A* Waypoint Optimization Using Genetic Algorithms

Stuart Bowman and Prof. Brian Dellinger

May 9, 2016

1 Introduction

A* is the undisputed champion of path finding for start-to-finish traversals of graphs and 2D planar environments. A* pathfinding, however, does 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 the 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]

1. Initialization of the base population to start the first generation

2. Evaluation of each data member's fitness relative to the goal

3. 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.)

4. The Crossover stage is known as the breeding step. 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.

5. Mutations are used to introduce small, random or nondeterministic changes in a child's traits that do not reflect the parents attributes. This can be used to prevent populations from becoming overly homogeneous.

6. Repeat the process, starting at step 2, and evaluate 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 for a perfectly 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 quicker and produce desirable paths.

2 Algorithm Overview

Our A* waypoint system began with several guidelines and constraints around which a genetic algorithm was modeled. These guidelines then dictated the type of graph we wished to achieve from running our genetic algorithm. Having understood A* search itself, an ideal waypoint graph should, as a result, 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. This resulted in a fully connected graph along all walkable edges in the graph.

2.2 Evaluation

After the initial generation had been built, the genetic algorithm must then handle evaluating each graph, scoring 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, since it is taking much more direct paths from start to finish. To simplify the crossover and tuning stages, the overall 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 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 between the two graphs is as follows: Parent graph a and b are both checked for their A* satisfaction rating. Let p represent the higher of these two A* satisfaction values. 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 new nodes are introduced at each breeding stage. 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 $(N * (1 - k))/2)$ nodes. Finally, the new graph is given an additional set of new nodes, adding $((1 - p) * N * V)$ nodes to c where V represents a normally distributed random float value. This step also allows the graphs to grow (or shrink) from one generation to the next and introduces graph diversity.

3 Implementation and Tuning

Following the genetic algorithm design detailed above, the implementation was constructed and tested in Unity Game Engine and written using `C#` .NET. The general structure of the system is as follows:

Genetic.cs The main driver class for our genetic algorithm. It primarily handles user interface callbacks, creates initial populations to feed to

the Generation class, and handles building game-objects on the fly to display the generated waypoint graphs.

Generation.cs Operates as a container for all of the graphs used in a given generation. It handles generation construction, evaluation, selection, breeding, and mutation, as well as tabulating and recording results to an output file for easy access once testing is complete.

Graph.cs Contains the graph nodes, as well as helper functions for use by the generation class to better facilitate evaluation and breeding. It also contains the class definition for the nodes, as well as the A* search implementation used by the genetic algorithm to check if a path can be traced between two given points on the maze.

BuildMaze.cs Borrowed from last semester's gngindiestudy, this class handles the process of parsing and reconstructing the maze in Unity. This allowed the Genetic class to pass maze geometry information along to the Generation and Graph classes for use in node creation.

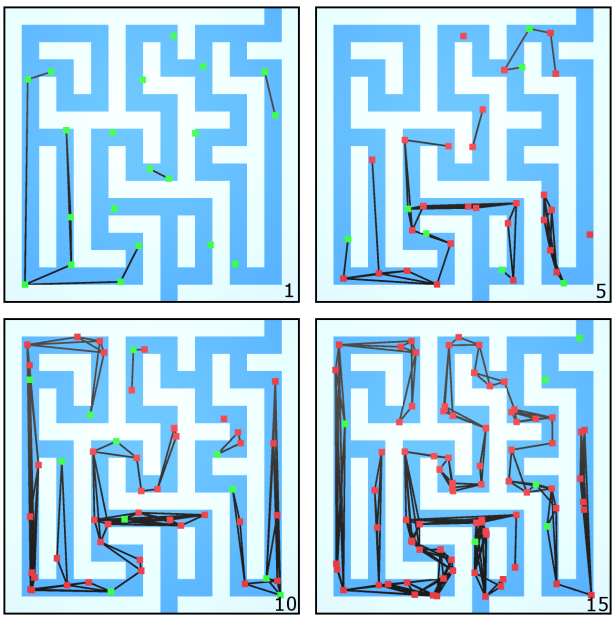

Figure 1: The most fit waypoint graphs from the third trial of our genetic algorithm. Sampled at generations 1, 5, 10, and 15 respectively. Red nodes are inherited from previous generations, while freshly created nodes are labeled in green.

