**Artificial Intelligence for Pathfinding**

**Contents**

*1.0 – Introduction*

*2.0 - Genetic Algorithms(GA) versus Artificial Neuron Networks(ANN)*

*3.0 - The Development of the Pathfinding AI*

*4.0 - Genetic Algorithms(GA) versus A Star(A\*): Experimenting*

*5.0 – Conclusions*

*A1.0 – Appendix: References*

*1.0 – Introduction*

In this report, I will be debating between few methods of pathfinding and elaborating on how and why I designed my own AI Pathfinding technique. The covered subjects will include A\* Pathfinding, Genetic Algorithms (referred to as GAs hereafter) and Artificial Neuron Networks (referred to as ANNs hereafter).

*2.0 - Genetic Algorithm(GA) versus Artificial Neuron Network(ANN)*

*2.1 - ANNS*

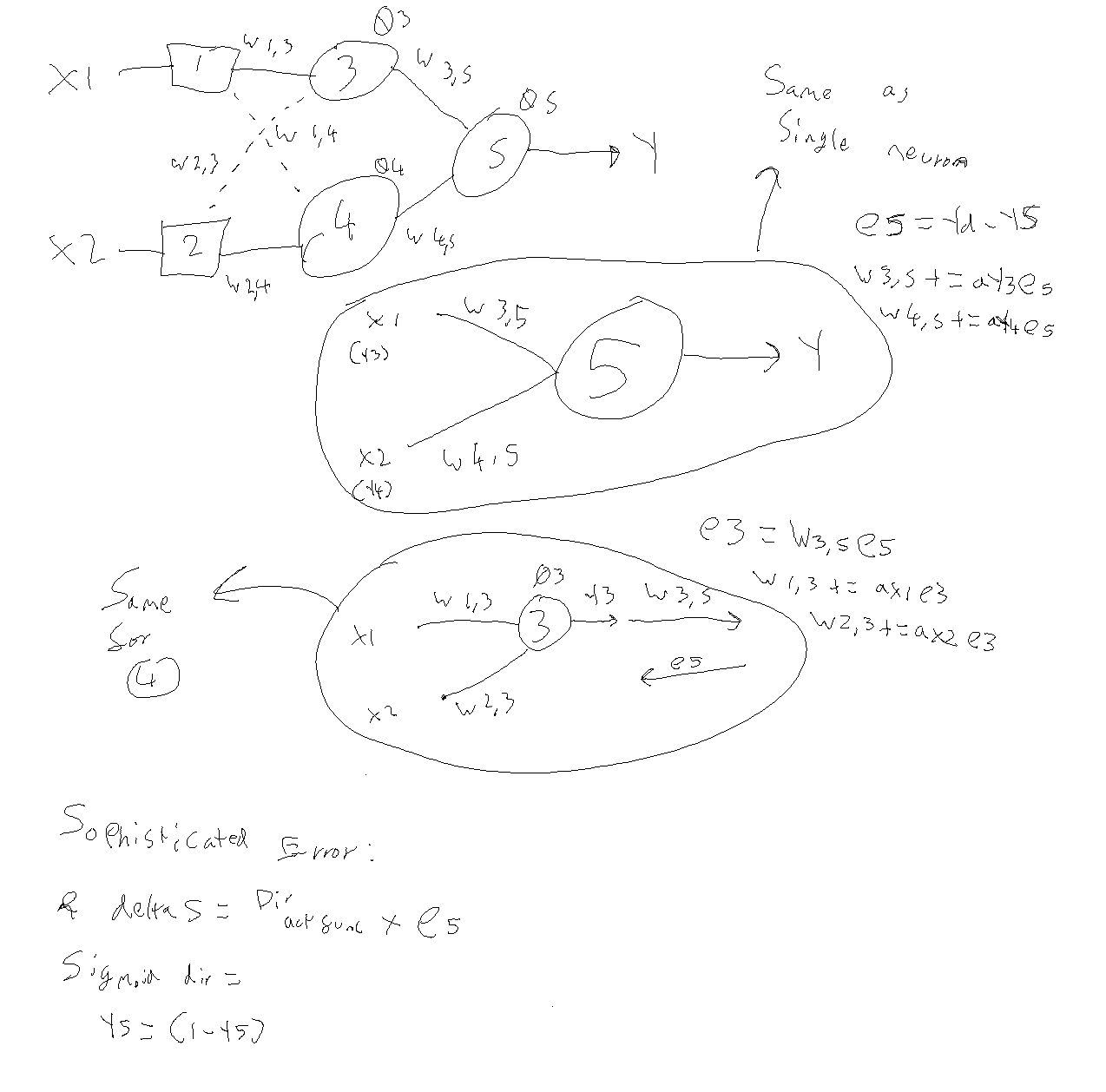
ANNs, coincidentally like GAs, take inspiration from naturally occurring principles. In the case of ANNs, they attempt to imitate the way a neuron functions within the human brain. A single artificial neuron can take information from an input channel, manipulate the information according to the weight (connectivity) of the connection between itself and the previous neuron before accepting the information into its main body. From there, it calculates whether the neuron should activate via an activation function and outputs the result of the activation function to the next neuron it connects with. Like the input it receives, it too is modified based on the weight (connectivity) of the bridge to the next neuron. A simplified model of this can be found in figure 1, below.

