Project 3 IT3708 Torgeir

A)

Parameter	Description
Cross over rate = 0.05	Remained the same for the whole development
Mutation rate = 0.05	Tried to vary this to create stable plots in non-static runs
Adults = 800	Started smaller, increased to see more variance
Generation per run = 100	Started out at 60 because I thought that was the maximum, then I upped it to 100. Evolution seems stagnant anyway, so it might not matter
Tournament probability = 0.05	Pretty arbitrary value, seems to produce good results, see no reason to change it
Tournament size = 80	One tenth of the population size, seems fine.
Produced children = 800	Was always the same as the adult population size. Effectively creating full competition between children and adults to survive

The fitness function works by looping over the cells the agent lands in after, potentially, moving, and adding a number to the score based on the kind of cell. After having looped over all the scenarios, the value is divided by the amount of scenarios. So the fitness score is the average of the fitness scores over the scenarios. Initially I was using sigma scaling, which forced the fitness values to not be negative, so I would have to normalize negative scores to positive. I did this after taking the average instead of inside the loop. Later on I changed to tournament selection, partly to allow negative scores.

The fitness function is not very math heavy. In practice it works like this.

```
Value = 0.0

For each scenario:

    For each move in scenario:

        If cell landed in is food:

            Value += 2.0

        If cell landed in is poison:

            Value -= 4.0

        If cell landed in is empty:

            Value -= 0.25:

Value /= (amount of scenarios)
```

These values are in practice what was experimented on the most. The reason to penalize empty moves is to encourage the agent to not move in circles and strongly encourage it to choose food when given the choice. Note that in cases where the agent does not move, that counts as an empty cell.

B)

The latest artificial neural network consists of an input layer of six nodes. The output layer has three nodes and there is a hidden layer that consists of five layers. Each node in the input layer corresponds to either a food or poison detector. If food is detected, the relevant node gets its value set to 1.0, otherwise it is set to -1.0 Likewise, the poison nodes are set to 1.0 or -1.0 depending on whether there is poison in the relevant direction.

All nodes other the ones in the input layer use a transition and activation function to assign its own value. The values of the parent nodes get multiplied by a weight given by the EA. The weights are effectively double values between -1.0 and 1.0 in 256 steps, because the genotype of the weights is a byte array. One weight is simply a byte sized value divided by 128. The next step is to take the average of these products by dividing by the length of the node layer above. Then the rest of the

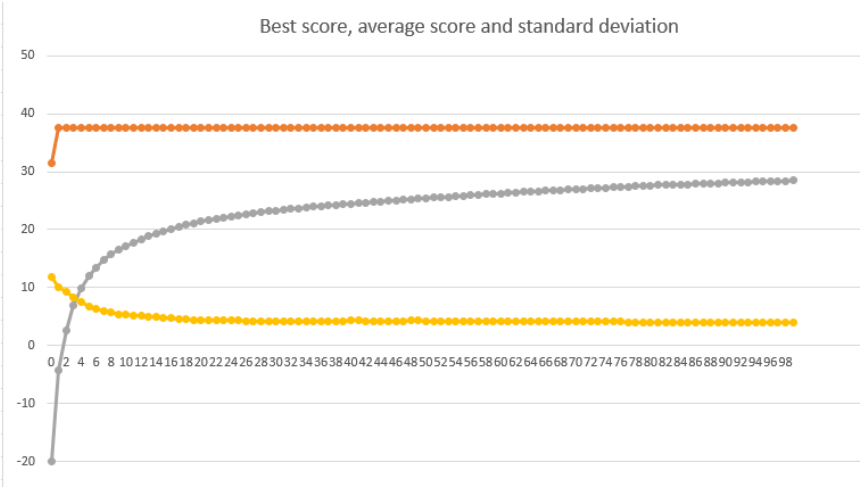
```
value = Math.abs(value) >= 0.5 ? Math.signum(value) : Math.sin(value *
Math.PI);
```

If the value is between -0.5 and 0.5, then a sinus function is applied. The sinus function is modified so that it has a minimum for a value of -0.5 and a maximum of 0.5. If the value is greater than 0.5 or lower than -0.5, then the value is set to 1.0 or -1.0 respectively.

