Introduction

Following the proposal of my project, I wanted to get my midterm project to work as I was unable to do so.

1. Drop a robot into a simple gradient environment and have it reach the source.
2. If that worked and I had time I would use a more sophisticated network or evolutionary algorithm.
3. Add a battery to the agent

I did however manage to make significant progress from my midterm project and managed to get that working.

The agent is on a 2D plane.

My code can be found here: https://github.com/Evolutionary-Robotics/programming-project-8-final-project-rhit-Dhillos

Component Descriptions:

For stage-1, I decided to reimplement my code from the midterm project to ensure that I did not have any bugs in the code. I have provided a description of the environment, body, fitness function, network and evolutionary algorithms in this section.

Environment

For the environment I settled on having a sound emitting object and the sound levels in the environment would be represented by a "sound gradient" which would be normalized from 0 – 1.

I tried various types of generations but ultimately settled on a rather simple but realistic design which would let me test out how the robot would behave in an open field with a sound source. The sound gradient would be strongest at the source and gradually reduce in a piecewise function way. I also tried multiple mathematical functions for this gradient.

One was just reducing it as a function of the inverse square distance rule. So the gradient would be: $g = \frac{1}{d^2}$ where g is the gradient value at a (x,y) coordinate and d is distance from source.

Shown in Fig 1.

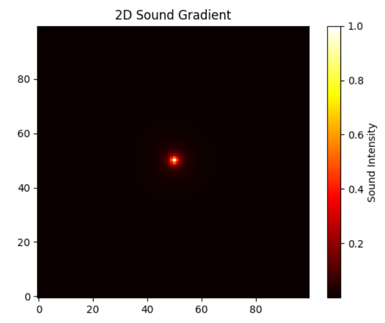


Figure 1: Gradient plot for inverse square distance rule. The gradient difference reduces faster than ideal and most of the grids have minute gradient changes

As seen in Fig 1, the gradient values using this method dropped much faster than I wanted, so I switched to a different method.

Another method was dropping the value by a certain drop-rate every couple of units based on distance from the source. So, the equation would be something like: $g = max - (dr \cdot d)$ Where g is the gradient value, max is the maximum value at source, dr is drop-rate and d is distance from source. Show in Fig x.

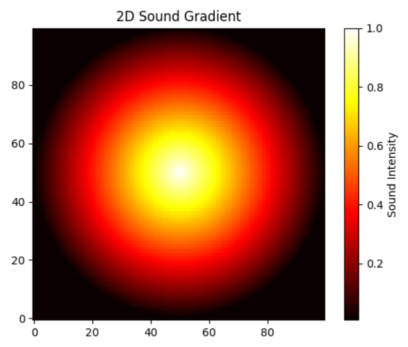


Figure 2: Gradient plot with piecewise and drop-rate based distance

Agent:

With the environment fixed I then decided on how the agent would work. I decided to give it only 4 degrees of movement meaning it would only go in 4 directions in a grid-like manner: North, South, East and West with a default step size of 1 unit. The sensors in the agent would look at the grid around it and take the values next to the agent in these directions. The diagram below illustrates how the agent sensor takes the gradient values (Fig x).

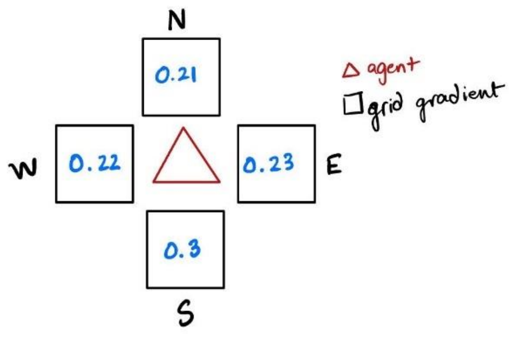


Figure 3: Agent input diagram. Shows agent looking at grid gradient around the agent's current position

From these values just a simple algorithmic function would be to choose the highest gradient value and move in that direction to move closer to the source. Which is what I tried next to verify my agent code was correct and that with the current gradient generation it was possible to navigate to the source. An example agent run is shown in Fig 4.

