Computing Efficiency of Genetic Algorithms over A\* in Pathfinding

Abraham Passmore, Hunter Boudreau

Computer Science Department, Quinsigamond Community College

GitHub: https://github.com/Abe638/CSC-212-Project

apassmore@qmail.qcc.edu, hboudreau@qmail.qcc.edu

**ABSTRACT**

Pathfinding algorithms are some of the most widely used algorithms in both AI and game design. The industry standard for pathfinding is the A\* search algorithm. The main drawback with the A\* search Algorithm is that it consumes a large amount of computing power. This is compounded by the fact that the algorithm is often executed multiple times each second to account for changes in the environment. Genetic Algorithms, such as the Ant Colony Optimization (ACO) algorithm, make use of computing power far more efficiently. They are able to solve complex paths in less time by utilizing generational learning. We will use the Unity engine coded in C# to demonstrate visually the algorithms in practice and to prove by use of metrics the increased efficiency of Genetic Algorithms.

**INTRODUCTION**

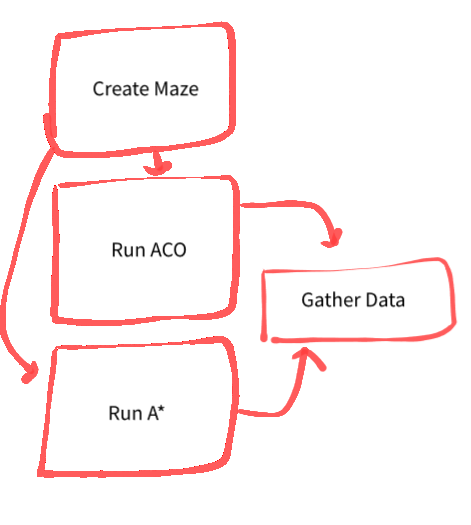
Pathfinding is a way to plot the path between two points. It is useful in finding a route between two points on a map, and is also useful in game design. The industry standard for pathfinding is the A\* algorithm. The A\* algorithm finds the shortest path between two points. The drawback of A\* is that it consumes a significant amount of cpu time. It is not an efficient algorithm. Genetic Algorithms on the other hand can be more efficient but less accurate. Our aim was to show a genetic algorithm was more efficient with a minimal error. For our purposes we focused on the Ant Colony Optimization algorithm ( ACO ). ACO finds relatively short paths but does so using a minimal amount of computing power. It works by creating a random gene or mutator that modifies the pathfinding function. Then the most successful gene from a generation is used for the next generation. Inherent randomness in the Genetic Algorithm means that the path will be suboptimal, although as long as the path isn’t too far off from the shortest path then it isn’t too concerning and might be in many cases the preferred algorithm.

**RELATED WORKS**

There are several papers that are related to our project. Foead, D., Ghifari, A., Kusuma, M. B., et al. (2021). *A Systematic Literature Review of A\* Pathfinding* covers A\* pathfinding and how it works. Other papers endeavor to compare A\* and genetic algorithms. This includes Noori, A., & Moradi, F. (2015). *Simulation and Comparison of Efficiency in Pathfinding Algorithms in Games* and Leigh, R., Louis, S. J., & Miles, C. (2007). *Using a Genetic Algorithm to Explore A\*-like Pathfinding Algorithms.* Finally, there are several papers which only explore the capabilities of genetic algorithms and/or more specifically the Ant Colony Optimization Algorithm. The papers that we have referenced which cover this are Alex Fernandes da V. Machado, Ulysses O. Santos, Higo Vale, et al. (2011) *Real Time Pathfinding with Genetic Algorithm*, Dorigo, M, Birattari, M., Stützle, T. (2006). *Ant Colony Optimization,* and Kaur, E. S. (2013). *Shortest Path Finding Algorithm Using Ant Colony Optimization*.

**PLANNING**

Our design is to have a main loop which we can run at the click of a button. The basic function of this loop is: it builds a maze, runs ACO and A\* on the maze, and saves the results to a .txt file, which we can then convert to a .csv file.