Figure 1: ANN Diagram

The model ‘learns’ by adjusting its own weight values relative to a provided error factor. When the final output is determined, it is compared to an expected value (which in the case of pathfinding, the ‘expected value’ would be the location of the goal you’re trying to reach). The difference between the final value and the expected value is taken and used to adjust the weights of the final neuron in the inverted-cascade chain relative to the learning rate (alpha), the error factor and the output that previously travelled across this bridge. For example, the weight 3,5 (located in the bridge between 3 and 5) would be adjusted by adding alpha(a) \* prior output(Y3) \* error factor 5 (E5). This continues all the way down the chain until it reaches the start, and all neurons in the sequence have ‘learned’. This entire procedure is then repeated until there are no errors. The error factor can be made more precise by calculating and applying the sophisticated error in place of the regular error factor. This is done by taking the derivative of the activation function and multiplying it by the error factor. A single neuron (perceptron) learns in the same way, only the error is not fed further down the inverted cascade-chain.

*2.2 – GAs*

GAs, while still deeply rooted in natural concepts, associate more with Darwinism: survival of the fittest. A population of virtual genes each hold an array of information. In the case of pathfinding, this is normally an array of integer or binary values representing directions in which to move: combining the moves forms a whole route. In any example, a genetic algorithm is usually a series of ‘steps’ or ‘instructions’ to achieve some sort of criteria or goal. At the end of the gene, it is tested for its ‘fitness’ to solve the problem. In a pathfinding example, its fitness would be relative to the distance from the goal it is, but in an accounting and investing example, the algorithm that gets the most profit on its investments would be rated the fittest.

The way a GA learns is entirely dissimilar to the ANN. Two genes are randomly selected to be ‘parent’ genes. This random selection is influenced by the fitness of each gene, meaning that a fitter gene has a higher chance to be selected as a parent. Once two parents have been selected, they breed and mutate. During this process, there is a chance that the two genes will swap information with each other in a process called ‘crossing over’ (see figure 2, below). This is made to resemble a child’s genes being a combination of both of their parents’ genes in animal reproduction. Additionally, there is a small chance that each chromosome in the gene (each value in the array of steps) will mutate into another entirely random chromosome, altering the data stored within the gene.

After the genes have (or, haven’t) crossed over, you take the results and save them as ‘offspring’. Once breeding has finished between all genes, the offspring replace the gene pool. This process causes the fittest solutions to be much more likely to breed, creating other fitter solutions. The degree of randomness however helps to prevent stalemating and encourages variation in the population. This is because on many occasions, even the fittest solutions need to take inspiration from the less fit solutions to progress and improve in the long-run.

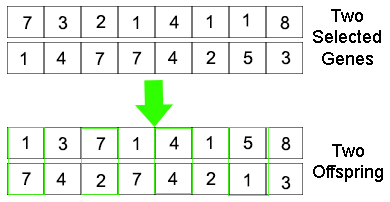


Figure 2: GA Diagram

*2.3 – GA or ANN?*

This particular task required pathfinding from location A to location B on an unknown 2D grid. In the end, I concluded that GAs would be the best solution forward.

First and foremost, in terms of complexity to produce, ANNs are significantly harder than GA’s and much more time consuming. To path find with ANNs, one would treat the output of each neuron in a feed-forward neural network (FFNN) as a direction. The activation function of each neuron would create a value which could be translated as a direction and added to an array of directions, not too dissimilar from a gene. The error factor would be relative to how far away the end is to the goal, and the weights and activation thresholds adjust accordingly (see figure 3 below). Each gene receives two inputs (X, Y co-ordinate), adjusts the values and then gives two outputs (a new location after it calculates the move). This repeats in a chain until the end is reached. Keeping in mind, this is only one **potential** example of an ANN for pathfinding, there are infinite ways you could apply them.

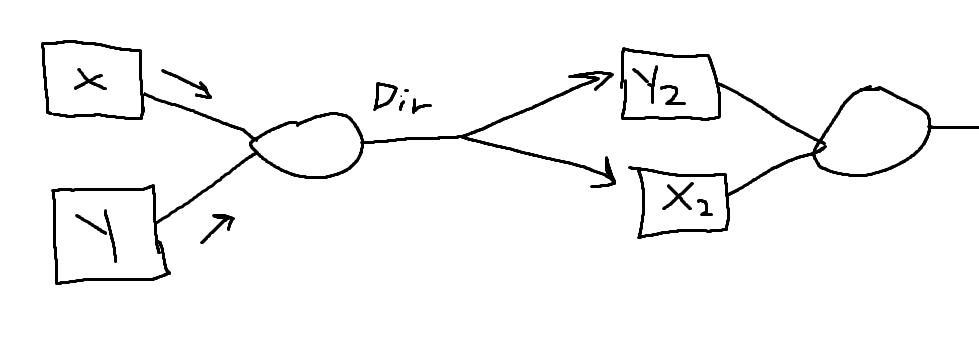


Figure 3: A FFNN

This would take a significantly higher period of time to create and tune than a GA due to its complexity. A gene is an easily readable variable which can be traced, followed and monitored. If an issue arose in the cross over function, it would be simple to pinpoint, unlike a FFNN. Its reliance on connectivity and its sequential series of calculations makes it very difficult to debug and monitor. If an unexpected result kept occurring, it becomes far more complicated to pinpoint the cause of the error.

