A compared to Genetic Algorithms and Artificial Neural Networks for grid traversal problems*

Kaloyan Penev

Bournemouth University 2019

1. Introduction of the considered methods

1.1. A*

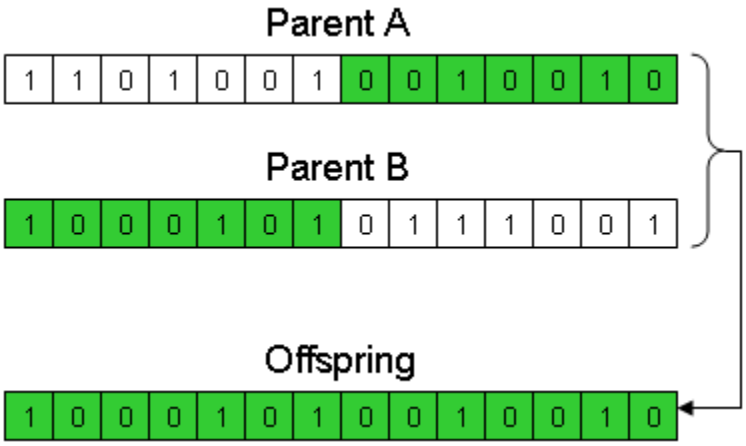
A*(star) is a deterministic path-finding algorithm which is an extension of Dijkstra's graph-traversal algorithm. It searches by analyzing adjacent grid position, with consideration to a given heuristic. Once it finds the solution, it optimizes the search area into the best path by tracking its progress. It caches already searched positions, which could make it computationally heavy in extreme cases, however even then, it often remains the best path-finding algorithm (W. Zeng and R. L. Church).

1.2. Genetic Algorithm

The genetic algorithm mimics evolution in nature, where the most adaptive individuals survive. The concept is to selectively breed genomes from a population of individuals based on certain criteria defined by a "fitness" function. I will consider a Genetic algorithm, in which:

1. Genomes are randomly generated paths, represented by 16-bit strings of zeros and ones. Two bits denote 1 move and whenever a move hits a wall, the individual *stays* at the same position.
2. Fitness function based on the Euclidean distance to the end of the maze:
$$f(\Delta x, \Delta y) = \frac{1}{sqrt(\Delta x^2 + \Delta y^2) + 1}$$
The function is bound in the range of 0 to 1, where the best result is 1.

3. Crossover method consisting of “slicing” the genetic properties of two individuals in half and building an offspring with the halves. This is a stochastic process with 70% chance of happening.



4. Mutation which is also a stochastic process with a small chance (0.001) for the offspring to mutate. A mutation would be a zero turning into a one.
5. Repopulation if the chromosome pool is filled with genomes optimized for a local optimum (see section 3.1.)

1.3. Artificial Neural Networks

Artificial neural networks are modeled after the human brain. They are built out of neurons – basic nodes which take input and produce output. In a neural network, connections to other neurons are governed by properties called “weights”. The “weight” is a unique value of a neuron that determines how the input is processed to produce the output. Whenever input is received, it is multiplied by the weight, the resulting value is then set to be the value of the “neuron” body. That value is then run through an “activation function” which determines the output of the neuron. The artificial neural network I will consider uses:

1. 1 Input layer of 2 input neurons denoting current X position and current Y position (possibly normalized?).
2. 1 Hidden layer of 4 neurons representing the Manhattan move set – up, down, left, right.
3. 1 Output layer of 1 output neuron denoting the move to be taken.
4. A population size of however many moves one would like to make.
5. Sigmoid activation function
6. A desired output of moving to an adjacent open space which is the closest to the end in Euclidean distance. Additionally, the last system individual has a desired output of moving to the end position.
7. Initial learning rate of 0.25.

2. Consideration: Artificial Neural Network or Genetic Algorithm

2.1. The Problem

The task at hand is to traverse and solve a maze represented by a 2D binary grid terrain map. This will be referred to as "The Problem" further on.

2.2. Analysis of Genetic Algorithms

Genetic algorithms are non-deterministic metaheuristics. To break it down, non-deterministic means that it employs stochastic methods for searching. Metaheuristic means that it is a problem-independent general technique that focuses on looking for heuristics, or solutions that are "good enough" in a computing time that is "short enough". What is "good enough" is defined by the fitness function. My fitness function unfortunately doesn't consider the number of moves or any sort of optimization. It is a primitive equation which evaluates only the Euclidean distance from the end point of the path to the solution point of the maze.