In the end, all the nodes in the output layer get activated. The whole point of the network is to choose a move. The chosen move depends on the values of the output layer only. More specifically, it depends on which of the output layer nodes has the highest value. Depending on which one had the highest value, the move is either left, up or right. However, the highest value needs to be higher than the given threshold for the agent to move. The threshold at the moment is -0.25. If the highest value is lower than this, the agent will not move. This threshold is low, so in practice that does not happen. Ideally the agent should only stop if all the options are poison and it is late in the run, which ultimately happens rarely, so it is not very important to make the agent more likely to not matter. Also note that until the weights are changed, the outcome of one set of input is fully deterministic.

There were a number of changes before this design was reached. Initially there was a design where the values ranged from 0.0 to 1.0 However, this was a bad design because it was hard to make the agent to anything else than either not moving or only moving left. This was partially because at the time, the first node above the threshold would determine the move. When the input layer only gets zero values, all values in the network will be zero, which is obviously bad. So I ended up changing to allow negative values, both in the nodes and in the weights. Also, in the beginning, the network had two layers of six nodes. Then I experimented with one layer of nine, one layer of six and a couple of other amounts. In the end a hidden layer of 5 nodes seemed to be sufficient, because there are only 27 possible input sets, and if we thing about the nodes as having values that are either low or high, then we realize that five bits should be enough to encode 27 possibilities. More nodes would only provide redundancy. The real downside would be that the amount of weights needed grows quadratically to the size of a layer. It would be hard for the EA to find optimal weights when there are ore weights to optimize over, so a low amount of nodes seems like a good idea.

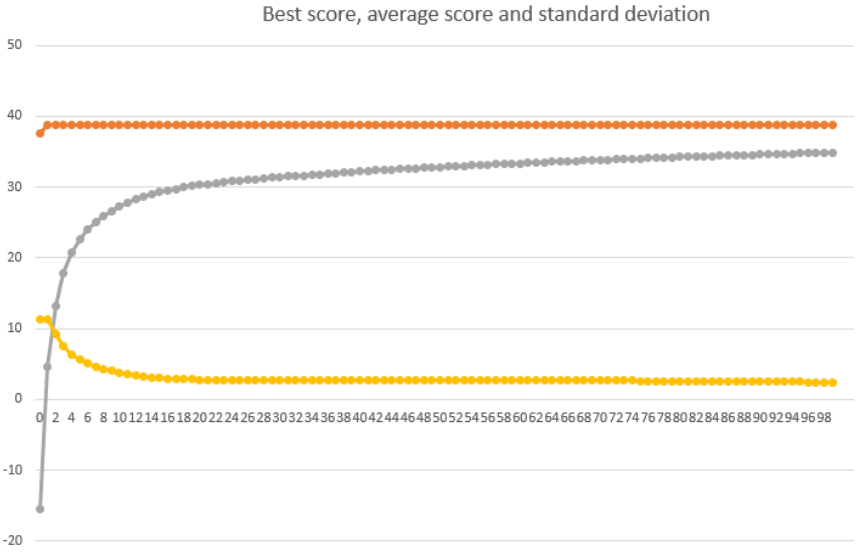
C)



This graph shows the best score, average score and standard deviation in a static run with one scenario.

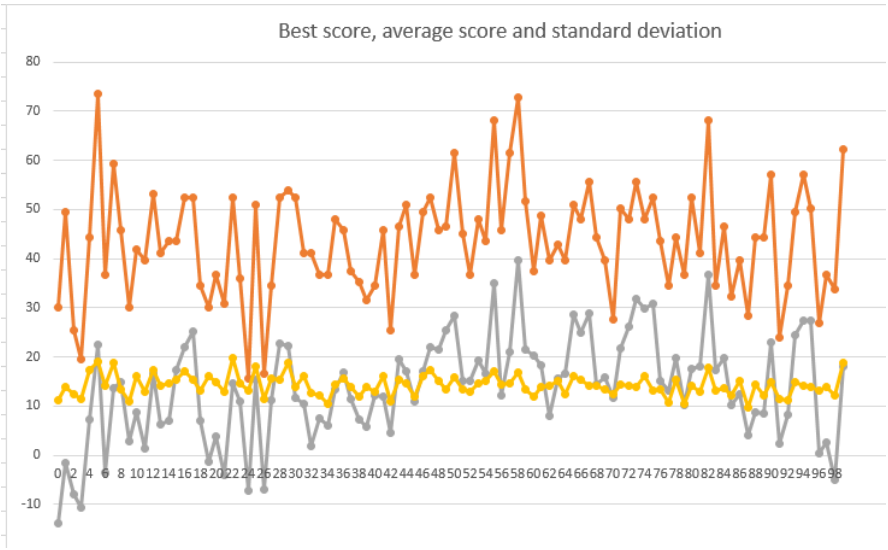
All of the moves done by the agent are optimal in the sense that it ate food when given the choice and avoided poison when possible. Some of the choices seemed somewhat arbitrary, probably because it was only trained on one board. For example, when there was poison to the right and nothing else, the agent chose to move left. After the 60 timesteps, the agent left eight food units out of the 28 that were originally present on the board. One poison was eaten because the agent was surrounded by poison.