3.1 Constraints

In addition to the algorithm structure and class structure detailed above, there were several constant values used to tune and regulate various components of the generational evolution.

Genetic.GraphInitialNodes Indicated the number of nodes the initial generation of graphs should contain. Graph size will fluctuate after the first generation, usually increasing and then beginning to level off after a certain point.

Generation.numEntitiesPerGeneration Dictates the number of graphs that will be created during the crossover stage. The generation population size is constant, meaning each generation will contain `numEntitiesPerGeneration` number of graphs.

Generation.numAStarPathChecks Indicates the number of random pairs of points that will be placed and checked each generation during the eval stage. The resulting percentage of point pairs that can successfully be mapped using A* will be stored under `Graph.AStarPathSuccess`

Generation.percentToReplace Percentage of population to remove during the crossover and mutation. The removal of nodes at this stage allows for the introduction of new nodes which keep the population from becoming overly homogeneous. Percentage value is represented as a float.

3.2 Algorithm Refinement

Over the course of our development we experimented with several maze configurations, constraint tuning, and crossover adjustments. Early on in development, a larger graph was used to demonstrate the basic functionality of the various genetic algorithm components. However, once the full algorithm was implemented, we chose to instead opt for a much smaller graph, which significantly improved runtime usability and allowed for a much easier comparison between graphs within the same population as well as comparisons from generation to generation. The smaller graph also allowed the initial population to begin with a fitness of 10-20% A* satisfaction, up from 1% with the large maze. This helped reduce the number of nodes required to start a generation, thus allowing the algorithm to run a little quicker.

3.2.1 Crossover Refinements

Many of the improvements that were made were also related to the crossover stage of the algorithm.

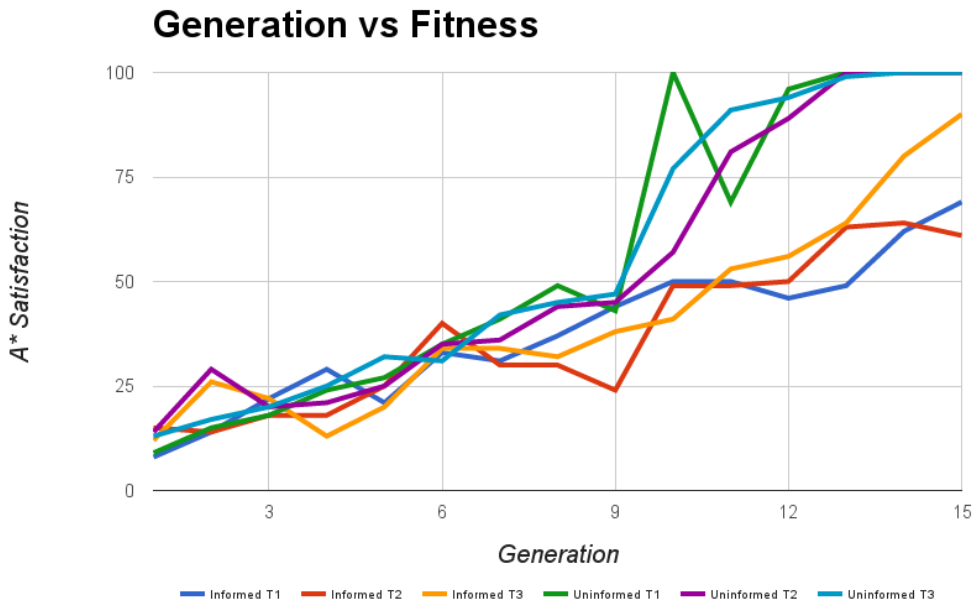

Figure 2: Red, Blue, and Orange show the trials using our genetic algorithm, while Green, Purple, and Cyan show trials using uninformed node placement.

