**Pathfinding AI Report**

This project is to compare the effectiveness of the widely used A\* algorithm to the non-deterministic AI methods, genetic algorithms (GA) and artificial neural networks (ANN). To do this I conducted research to determine the most effective AI method to implement and compared it to A\* when put to the same pathfinding problem.

**AI Methods**

The methods I have been asked to consider are a simple GA or ANN. An ANN is a system that attempts to mimic the processes of the human brain. It is made up of a set of neurons that sum the all their inputs multiplied by the weights of those inputs. The inputs are carried along synapses to the next layer and each synapse has a weight. The ANN changes its thought process by editing these weights.

The problem can be abstracted to pattern recognition where finding the correct weights will solve the pattern. ANNs excel at extrapolating a pattern from a limited data set. Their limitations come when faced with a problem without a pre-solved data set such as finding the path through a maze. Without a data set to test on there isn’t an easy way to train the ANN. A possible solution to this could be making an ANN able to solve any new maze after training on mazes for which the optimal path is already known. This could possibly allow the ANN to generalise rules for solving mazes and act like a living creature trying to solve a maze from the inside.

Another issue ANNs encounter is that they’re rather heavy on processing power and memory space as many sets of weights must be saved along with input sets and the output of every single neuron in the network. There isn’t a real solution to this problem as neural networks require a lot of data in order to function and there isn’t a way to simplify them for a computer.

GAs don’t have this problem as they are designed to mimic evolution. It does this by generating a population of individuals, each with a set of decisions. Each decision is considered a gene and the decision set is a chromosome. The decisions are processed to generate data which determines how effective an individual is at the task. This is then put through a fitness function, mine maps all values between 0 and 1 so that a perfect individual has a fitness close to 1 and a weak individual tends towards 0. This fitness is divided by the total fitness of all individuals, which finally determines the individual’s chance of passing on their genes. More fit individuals have more chance of being chosen to breed. Individuals breed by swapping a chunk of their genes with each other which is called crossover. Then a small mutation chance is applied to every single gene, in my case I used 0.1%, and if a mutation happens that gene is flipped.

This method uses much less processing power and memory space than an ANN because the only storage needed is the decisions of each individual and the only training done is using a random chance for a very simple function to occur. This allows a GA to run and train much faster than an ANN. It also has the advantage of not needing a training set to solve a problem. The only information the GA would need is its distance from certain points at the end, and in my case, I improved its intelligence by penalising moving into a wall in the fitness function. Due to these significant advantages I have chosen to implement a GA to solve this pathfinding problem over an ANN.

**The Competition**

The algorithm I will be comparing my AI to is called A\*. This is an algorithm which always finds the most optimal path to a goal through a maze through a logical process. It makes use of 2 dynamic lists, an open list and a closed list, containing every node that it has checked so far. First it places the start node into the closed list and adds every valid node surrounding it into the open list and calculates their cost, defined as the distance from the start + the distance from the end. Then it selects the node in the open list with the lowest cost and repeats the process. When nodes are added to the open list, they contain a pointer to the node currently being checked when they are added. This allows the algorithm to trace back the final path once the end has been reached. The algorithm terminates if either every node has been checked, in the worst case, or the end node has been reached. This allows it to optimise by not checking any nodes it doesn’t have to. Once the end node has been found the pointers are traced backwards all the way to the start node in order to generate the final path to be outputted.

This algorithm is extremely fast and is almost always the optimal solution for a pathfinding problem as it will find the optimal route by checking as few nodes as possible. This algorithm will provide the benchmark for testing my non-deterministic AI against.

**Testing Criteria and Methods**

For this problem I will be comparing the processing time of my GA to that of A\* in order to find which one finds the optimal path faster, and if my GA can find the optimal path. For each map I will run the A\* algorithm to find the optimal route 5 times and record the processing time of each run using the ctime library’s clock function. This will record the time at the start and end of the algorithm then output the time taken in milliseconds. I’ll run the algorithm 5 times and take the mean as there is some variance in processing time due to background applications on the computer affecting the program. This will provide a benchmark for my GA in terms of the path it must find as well as its processing speed.

After testing A\* I will run my GA on the map 5 times and record the processing time the same way as with A\*. this will show me the time it took in milliseconds again and allow me to compare to the A\* algorithm. I will also follow the path the GA generates and check to see if it follows the most optimal route towards the goal. this systematic method will give me a clear view of the GA’s abilities compared to the deterministic method of A\*.