For this Genetic Algorithm(GA), the process of searching for a solution is basically generating random paths, evaluating the area that the paths have traversed and crossing them over with other individuals which results in exploration and eventual solution. Since the fitness function prefers paths that get closer to the end, it is important to note that an effective half of one path could be awful in a combination with a different half-path. However, that doesn't necessarily spell disaster, as in those scenarios it increases its search area, as it traverses new combinations, and consequently increases its chance for finding a solution. As so, the algorithm can't reliably find the best path as it's methodology is more of *trying to generate **a** solution*, and less of *actual optimization of the path*.

2.2. Analysis of Artificial Neural Networks

Artificial neural networks are also non-deterministic methods, but not metaheuristics. To expand, it is a system that uses a stochastic optimization algorithm called the stochastic gradient descent to train itself for a given problem. It doesn't necessarily compute for *any* solution, as it keeps optimizing itself until the solution is "good enough" according to a given criterion (sum of squared errors). And while the genetic algorithm has a predetermined jumbled mess of a search area that it tries to optimize, the neural network is designed to search *by* expanding. However, since it doesn't have any prior knowledge of how the system of the maze works, it has to explore and search for a long time to be able to make sense of its environment (hence the big learning rate that I propose for the implementation). Because it has to explore, the algorithm is slower than GA, although with higher potential, although every time you change any parameter of the maze, it would have to re-learn. In The Problem though, speed is king, so I will *implement only the Genetic Algorithm*. ANN is a method good at battling large-scale problems in which there are too many variables to consider. Artificial Neural Networks thrive in those scenarios, as the computational load is simply too massive to be solved by traditional methods and algorithms.

*3. Implementation in Python: Genetic Algorithm and A**

3.1. Evaluation of the Genetic Algorithm

After implementing the algorithms, a notable reason of why GA struggles with The Problem is revealed - local optima. As the genetic algorithm is a metaheuristic, by design it searches for solutions that are "good enough". In path-finding problems, local optima occur often, as the closest Euclidean distance to the finish might be blocked by a wall (fig. 1) However, in The Problem, local optimum convergence is a weakness.

