an autonomous agent's decision-making can lead to significant improvements in both survivability and success rates. While trade-offs might exist in terms of potential delays or limitations in highly restrictive situations, the "safety over progress" paradigm offers a robust and effective strategy for navigating unpredictable hostile settings

3.5 Comparative Discussion

3.5.1 Consistent Failure Points

Previously, when I referred to K_N, I was referring specifically to $K_N(1)$. I would like to shift my analysis towards examining $K_N(x)$, as defined above, for various values of x. As a reminder, we may interpret the evaluation of $K_N(x)$ as the number of aliens after which a bot consistently succeeds less than x% of the time. To put it in another way using an example, this means that all previously mentioned K_N were the amount of aliens after which the bots consistently fail 99% of the time. From the discussion about the bots above, we see that $K_N(1)$ all fall within 5% of each other at around the 255 alien range. The conclusion that I have established from this is that, regardless of the bot's strategy, this amount of aliens proves to be too crowded for any bot to succeed.

So, what happens when we vary x? Figure 6 shows us $K_N(x)$ for each bot with success rate thresholds from 1% to 25%. When examining this graph, we immediately see a clear trend: as the success rate threshold *increases*, the $K_N(x)$ value for all bots *decreases*. This means that a higher threshold for success necessitates a smaller number of aliens to guarantee that the success rate exceed the threshold. This makes sense intuitively, as a less strict success criterion becomes easier to satisfy with fewer aliens.

Bot 1 consistently demonstrates the weakest performance across all success rate thresholds, requiring the lowest K to meet each target. This means that it requires fewer aliens than all other bots to consistently be below any success threshold x. Bot 2 and 3 fall in the middle, with values close to each other across all x. Bot 4, again, generally performs better than all bots across all success thresholds.

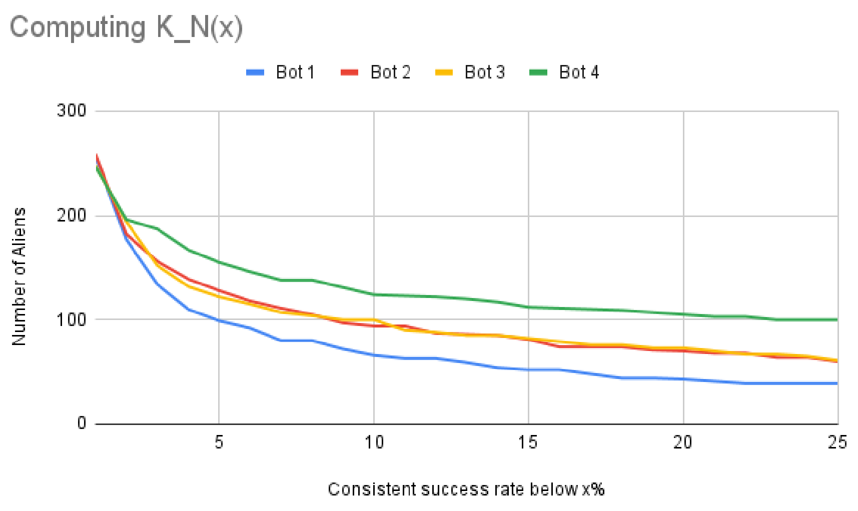

Figure 6: The graph of $K_N(x)$ for each bot

As an example, let us consider $K_N(10)$. Bot 1 would require more than 64 aliens to have success rates consistently below 10%. Bot 2 and 3 come in much higher at 94 and 100 respectively, while Bot 4 showcases a significantly better performance requiring 124 aliens.

Importantly, the differences between the $K_N(x)$ values for the bots become smaller as the success rate threshold becomes more stringent. This suggests that under extremely crowded conditions, the performance of all bots start to converge as the large amount of aliens render their strategies useless in saving the captain. Thus, by analyzing $K_N(x)$ for different success rate thresholds, we can gain a more nuanced understanding of how each bot's performance relies on varying amounts of aliens.

4 Bonus 1

This problem tasks us with discovering/designing a more effective ship design via some algorithmic process. What came to my mind immediately was a genetic algorithm, where we combine the best performing ships in hopes of producing an even better ship.

4.1 Methodology

In order to complete the bonus, I had to create a new Scene with a revamped "Game Manager", the `GeneticLogic` class, which utilized a more stateful approach to running simulations. In addition, I had to specify a separate `Simulation` class so that multiple simulations could be run and managed at the same time. I also created a new `GeneticShip` class which extends the `ShipManager` class from before. This class allows the ship to maintain the same layout across all simulations. Note, all relevant code is found in the `Genetic/` directory.

