# Chapter 34 - Genetic and Evolutionary Algorithms

## Artificial Evolution

### 34.1 – Introduction

Some computational tasks are exceptionally hard for computers to solve within a reasonable amount of time, and a reasonable amount of resources (e.g., memory). A special class of these tasks is even said to be NP hard. n computational complexity theory, NP-hardness (non-deterministic polynomial-time hardness) is the defining property of a class of problems that are informally "at least as hard as the hardest problems in NP". This is a rather vague definition, so let us also share a more intuitive one. A problem is considered NP-hard when we believe there exists no algorithm that scales in memory resources and time needed to test all possible solutions to a certain input. Typically, NP hard problems suffer from a combinatorial explosion of solutions. Think of even a modestly sized graph of n nodes, the number of edges is n \* (n-1) if nodes cannot be connected to themselves and the graph is directed. So, for a graph of 3 nodes, there are 6 edges, and for a graph of 10 nodes there are 9900 edges. Now imagine of the number of possible paths that visit every node n once and only once. For a graph of 3 nodes, we have 6 paths possible paths, but already at 10 nodes, we have 3.6 million+ paths, as the number of possible path p for n nodes is given by n! (n factorial, computed as n \* (n-1) \* (n-2) \* .

### 34.2 – Traveling Salesman Problem

The above highlights the difficulty of the classic traveling salesman problem (TSP) in computer science: If our salesman needs to visit n number of cities at least once, how do we efficiently finds the shortest path between cities that accomplishes that? To find the path, we will have to iterate through every possible path and compare it to some current global shortest path in memory. The amount of time such a brute force algorithm will take is given by the number of paths, and for this problem, there isn't a smarter more efficient algorithm that can do it faster, as far as we know.

### 34.3 – Solving Problems Through Genetic Algorithms

Instead, problems like the Traveling Salesman can be solved, at least by approximation, by evolutionary or genetic algorithms. Both genetic or evolutionary algorithms rely on the mechanisms of evolution found in nature to the problem of finding an optimal solution: fitness, selection, mutation, and crossover, which we discuss in greater detail below. After discussing these mechanisms, we will have the necessary vocabulary to explain the precise difference between evolutionary and genetic algorithms. In both types of algorithms, we explore many possible solutions to our computational problem at once.

### 34.4 – The Evolution of our Algorithm - Initialization

First, we generate a large number 'individuals' that each encode a potential solution to our problem. For example, in the traveling salesman problem, we initialize our population of individuals by given each one a random ordering of the n cities to be visited. You can think of this sequence of n unique nodes as the individual's DNA, the genetic code we are trying to optimize. If the number of individuals we initialize is small in comparison to the number of possible paths, it is likely none of these individuals is already the most optimal solution (i.e., the shortest path). Instead, some individuals might have partially optimal solutions, meaning they got some transitions of the most optimal path right by chance. But the majority will encode a path that is far from optimal. The optimality of an individual's solution (i.e., path) is used to rank the individuals based on their fitness. The shorter the total length of the path encoded, the higher the fitness. The use of a population of solutions helps the evolutionary algorithm avoid becoming "trapped" at a local optimum, when an even better optimum may be found outside the vicinity of the current solution. The population explores the state space of solutions in many different directions, in contrast to for example gradient descent where a single solution is optimized continuously by adjusting the parameters of the model. The exploration is iterative, and consists of 3 steps: selection, cross-over and mutation.

### 34.5 – The evolution of our Algorithm - Selection

After the initial population has been created and each individual assigned a fitness value, we select 50% of the individuals in population associated with the highest fitness values and divide them into two groups of equal size. The remaining 50% of individuals with the lowest fitness scores are removed from the population.

### 34.6 – The evolution of our Algorithm - Cross-over

Borrowing from biological reproduction, every individual from one group is combined with one member of the other group. These 'parent' individual then create 2 offspring individuals using a technique called cross-over. There are more than one way to do cross-over, but all work by selectively recombining the 'genetic material' from both parents. Each parent may have had a sub path (e.g., 2 -> 3 -> 4 of the most optimal paths (1 -> 2 -> 3 \_> 4 -> 5) already encoded, and the goal is to end up with offspring inheriting and recombining these optimal sub path. Of course, it is equally likely the offspring inherits the worst sub paths from its parents, resulting in a lower fitness score for the offspring , relative to the parents. As such, it is unlikely not be included in the next set of parents selected to produce new offspring, so their lineage encoding a poor solution will not continue.