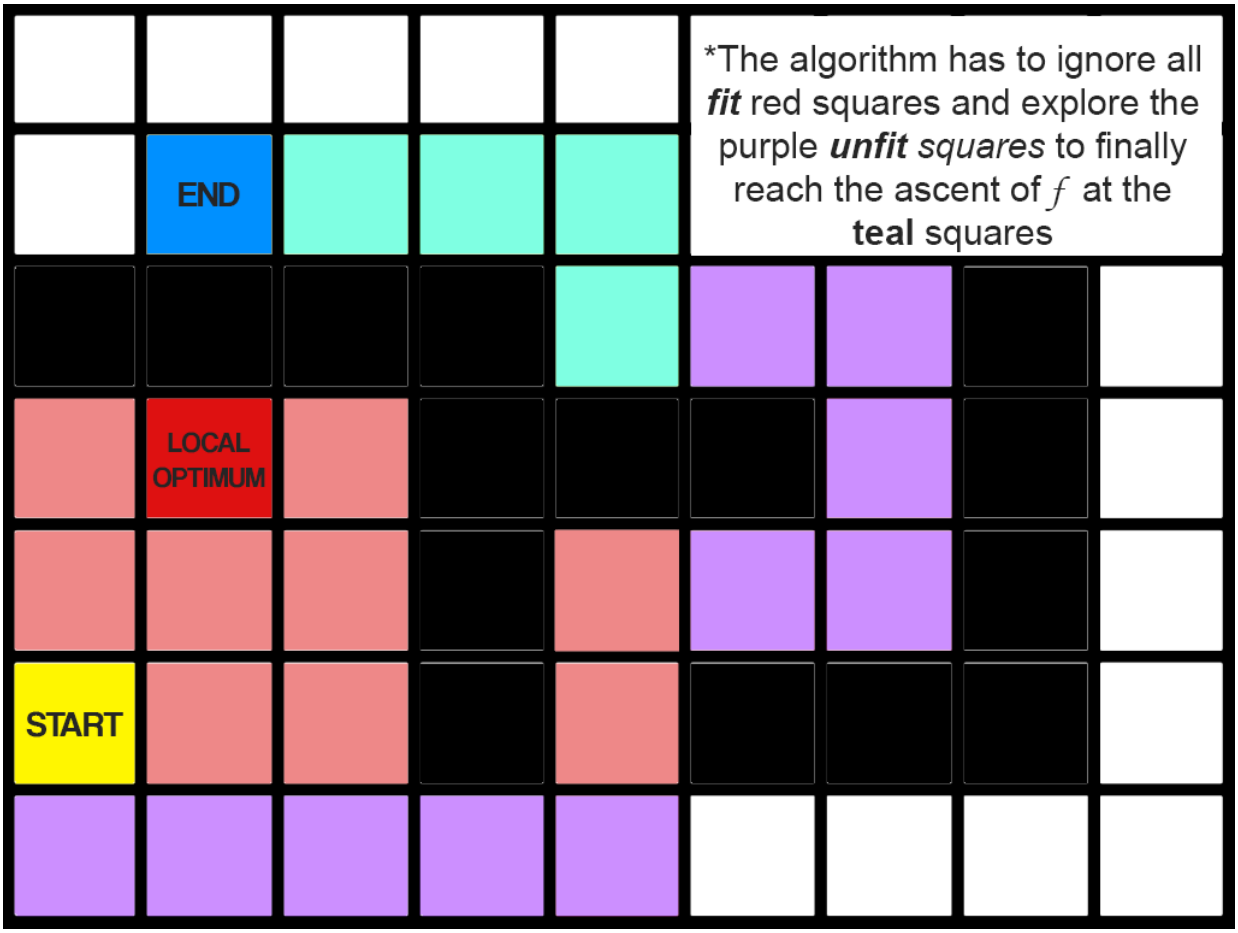


Figure 1. *Special case maze (terrain4.txt)*

As the fitness function peaks at the (4, 2) block, the individuals who end up there have a high fitness value. Individuals who deviate from the local optimum will have a smaller chance of survival as their fitness value is lower, and as a result the chromosome pool will be filled with individuals good at finding the local optimum.

The Mutation method was designed to battle this, however, in practice, as a result of the constant striving for the local optimum, the search area gets more precise, making deviations harder to achieve through Mutation. The genetic algorithm is greedy and doesn't willfully want to decrease its effectiveness, so it only looks to increase its fitness, however in reality it has to decrease its fitness first and explore

more to find the solution. (see Fig. 2) That would be going against the principles of the heuristic nature of the method. To tackle this, I have implemented a function which repopulates the genome pool whenever the fitness function doesn't change in a reasonable amount of generations (implying that the genome pool is most likely filled with "locally optimized" individuals). In special cases where that still doesn't work, a workaround is increasing the algorithm's raw search area – by increasing the genotype length (increasing population size might have the opposite effect – see section 4.). Space and time complexity increase, but it *does* get to the solution. In practical applications, one must consider the worth of computational power against the worth of a solution.

