Evolutionary Computation with a Maze

1st Jonathan Na
Information and Computer Sciences Department
University of Hawaii at Manoa
Honolulu, Honolulu
jna6@hawaii.edu

***Abstract*— This paper is a review on the research of Evolutionary Computation with a maze project. The maze program is built with the Unity Engine and the genetic algorithm in C#. The goal of the project is for the Artificial Intelligence to traverse through the maze from the spawn point to the exit without any human input or solution comparison. The paper further goes into details on the results from the trials, including comparison between recombination and mutation and different fitness evaluation sizes, which will be covered in the paper.**

Keywords—Artificial Intelligence, maze, Evolutionary Computation, recombination, mutation, fitness, Unity, Genetic Algorithm

I. Introduction

A. Maze

The oldest known maze is the Labyrinth of Egypt, built by the Egyptians nearly 2 thousand years ago and was first documented by a Greek writer named Herodotus [1]. Mazes are an interesting way to learn evolutionary computing as there are many research papers done on the performance and solution to solving mazes. One such paper "Maze Benchmark for Testing Evolutionary Algorithms", tested a method on maze solving with evolutionary algorithms by using the Connectivity Based Maze Generation Algorithm [2]. The maze created in this project is done on Unity as shown in Fig. 1 below, the yellow circle indicates the starting point and the red line in the bottom right corner indicates the exit. The black dots in Fig.1 indicates the Artificial Intelligence (AI) that traverses through the maze to find the exit:

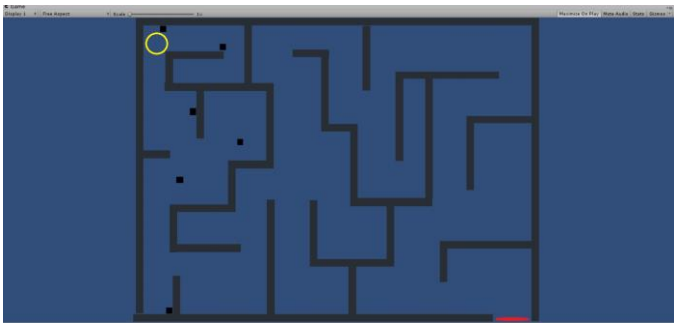

Fig. 1 (Maze) traced by image on Wikipedia [3]

B. Purpose

The purpose of the project is to conduct a basic review on Evolutionary Computation with a maze project. The paper will go over the method of genetic algorithm that was used in the project. As well as any data gathered from the trials. Comparison between Recombination (Crossover) & Mutation (abbreviated to RM) algorithm vs. Mutation Only (abbreviated to MO) algorithm is done and evaluation between two different types of fitness search space.

C. Goals

The primary goal of the project is for the AI to traverse through the maze from the spawn point to the exit without any human input or solution comparison. To further elaborate, an earlier version of the project needed a human solution of the optimal path, inputted into the AI. The AI would use this optimal path given by the human as the search space and fitness evaluation, if the AI was on the correct path in the maze. The current version of the project can now traverse through the maze without any needed human input or solution comparison.

D. Program Langauge and Software

The Software used to create the maze project is a video game engine called Unity. Unity is free and has a lot of online documentation and support. The main program language that the genetic algorithm was built is in C#.

II. Genetic Algorithm

A. Search Space, Fitness, and Termination

The maze is filled with invisible tiles called food as shown in Fig. 2 below, when an AI touches a food then it increases its fitness level. However, if the AI touches the same food again then it decreases its fitness level, this is all done to reduce backtracking. Backtracking is when the AI decides to move in the opposite direction, away from the exit and regresses. Backtracking wastes time and reduces the chance of finding an optimal path. In this project the program starts with a population of 100 AI's which will act independently from each other. If an AI touches any of the walls in the maze, then it dies and if all the AI's are dead then the AI will proceed to the next generation. The termination requirements are:

- If an AI reaches the exit of the maze.
- The generation reaches one thousand.