### 34.7 – The evolution of our Algorithm - Mutation

In addition to crossover, another biological process is also used to generate variability: mutation. Mutation in our traveling salesman problem can be accomplished by randomly swapping one or more of the nodes in an individual's encoded path, with some low probability.

### 34.8 – The evolution of our Algorithm - Offspring

Since we used 50% of the initial population to generate offspring by taking two individuals and having them produce exactly 2 offspring means that the population size stays the same. The individuals that had a fitness lower than the population median were removed, and replaced by an equal number of new individuals, the offspring of the individuals who had a fitness higher than the population median. Selection, cross over and mutation comprise a single step in the evolution of the population and is repeated until the fitness of the population appears to stabilize. At this point, we can take the most fit individual(s) as the solution of our problem. However, it is important to remember that the most optimal solution only optimal relative to all other solutions the population explored. It is not guaranteed to be the optimal solution across all possible paths, especially when the number of possible paths is high. If we wanted to know whether it was the most optimal solution theoretically, we find ourselves having to apply the brute force approach to first find the most optimal solution to compare to what our evolutionary process has produced over many generations, defeating the purpose of the evolutionary approach all together). However, many different genetic and evolutionary algorithms have found solutions for problems that are optimal or very close to optimal. In addition, and perhaps more exciting, genetic/evolutionary algorithms have found solutions to problems more optimal than the ones created by humans, solutions that often appear counterintuitive to those same human experts, but work, nonetheless.

### 34.9 – The Genetic or Evolutionary?

Returning to the definition of and distinction between the terms genetic and evolutionary algorithms, we note that they are often used interchangeable, but that they are not the same. All algorithms that adapt and evolve in way that is analogous to natural selection and evolution in biological system can be called evolutionary algorithms. Genetic algorithms can be seen as a subclass of these algorithms. These algorithms represent their solution as a genetic code similar to DNA, typically as bit strings that the individual uses as instructions to determine its fitness. Encoded in this way changes how new individuals are produced by combining parent individuals during cross over and mutation. For example, imagine an evolutionary algorithm finding the best solution to the following set of equations

y = ax + bx2 + cx3 + d

In other words, we need to the best values of a, b, c, and d as to produce an outcome as close as possible to some ground truth. When two individuals are paired to produce new offspring, one way they can recombine is to represent each individual of having 4 unique values. When two individuals have a high fitness, we can recombine these individuals by perhaps producing offspring with parameters a, b, c, and d that are some weighted averages of these values in their parents. The idea behind this is that two individuals have a high fitness because their own parameters a, b, c, and d are relatively close to the optimal solution, then a weighted average between individuals has a reasonable chance of producing an even higher fitness. In a genetic algorithm, we instead encode these four parameters as bit strings. During cross-over, individual bits are swapped, without regards to the thing they help encode (the parameters). This is a more subtle way of recombining what we deem as 'fit' solutions into even fitter ones, and one that works well for certain problems, and less well for others. We will provide an example of each in the following two sections and accompanying demo notebooks.

### 34.10 - Genetic Algorithms for Protein Folding

The games offer an intuitive way to explain the concepts behind evolutionary and genetic algorithm, but the application of these algorithms to other domains has generated far greater interest. One such domain is the problem of protein folding. Proteins are large molecules that serve many different roles and functions in biology. As molecules, proteins are long strands of connected atoms that are folded in intricate ways. Not only does that make them fit inside the small confines of a cell, but it also dictates their properties and interactions with the biological system they are part off. Understanding how a protein is folded and how this changes its function would greatly advance and speed up medical science. The goal is to reverse engineer the proteins, starting from their beneficial medicinal properties and ability to cure a disease to the folding of the protein this function requires. This is in contrast with the current situation where a lack of understanding protein folding and function forces pharmaceutical research to take on a trial-and-error approach, cycling through many different folding permutations before hopefully landing on one that has the desired properties. Evolutionary and genetic algorithms have been successfully used in revamping this cumbersome expensive and time-consuming process and provide a way forward to engineer ever more complex proteins to cure our collective ailments and diseases.

### 34.11 - Demos

### 34.12 - References