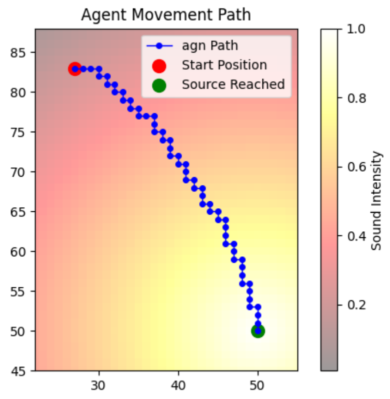


Figure 4: Ideal path taken by agent using algorithmic methods

Genetic Algorithm

I used the MGA from previous projects, with the genotype being the neural network weights and biases. The evolution used tournament style to select higher fitness individuals which means that it selected neural networks which performed better and merged them into lower fitness networks. I used default recombination and mutation probabilities at 0.5 and 0.01 respectively. Although I did try to manipulate these probabilities as I will describe in the results later.

Neural Network

The neural network was an FNN used in the previous programming projects. However, the inputs and outputs were different. The network was connected to the agent's "sound" sensors meaning the gradient values in the 4 directions (N, S, E, W) the agent takes in act as inputs to the neural network – so it has 4 inputs which are the gradient values to the agents North, South, East and West. There are 4 outputs which tell the agent the direction it should move in. It is a one-hot encoded output which means the direction with the highest gradient value will be 1 and the rest will be 0. An example would look something like so:

Directions: [N, E, S, W]

Input: [0.21, 0.23, 0.30, 0.22]

Output: [0, 0, 1, 0]

The hidden layer size and number varied and is section specific. I will mention it where relevant.

Fitness Function

I added agents on multiple starting points by fixing the starting point for the agent for each run (Fig 5.). The fitness function would average out whatever fitness each agent gave. This would have the result of trying out many spots and routes for an agent on each run helping the network evolve and coming up with solutions faster. I can vary where each agent starts and how many agents start for each iteration. This method has the benefit of teaching the agent to reach the source from any direction and theoretically it can find more robust and numerous solutions.

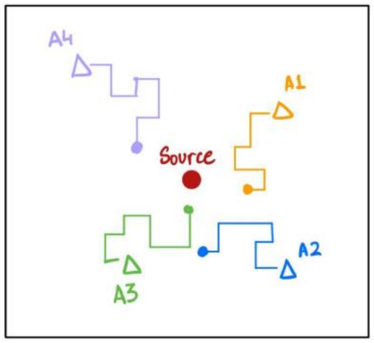


Figure 5: Agent starts diagram. Visual representation of each generation runs. Triangle is the starting position of agent. Circle is the last position for agent. The red circle in center is sound source

Function 1:

This fitness function was:

Simply calculating the distance at the end of the agent run and averaging with other agents and then normalizing the calculation between 0 and 1. The higher the distance from the source the lower the fitness (i.e. closer to 0).

This was the final fitness function I tried for the midterm project. First, I wanted to fix the problem from the mid-term project where the fitness function was high even if the agent did not really have impressive behavior. For example, the fitness function would assign a high fitness function for an agent starting close to the source and moving a few steps and oscillating back and forth (Fig 6).