4.1.1 Setup

Due to time and computational constraints, I opted for a more conservative configuration. The dimension of the ship remain the same as before, 32×32. Each generation would consist of 50 ships, each running 25 simulations. The simulations would use Bot 4 against 64 aliens. Per generation, we take the top 10 performing ships to produce the next generation of 50 ships. The first generation would be generated procedurally as before, but all proceeding generations would be generated from the prior.

4.1.2 Combination

To generate 50 ships for the next generation from the 10 best ships of the current generation, I employ a method of uniform crossover. I take two ships, and consider their nodes in corresponding positions to determine the node in the child. If the parents' nodes are the same (both open or both closed), then the child will take on that value. If the parents' nodes are different, then the child will take on the value of the first parents' node with the probability $p = \frac{x_1}{x_1+x_2}$ where x_i is the number of successes each parent ship had. In other words, we prefer taking the value of the parent with the higher success rate.

To choose which parents to pair up, I use roulette wheel selection. The probability that a ship is selected to be a parent is dependent on the proportion of the success rate of that ship to the sum of the success rates of all ships. Again, this gives us a preference for the best performing ships. We generate 47 pairs and generate 47 new ships. As for the remaining 3, I use the concept of elitism, and take the best 3 ships from the previous generation to succeed to the next.

4.2 Results

4.2.1 First Pass

Running the algorithm for the first time, there were many glaring issues with my implementation. When looking at the final generation, I found that all ships were the exact same. Furthermore, as generations went on, the success rate plummeted. I found that the success rate of the procedurally generated generation was approximately 36%, but the success rate for the last generation was around 20%.

The issue with the homogeneity of the final generation could easily be attributed to how I combined ships. There are multiple areas in which the traits of the best performing ships are preferred, from parent selection, to child node determination, to elitism. Combining these lends itself to a quick homogenization of the population. Furthermore, there is no notion of random mutation. While I figured that the slightly random chance of determining which parent a node would be determined by would be enough to create novel ideas, it isn't random mutation in its true sense. The child node will still always be determined by its parents. To mitigate these factors, I implement these changes to how a child node is determined:

- Corresponding nodes that are not the same in the parents now have equal probability of being inherited in the child
- Corresponding nodes that are the same have a 5% chance of being different from their parents.

However, with these changes, the sudden decline in success rate still persists. But, one thing I did notice among these ships is that, as generations progress, the distribution of closed nodes increases. So, I "guide" the generation of ships by making the conclusion that a more open ship is advantageous. Thus, I modify the first point as such:

- Corresponding nodes that are not the same in the parents have a 65% chance of being open in the child.

The 5% mutation rate comes in handy now, as it aids in creating interesting closed structures.

4.2.2 Final Pass

After implementing these tweaks, we end up with the ship seen in Figure 7, having a final success rate of 64% across 25 simulations. This is better than the overall success rate we see from the first generation of procedurally generated ships, which was around 36%. When observing the ship, we see the presence of large open spaces scattered throughout, with plenty of ways to enter and exit those spaces. Furthermore, the amount of dead ends where the bot could get stuck is limited. These dead-ends still exist, indicating there is still space to improve. When comparing to a procedurally generated ship, such as in Figure 1, there are plenty of deadends and narrow spaces with only one or two exit routes.

However, this genetic algorithm is not perfect. If we examine Figure 8, we actually find a much