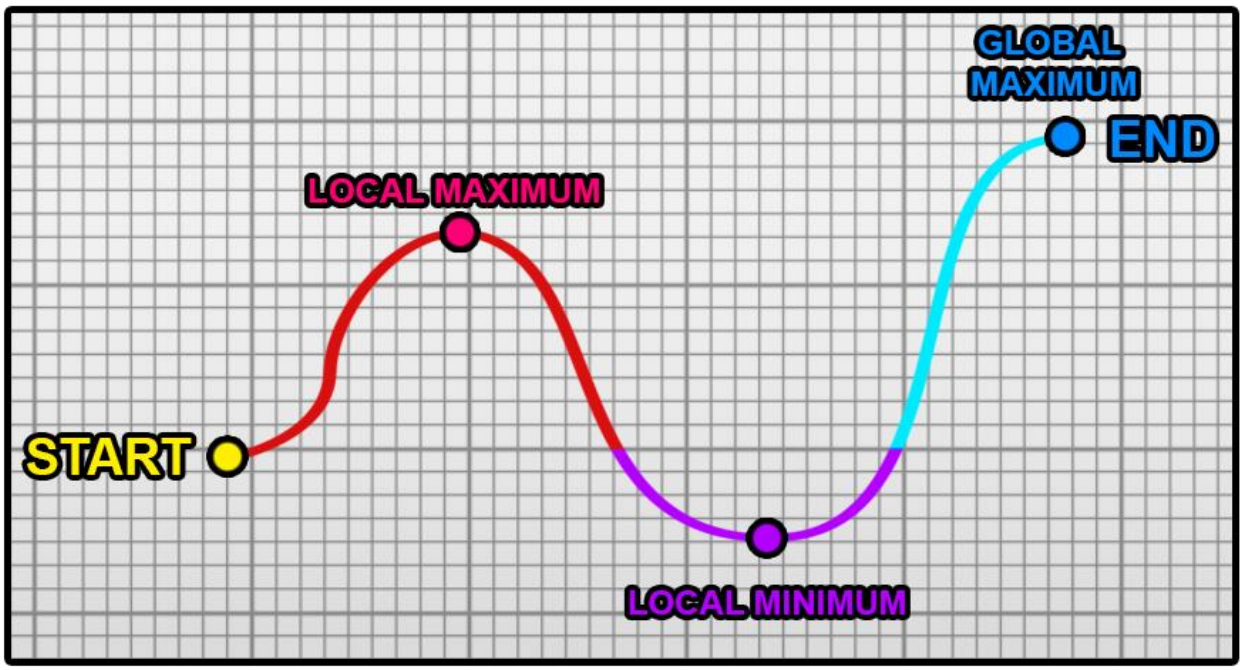


Figure 2. *Curve of the fitness function.* Colors are represented as tiles in figure 1.

3.2. Evaluation of A*

On the other hand, A* was created precisely for this type of problem. It is a highly specific and reliable algorithm which is optimized to be as fast and effective as possible in path-finding problems. No special or extreme case can defeat it and it deals with local optima swiftly, as it continues to iteratively explore until it finds the global optimum (see Fig. 3). As a result, in my implementation, GA couldn't outperform A* even when the GA managed to generate the solution in its first generation (A* - 4ms, GA – 5ms).

