AI Game Programming

Report

Maciej Legas

s4922675

Contents

Introduction

Nowadays, there is a lot of development concerning Artificial Intelligence. Some of the biggest companies, such as Tesla or Google, are investing a lot of money and time into implementing AI into their projects. Concerning this as games developers, we should investigate if AI could not be used in the games industry in areas dominated nowadays by algorithms. One of the possibilities is pathfinding, which is one of the most commonly implemented mechanic in games history. Compared to currently used algorithms, could AI be more efficient? We shall begin by introducing two of the most widely AI techniques used: Artificial Neural Networks and Genetic Algorithms.

Artificial Neural Network

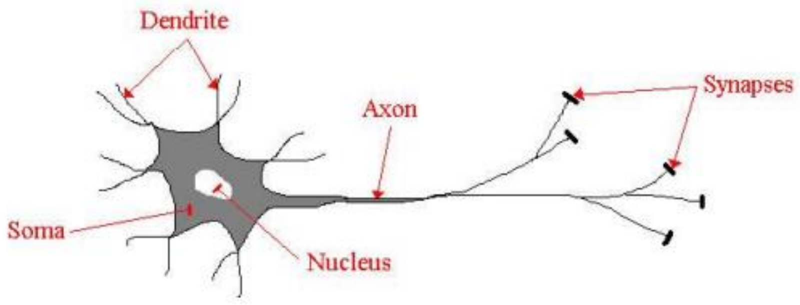


Figure 1 Biological neuron. (Graham et al., 2003)

Artificial neural networks are based on the concept of neural nets in our brains, where in simple words neurons send each other signals to dendrites, which somas sum together and depending on the signals' strength a soma may send a signal to the axon, which sends it across the neural net to other neurons (Graham et al., 2003).

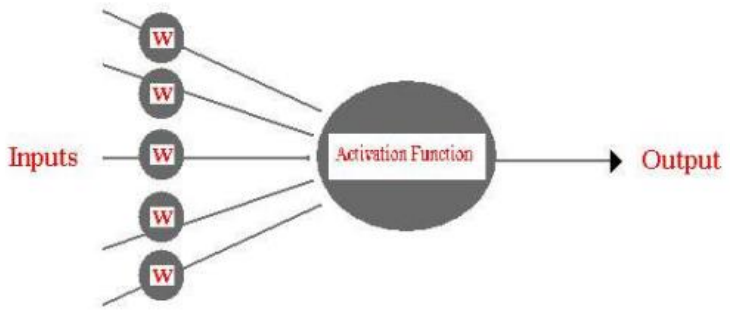


Figure 2 Artificial neuron. (Graham et al., 2003)

In an artificial neuron design we change the signals' strength into inputs with weights, which allow us to calculate an output in the activation function, which would be a soma in a biological neural network (Graham et al., 2003).

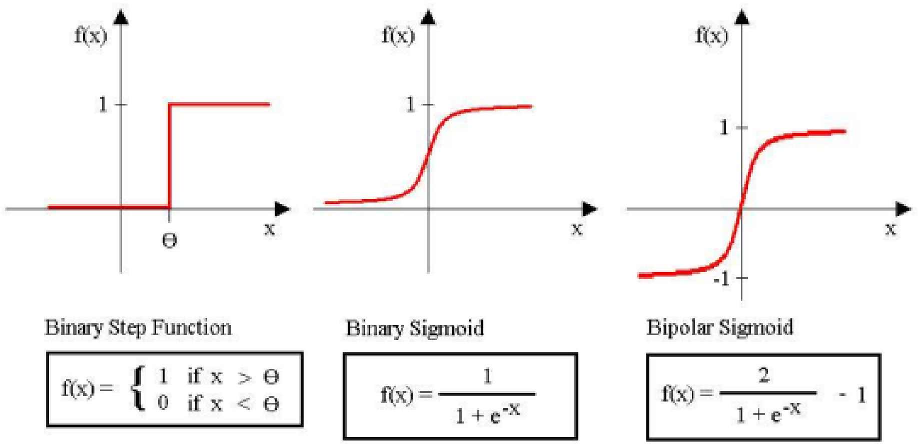


Figure 3 Activation functions. (Graham et al., 2003)

Of course, we can connect neurons together to create nets in case of, for example, inputs depending on outputs from previous neurons. This allows us to put them together as layers, being connected with each other (Graham et al., 2003).