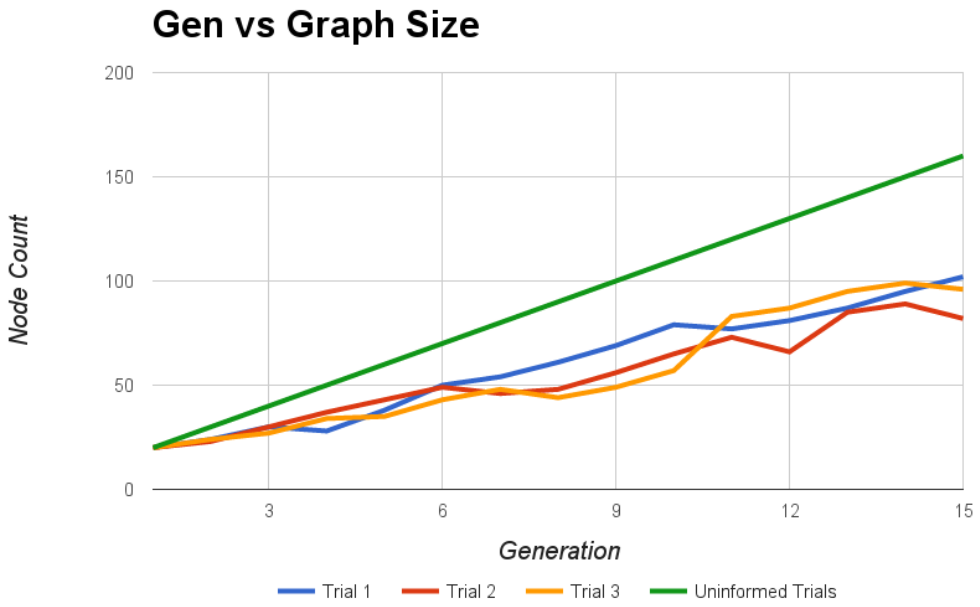

Figure 3: Green shows graph size growth for uninformed trials, while Red, Green, and Orange show graph size for trials using our genetic algorithm. The uninformed system uses significantly more nodes, which is undesirable for A*.

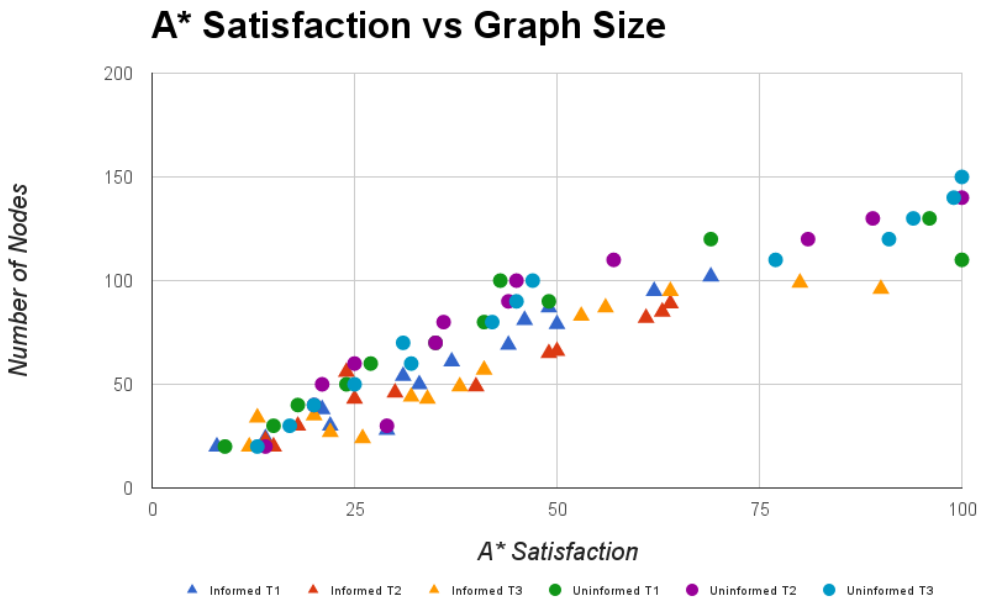

Figure 4: The Genetic algorithm (Triangles) consistently demonstrates that it requires far fewer nodes than the uninformed system to achieve approximately the same A* Satisfaction rating. (Circles)

Specifically, there were several major changes that were made to improve generational A* satisfaction progress, as well as reduce the number of nodes required to reach higher satisfaction values.

