

# A\* Graph Optimization Using Genetic Algorithms

Stuart Bowman

May 6, 2016

## 1 Introduction

A\* is known to be the undisputed champion of path finding for start to finish traversals of graphs and 2D planar environments. A\* pathfinding does however require a set of waypoints to analyze viable paths and traverse terrain. While it isn't unreasonable to manually create a set of waypoints for most applications or use cases, it would be ideal to automate the process of generating and connecting the waypoints needed to run A\* on a given environment. Genetic algorithms may provide a vehicle for creating, modifying and evolving data in order to provide the optimized graphs needed to run A\*.

Genetic algorithms, as defined by John McCall, are a heuristic search and optimization technique inspired by natural evolution. [2] In their broadest sense, they represent a small branch of artificial intelligence that exists to construct solutions for problems that require optimizing many different attributes or parameters simultaneously. They are particularly useful under contexts where solutions can be optimized through repeated generational selection and breeding of entities containing desirable traits. [1]

Genetic algorithms provide an excellent platform for rapidly building, evolving and mutating data to follow a series of metrics through heuristical analysis. The full process can be broken down into several steps: [1]

- Initialization of the base population to start the first generation
- Evaluation of each data members fitness relative to the goal
- Selection of the most fit individuals in the current generation to be used to breed the next generation. This selection should usually strive to direct the population towards a generation of individuals with preferred traits.
- Crossover represents the actual breeding phase. This typically involves borrowing a mix of traits from each of the parents to create a new child

entity or entities to be evaluated in the next generation.

- Mutations are used to introduce small random or nondeterministic changes in a child's traits that does not reflect the parents attributes. This can be used to prevent populations from becoming overly homogeneous.
- Repeat the process, starting with the evaluation of the most recent generation.

[1]

Additionally, genetic algorithms offer the flexibility and functionality of being able to find an optimal solution in a reasonable amount of time. This means that rather than waiting unreasonable amounts of time for an optimal solution, a genetic algorithm can be stopped at any iteration when the desired attributes have met or exceeded a given threshold or local optimum. [1]

With the base concept and definition of genetic algorithms in mind, the next step is to decide how genetic algorithms will play into the iteration and optimization of waypoint selection and generation. One of the primary concerns with waypoint generation for use in A\* is to reduce the search space as much as possible without losing valuable information. Having a smaller set of valuable waypoints will allow A\* to run quickly and produce desirable paths.

## 2 Genetic Algorithm Overview

Our A\* waypoint system began with several guidelines and constraints around which to model a genetic algorithm. These guidelines would dictate the type of graph we wished to achieve from running our genetic algorithm. Having understood A\* search itself, an ideal waypoint graph should only contain the **minimum number of nodes** required to quickly and efficiently traverse the maze through the search. Secondly, the graph should aim to achieve **full connectivity** of the maze's walkable space, meaning that

any random point placed in the maze should be walkable to any other point in the maze by running A\* search using the generated graph.

With the attributes of an ideal A\* waypoint graph in mind, we next considered the construction of a genetic algorithm to handle the evolution sequence for generating such a graph.

## 2.1 Initial Population

The initial starting population contained a set of  $x$  graphs, each of which would be generated using  $y$  randomly placed nodes in the walkable space of the graph. Each node in this set would be connected to every other node where line of sight visibility was present. Line of sight, for the purpose of this algorithm, was handled by Unity Engine’s raycasting system. This means that for any given nodes  $a$  and  $b$  in the graph, such that  $a \neq b$ , if a ray could be cast from  $a$  to  $b$  without intersecting with any wall of the maze, then  $a$  and  $b$  should be connected. The resulted in a fully connected graph along all walkable edges in the graph.

## 2.2 Evaluation

After the initial generation has been built, the genetic algorithm must then handle evaluating each graph, evaluating fitness for use later in the selection and crossover stages. Three immediate factors were used to score graphs in this implementation. The first, and most obvious factor, was the **graph’s node count**. Lower node counts result in A\* running faster and requiring fewer traversals of adjacent nodes. The second characteristic scored was the graph’s overall **A\* satisfaction rating**. This rating was computed by randomly placing 100 starting and ending points in the graph’s walkable space. An A\* search was then attempted on each of these pairs using the given waypoint graph, with the resulting satisfaction rating representing the percentage of successful A\* paths that could be traced from the pool of 100 pairs of points. It is important to note that a fresh set of 100 random starting/ending points were cycled with each new generation, preventing the graph from becoming “overly fit” for just one set of A\* maze traversals. The third attribute measured on each graph was the **average length of successful A\* path traces** from the A\* satisfaction rating evaluation. This value indicates how successful the given graph is at effectively traversing the maze. Lower A\* path lengths mean that A\* is able to traverse the graph much more efficiently, taking more direct paths from start to finish. To simplify the crossover and tuning stages, the over-

all heuristic score was strictly limited to the A\* satisfaction rating, however further tuning could be made by setting the heuristic score to a weighted combination of both A\* satisfaction and average path length.

## 2.3 Selection and Crossover

With the population evaluated and sorted by fitness, the algorithm can then go through the list of graphs and begin breeding. The breeding system involves systematically running through the population and “mating” the most fit graphs first. The process continues until enough offspring have been generated to replace the previous population. If every graph in the current population has had a chance to breed but there are not enough offspring to replace their elders, the process will start again with the two most fit graphs.

The breeding process itself between two graphs is as follows. Parent graph  $a$  and  $b$  are both checked for their A\* satisfaction rating. Let  $k$  represent the percentage of crossover nodes to be removed from the breeding process. By cycling out a certain percentage of nodes during each breeding phase, inbreeding is prevented and additional diversity of graphs are reintroduced at each breeding stage. Let  $p$  represent the higher of these two A\* satisfaction values. Let  $N$  represent the total number of nodes in parent graph  $a$ . The new child graph  $c$  is then created using a randomly selected node  $n1$  from graph  $a$  and a randomly selected node  $n2$  from graph  $b$  (provided that  $n1$  and  $n2$  have not previously been selected from  $a$  and  $b$  respectively). Nodes are randomly selected from  $a$  and  $b$  until graph  $c$  contains  $(N * (1 - k))/2$ . Finally, the new graph is given a boost of new nodes, up to  $(1 - p) * N * \text{normallydistributedrandomfloats}$ . This step also allows the graphs to grow (or shrink) from one generation to the next and introduces graph diversity.

## 3 Implementation and Tuning