However, for an artificial neuron to know how to deal with the inputs, it needs to learn the expected behaviour. Due to this, it is a popular choice for tasks where there is a common pattern happening, such as game AI – if an enemy AI sees the player, it can make a decision on whether to attack the player or flee from him, depending on the current health of the enemy, which can be treated as one of the inputs. One of the most common approaches is teaching ANN through Genetic Algorithms, which I will be talking about next.

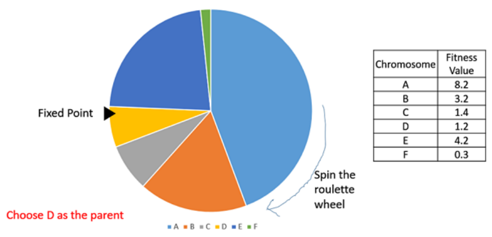
Genetic Algorithm

Genetic algorithms are based on the natural evolution of successful organisms – survival of the fittest. It simulates a cycle of generations, with most of the genotypes crossing over with other ones in their generation, usually the fittest ones and rarely but still – a few of them mutating as well (Graham et al., 2003).

Due to this characteristic, it has been widely used as an optimisation tool to quickly find solutions to problems with hard to describe patterns.

There are three key things to consider when thinking of genetic algorithms:

- Selection
    - How do we choose which chromosomes get chosen to crossover with each other?
- Crossover
    - In what kind of way offspring will be made?
- Mutation
    - How will a chromosome mutate?



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Figure 4 Lottery wheel selection. (Tutorialspoint, 2018)

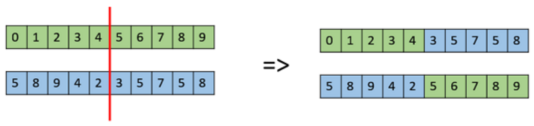


Figure 5 One Point Crossover. (tutorialspoint, 2018)

Choice evaluation

In theory, a highly trained Artificial Neural Network should be able to maintain a reasonable speed compared to A*. However, there is still a possibility that an extreme case shows up which has never been trained before, possibly stopping the whole neural network. Another important thing to remember is that even though the average execution time will be in favour of ANN with the moment it is fully trained, a Genetic Algorithm should be able to solve the problem quicker than an ANN if it deals with untrained data sets. An advantage of GA is that due to their randomness at creation, they help to solve the problem of unexpected outcomes with the maze. Due to these reasons I chose GA for my AI pathfinding task.

A* Algorithm

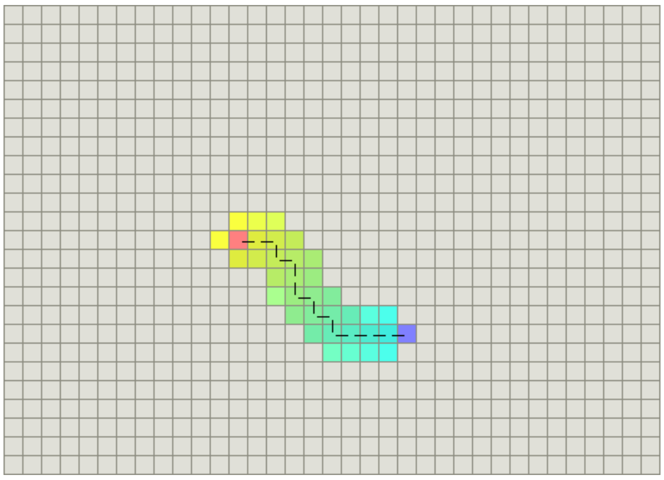


Figure 6 A Algorithm. (Patel, 2011)*

The A* algorithm is an expansion on Dijkstra's path finding algorithm, having its initial version invented by Nils Nilsson in 1964 under the name of A1, and being brought to the current shape we know today by Peter E. Hart in 1968 (Reddy, 2013). The A* algorithm implemented a heuristic-based approach, so that when using A* for traversing through a graph, it will not only check whether the distance cost to travel to a neighbouring node is low – g(n), but also whether is it getting any closer to the end – h(n) (Reddy, 2013).