Initially, crossover was implemented simply as a breeding stage, merging half of the nodes from each parent into a child graph. As the algorithm began to take shape, however, it was quickly discovered that overly homogeneous populations were causing the graphs to decline in fitness from generation to generation. To rectify this issue, we modified the crossover stage to instead cycle out approximately 10% of the nodes from the parents and replace them with new nodes.

Additionally, we recognized that the graph size would need to fluctuate in order to properly cover the maze's walkable area and improve fitness. Initially we had the graphs grow at a steady rate at each generation. However, after testing this system, we found that high A* satisfaction was likely being reached only by over-saturating the maze with nodes. Our alternative approach involved modifying how many nodes would replace the 10% that were originally removed. In order to prevent the graphs from growing indefinitely, we implemented a system to grow the graph depending on its parent's A* satisfaction rating. The higher a parent's A* satisfaction rating, the fewer nodes will be added to its children. This also allows the node count to plateau after a certain number of generations, and in some cases decrease after a certain point.

4 Results

The testing and evaluation of our **informed** genetic algorithm took the form of three trials, each running for a series of 15 consecutive generations at 200 graphs per generation. Initial generations began with 20 nodes per graph, with 10% of the node pool being dropped each generation to make room for new nodes. We also ran three trials without generational inheritance to simulate the process of randomly placing nodes on the graph. This **uninformed** system served as a baseline to compare our algorithm against. Each iteration of the uninformed tests involved generating a fresh population of graphs, each of which contained 10 nodes more than the previous, allowing for a predictable, linear growth in graph size. Additionally, it should be pointed out that our Unity Engine implementation color-coded graph nodes based on inheritance. If a particular node was inherited from an ancestor, the node is colored red; otherwise if the node was freshly created during the previous crossover stage, the node is colored green. This color-coding system allowed us to easily identify which attributes were inherited and which were freshly intro-

duced.

Although the informed trials did not quite reach 100% A* satisfaction by the end of 15 generations, there are several advantages our genetic algorithm had over the uninformed method of simply placing nodes randomly in the maze. Primarily, as demonstrated in Figure 3, our informed genetic algorithm is conscious of the graph size, avoiding over-saturation and preventing the node count from growing out of control. Graph size is critical to a well-balanced A* waypoint graph. If a waypoint graph has too many nodes, then A* will not be able to efficiently find a path through the maze.

Figure 3 demonstrates how the graph size scaled with each consecutive generation during each of the informed and uninformed trials. As the uninformed generations grew at a predictable rate, the informed trials followed a more shallow linear progression, and in some cases decreased the number of nodes toward the end of the trial.

Additionally, Figure 4 demonstrates how graph size changed with respect to A* satisfaction for each trial. We can see that in almost all cases, *the genetic algorithm was able to build waypoint graphs that obtained the same A* satisfaction rating as the uninformed system, yet required far fewer nodes.* This was a huge achievement for the genetic algorithm system and demonstrates its ability to intelligently retain graph attributes that benefit its fitness.

5 Summary

Overall, there were many challenges to overcome in our genetic algorithm implementation, and there is still much more that could be done to further improve that system that is in place. Further work could improve the overall runtime of the algorithm, including parallelizing the evaluation and crossover stages. Additionally, the mutation stage could also be improved by moving pre-existing nodes, or adding and deleting edges. Several constraints could also use tuning, including the number of graphs required in each generation or the percentage of nodes cycled out during each generation. Finally, it would help to run this algorithm with larger mazes and see how it operates in a larger environment where obtaining a high A* satisfaction requires even more thorough maze coverage.

The algorithm in its current state functions as a strong proof of concept for the benefits of genetic algorithms in waypoint graphs. The system was able to achieve a similar A* satisfaction rating to that of a completely uninformed system, while using substantially fewer nodes in the process. With some additional tuning, this system could serve as a very useful, optimized, fully automated environment waypoint mapping algorithm.

6 Citations

1. Jacobson L. (2012, Feb 12). "Creating a Genetic Algorithm for Beginners" [Blog].

   Available: http://www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3

2. McCall, J."Genetic Algorithms for Modelling and Optimization" Journal of Comp. and Applied Math, vol. 184, no. 1, pp. 205-222, Dec 2005