Daniel Hawkins

Richard Tilquist

CSCI 411

30 October 2023

Nature's Algorithm: Decoding Evolution's Logic

Introduction

Genetic algorithms (GAs) draw inspiration from Darwin's theory of natural selection. These algorithms are specialized tools for optimization and search, using processes like mutation, crossover, and selection. Early computing pioneers, such as Turing, von Neumann, and Wiener, used biology as an inspiration to simulate evolution. John Holland furthered this approach in the 1960s, introducing mechanisms that mirror natural selection and genetic variation. (Mitchell) Over the years, GAs have become more sophisticated and intertwined with other methods in evolutionary computation.

They have applications across diverse sectors such as finance, video game development, robotics, and medical research. Specifically, GAs have played a role in areas such as improving chemical reactions in plasma-catalytic conversions, designing advanced frequency-selective surfaces for electromagnetic applications, and creating enhanced tools for graph visualization. Additionally, they synergize with neural networks and simulated annealing to improve tasks like scheduling, identifying gene-regulatory sequences, and simulating human voice synthesis. (Roeva)

Intuition

The goal of GAs is to find the optimal solution for the problem you are trying to solve from a (usually large) solution set. Much like how nature evolves species to adapt

and survive in their environments through the process of natural selection, GAs use principles from genetics to evolve solutions to problems. In GAs, each potential solution is thought of as an individual "chromosome" made up of smaller components known as "genes". In the context of this project, the solutions being looked at are in the form of a 10-element array of bits, which are either 0s or 1s. Each bit in this array is like a gene in the chromosome. This project's ultimate goal is to identify a pattern composed entirely of 1s, as this represents the optimal solution.

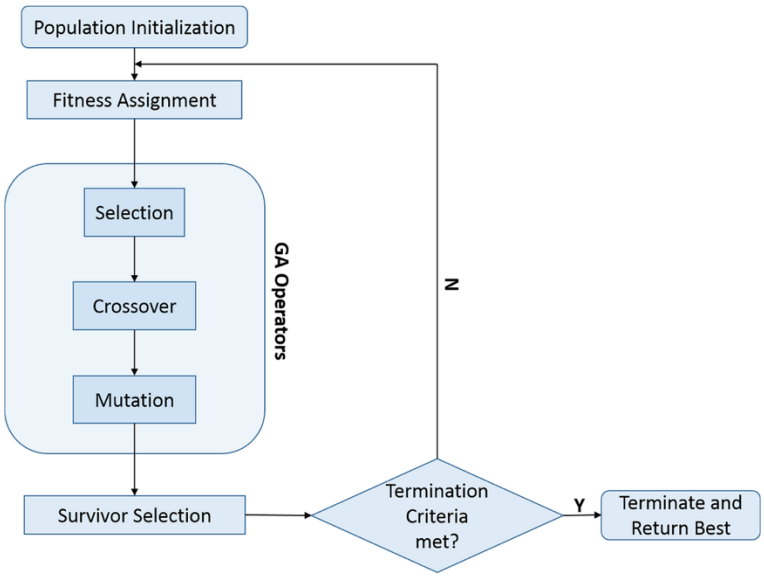

Fig. 1. Flow Chart of Genetic Algorithm (Anand Deshpande)

Figure 1 shows a high-level flow chart outlining the GA procedure. The process begins with generating an initial set of potential solutions. A potential solution for this project might look like [0, 0, 1, 1, 0, 0, 1, 0, 1, 1]. The next step involves assessing each solution's effectiveness by assigning a fitness value using a designated fitness function. This function is tailored specifically to the problem trying to be solved and often is one of the most complex components of the GA. For this project, the fitness function calculates

the sum of all elements within a solution (see Figure 3). The closer a solution is to an array comprised entirely of 1s, the higher its resulting fitness score.

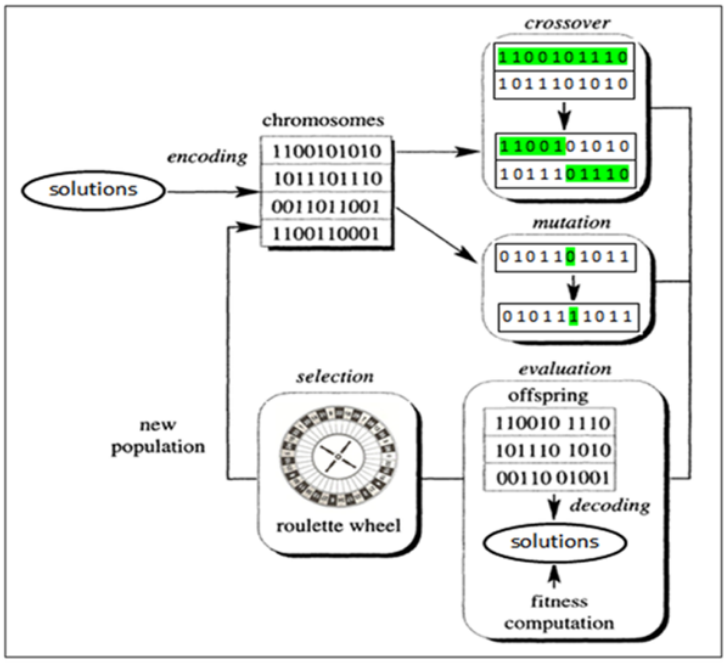

Fig. 2. Flow Chart of Genetic Algorithm (Mitsuo Gen)