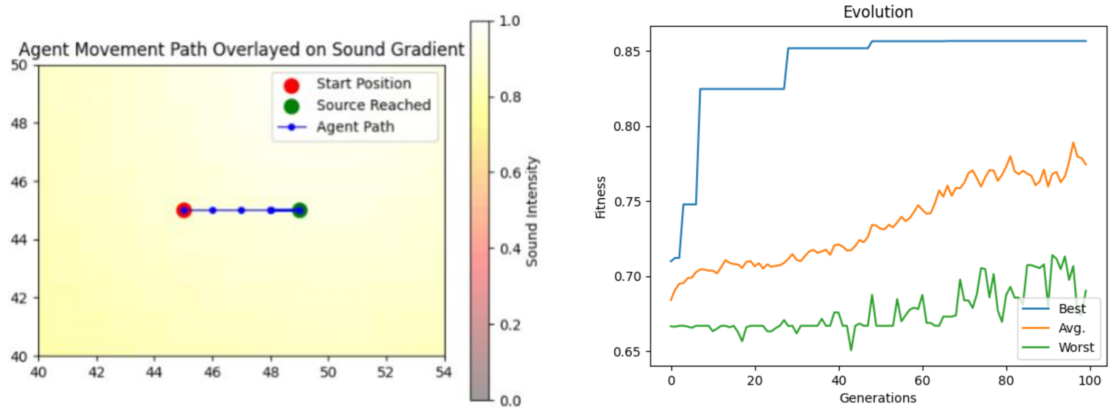


Figure 6: Path showing agent being stuck at some point still move closer to the source (indicated by the thicker line near the green dot). This agent gets a fitness close to 87% out of 100%. This is pure luck, and the agent should not get high fitness for this.

Iteration 1

This was the first problem I tried to fix. I did so by punishing the agent for not reaching the source (or an acceptable radius around it typically set to 1.5units) by dividing it by a scalar multiplier (by default I did 0.5, so halving the value). I also multiplied the fitness by a scalar multiplier if it did reach the source (by default set to 2, so doubling the fitness).

I am unsure if this was the point that did the trick or if it was just me rewriting all the code that removed a bug, but I was able to see my first success in training the network.

This was on all the default settings for evolution – mutation probability 0.05, recombination probability 0.5, tournaments 100, population 100 – and an FNN of structure [4, 4] and no hidden layers – as in the midterm project my experiment showed me the problem is simple enough to not require hidden layers.

As advised, I took the problem in steps and tried to train the agent with only one starting point at (30, 30) with the source at (50, 50). I also tested it at the training point first to see if I saw expected behavior.

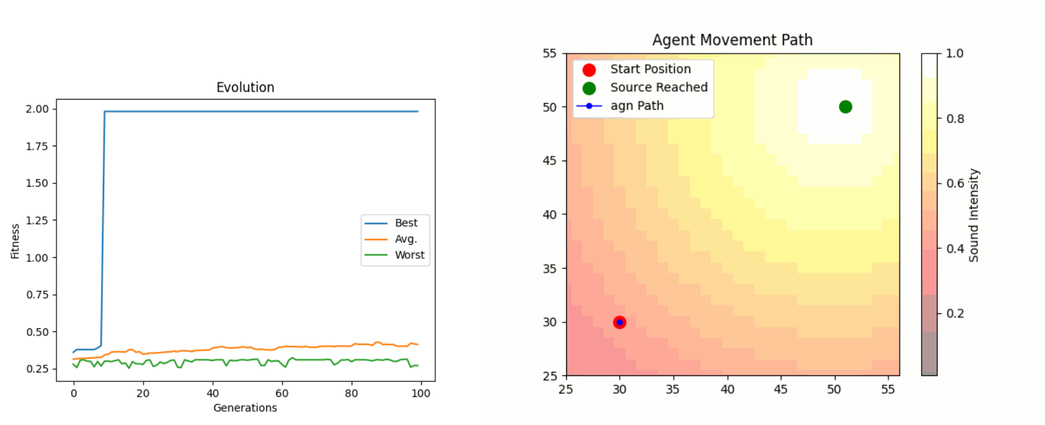


Figure 7: Fitness and movement path of agent with Iteration 1 fitness function. The fitness is 2 because of the doubling on reaching the source, I found it hard to normalize that, so the max fitness is 2 in this instance while lowest is still 0.

As it can be seen here the agent is successfully able to reach the source. Indicating the network is learning correctly. It also works at other points in the quadrant most times, but is susceptible to fail sometimes.