On another static run, this time with a new random board for the visualization, the agent was mostly doing well. The agent completely avoided poison, and was not even forced to eat poison. However, the agent moved to the right when there was food on the left side and the other directions had empty cells. This seems very arbitrary, and might have been a property that made the agent more optimal for the board it was trained for



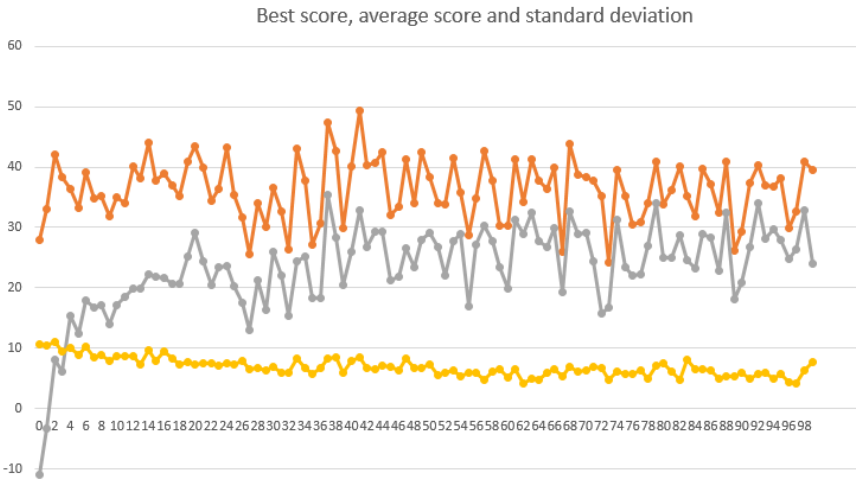
This plot is from a static run with five scenarios. This time the agent made all the reasonable choices with one exception. When presented with poison on both the left and right side and nothing in front of it, the agent would eat the poison to the left. And in one of the scenarios, the agent got caught in a circle because it was moving to the right both when seeing one poison in front of it and when seeing one poison to the left.

Now in a static run with five scenarios and new random scenarios for the visualization. This time the agent had one defect, specifically when it saw two poison units, positioned in front of it and to the left. Then it would eat the poison in front of it. It could have learnt this because the five scenarios it was trained on provided this choice very few times. Otherwise the agent made all the smart choices.



This plot is from a dynamic run with one scenario per generation. The fact that the board layout is random between generation means there is a lot of variance between the generations.

The agent had two defects. When it detected poison to the left and right at the same time, it ate the one on the right. This only happened once. The other defect happened several times, which made the agent look really bad, was that when it sensed food to the left and poison in front of it, it moved to the right. Otherwise it made smart choices.



Last plot is from a dynamic run with five scenarios.

This time it had two defects, which were similar. When it detected food to the right and nothing else, and when it detected food on the right and poison to the front, it did not eat the food, rather it would move forward or to the left respectively.

Differences between these cases are interesting. On static runs, the agent can learn habits that are good only for the boards it is tested on. During dynamic runs, a random new board can potentially give a huge penalty for a small defect. Also, the agents are not tested on the new boards generated for visualization, so the best agent for these might not be the same as the best agents for the boards in the last generation of testing. My static runs had the best score stagnate and stay the same for the entire run. Logging reveals that the stagnant scores are due to the same object being the best option in all the generations. In the dynamic runs, using more scenarios per generation seems to have reduced the variance, which is obviously great.