Following the fitness evaluation, a collection of solutions is selected for the mating pool based on their fitness scores. While there are various methods for selection, such as the roulette wheel (see Figure 2), this project utilizes truncation selection. This deterministic approach selects parent solutions from the top-performing solutions. For this project, that means choosing from the top 50% for the crossover (reproduction) process.

```
fitness([0,1,1,0,0,0,1,1,0,1]) = 5
fitness([1,1,1,1,0,1,1,0,1,1]) = 8
```

Fig. 3. Fitness Function Example

Solutions proceed to the "crossover" stage which mimics biological reproduction. Genes (or bits) from two parent solutions combine to produce offspring (See Figure 4). For this project, crossover involves the random swapping of bits between two parents, creating the next generation of solutions for the GA process.

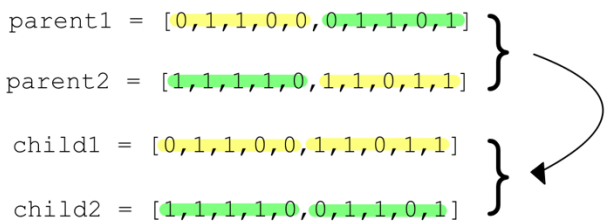

Fig. 4. Crossover Example

To ensure diversity, and avoid premature convergence, random mutations are introduced to the offspring. This not only ensures variety but can also steer solutions closer to the optimum. The mutation rates can vary, they typically range between 0.1% and 1.0% of the genes (bits) mutated. For this project, a mutation translates to flipping a bit from either 0 to 1 or vice versa (See Figure 5).

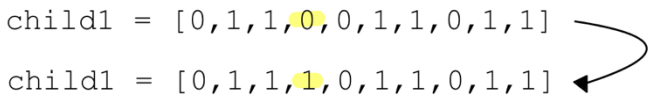

Fig. 5. Mutation Example

Post mutation, the newly formed population advances to the succeeding generation. The iterative process of selection, crossover, and mutation is repeated. With each iteration, the population's overall fitness tends to improve. The algorithm terminates when certain conditions are met. These might include: a solution meeting minimum or optimal criterion, a predetermined number of generations being reached, minimal variance among individual solutions suggesting population convergence, or insignificant

improvement is being observed over several generations. Upon meeting the set criterion, the GA identifies the most fit chromosome. For this project, our algorithm will terminate after a specific number of generations or if the optimal chromosome has been found.

Pseudocode and Detailed Description

Terminology:

- Chromosome: A solution represented as a list of binary integers (0s and 1s). Each binary integer in this context is termed as a 'gene'.
- Gene: A binary digit (either 0 or 1) within a chromosome.
- Population: A list of chromosomes, representing a set of potential solutions.

```
function initPopulation(popSize, chromLen)
    population = []
    for i = 1 to popSize
        chrom = []
        for j = 1 to chromLen
            bit = randomChoice([0, 1])
            chrom = chrom + [bit]
        population = population + [chrom]
    return population
```

Description:

- Initializes an initial population for the Genetic Algorithm.

Arguments:

- `popSize - int`: The desired number of chromosomes in the initial population.
- `chromLen - int`: The number of genes in each chromosome.

Return Value:

- `list[list[int]]`: The generated population consisting of `popSize` chromosomes, each with `chromLen` genes.

```
function fitness(chrom)
    return sum(chrom)
```

Description:

- Computes the fitness of a chromosome by summing the values of its genes.

Arguments:

- `chrom – list[int]`: A chromosome.

Return Value:

- `int`: The fitness score of the chromosome, determined by the total sum of its genes.

```
function selectParents(population)
    if length(population) == 1
        return population[0], population[0]
    elif length(population) == 2
        return population[0], population[1]
    parent1 = randomChoice(population[0:length(population)/2])
    parent2 = randomChoice(population[0:length(population)/2])
    return parent1, parent2
```

Description:

- Selects two parent chromosomes randomly from the first half (top-performing) of the given population for genetic crossover. If there is only one chromosome in the population, the chromosome crosses with a clone of itself.

Arguments:

- `population – list[list[int]]`: A population sorted by fitness, starting with the largest.

Return Value:

- `tuple(list[int], list[int])`: A pair of selected parent chromosomes from the population.

```
function crossover(parent1, parent2)
    point = randomInt(1, length(parent1) - 1)
    child1 = sublist(parent1, 0, point) + sublist(parent2, point,
length(parent2))
    child2 = sublist(parent2, 0, point) + sublist(parent1, point,
length(parent1))
    return child1, child2
```

Description:

- Performs a single-point crossover between two parent chromosomes to produce two child chromosomes. The crossover point is randomly selected, and genes before this point are taken from one parent, while genes after this point are taken from the other parent, for both children.

Arguments:

- `parent1 – list[int]`: The first parent chromosome.
- `parent2 – list[int]`: The second parent chromosome.