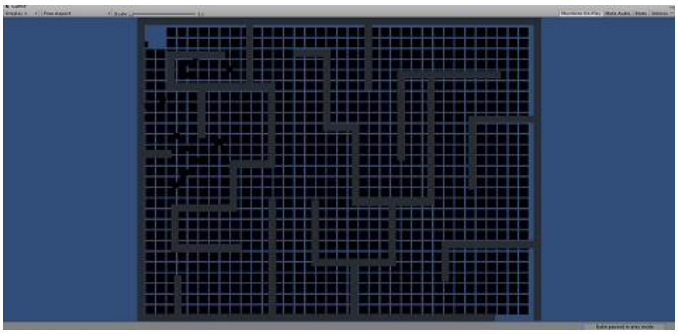

Fig. 2 (Maze with food (tiles))

B. AI Movement (Path Array)

Each AI starts with a path array size of 400 and prefilled with one random number ranging from 1 to 4. The movement of the AI is shown in Fig. 3 below.

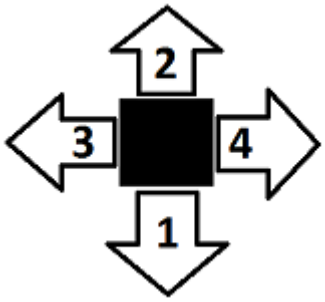

Fig. 3 (AI with directional pointers represented by its numbers)

After each generation the AI's array are either swapped with another AI or mutated that will change the numbers in the array. If the AI goes over the path array 400 size limit, then the AI will die since the AI couldn't reach the exit (the average array size needed to complete the maze is approximately 200-300). The idea of the AI dot is influenced by the paper: "Evolving Sparse Direction Maps for Maze Pathfinding", where the "Agent" traverses through the maze like the AI dot. However, the agent can move in sixteen different directions but relies on the directions stored inside the map instead of an array [4].

C. Selection

The AI uses a Tournament Selection, a type of selection strategy that doesn't require any knowledge of the population but does need a fitness ranking system. The tournament selection chooses a tournament size of *k* and compare those AI's in *k*, the winners then become the parents for the next generation [5]. Another reason why the tournament selection is used in this project is because there is a chance for the AIs to get a negative fitness level due to the nature of Unity updating every frame, meaning that an AI that back tracks can hit the same food multiple times leading up to a negative fitness level. The project randomly chooses two AI's from the population and compares them to see who the winner will be. The winner will then be removed from the tournament pool in order to give the weaker AI's a chance in the selection. Only a possible of fifty winners will be selected for the parents for the next generation.

D. Recombination (Crossover)

Recombination or aka crossover is a binary variation operator when two parents' mate together to create one or two offspring. This may or may not improve the offspring features due to recombination being a stochastic operator [5]. In this project, recombination will occur 100% of the time, and a one-point crossover is used between two parent AI's that won in the tournament selection. A point is randomly selected in the path array, then the array is split into two sections, *a* being the first portion of the array, {R} the location of the split, and *b* being the second portion of the array. The first parent *b* will swap with the second parent *b*, as shown below in Fig. 5.

a {R} b

Before Crossover:

Parent1 [1][1][1][2][2][2][2][2][3][3] {R} [1][1][4][4][4]

Parent2 [1][1][1][2][2][2][3][3][3][1] {R} [1][4][4][4][2]

After Crossover:

Parent1 [1][1][1][2][2][2][2][2][3][3] {R} [1][4][4][4][2]

Parent2 [1][1][1][2][2][2][3][3][3][1] {R} [1][1][4][4][4]

Fig. 4 (Example: one-point crossover array)

E. Mutation

Mutation is unary variation operator it mutates the offspring into an improved version of the parent. However, this may not always be the case since mutation is also a stochastic operator [5]. In this project mutation will occur for both the offspring's and parents by using a modified version of the Flip Mutation. Flip Mutation is when normally used in binary encoding and its process is to flip a 0 to a 1, vice versa [6]. In the project the modified flip mutation instead flips a whole section of an array into one number. The array location area where the AI died is randomly selected (about 1-5 indexes away from the location. With *a* being the first portion of the array, {D} the location of picked near its death, and *b* being the second portion of the array, shown in Fig. 5 below. The portion *b* is then flipped into an integer that isn't:

1) The previous index: This prevents the AI from colliding into the same wall again.