$$f(n) = g(n) + h(n)$$

The A* star algorithm uses this function in a way that it checks its neighbouring nodes and prioritises movement through the nodes returning the lowest values using the given heuristic, returning search back to other nodes with lowest distances in case a node with the lowest distance returns no possible neighbouring nodes to visit, which can happen if the algorithm already travelled nearby or if the node is blocked by obstacles.

It is an improvement compared to Dijkstra's algorithm, which only focuses its searches for the nodes costing the least distance to go to:

$$f(n) = g(n)$$

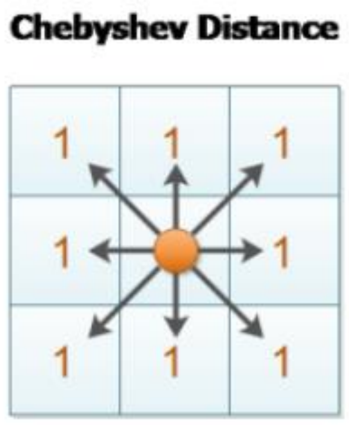


Figure 7 Chebyshev Distance. (IEEE Morocco Section, 2016)

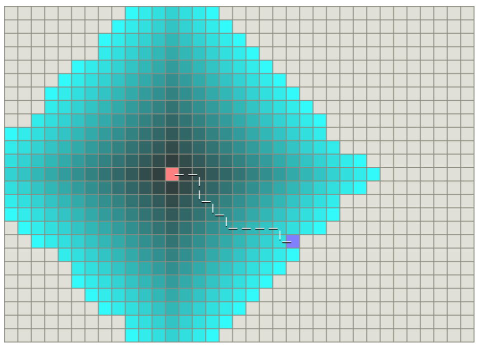


Figure 8 Dijkstra's Algorithm. (Patel, 2011)

In a 2D grid map, using Chebyshev's distance for simplicity, this would result in visiting all of the nodes around the start first and expanding upon that, creating a great circle around it.

Implementing criteria

To begin with, let's start with a simple overview of what should our Genetic Algorithm do.

```
Read the maze file
Calculate the chromosome length
Generate a population of chromosomes
While end has not been found
    Traverse each chromosome from the current generation through the maze
    Select chromosomes for crossover
    Crossover the chromosomes
    Mutate chromosomes
If route has been found
    Display the route
Else
    State that no route has been found
```

To calculate the chromosome length, I firstly looked at the most difficult maze possible for any grid in order to try to generalize the length somehow. Here is an example of a 5x5 grid with the maze.

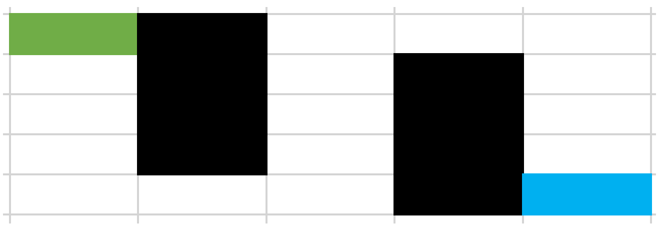


Figure 9 An extreme case.

As we can see, the longest route GA can take should be 16 moves, if it is does not move diagonally. As there are 25 tiles overall, we can create an equation for the chromosome length in case of a scenario like this.

$$x * y - (smaller - 2) * (\frac{(int)higher}{2})$$

Calculating this for a 5x5 grid gives us 19 moves. The division with the variable named "higher" calculates how many possible walls of obstacles like that could be in the maze, while the "minus 2" gives a little bit of extra moves for the chromosome in case it gets stuck on a wall.

What if the grid is 20x20, but the start and end are nearby then? I decided to implement as another criteria an optimization function for GA – if it finds a path and its length is smaller than the current chromosome length, it should resize the chromosomes and search again until it either is too short to get to the end, or it cannot find a better solution. This should bring the results of GA closer to the preciseness of A*.