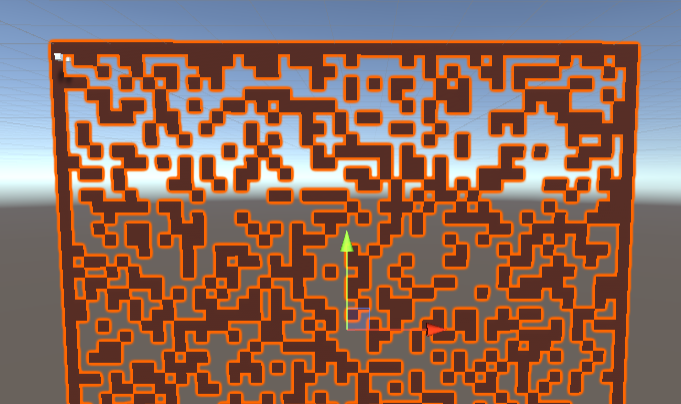


**IMPLEMENTATION**

All our codes is in C# and was run in Unity.

**MAZES**

Mazes were created by allocating a 2D array representing tiles and then randomly deciding which tiles are walkable and which aren’t. A validity checker was then run to ensure the maze was completable. This was done using a pseudo-A\* type algorithm. If the maze was invalid it would continue this cycle until a valid maze was created.



Here is a picture of a maze.



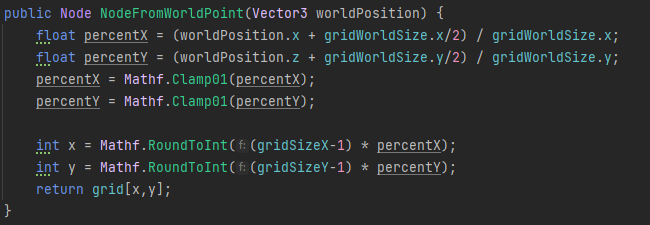
This is the beginning of our maze class. As you can see there are only three arguments: tile length, maze length in tiles, and cutoff. *Tile length* is the length and also the height of tiles, as they are square. *Maze length in tiles* is the number of tiles across and down a maze, as the mazes are also square. *Cutoff* is the percentage of non-walkable tiles that will be spawned ( between 0 and 1 ). If that number gets too close to 1 it becomes impossible to create a valid maze. The function createMaze attempts to create a maze. It returns whether the maze is valid or not. If it’s not valid it will be re-run up to 1000 times.



This is part of the code that tests if a maze is viable, which is to say that it is completable from the top left corner to the bottom right corner. It basically runs the A\* algorithm but doesn’t keep track of the exact path.

**GRIDS**

The ACO and A\* algorithms were kept separate from the internal workings of the maze code in order to ensure modular independence. Each algorithm created a grid of its own to determine walkable and unwalkable tiles.



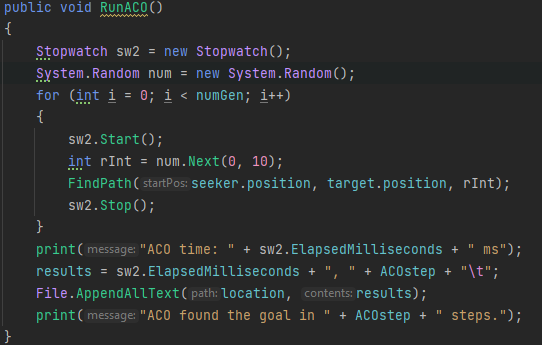
The most important function of the grid is the NodeFromWorldPoint() function. This helps to convert from unity’s ingame objects, such as the maze, into nodes on the grid. This can be used to get the grid coordinates of any node that is intercepted by an object of a specific layer, and therefore can be used to determine what nodes are unwalkable because they are intersected by an object in the unwalkable layer.

**A\* ALGORITHM**

This is the main loop of A\*, a function called FindPath(). This works by storing every node that is yet to be checked in the openSet List while all of the nodes confirmed to be in the path are stored in a Hash set called closedSet. The first for loop determines what the current node is, then the current node gets added to closedSet and removed from openSet. After the walkable neighbors are determined the function checks each neighbors distance from the goal and adds them to openSet. Finally if the new neighbor has a lower h cost than the current node, then the variables get updated and the neighbor with the lowest h cost becomes node.