Return Value:

- `tuple(list[int], list[int])`: A pair of child chromosomes.

```
function mutate(chrom, mutRate)
    newChrom = []
    for i = 1 to length(chrom)
        if randomFloat(0, 1) > mutRate
            newChrom = newChrom + [chrom[i]]
        else
            if chrom[i] == 0
                newChrom = newChrom + [1]
            else
                newChrom = newChrom + [0]
     return newChrom
```

Description:

- Mutates the genes within a chromosome based on a given mutation rate. For each gene in the chromosome, a random number between 0 and 1 is generated. If this number exceeds the mutation rate, the gene is left unchanged. Otherwise, the gene is flipped (0 becomes 1 and 1 becomes 0).

Arguments:

- `chrom – list[int]`: The chromosome to be mutated.
- `mutRate – float`: The mutation rate, a value between 0 and 1, indicating the probability of each gene undergoing mutation.

Return Value:

- `list[int]`: The mutated chromosome.

```
function geneticAlgo(numGen, popSize, chromLen, mutRate)
      population = initPopulation(popSize, chromLen)
      for i = 1 to numGen
          population = sort(population, key=fitness)
          if fitness(population[0]) == chromLen
            return population[0], i + 1
          newPop = []
          while length(newPop) < popSize
              parent1, parent2 = selectParents(population)
              child1, child2 = crossover(parent1, parent2)
              child1 = mutate(child1, mutRate)
              child2 = mutate(child2, mutRate)
              newPop = newPop + [child1, child2]
          population = newPop
      return max(population, key=fitness)
```

Description:

- Implements the GA for a given number of generations. The GA starts by initializing a population of chromosomes. For each generation, the population is sorted based on fitness, checked to see if the optimal solution has been reached, and new offspring are produced through crossover and mutation. By the end of

the specified number of generations, the function returns the best solution (chromosome with the highest fitness) from the final population.

Arguments:

- `numGen – int`: The number of generations the GA should run.
- `popSize – int`: The size of the population in each generation.
- `chromLen – int`: The length of each chromosome in the population.
- `mutRate – float`: The mutation rate, a value between 0 and 1, indicating the probability of each gene in a chromosome undergoing mutation.

Return Value:

- `list[int]`: The best chromosome (solution) from the final generation.

Run Time Analysis

Starting with the `initPopulation` function, the outer loop runs for `popSize` iterations, and within it, an inner loop runs for `chromLen` iterations. The assignment, random choice, append, and return operations are all constant. Given this, the overall complexity of the `initPopulation` function is `O(popSize*chromLen)`.

The `fitness` function calculates the sum of all bits in a chromosome. It iterates through the chromosome with length `chromLen` once so its complexity is `O(chromLen)`.

The `selectParents` function randomly selects two parents from the top half of the sorted population. Since selecting a random element from a list is a constant time operation, the complexity of this function is `O(1)`.

The `crossover` function uses the `sublist` function to split the parents when creating the child chromosomes. This complexity of creating these child chromosomes is

linear with respect to the length of the parent chromosomes, which is `chromLen`. Therefore, the overall complexity for `crossover` is `O(chromLen)`.

In the `mutate` function, each gene in a chromosome is iterated through to determine if it should be mutated. This function runs in `O(chromLen)` time, as it potentially modifies each gene in the chromosome.

Finally, the `geneticAlgo` function contains the main loop, which runs for `numGen` iterations. Within this loop, a number of operations take place. The population is sorted based on `fitness` using Python's sort (Timsort), which has a complexity of `O(popSize*log(popSize))`. Check to see if the optimal solution has been reached (`fitness/O(chromLen)`), return the population and number of generations (`O(1)`). After sorting, until the new population is fully formed, parents are selected (`selectParents/O(1)`), crossed over to produce offspring (`crossover/O(chromLen`), and these offspring are potentially mutated (`mutate/O(chromLen`). Given that each of these operations either scales with `popSize` or `chromLen` and are nested within the `numGen` loop, the overall complexity of the main function is `O(numGen*popSize*chromLen+numGen*popSize*log(popSize))`.

Genetic Algorithms, by nature, are highly flexible and can be tailored to diverse problems leading to significant variations in runtime. Factors like population size, mutation and crossover techniques, selection methods, and the complexity of the fitness function can substantially impact their efficiency. Additionally, the stopping criteria, whether based on a fixed number of generations or a convergence threshold, further adds variability. Thus, two genetic algorithms, even if applied to the same problem, might exhibit vastly different runtimes due to these design and parameter choices.

Works Cited

Anand Deshpande, Manish Kumar. *Artificial Intelligence for Big Data: Complete guide to automating Big Data solutions using Artificial Intelligence techniques*. Birmingham, UK: Packt Publishing, 2018.

Mitchell, Melanie. *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT, 1998.

Mitsuo Gen, Runwei Cheng. *Genetic Algorithms & Engineering Design*. Hoboken, NJ: John Wiley & Sons, Inc, 1997.

Roeva, Olympia. *Real-World Applications of Genetic Algorithms*. Rijeka, Croatia: InTech, 2012.