Additionally, when finding the most optimised route, a GA’s gene can be far more easily manipulated. If a gene reaches the goal using only 10 of its 12 hypothetical chromosomes, it can be shortened and refined simply. Neurons do not have this feature, at least not nearly to the same degree of simplicity. Instead, it would rely on shortening the neuron-chain length (in the example given above. Other ANNs may vary), which would have an enormous knock-on effect to the rest of the neuron chain and probably alter the result entirely. In summary, GA generated paths are better at refining and optimising themselves to become shorter than ANNs.

One reason that ANNs could be considered over GAs however is they are vastly more efficient.

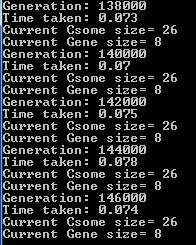
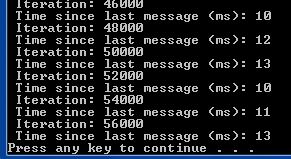


Figure 4: Neuron iteration time

Figure 5: Gene iteration time

In the above figures, 2000 iterations of a GA takes approximately 75ms to complete. This contains 8 genes of 26 chromosomes and a 20x20 grid. This program also uses four threads to split the workload. On the right, 2000 iterations of an ANN took about 11ms. This is on the same machine with the same settings. Now, granted the scenario here is different, as this ANN is not trying to path find, it instead uses 3 neurons to determine an enemy’s response action. Regardless, the process of training a neuron is the same no matter the task it handles. This means that in path finding, if 3 neurons are 7x more efficient than 8 genes of length 26. You could get approximately 21 neurons in a network to run as quickly as 8 genes, before factoring in multithreading. Whilst testing the same scenario would be able to give a more precise conclusion, this is sufficient enough to at least declare that ANNs are somewhat more efficient and can produce more iterations per second. Furthermore, if you consider the calculations involved within the program, the GA has several loops that iterates through the entire population and its chromosomes a few times per frame. This can take a long time over a lot of frames. ANN doesn’t have this problem, as all the neurons are only ever iterated-through twice per iteration (once going forwards with the output, and the second time during backpropagation with the error factor). GA contains significantly more loops which get exponentially bigger with a bigger input, like “For (0 – population size): For (0- population size)”. This causes a population size squared number of iterations, but is a necessary evil.

Overall, despite the potential for increased efficiency, the GA was selected as the most suitable solution for the given task. Partly due to time constraints demanding the easier-to-tune solution, but mostly due to the superiority in fine-tuning and manipulability. For example, if I wished to manipulate my GA to favour straight paths over diagonal paths, doing so would be a simple case of taxing the diagonal move in the fitness. With an ANN, this would be overly complex. One thing worth considering however, is a combination of GA with ANN in a hybrid solution. The weights of neurons in an ANN would be stored as genes and tested for fitness against other networks, retaining and breeding only the fittest networks. This could also be made to factor in a gene made of the final neuron outputs and breed entire ANNs rather than breeding their weights. This example is almost as fast as a pure ANN, as the GA isn’t used as much, while keeping the benefits of manipulability of GA. That said, doing so is more complex than anything previously mentioned and beyond difficult to debug. For this reason, the decision of using GA will be kept to.

*3.0 - The Development of the Pathfinding AI*

Many special features and considerations had to be taken into account for the optimisation of the GA.