**Results of Testing**

**Map 1: A\***

Run 1: 2 milliseconds, down, right, down, right, right, right

Run 2: 1 millisecond, down, right, down, right, right, right

Run 3: <1 millisecond, down, right, down, right, right, right

Run 4: <1 millisecond, down, right, down, right, right, right

Run 5: <1 millisecond, down, right, down, right, right, right

This gives an average processing time of ~3/5 milliseconds. This isn’t perfectly accurate as the processing time for 3 tests was too small for the time resolution to measure. A\* also provided the exact same solution every single time as was expected for a deterministic algorithm.

**Map 1: GA**

Run 1: 3041 milliseconds, down, right, down, right, right, up, right, down

Run 2: 1245 millisecond, right, up, down, down, down, right, right, right

Run 3: 1299 millisecond, right, down, down, right, right, up, right, down

Run 4: 1235 millisecond, right, down, down, right, right, right

Run 5: 1224 millisecond, right, down, down, right, right, right

This gives an average processing time of 1608.8 milliseconds. The first value could be considered an anomaly but that was the only run where the algorithm didn’t immediately find the solution on the first generation. This is because I’m running with a population of 1000 which may be excessive for this problem. Even disregarding the first result it’s clear to see that GA is magnitudes slower than A\* in processing time as it must perform rather complex calculations due to the fitness function and the randomisation, whereas A\* only needs to perform addition to find the cost of each node and compare these results. The final 2 individuals found an alternative optimal path to A\* which uses the same number of moves, however the other 3 took a short detour which added an extra 2 moves to the path.

**Map 2: A\***

Run 1: 1 millisecond, down, right, down, down, right, down, right, down

Run 2: 1 millisecond, down, right, down, down, right, down, right, down

Run 3: <1 millisecond, down, right, down, down, right, down, right, down

Run 4: <1 millisecond, down, right, down, down, right, down, right, down

Run 5: 1 millisecond, down, right, down, down, right, down, right, down

As before A\* finds the same path with each run. The average was the same this time as well.

**Map 2: GA**

Run 1: 3769 milliseconds, down, right, down, down, right, down, down, up, down, right

Run 2: 1356 millisecond, down, right, down, down, right, right, down, left, right, down

Run 3: 20490 milliseconds, down, right, down, down, right, down, down, up, down, right

Run 4: 52106 milliseconds, down, right, down, down, right, down, down, up, down, right

Run 5: 25839 milliseconds, down, up, down, right, down, down, right, right, down, down

This map is much more complex than map 1 and this causes the GA to take much longer to find the correct path. It still often takes detours along the optimal path, but every detour is followed by moving back to where it was resulting in 2 wasted moves but remaining on the optimal path. With this map is took many generations to find the optimal path which is why there is a much greater processing time to find the optimal path. On this map the GA got a much higher average of 20712 milliseconds.

**Evaluation**

From these experiments it is abundantly clear that in terms of raw processing speed GA is at least 1000 times slower than A\* which is multiplied further as the map complexity increases. If the goal of this solution is purely to find the most optimal path as fast as possible GA will never be better than A\* regardless of how optimised it is. However, a GA might be more appropriate if judged by other criteria.

Firstly A\* uses a significant amount of memory space and this usage increases with the map complexity. A GA on the other hand uses no additional memory regardless of the map complexity as the algorithms work for any problem. The only time complexity affects processing time is if the chromosomes aren’t long enough to reach the end and need to be extended, which would still cause a smaller increase in memory usage than A\*.

Another consideration is the context of the problem. While A\* will always find the optimal path, it looks very mechanical and computerised in the way it finds the path. Meanwhile a genetic algorithm could be more appropriate for a game as an example. This is because a genetic algorithm can appear more alive in the way it solves problems through trial and error without knowing where the end is. This looks much more like a creature lost in a maze than A\* which knows where the end is and always goes directly towards it.

In conclusion the method chosen to use should depend on what criteria the client is looking for. If all that matters are the speed at which the problem can be solved, non-deterministic AI won’t ever be better than using the A\* algorithm which can find the optimal path in a matter of milliseconds. However, if the algorithm is for a context where the system should appear more alive and thinking, non-deterministic methods may be more appropriate for solving the pathfinding problem.