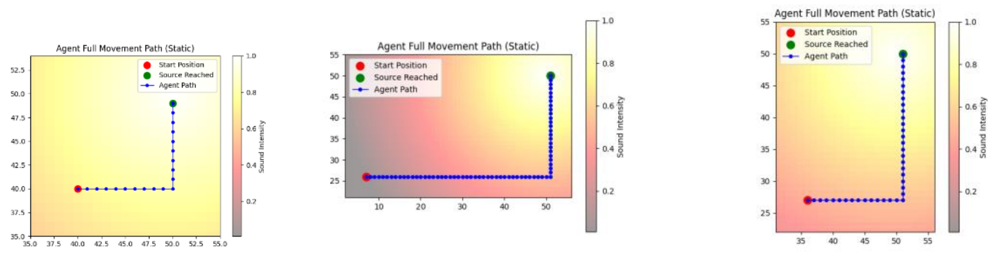


Figure 8: Example successes using Iteration 1 fitness model. Note that all lie in the bottom left quadrant which is one limitation discussed later.

However the big problem with this result was that it only worked if the agent was in the same quadrant as the training agent position. Testing starting the agent at another quadrant gave the same problems as the midterm with it getting stuck and oscillating or not consistently moving towards the source (Fig 9).

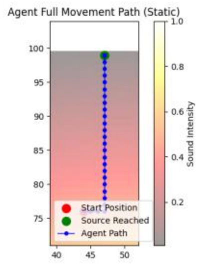


Figure 9: Example failure lies in the top left quadrant thus performs poorly

The process of solving this problem is also my experiment and is detailed in the Experiment section. I also tried a second fitness function that is detailed in next section Iteration 2.

Iteration 2:

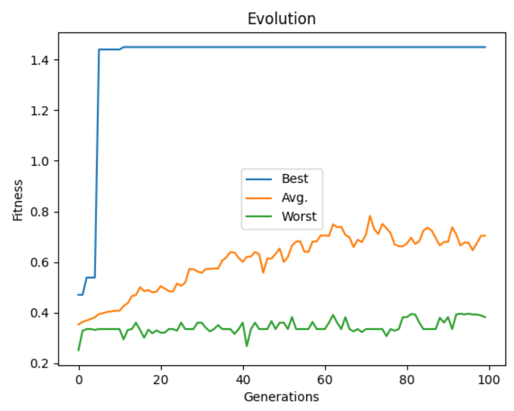
This fitness function is a more sophisticated fitness function. The agent's behavior is assessed using three components: final distance from the source, cumulative progress toward the source, and total reduction in distance.

The fitness function was designed to address issues where agents either failed to approach the source or oscillated near it without reaching an acceptable proximity (typically set to a radius of 1.5 units). My main goal was to reward meaningful progress while penalizing inefficient behavior.

I began by normalizing the final distance from the source, assigning higher fitness to agents closer to the source. If an agent reached the source (or the defined radius), I rewarded it by doubling the final distance score. Conversely, I halved the score for agents that failed to get close. To ensure that agents were consistently moving towards the source, I tracked the reduction in distance over time and the proportion of moves contributing to progress. A progress score punished agents whose movements were erratic or oscillatory by scaling their fitness down if fewer than 90% of their moves reduced the distance.

This restructured function incorporated weighted contributions from final distance (60%), cumulative distance reduction (20%), and progress consistency (20%). These values can be changed according to what I weigh as more important. I weighed the final distance the heaviest since that is the most important factor. It should contribute the most.

This fitness function showed more optimized paths and was more reliable in reaching the source when starting from the same quadrant the source was in. An example success is shown below. For this one I picked the training point in the top right quadrant as opposed to bottom left to show that I can now reliably train the agent to work from any quadrant (Fig 10).



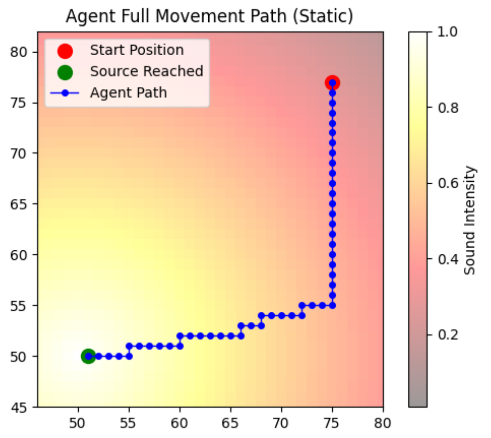


Figure 10: Fitness graph and path visual for Iteration 2 of Fitness function. The Fitness is over 1 because of the bonus upon success, same as before I wasn't sure how to normalize that correctly. The example path shown is much more optimized however due to the added optimization factors in the fitness function.