Generating the chromosomes will be random.

To select chromosomes for crossover, I used a tournament selection, mainly as it is generally way better in terms of improving the overall fitness of chromosomes. I made this choice basing on a graph comparison from one research I have found (Kenan, c.a. 2011):

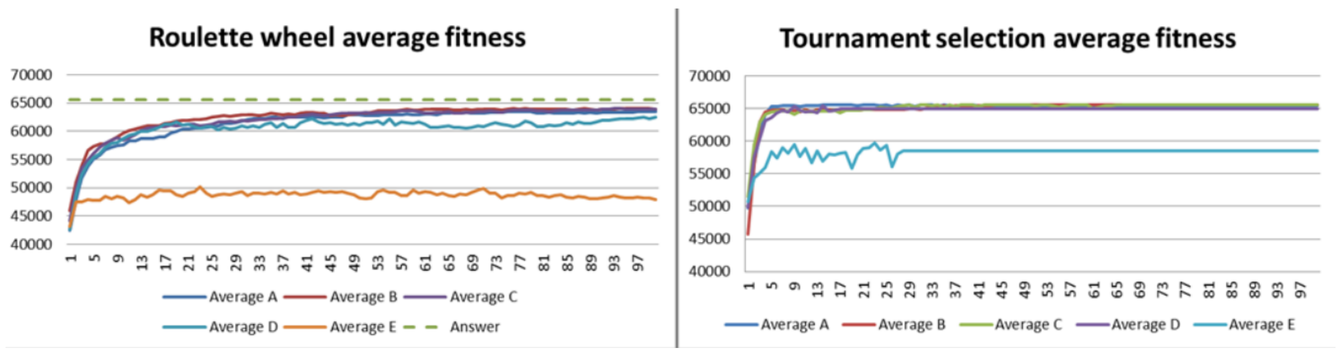


Figure 10 Tournament selection vs roulette wheel. (Kenan, c.a. 2011)

A tournament selection selects a few random chromosomes, picks the one with the best fitness out of them and puts it on the crossover pair list. This ensures that even if a few of the weakest chromosomes have been picked, it is always the best one being taken.

Moving onto the crossover procedure, I picked uniform crossover as in comparison to one-point or multi-point crossover, it allows the chromosomes to mix together in every possible way instead of having just a few crossover points.

As for mutation, since I decided to use integer values instead of binary for the movement mechanic, I have decided to set it to randomize a move whenever a mutation applies, as my tests with using swap or scramble mutation caused my chromosomes to actually get a lower fitness value than before using it.

Having all this, my aim is to implement GA with an accuracy close enough to A*, possibly affecting the execution time on the way. Therefore, the moves possible by the chromosomes will be diagonal as well.

Here is an updated overview of my GA:

```
Read the maze file
Calculate the chromosome length
Generate a population of chromosomes
While end has not been found
    Traverse each chromosome from the current generation through the maze
    Select chromosomes for crossover
    Crossover the chromosomes
    Mutate chromosomes
If route has been found
    Try to find a better path
Else if no route has been found
    State that no route has been found
Else if optimal generation search count is over a limit
    Reverse to last path and display it
```