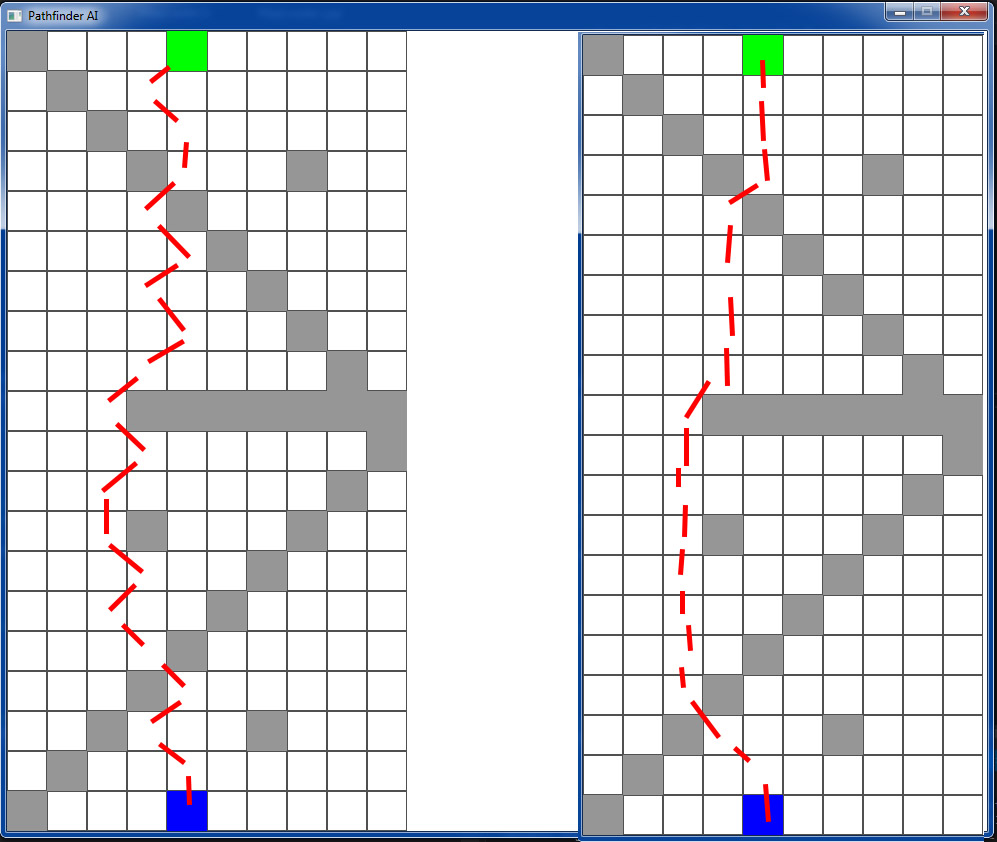


Figure 6: Two routes

In the figure above, both paths use the same number of moves, but the rightmost route appears more natural. The route on the right is the result of penalising diagonal movement slightly so that the GA is encouraged to walk in straight lines when diagonals are unnecessary. The issue with this however, is that in some cases it would cause the AI to refuse to cut corners as it would calculate that it was less efficient (despite using more moves). In the interest of trying to stick to the least moves that it can determine, the more ‘natural’ looking alternative was scrapped for the leftmost example.