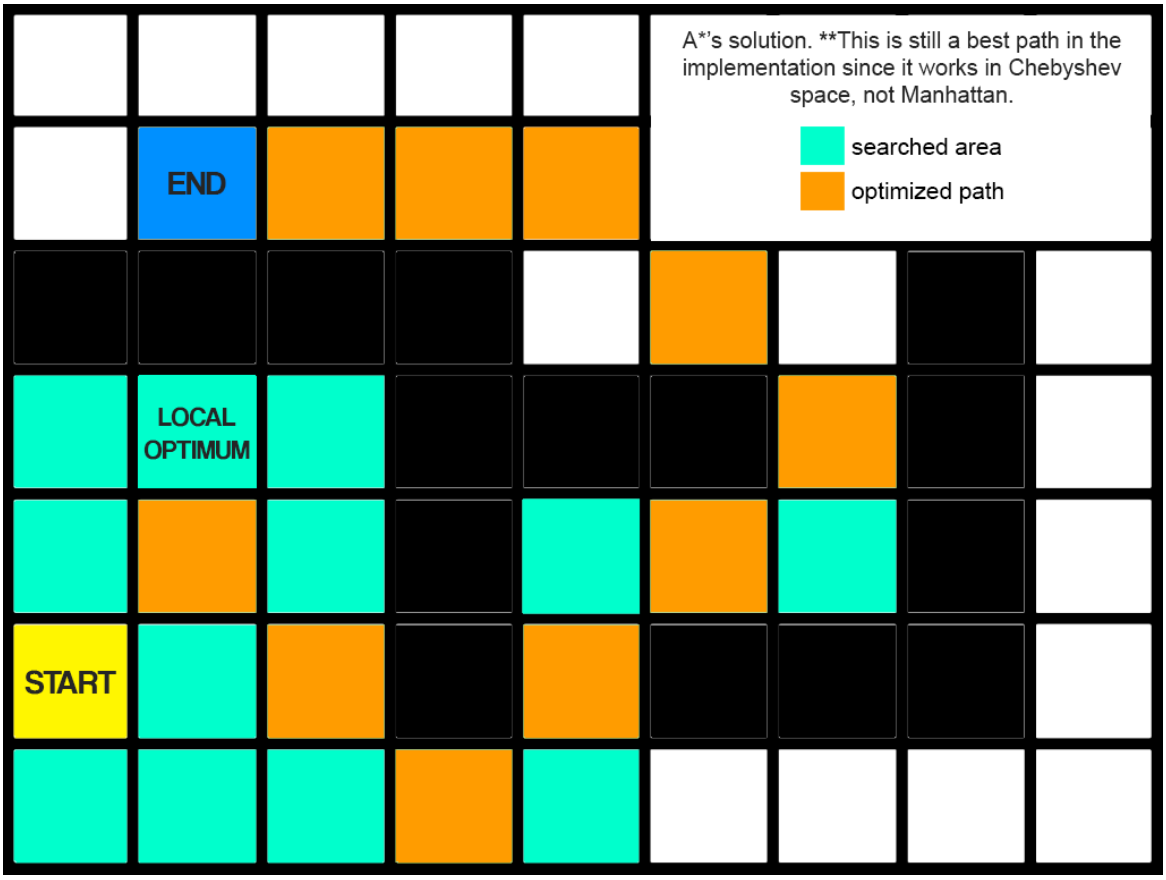


Figure 3. *A* implemented in the same maze.*
Note: The algorithm is implemented in Chebyshev (or Haar) space.

*4. Performance benchmarks: Genetic Algorithm and A**

4.1. Performance of Genetic Algorithm

The genetic algorithm is a stochastic algorithm, resulting in extreme deviations in time of convergence. As a result, making an accurate estimation for performance time is impossible, yet space(memory) and time complexity both grow *linearly*, as the crossover, mutation and fitness functions are all methods that take a constant and known amount of time. As so, for
- N - population size ; and
- L – length of genotype;
we could denote complexity through the Big O notation as:

$$O(n) = (NL)$$

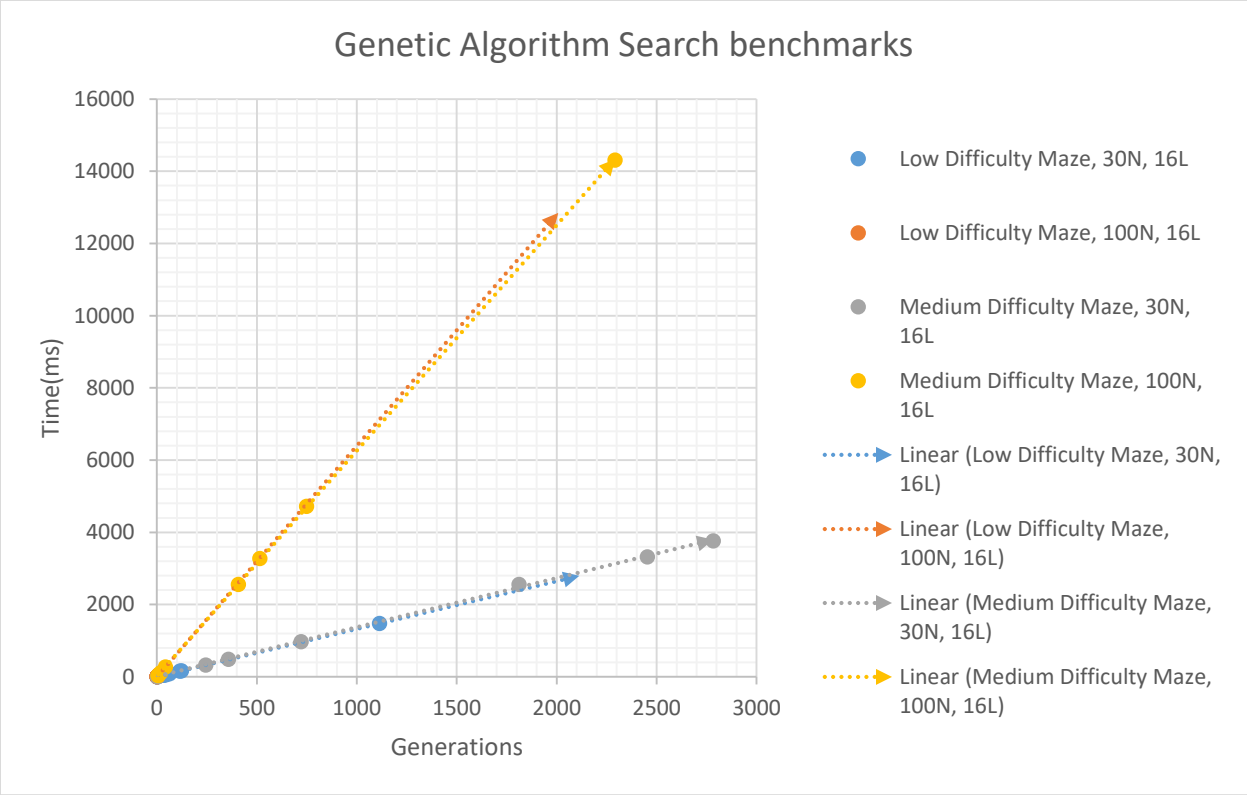
We can conclude that with size of input increasing, resource requirements increase in a linear trend.