2) The opposite of the previous index: This prevents the AI from backtracking as this will regress the AI and waste time.

There is an 80% chance for the AI to only choose 1-5 indexes from the last death location and a 20% chance to choose 1-10 indexes.

a {D} b

Before Mutation:

Array1 [1][1][1][2][2][3][3][3][2][2] {D} [2][2][4][4][1]

After Mutation:

Array1 [1][1][1][2][2][3][3][3][2][2] {D} [3][3][3][3][3]

Fig. 5 (Example: Flip Mutation array)

III. Data Gathered

In this project two tests were performed, the Recombination & Mutation (again abbreviated as RM) algorithm vs Mutation only (again abbreviated as MO) algorithm and the test between the small food size vs the large food size. A video sample of the project can also be viewed on the Github page in the Sample.zip folder. Also, the project can also be downloaded from the Github page to the Unity Engine, further instructions can be found in the readme file: https://github.com/jjhna/Mini

1) RM vs. MO: Test difference between MO algorithm and RM algorithm (note that this test was done using the 0.7 x 0.7 food size).

2) Different Food sizes: Test between two different food sizes (0.2 x 0.2) vs. (0.7 x 0.7) to observe any differences (note that the test was done using the RM algorithm) .

A. RM vs. MO results

The results shown in Fig. 6, is the results of trial 1 from MO and trial 2 from RM from Table I. Which shows that the MO algorithm performed better than the RM algorithm in two places. Overall the MO algorithm performed better in the end.

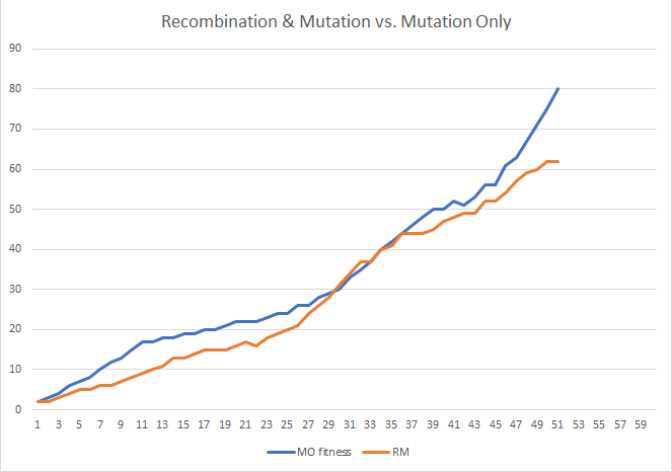

Fig. 6 (line graph of the RM vs. MO algorithm fitness)

TABLE I. RM vs. MO

Trials	Generation after completion	
	***RM*[a]**	***MO*[b]**
1	77	52
2	61	49
3	104	48
4	75	41
5	106	48

[a] RM = Recombination and Mutation.

[b] MO = Mutation Only

B. Food size changes

The results shown in Fig. 8, is the results of trial 5 from both the large and small food size from Table II. Which shows that the smaller food size has gained more fitness due to the increased amount of food tiles to be covered in the maze. However, both tests have completed the maze in the same amount of generations, but the large food size have completed the maze quicker than the small food time numerous times.

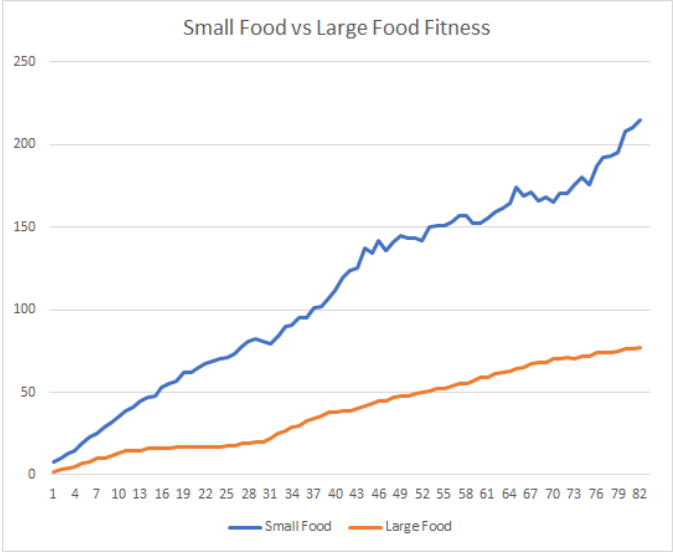

Fig. 7 (line graph of large vs small food fitness)