But it showed similar problems when starting agent from a different quadrant that the training point did not lie on.

A partial solution was found by incorporating the multiple start points in each run and averaging them out. This is also what my experiment is based on.

Experiment:

From my experiment I wanted to try and see how the number of starting points translated to the agent being successfully able to reach the source from any direction and not just from the quadrant it was trained in. The logic is that by averaging the fitness results from starting at different quadrants in each run the agent can learn more robust behavior and can successfully reach the source from any direction. Refer to Fig 5 for a visual of this explanation.

To test the evolved network performance, I will start the agent at 100 different points around the source in concentric circles. Each circle will have 25 positions for the agent uniformly distributed. These 100 points will be fixed for every experiment run (Fig 11).

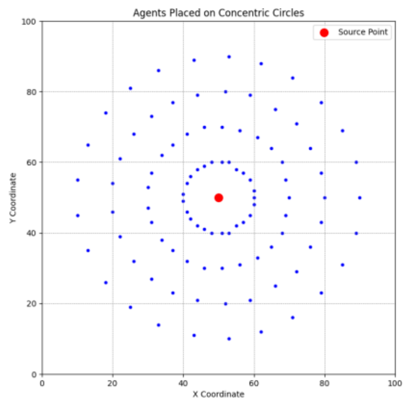


Figure 11: Position of agents (blue) and source (red). Circles are of radius 10, 20, 30 and 40 with 25 points each

Neural Model

FNN network → [4, 4]

4 inputs, 4 outputs and no hidden layers

Activation = sigmoid

Evolution/Training Parameters:

Population size = 100
Recombination Probability = 0.5
Mutation Probability = 0.05
Tournaments = 100 * Population Size
Source = (50, 50)

Observable / Measure of Interest

How the number of agents starting in the training phase affects the overall success of the solution.

Variable:

Agent starting positions.
1 agent: Starts at (30,30)
2 agents: Start at [30,30) and (70,70)]
4 agents: Start at [30,30); (70,70); (30, 70); (70, 30)]
8 agents: Start at [30,30); (70,70); (30, 70); (70, 30);

(80,80); (20,20); (80, 20); (20, 80);]
12 agents: Start at [30,30); (70,70); (30, 70); (70, 30);
(80,80); (20,20); (80, 20); (20, 80);
(80,80); (10,90); (90, 10); (90, 90);]
16 agents: Start at [30,30); (70,70); (30, 70); (70, 30);
(80,80); (20,20); (80, 20); (20, 80);
(80,80); (10,90); (90, 10); (90, 90);
(40,60); (60,40); (40, 40); (60, 60);]

Repetitions:
generations = 100
The evolution took place for 100 generations for each number of starting positions.

Results:
1 Agent:

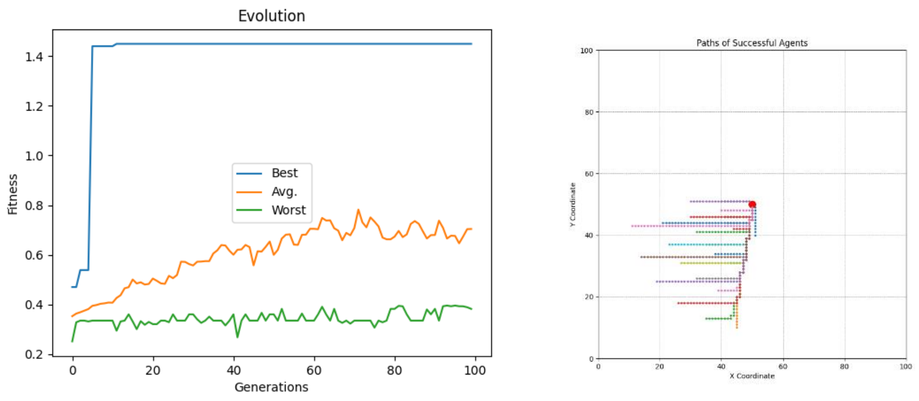


Figure 12: Fitness function (right) and successful agents paths graphs (left) for training on 1 Agent. The agent only learns behavior for training quadrant as mentioned above in the report and seen in the graph paths.