4.2. Time Benchmarks for the Genetic Algorithm

Approximate time evaluations were done on a CPU with the following relevant specifications, with no notable background processes running:

CPU: Intel Core i5-2500k @ 4400MHz
RAM: DDR3 16GB
DISK: SSD 250 GB

Pelikan and Lobo (2000) state, “if the quality of a found solution does not grow monotonously with the population size, then GA is probably not the best way to approach this problem”. To test if this statement affects The Problem, I ran a few benchmarks as seen in Graph 1.



Graph 1. 10 Searches performed by Genetic Algorithm

**It is important to note that data has been taken only from searches which converge in a reasonable time (For Medium Difficulty Maze-100N-16L, one in three executions takes >30s to reach convergence).*

Most solutions clutter in the 1-100 generation range; however, we can graph their linear growth by extrapolation. From the noticeable decrease in time performance of the algorithm when using the larger population size (100N) and the low convergence rate in the Medium Difficulty – 100N maze, one can confidently conclude that the quality of the solution does not grow linearly with the population size. As a result, according to Pelikan and Lobo (2000), GA is a bad fit for The Problem.

4.3. Performance of A*

A* is a deterministic method and as a result will always solve the maze in a pre-determinable amount of time with a known amount of memory needed. Additionally, it's space and time complexity is a constant, making it just as good with increased size of input. Additionally, in my empirical evaluations, A* never failed to converge and managed to solve a High difficulty maze in less than ***5ms***. I could not find any flaws in the algorithm's methodology in special and extreme cases of The Problem.

5. Conclusions

For graph traversal path-finding problems, A* is indisputably the best algorithm to use. It scales well with size of input. It's iterative, fast and easy to implement. Finally, it could be modified to work in special conditions.

Bibliography

1. Michael A. Nielsen. 2015. *"Neural Networks and Deep Learning".* Determination Press
2. Negnevitsky, M. (2011). *Artificial intelligence: A Guide to Intelligent Systems*. Harlow: Addison Wesley.
3. Pelikan, M., & Lobo, F.G. 2000. *Parameter-less Genetic Algorithm: A Worst-case Time and Space Complexity Analysis*. GECCO.
4. Wang SC. (2003). Artificial Neural Network. In: *Interdisciplinary Computing in Java Programming*. The Springer International Series in Engineering and Computer Science, vol. 743. Springer, Boston, MA
5. Brownlee, J. (2018). "*Why Initialize a Neural Network with Random Weights?"*. Machine Learning Mastery. URL: https://machinelearningmastery.com/why-initialize-a-neural-network-with-random-weights/
6. Zeng, W., & Church, R. L. (2009). *Finding shortest paths on real road networks: the case for A**. http://doi.org/10.1080/13658810801949850
7. Sörensen, K. & Glover, F. 2015. *Metaheuristics*. Scholarpedia, 10(4):6532
8. Prakoonwit, S. 2019.: AI Game Programming Lecture E: *A* and Evaluation Methods*. Bournemouth University.
9. Prakoonwit, S. 2019.: AI Game Programming Lecture D: *Genetic Algorithm*. Bournemouth University.
10. Swift, N. 2017. *"Easy A* (star) Pathfinding".* Medium. URL: https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2