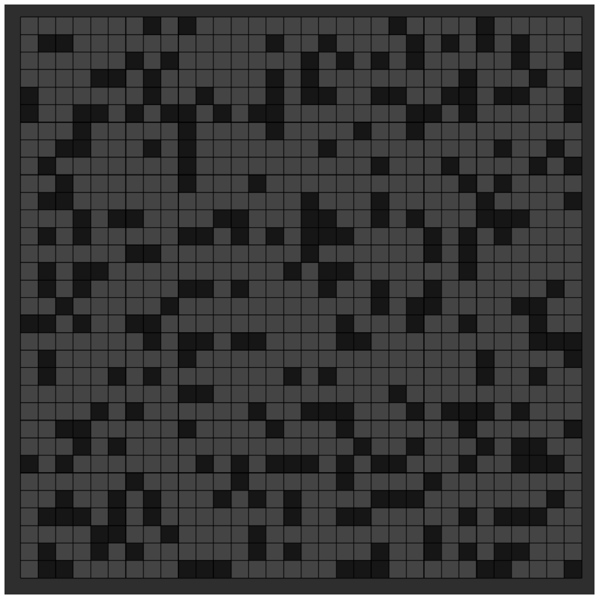

Figure 7: The best performing ship from generation 50

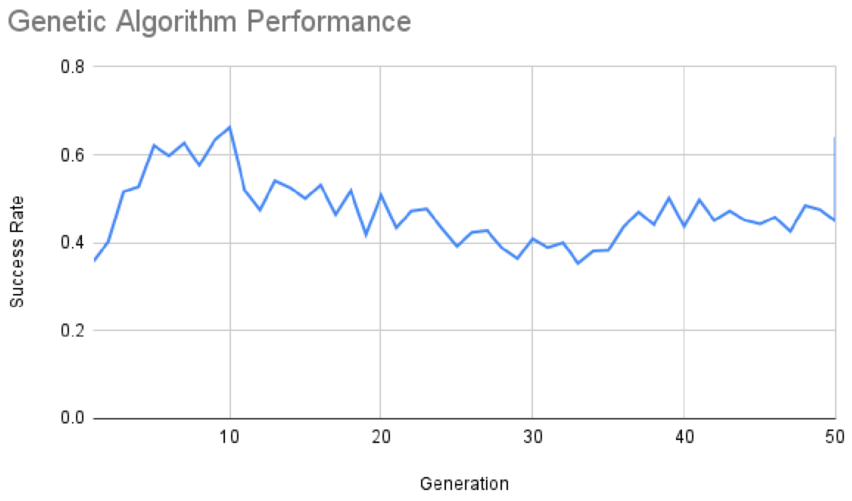

Figure 8: The performance of each generation

better performing generation way earlier, where each ship is way more varied from the procedural generation. As we hone in on a specific ship layout, our success rate drops before seeing incremental improvement. There could be many reasons for this, from the parameters I picked to the specific crossover algorithm I used. If I gave myself more time, perhaps I could've explored this further.

5 Closing thoughts

5.1 Conclusion

This project explored the challenges of autonomous navigation and decision-making within dynamic, adversarial environments. The data illustrates a clear performance hierarchy among the implemented bot strategies. Bot 1, relying on a static path, struggled immensely in the face of unpredictability. Bots 2 and 3 introduced adaptability with dynamic path replanning, leading to improvement. However, even their adaptability was overwhelmed in highly dense alien scenarios. Bot 4 emerged as the most effective strategy across various alien densities due to its fundamental emphasis on safety and self-preservation.

The analysis of performance under varying success thresholds ($K_N(x)$) revealed further nuances. Importantly, under extremely crowded conditions, even Bot 4's success rate declined rapidly, suggesting that all strategies tend to converge as the environment becomes too restrictive. The importance of striking a balance between safety and progress is thus underscored.

Furthermore, the bonus exploration demonstrated the potential of genetic algorithms in the realm of procedural ship design. While initial results were suboptimal, modifications to the algorithm—prioritizing open layouts and incorporating mutation factors—yielded a somewhat better final ship design. This design achieved a notable 64% success rate, surpassing the performance of procedurally generated counterparts. The process highlights the iterative nature of genetic algorithms and suggests directions for further optimization, potentially leading to even more effective ship designs for this specific task.

If time permits, I may explore this project further, as I learn more about artificial intelligence and discover new ideas for potential strategies. I also wish to explore the genetic algorithm further, as I see there are major areas of improvement and I'm not really satisfied with my results. Nonetheless, this project serves as a foundation for continued exploration of autonomous agent behavior, highlighting the importance of adaptability, safety considerations, and the limitations inherent in navigating unpredictable and hostile environments.

5.2 Unity

As this is my first major project using Unity, I know that there are many improvements to be made regarding good coding practices. For one, the `Logic` class, which serves as the "manager" of the scene, could be refactored to use a more stateful approach in managing the game. Furthermore, there was probably a better way for long processes to be handled in the background. And also, while using Unity allowed for an easy way for me to visualize the project during development, I feel I may have shot myself in the foot. I lack the ability to interact with the program completely through the command line, meaning that resources like the iLab are out of the question for me. Regardless, I'm grateful for this opportunity to be able to further my skills and learn C#, Unity, and game development.

5.3 Credits

Certain libraries were not included in the version of C# I'm required to use in Unity. So I had to take some code from the internet:

- `Utils/PriorityQueue.cs`: The official .NET priority queue implementation ported from GitHub.
- `Utils/ThreadSafeRandom.cs`: For use in `ParallelShipGen.cs`, which relied heavily on randomization in parallel environments. By Andrew Lock.