2 Agents:

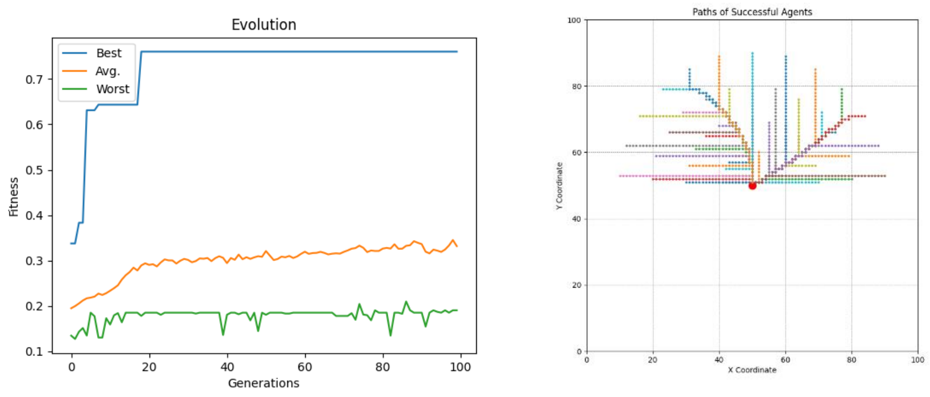


Figure 13: Fitness function (right) and successful agents paths graphs (left) for training on 2 Agents. The agent is now learning behavior for 2 quadrants as seen in the graph paths.

4 Agents:

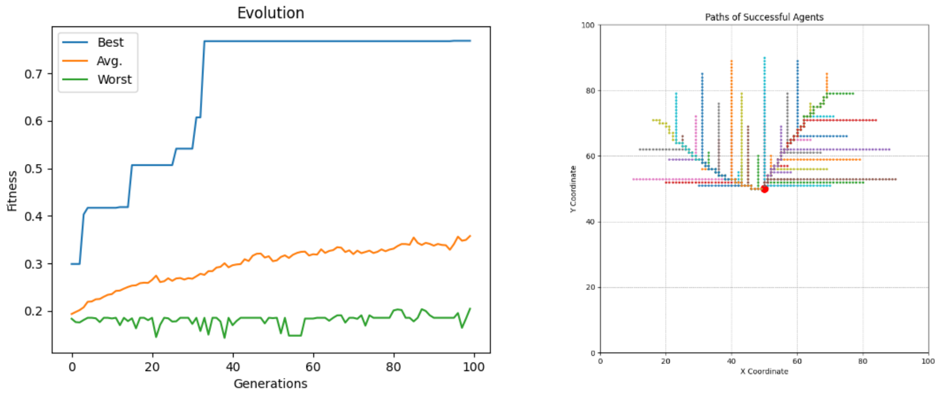


Figure 14: Fitness function (right) and successful agents paths graphs (left) for training on 4 Agents. The agent is still learning behavior for 2 quadrants as seen in the graph paths however the paths are different, indicating it is still learning.