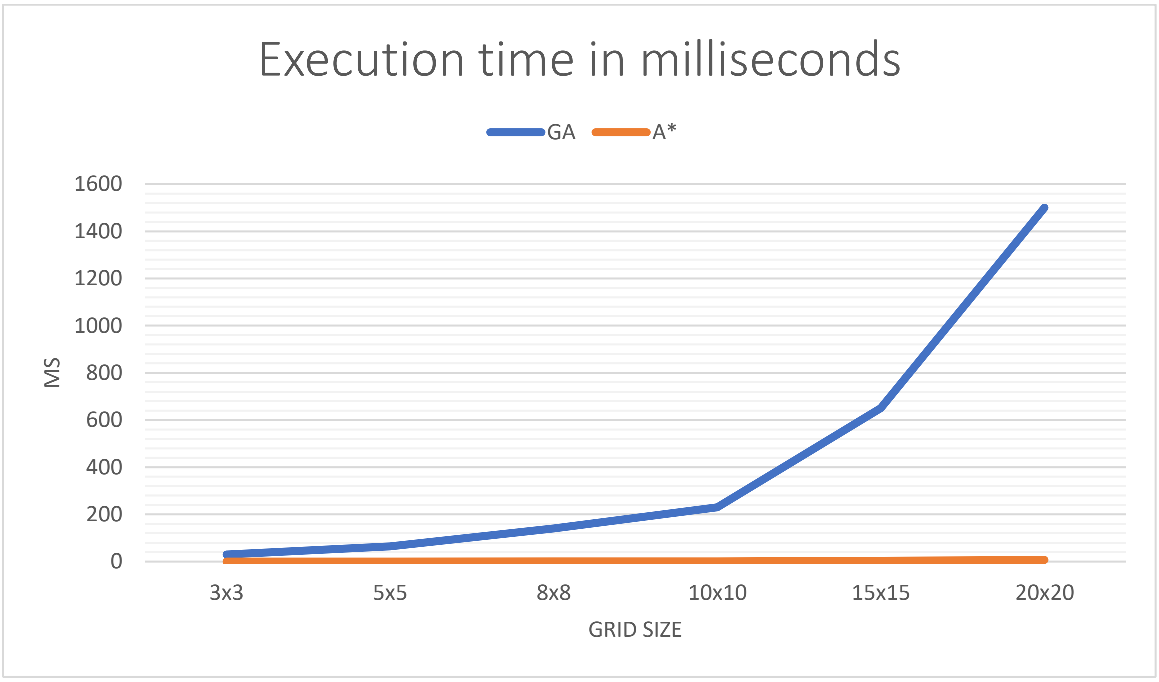
Comparing A* and GA – analysis

Due to the methods used, GA should be able to find routes similar in length to A*. The biggest drawback is waiting for GA to optimize the chromosome length after it has found the shortest chromosome length (as in – shortest route), as it might try to search for a route that would take way too long for the chromosomes to adjust their fitness before returning to a previous chromosome/route length. However, as my main aim was to try to shortest route possible even if that meant a longer execution time, I am satisfied with the solution.

The program was run on a MSI GX70 laptop with an AMD A10-5750m quad-core processor, which is currently a low-end processor compared to the processors of the newest generation, therefore the execution times are certain to differ than running it on a modern PC, especially since it is a notebook processor as well. However, this allows to see how it runs on slower machines, which is a valuable information.

Initially, I checked how the algorithm ran on an empty map with no obstacles, with the start on the north-west corner, and the end on the south-east, increasing the size each time. Here is a diagram showing the stated execution times, using the clock from C-language standard libraries. I took an average of 10 tries for each timing.

These cases are average for GA, and good ones for A*, as GA has to resize its chromosomes each time in order to find the shortest path, but in compensation the path should be easy to obtain as there are no obstacles on the way for the GA, allowing it freely adjust its fitness values and moves in each chromosome.



As it is clearly seen, the execution time for GA grows in an exponential manner as the size grows, while A* barely moves from the 0-1 milliseconds value, getting to 8 milliseconds at most. This shows that A* is the clear winner in terms of execution time, and that GA is currently simply impractical for use in pathfinding.

Why is that happening? It seems that it is due to the amount of operations that GA has to maintain during each generation pass – selection for crossover, crossover itself, as well as mutation. There is a lot of n computational time loops, with a few n^2 computational time loops in code as well in cases where chromosomes need to be set move after move. It is important to remember that the exponential growth is due to the fact that with each growth of the size the amount of operations multiplies.