 A modified version of this function is then used in the Genetic Algorithm’s main pathfinding loop.

**GENETIC ALGORITHM**



The main difference between the two is the random nature of the genetic algorithm. FindPath() now has a parameter for a mutation that starts as a random int between 0 and 10. This initial mutation will be used to help alter the path taken. The fitness of the path is then evaluated by the Ant Colony Optimization Algorithm, this essentially just retraces each path, counting the number of steps taken. Whichever mutation was deemed the “fittest” will then affect how the future mutations are generated by dictating the new range of integers that will be used to determine the mutations.

**FUTURE WORK**

While we have no plan on continuing work on this comparison, there is much room for improvement. A\* is a tried and tested algorithm which can be easily replicated at peak efficiency, on the other hand Genetic algorithms still remain new and relatively untested. This means that as Genetic algorithms are tested and developed in the future they will become more and more preferable to A\*. Our Genetic algorithm was also very simple, we were limited on time so our mutation is a simple constant applied as a modifier to certain equations. This could be further improved and even converted into a more modern “AI” that learns over time and doesn’t reset its “knowledge” after every test.

**CONCLUSION**

This project was very illuminating as far teamwork was concerned. Abraham and I tackled separate areas of code. I focused mainly on the mazes, and Abraham focused mainly on the two algorithms. When it came to putting two pieces together there was a bit of confusion. Abraham and I had to communicate back and forth a significant amount in order to combine each of our sections. Our strategy was very ad hoc. We just started working on what the other wasn’t doing. Ultimately, we succeeded. If we were to start this project again I think we would draw out a clearer road map of what needs to get done, and who is doing what, as well as what features need to connect between our code.

**REFERENCES**

Foead, D., Ghifari, A., Kusuma, M. B., et al. (2021). A Systematic Literature Review of A\* Pathfinding. Procedia Computer Science, 179(07-514), www.sciencedirect.com. https://reader.elsevier.com/reader/sd/pii/S1877050921000399?token=A18B181510BA76DC648DD40324EB2450C18CA9FDFF37B37C8547FE9F0874BBDDF15B13B9C39A5C3B7169F87C3EFE2603&originRegion=us-east-1&originCreation=20220219152528

Noori, A., & Moradi, F. (2015). Simulation and Comparison of Efficiency in Pathfinding Algorithms in Games. Ciencia e Natura, 37(230-238), semanticscholar.org. https://pdfs.semanticscholar.org/f4ae/638f2297300ad377cf7e4bb1d48d070bd44e.pdf

Leigh, R., Louis, S. J., & Miles, C. (2007). Using a Genetic Algorithm to Explore A\*-like Pathfinding Algorithms. IEEE Symposium on Computational Intelligence and Games, www.cse.unr.edu. https://www.cse.unr.edu/~sushil/class/gas/papers/Using%20a%20Genetic%20Algorithm%20to%20Explore%20A\_-like%20Pathfinding%20Algorithms.pdf

Alex Fernandes da V. Machado, Ulysses O. Santos, Higo Vale, et al. (2011) Real Time Pathfinding with Genetic Algorithm. 2011 Brazilian Symposium on Games and Digital Entertainment, ieeexplore.ieee.org, https://ieeexplore.ieee.org/abstract/document/6363236

Dorigo, M, Birattari, M., Stützle, T. (2006). Ant Colony Optimization. IEEE Computational Intelligence Magazine, 1(28-39), ieeexplore.ieee.org, https://ieeexplore.ieee.org/document/4129846

Kaur, E. S. (2013). Shortest Path Finding Algorithm Using Ant Colony Optimization. International Journal of Engineering Research & Technology, 2, www.ijert.org. https://www.ijert.org/research/shortest-path-finding-algorithm-using-ant-colony-optimization-IJERTV2IS60197.pdf