8 Agents

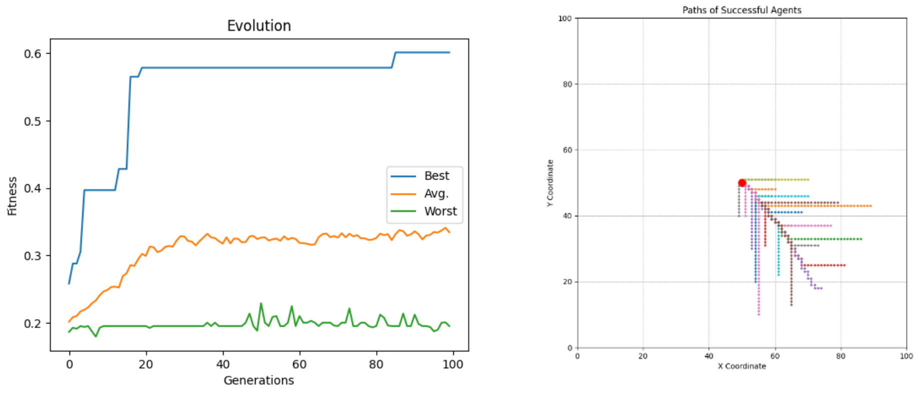


Figure 15: Fitness function (right) and successful agents paths graphs (left) for training on 8 Agents. The agent learns traversing in a different quadrant and performs worse. Could be overfitting or just unlucky.

12 Agents

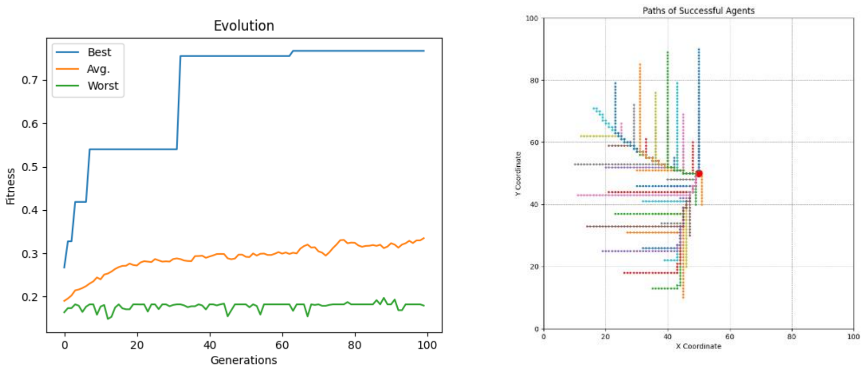
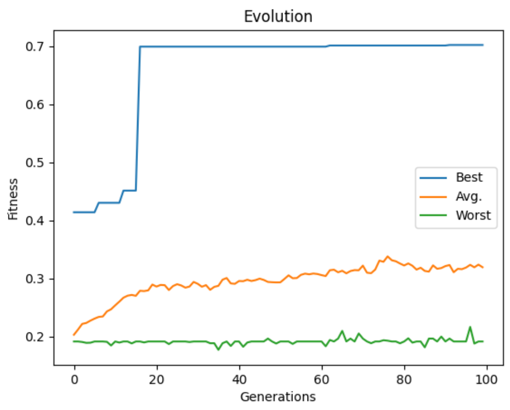


Figure 16: Fitness function (right) and successful agents paths graphs (left) for training on 12 Agents. The agent is performing like 4 agents, but in different quadrants. Quadrants could be different because of luck

16 Agents:



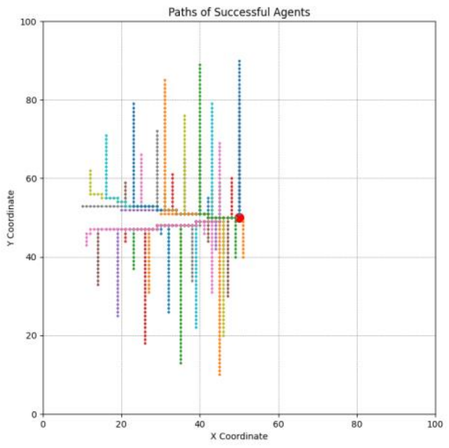


Figure 17: Fitness function (right) and successful agents paths graphs (left) for training on 16 Agents. The performance seems to be stagnating