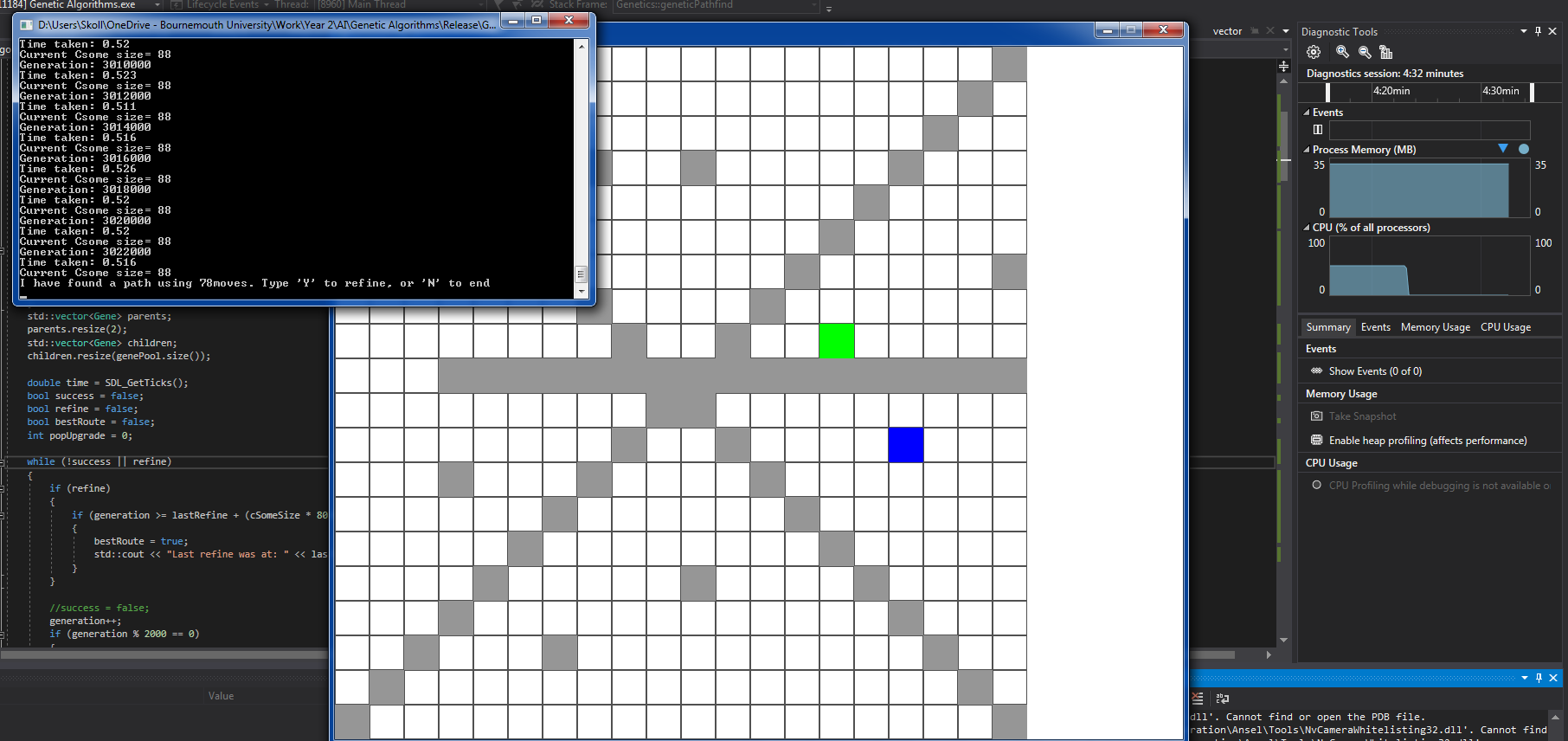


Figure 7: The 'X' map

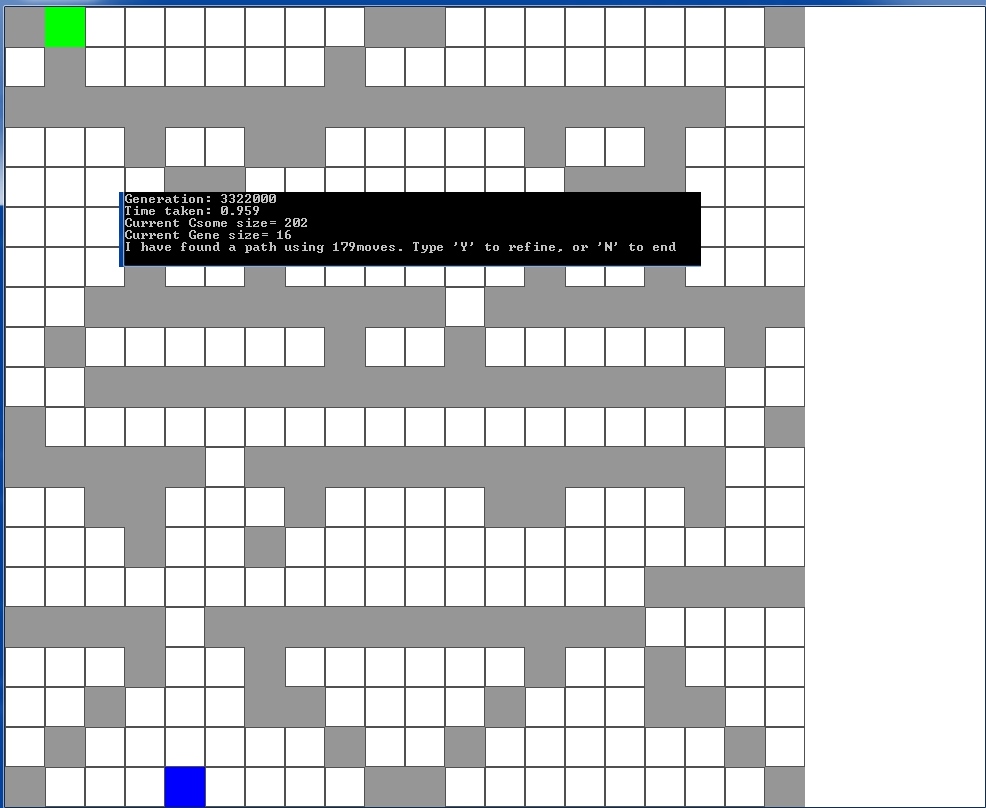


Figure 8: Nightmare Map

Other necessary adjustments and implementations include dynamically deciding the initial population size and gene length based on map size, including multi-threading to optimise the workload, a genocide feature that wipes the population after a certain number of generations to prevent stagnation, gene and population sizes that grow after certain numbers of generations in order to beat the complex maps without being wasteful on the simple maps, population sizes that shrink when refining the answer to find the most efficient solution, a half-genocide feature that replaces half of the population during refinement to prevent stagnation whilst not losing the answer (because half of the population stay) and a simple few lines of code that prevents backpedalling. These features and functionality have allowed me to handle even extreme maps such as the figures above… Eventually.

Fine-tuning these values was excessive trial and error. For instance, I found that genocide was best after roughly 150,000 generations by testing the maximum number of iterations a difficult map needed to succeed. This was apparently around 150,000 iterations. This meant that if it takes any longer, it has probably become stagnant and is in need of wiping. Additionally, the most efficient number of threads to use was 4, according to the time taken per 2000 iterations. I also started by increasing the length of the genes by 2 every genocides, but this caused extreme maps to take far too long, which quickly made its way to a cumulative increase that gets larger every genocide in order to handle the more extreme maps faster (see figure 9 below).