I tried to compare GA to A* for an extreme, or worst, case like this, where A* had no problems solving it, as shown below:

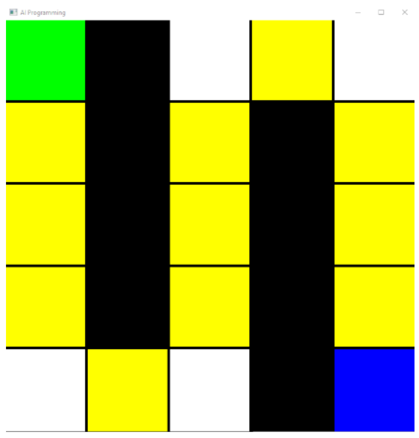


Figure 11 A dealing with extreme case.*

But unfortunately I found out that GA managed to break down on the most difficult mazes, most of the times breaking somewhere in the maze, which is considerably odd as I programmed it to only either find the end or to stop searching if it cannot find the end after a certain generation limit.

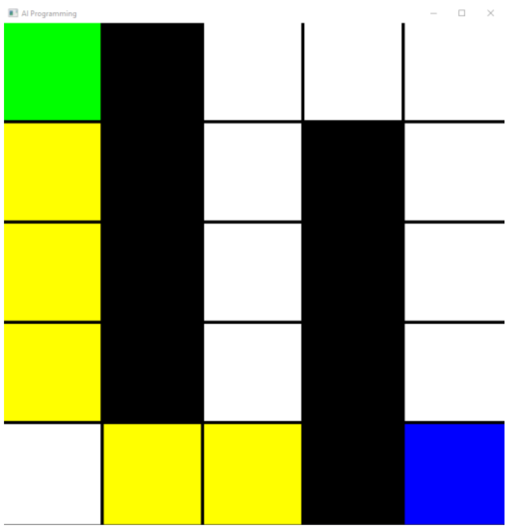


Figure 12 GA breaking down at extreme case.

Interestingly, most of the times changing one of the obstacles from one of the walls worked wonders for GA, allowing it to pass the maze without any problems. It seems that GA's efficiency drops massively when it is forced by the maze layout to follow long, constant movements with sharp turns, instead of being able to initially find a seemingly random path and then improving it with the crossover mechanism.

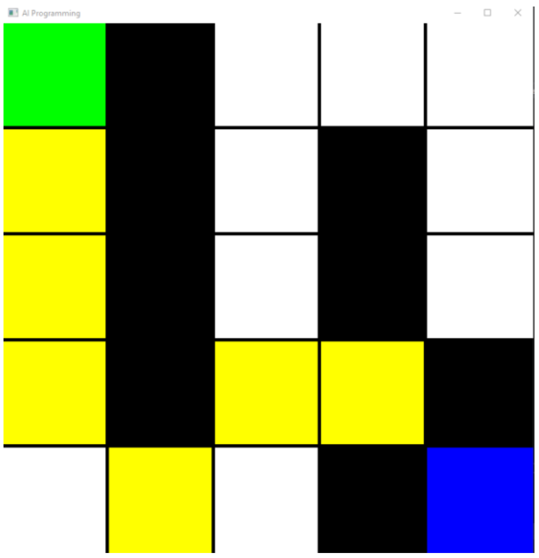


Figure 13 Finding a reason for the GA breakdown.

To ensure myself on this, I tried out a 7x7 grid with just a few obstacles to see if GA would break on it.