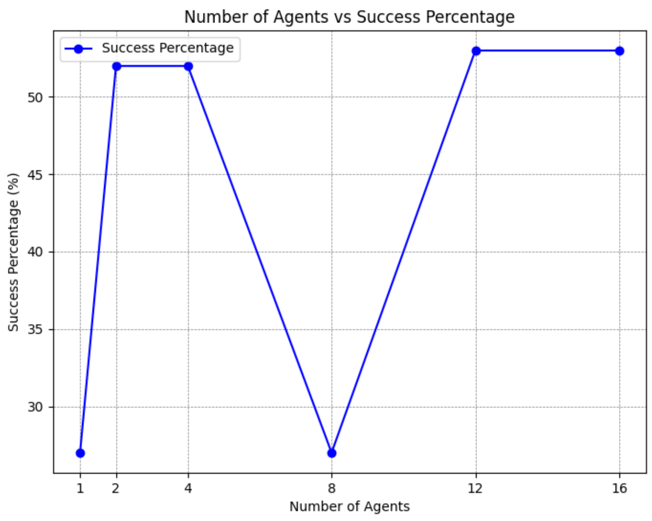


Figure 18: Success rates out of 100, for agent numbers 1 to 16. Ignoring 8 agents result as an anomaly the learning seems to stagnate, and the agent is only able to learn how to reach source from at most 2 quadrants.

Interpretation:

As seen by the results above, the fitness decreases but performance increases and then stagnates. The decrease in fitness could just be caused due to the average fitness across the agents. The agent performs better training from 2 and 4 agents than just training from 1 agent. However, it decreases again with 8 agents, this could be a one of anomaly due to bad luck in training or resulted by overfitting. The latter option is less likely however due to the run with 12 agents picking back up again.

Further Questions:

Given more time I would like to explore whether this is stagnation due to the network stopping learning or overfitting.

I would also like to see if it is indeed stagnating or overfitting then why? Is it because the network size is too slow or some other more nuanced and complicated reason.

I did try it with a higher network size but the agent was still only learning to reach the source from 2 quadrants so I would like to explore this result further given more time.

Conclusions and Limitations:
There are still several improvements that can be made. For one having the agent successfully be able to reach the source starting from any quadrant is still not achieved. I expanded on my thoughts about this in the Interpretation for the Experiments section.
Ultimately this could be caused by a wide variety of reasons, some of which I can try and expand upon in the future. I had limited time and was unable to spend too much more than the recommended time on the project. Another huge limiting factor was my hardware. Each training run with more than 2 agents took several minutes to close to an hour and it was very difficult to find and fix implementations due to this.

If I had more time or made use of the external, more optimized training libraries like TensorFlow I might have been able to progress further

<u>Future Work:</u>

One way is to successfully implement a more advanced Neural Network like a CTRNN which allows for more robustness for tasks concerning robotics or an Evolutionary algorithm like the Elitism which could also help find more sophisticated solutions and in fewer generations.

I would also like to get to the true optimizing part by giving an energy meter to the agent.

Overall, this time around I had much more success and was much more systematic in my approach as hopefully it shows. I obtained better results and was able to show my results in an appealing visual manner with the experiment.

Extras:

There are a lot more things I tried and failed at which I haven't mentioned for the sake of brevity and clarity in the report. I tried to implement TensorFlow, but it was not successful as I had trouble combining it with the evolutionary algorithm – I had bugs in the code and errors which I was unable to fix so I moved on to other methods. I also tried to evolve the weights in a different way with multiple agents. Instead of training them simultaneously in the fitness function I tried to instead do it one by one. For this I changed the code to be able to pass in the evolved genotype (i.e. weights and biases) of a previous run and feed them into the next run and evolve behavior that way. However, this was unsuccessful as the fitness did not meaningfully increase and the results instead got worse – this could be due to a bug in the implementation, but I am not sure.

I also tried to change the evolutionary algorithm from tournament style to elitism however that was also unsuccessful as my fitness was stagnating – again could be due to bugs in the code.

There are also quality of life and convenience features I implemented which helped me throughout the project. For example, I used the 'pickle' library to store my final evolved weights and biases such that I can utilize them later and make the code more object oriented and clear. It also helped me on several occasions to go back and verify previous results, make changes to visualizations etc.