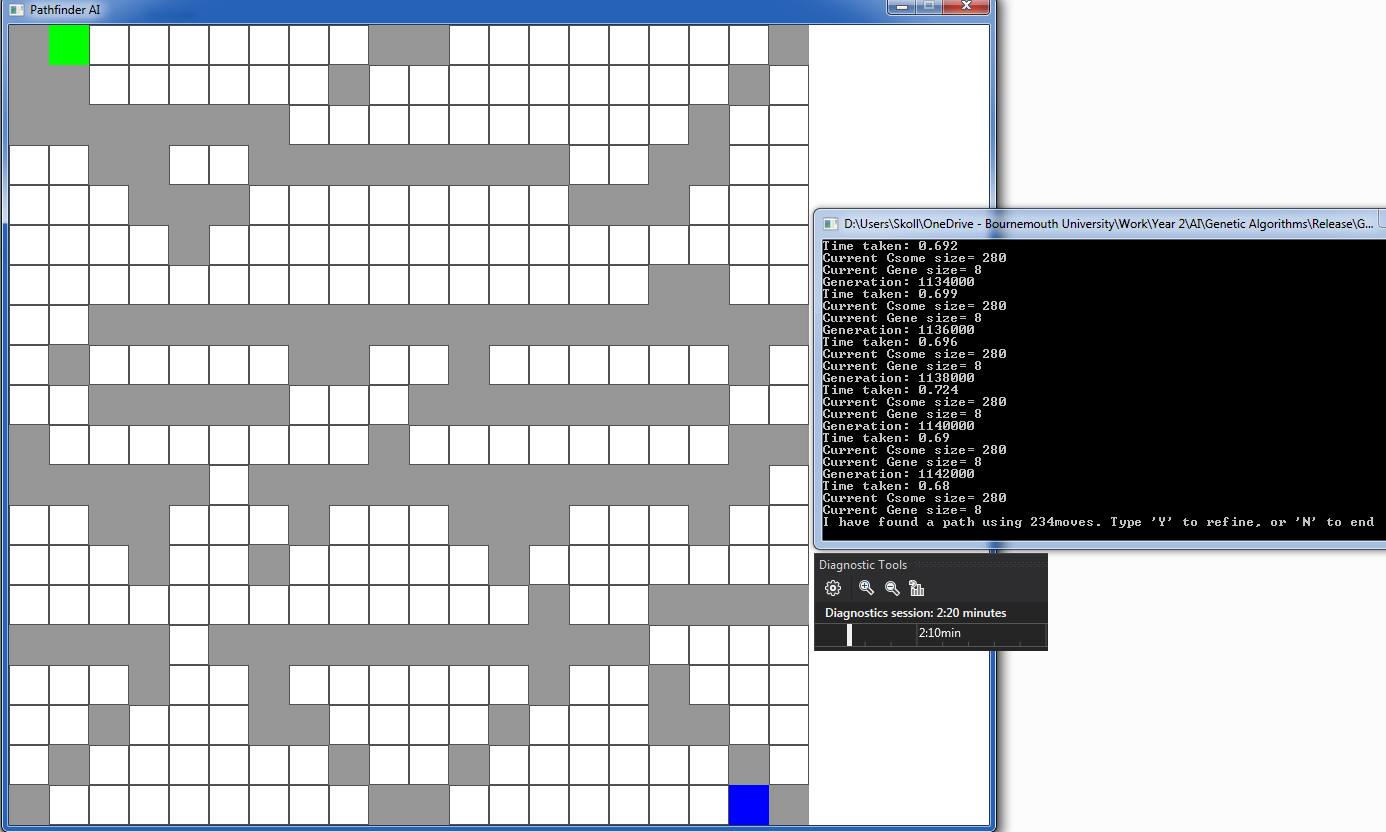
This map is very slightly different to the aforementioned ‘nightmare’ but as shown in figures 9 and 8, after just tweaking the chromosome growth rate, the program succeeded in one third of the generations. It did however use more moves to reach the goal, but refinement should even that out.

Figure 9: A faster success

In the below figure, backpedalling became an illegal move, nearly halving the required moves. The time taken was just shy of 2 minutes (the screenshot was taken slightly late, so the time counter in the figure is inaccurate)