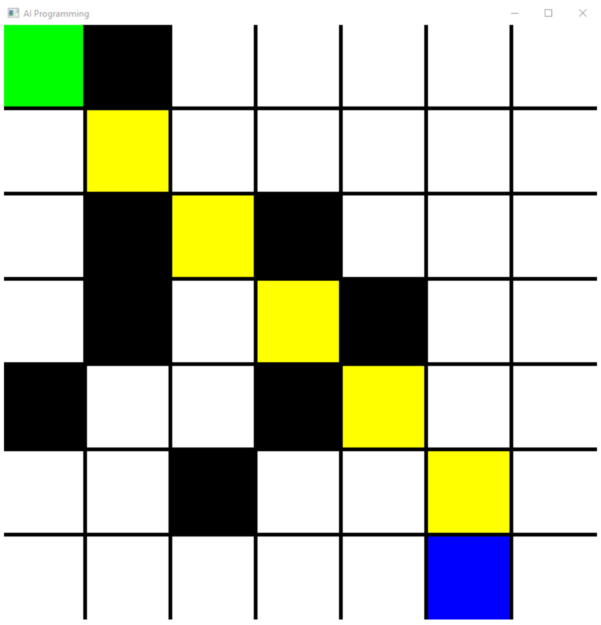
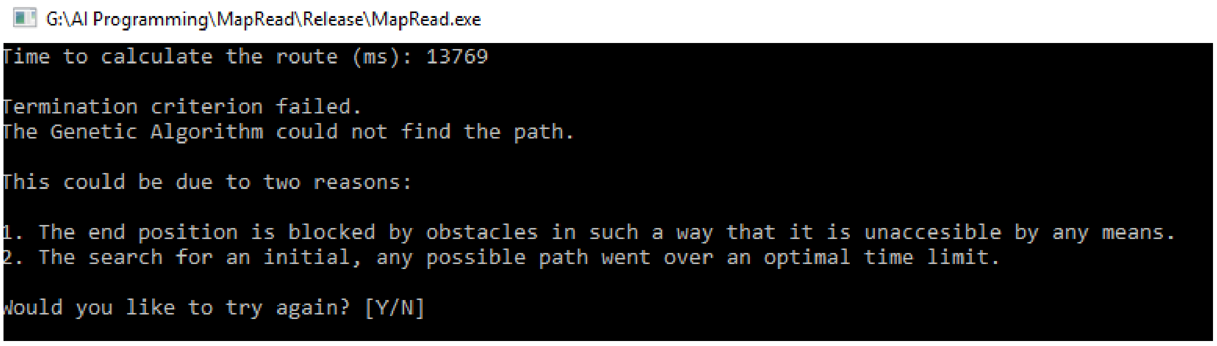


Figure 14 Learning the behaviour of GA.

And it seems that my point of view is correct.

What if a path simply does not exist? I implemented a 3x3 grid with the exit being covered with obstacles to check it. Naturally, the A* immediately found that there is no path due to all nodes being closed, but what about GA? I implemented a generation check so that when the program is searching for a first path, it will eventually stop if it does not find any path at all.



```
Time to calculate the route (ms): 13769

Termination criterion failed.
The Genetic Algorithm could not find the path.

This could be due to two reasons:

1. The end position is blocked by obstacles in such a way that it is unaccesible by any means.
2. The search for an initial, any possible path went over an optimal time limit.

Would you like to try again? [Y/N]
```

As we can see, a large drawback is that the generation count must be high in case that there is a very difficult maze for GA to go through, therefore lengthening the execution time. I have previously set the program to a similar scheme as the chromosome length optimization function – if there is no fitness improvement found for a few hundred generations, it would eventually stop. This however made it stop when traversing difficult mazes, and had to be reversed.

An interesting occurrence also happens for my GA if the grid size is overly higher than needed, as in the example below. This should be in theory one of the “best cases” for the algorithm, but as I implemented the chromosome length optimization function, it is actually one of the worse ones. If the end is right nearby the start, there is an around 1 in 20 chance that the chromosome will “escape” from the end condition when drawing the path and continue its path, which is against the “find best route” code written, as it should automatically resize the chromosome to the moment when it has hit the end. I believe this could be probably fixed by either stopping going through the chromosome when it hits the end when drawing the path, or checking if the chromosome's fitness is not below 1 before declaring that the end has been found. However, this could be also somehow possible due to GA not finding a better path when having a large chromosome length, which could be another drawback compared to A*.

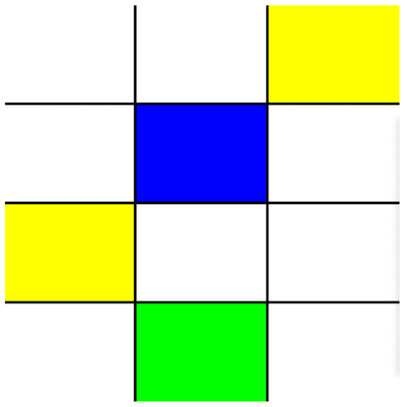


Figure 15 A chromosome going over the end.

What about the accuracy? I found that GA in general has no problems keeping up with A* until the grid size gets over around 8x8, where it start to sometimes get 1-2 moves more on average than A*, but that of course always depends on the map.