TABLE II. LARGE FOOD VS. SMALL FOOD

Trials	Generation after completion	
	Large Food Size	***Small Food Size***
1	86	96
2	75	69
3	75	83
4	43	93
5	84	83

IV. Conclusion

After numerous testing and results, overall the best method to traverse the current given maze is a MO algorithm with a selection tournament and using large food sizes.

A. Conclusion

The following conclusions to the two tests are:

1) RM vs. MO: Overall the MO algorithm performed quicker than the RM algorithm. As shown in Table I. the MO algorithm completed the maze quicker than the RM algorithm.

a) RM: The RM algoirthm prefomed poorly than the MO algoirthm. This is due to the possibility of

regression when the AI chooses where to cut the array from or the possibility of paring up with a weaker AI making both AI's weaker.

b) MO: The MO algorithm performed quicker than the RM algorithm. This is due to the simple process of selecting the best parents with the tournament selection and creating a clone of it, doubling the chance of progression. This method appears to be feasible on the current maze but further testing on a harder maze is desired.

2) Different Food sizes: Overall the bigger food sizes performed a little better than the smaller food size in the tests. The bigger food size would end around generation 75, while the smaller food size would end around generation 90. With the smaller food sizes the AI would take more U-turns due to its greed in increasing its fitness rating. Since the food sizes are smaller the AI's can only cover so much ground in a period of time which would explain the U-turn and backtracking.

a) Bigger Food: The AI completed the maze sometimes quicker than the smaller food and have completed the maze multiple times without fail.

b) Smaller Food: The AI would sometimes complete the maze quicker than the bigger food but would also get stuck in some areas leading to a completed generation at 200 or failing to complete the maze itself.

B. Additional Future Work

Additional future works would include changing the current maze system into a more tile or grid-based system. This will allow an easier approach in fitness evaluation and building additional mazes. Additional testing on harder mazes is also highly recommended and perhaps a mini-game on the Unity Engine.

ACKNOWLEDGMENT

I would like to thank Professor Lee Altenberg for providing me guidance to write this paper and project. I also would like to thank my parents for their continuous support of me. Finally, I would also like to thank the Unity Community for the numerous solutions to Unity bugs and issues.

REFERENCES

[1] W.H. Matthews. 1922. Mazes and labyrinths: a general account of their history and developments, New York: Longmans, Green and Co., pp.6-7

[2] Camilo Alaguna and Jonatan Gomez. 2018. Maze benchmark for testing evolutionary algorithms. Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO 18 (July 2018), 1321–1328. DOI:http://dx.doi.org/10.1145/3205651.3208285.

[3] Jkwchui. 2011. File:Maze simple.svg. (April 2011). Retrieved December6,2018from https://commons.wikimedia.org/wiki/File:Maze_simple.svg.

[4] V.Scott Gordon and Zach Matley. 2004. Evolving sparse direction maps for maze pathfinding. Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753) (June 2004). DOI:http://dx.doi.org/10.1109/cec.2004.1330947

[5] Eiben, A. and Smith, J. (2015). Introduction to Evolutionary Computing, Second Edition. 2nd ed. [Place of publication not identified]: Springer., pp. 84-86 and pp. 32-33 and 31-32

[6] Soni, N. and Kumar, T. (2014). Study of Various Mutation Operators in Genetic Algorithms. International Journal of Computer Science and Information Technologies, [online] 5(3), pp.4519-4521. Available at: https://ijcsit.com/docs/Volume%205/vol5issue03/ijcsit20140503404.pdf [Accessed 4 Dec. 2018].