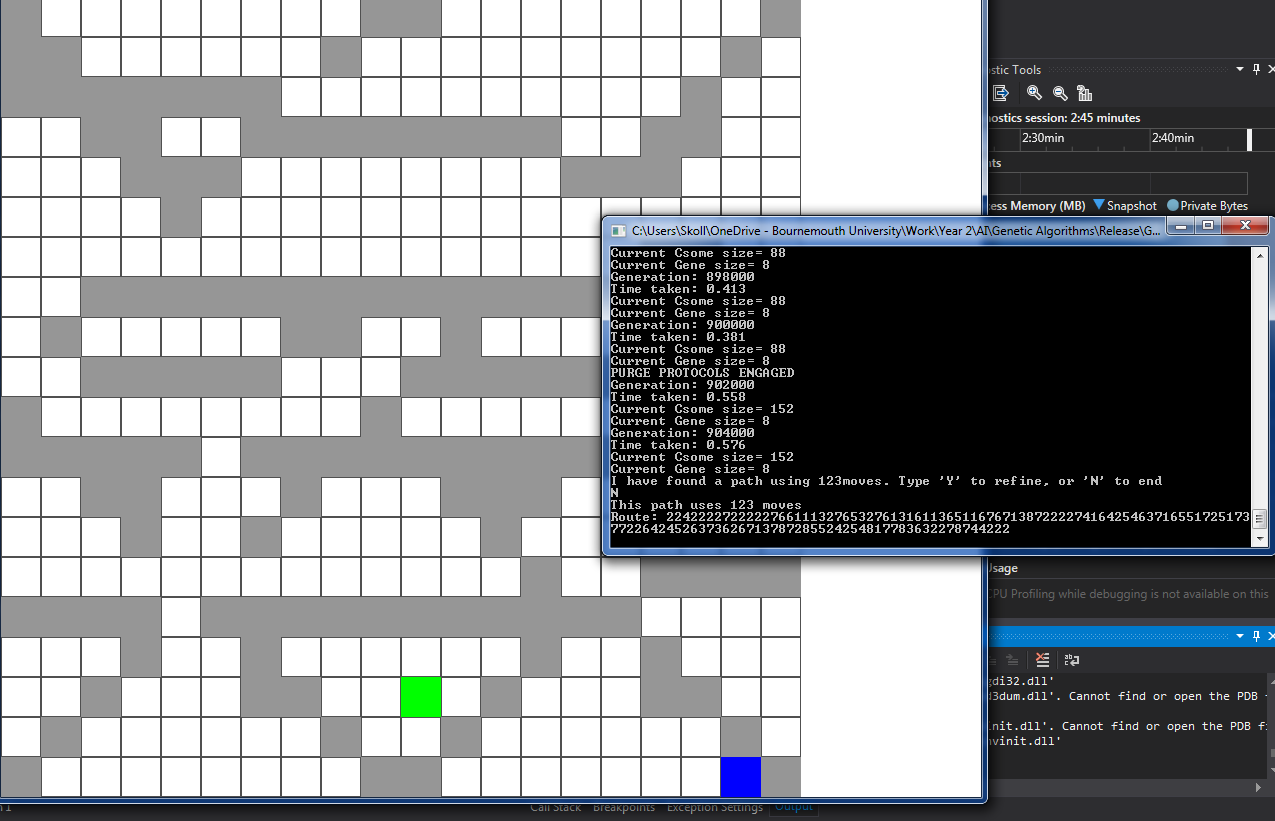


Figure : Banning backpedal

*4.0 - Genetic Algorithms(GA) versus A Star(A\*): Experimenting*

The efficiency of the GA was rigorously tested. On simpler maps, it was able to complete them near-instantly. See figure below:

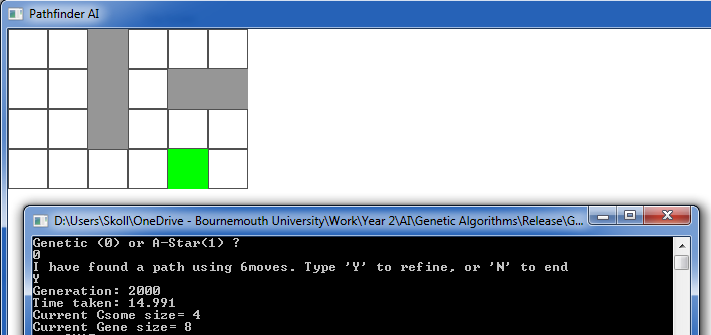


Figure 11: Simple map

In this example, the program took less than 2000 iterations to solve. The “time taken” shown is overly long as it waited for my input. Within 10ms it found a route, and then found the shortest route (4 moves). A\* was also able to find the shortest route on the same map without any difficulty, and in approximately 0ms. Of course it did take time, but it was so small that it was rounded down to 0ms. Noticeably, while both solutions can find an answer near-instantly, A\* is already appearing to be faster in simple cases.

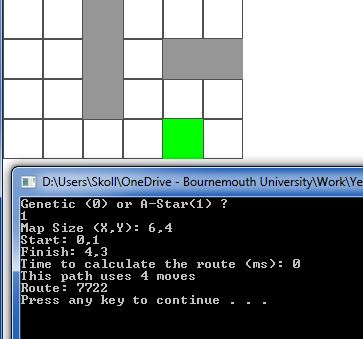


Figure 12: Simple map A\*

Every comparison made between the two is being done on the same machine, and so should be considered relative to each other rather than considering the actual time taken, which will vary on other machines. The GA is multi-threaded between 4 threads, whereas the A\* only uses (and only needs) one. The processor used is an Intel Xeon E31245 3.30GHz (4 core).

In extreme cases, such as the ‘nightmare’ map, A\* still outperforms GAs by an enormous margin. In the below figure, an A\* algorithm not only solved the map instantly (close to 0ms) but did so in the shortest possible number of moves. GA, as shown in figure 9, took just over two minutes to solve and did so in roughly 5 times the number of moves. Refining that answer usually optimises the route down to approximately 150 moves, but refining a lengthy path takes excessive amounts of time (refining takes approximately 5x as long as however long it took to find the path in the first place) and still isn’t the shortest possible number of moves.

In terms of memory usage, GA and A\* are approximately equal. The memory required to store the map is the same in both cases, so the difference between GAs and A\* is that GAs save the genes, store the child genes and a random generator. The A\* stores an open-nodes map and a closed-nodes map, an array of integers. On smaller maps, the A\* has slightly less to save, but overall, the memory usage difference between them is negligible.