Summary

Considering the overall accuracy of GA, it could be used if A* was unavailable for shorter paths. Otherwise however, A* beats GA not only in terms of performance, but accuracy as well. Possibly a path of improvement would be to create an Artificial Neural Network trained by Genetic Algorithms, which would result in gaining advantages of both sides.

References

Figure 1: Graham R., McCabe H., Sheridan S. (2003). *Pathfinding in Computer Games*, p. 70. [online] The ITB Journal: Vol. 4: Iss. 2, Article 6. Available at: https://arrow.dit.ie/cgi/viewcontent.cgi?article=1063&context=itbj [Accessed 13 Dec 2018].

Figure 2: Graham R., McCabe H., Sheridan S. (2003). *Pathfinding in Computer Games*, p. 70. [online] The ITB Journal: Vol. 4: Iss. 2, Article 6. Available at: https://arrow.dit.ie/cgi/viewcontent.cgi?article=1063&context=itbj [Accessed 13 Dec 2018].

Figure 3: Graham R., McCabe H., Sheridan S. (2003). *Pathfinding in Computer Games*, p. 71. [online] The ITB Journal: Vol. 4: Iss. 2, Article 6. Available at: https://arrow.dit.ie/cgi/viewcontent.cgi?article=1063&context=itbj [Accessed 13 Dec 2018].

Figure 4: Tutorialspoint.com (2018). *Genetic Algorithms – Parent Selection.* [online] Available at: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm [Accessed 13 Dec 2018].

Figure 5: Tutorialspoint.com (2018). *Genetic Algorithms – Crossover.* [online] Available at: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm [Accessed 13 Dec 2018].

Figure 6: Patel A. (2011) *The A* Algorithm.* [online] Available at: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html [Accessed 13 Dec 2018].

Figure 7: IEEE Morocco Section. (2016) *Distances in classification.* [online] Available at: http://www.ieee.ma/uaesb/pdf/distances-in-classification.pdf [Accessed 13 Dec 2018].

Figure 8: Patel A. (2011)*The A* Algorithm.* [online] Available at: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html [Accessed 13 Dec 2018].

Figure 10: Kenan E. (c.a. 2011) *Tournament selection vs roulette wheel.* [online] Available at: https://www.cs.bgu.ac.il/~elirank/evo%20web%20crap/q3.htm [Accessed 13 Dec 2018].

Graham R., McCabe H., Sheridan S. (2003). *Pathfinding in Computer Games*, p. 70. [online] The ITB Journal: Vol. 4: Iss. 2, Article 6. Available at: https://arrow.dit.ie/cgi/viewcontent.cgi?article=1063&context=itbj [Accessed 13 Dec 2018].

Graham R., McCabe H., Sheridan S. (2003). *Pathfinding in Computer Games*, p. 72. [online] The ITB Journal: Vol. 4: Iss. 2, Article 6. Available at: https://arrow.dit.ie/cgi/viewcontent.cgi?article=1063&context=itbj [Accessed 13 Dec 2018].

Graham R., McCabe H., Sheridan S. (2003). *Pathfinding in Computer Games*, p. 74. [online] The ITB Journal: Vol. 4: Iss. 2, Article 6. Available at: https://arrow.dit.ie/cgi/viewcontent.cgi?article=1063&context=itbj [Accessed 13 Dec 2018].

Reddy H. (2013). *PATH FINDING – Dijkstra's and A* Algorithm's*, p. 4. [online] Available at: http://cs.indstate.edu/hgopireddy/algor.pdf [Accessed 13 Dec 2018].

Reddy H. (2013). *PATH FINDING – Dijkstra's and A* Algorithm's*, p. 7. [online] Available at: http://cs.indstate.edu/hgopireddy/algor.pdf [Accessed 13 Dec 2018].

Kenan E. (c.a. 2011) *Tournament selection vs roulette wheel.* [online] Available at: https://www.cs.bgu.ac.il/~elirank/evo%20web%20crap/q3.htm [Accessed 13 Dec 2018].