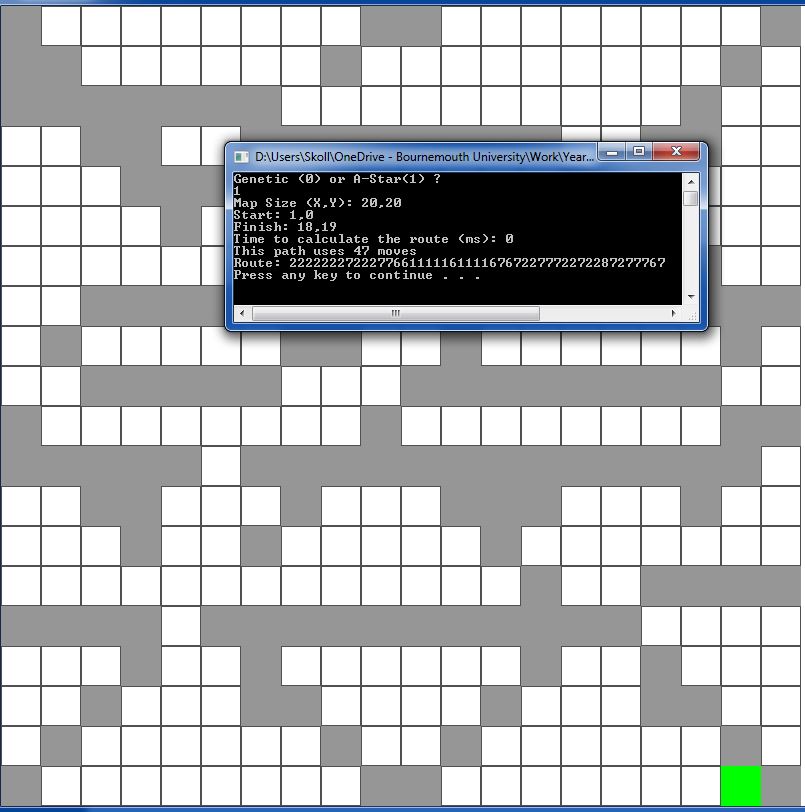


Figure 13: A\* Solves Nightmare

A\* can heavily outperform due to the way it functions. Rather than expensive and enormous loops, it tests travel in every available direction using its open nodes map. Every enterable square is considered as a node, of which there is a total maximum of 400 nodes. Walls and failed paths are treated as ‘closed nodes’. It simply tests any available direction every iteration and retains the direction it travelled to reach that square. If a direction fails or reaches a dead-end, it backpedals and tries again in a new direction. After a mere maximum of 3200 calculations (400 maximum nodes \* 8 possible directions, as the previous direction doesn’t count) it will always, 100% of the time, have successfully tested every possible square on the map. Once it runs out of legal moves it can conclude that the path is impossible, which is something GA cannot possibly do without saving every single square it passes through. This means that A\* will **always** find the goal (or exhaust every possible move) and usually in the shortest possible moves, using thousands or millions fewer iterations than GA.

On the other hand, due to its deterministic nature, A\* cannot ‘learn’ and will always be very easily predictable. GAs and ANNs can learn, generalise and adapt. They become difficult to predict due to their random elements and learning features. In pathfinding, this means that a nondeterministic AI like a GA will act more spontaneously and unusually. This isn’t always a benefit but does make GA worth considering when less direct behaviour is demanded.

*5.0 – Conclusions*

There are a few conclusions that can be drawn from the results of this assignment. Firstly, A\* is largely superior to GA for pathfinding. In terms of features, efficiency and simplicity. The only real drawback that can be claimed is that A\* is deterministic and predictable, which is rarely an unwanted feature in pathfinding. But on the rarer occasions where it is unwanted (such as making bot AI’s for games where the bots are to imitate player actions) a GA may be more suitable, although difficult to manage in real time. A better real time solution to that would be an ANN GA hybrid, although it would still be more expensive than A\*, it would retain its non-deterministic nature.

Secondly, a GA is superior to a non-hybrid ANN. What it lacks in efficiency, it makes up for in simplicity and manipulability.

Finally, while GA may not be an ideal solution for pathfinding, it still has much wider applicability. A\* may be excellent at pathfinding, but it is capable of only that. GA should be considered in other fields such as accounting, map creation and character creation where its ability to make “choices” can be utilised.

*A1.0 – Appendix: References*

Information on ANN GA hybrids:  
Graham, Ross; McCabe, Hugh; and Sheridan, Stephen (2004) "Neural Networks for Real-time Pathfinding in Computer Games," *The ITB Journal*[online] Vol. 5(Iss. 1). Available from: <http://scholar.google.co.uk/scholar_url?url=https://arrow.dit.ie/cgi/viewcontent.cgi%3Farticle%3D1204%26context%3Ditbj&hl=en&sa=X&scisig=AAGBfm0KHnrXU8hvoCS8WZuuHty4PKB7sA&nossl=1&oi=scholarr> [Accessed 11/12/2018]

Pseudocode from which the ANN research was based:  
Prakoonwit, S., 2018. *AGP\_AY201819\_Labs11and12 v6* [online]. Poole: Bournemouth University. Available from: <https://brightspace.bournemouth.ac.uk/d2l/le/content/30724/viewContent/185288/View> [Accessed 12/12/2018]

A\* Algorithm used for comparison:  
Wikipedia, 2018. *A\* Search Algorithm* [online]. San Francisco. Available from: [https://en.wikipedia.org/wiki/A\*\_search\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm) [Accessed 12/12/2018]  
Note: The code used from this page seems to have been updated and drastically altered since the version of the code provided to us. However, credit still belongs to the aforementioned, even if the information is now unobtainable.

Code to organise an array:  
Unknown, 2018. *Write C++ program to sort an array in ascending order* [online]. Unknown: TechStudy. Available from: <https://www.techstudy.org/CplusplusLanguage/Write-C-plus-plus-program-to-sort-an-array-in-ascending-order> [Accessed 01/12/2018]

Code for the random generation:  
Unknown, Unknown. *Uniform real distribution* [online]. Unknown: Cplusplus. Available from:   
<https://scontent-lht6-1.xx.fbcdn.net/v/t1.0-9/47685597_2404592459868024_7160583065167724544_n.png?_nc_cat=107&_nc_ht=scontent-lht6-1.xx&oh=83f252e06dfbcd6e93219a0987c9fc79&oe=5CAA8156> [Accessed 01/12/2018]