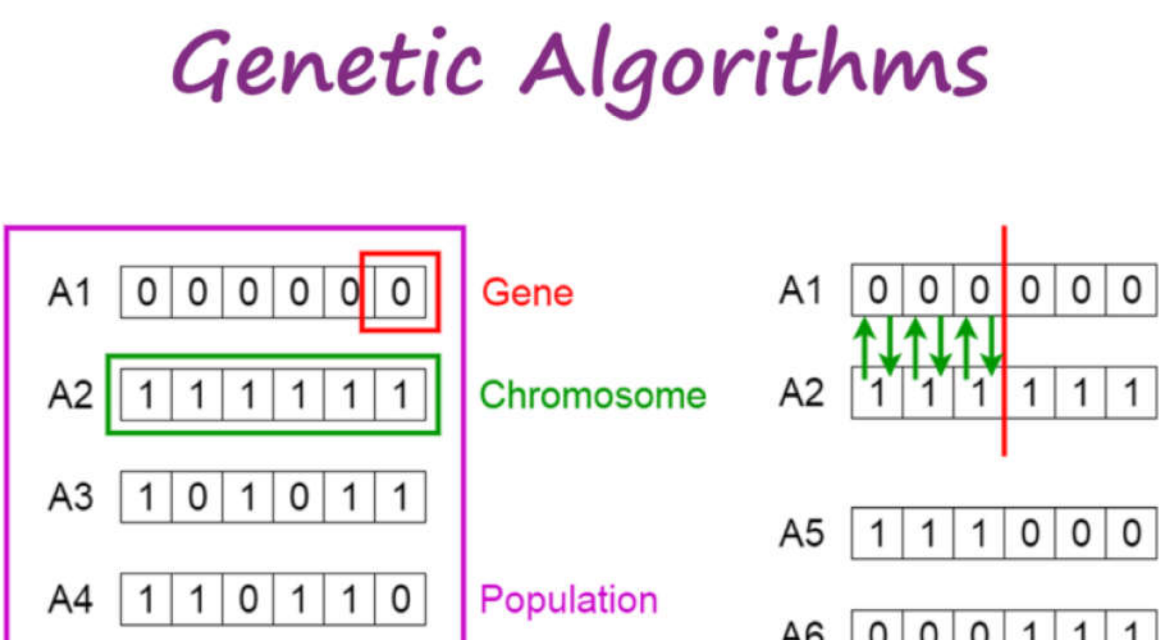
Introduction to Genetic Algorithms — Including Example Code

Vijini Mallawaarachchi Jul 8, 2017 · 4 min read ★

A **genetic algorithm** is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.



Notion of Natural Selection

The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found.

This notion can be applied for a search problem. We consider a set of solutions for a problem and select the set of best ones out of them.

Five phases are considered in a genetic algorithm.

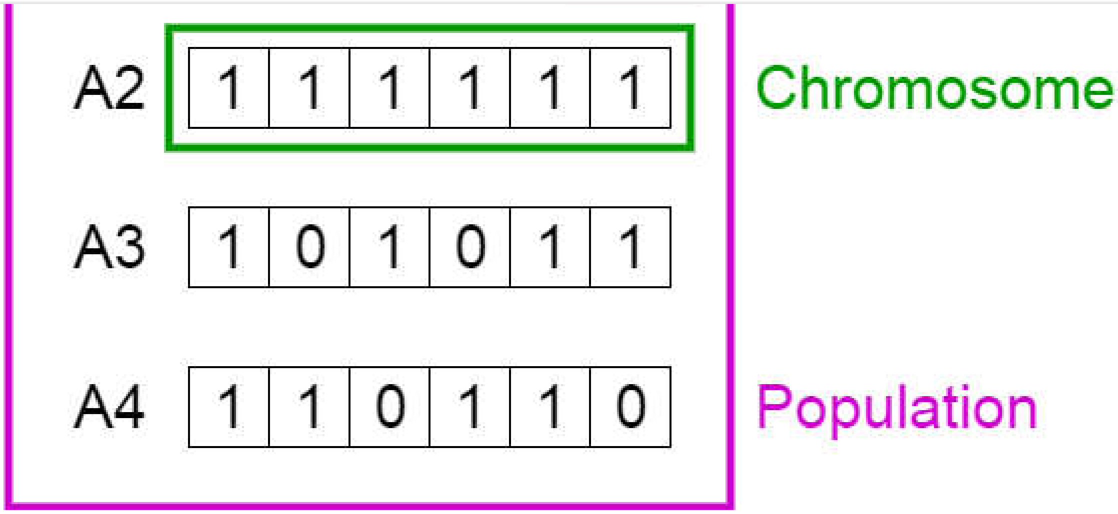
1. Initial population
2. Fitness function
3. Selection
4. Crossover
5. Mutation

Initial Population

The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve.

An individual is characterized by a set of parameters (variables) known as **Genes**. Genes are joined into a string to form a **Chromosome** (solution).

In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s). We say that we encode the genes in a chromosome.



Population, Chromosomes and Genes

Fitness Function

The **fitness function** determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a **fitness score** to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.

Selection

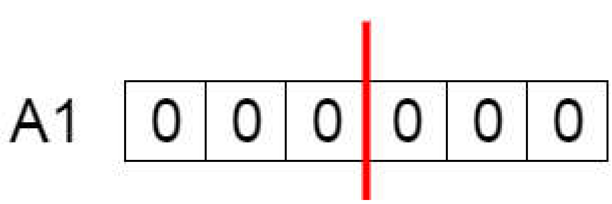
The idea of **selection** phase is to select the fittest individuals and let them pass their genes to the next generation.

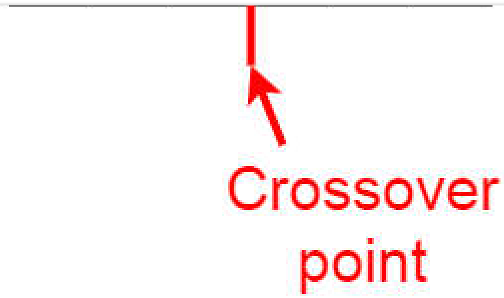
Two pairs of individuals (**parents**) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.

Crossover

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a **crossover point** is chosen at random from within the genes.

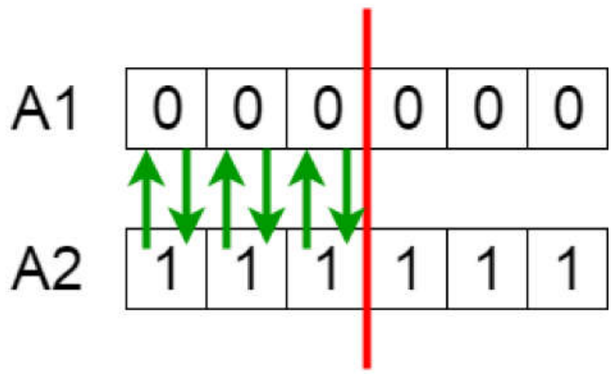
For example, consider the crossover point to be 3 as shown below.





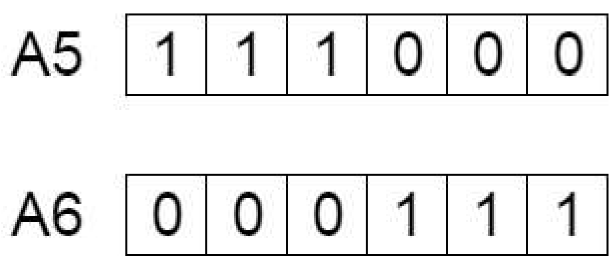
Crossover point

Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached.



Exchanging genes among parents

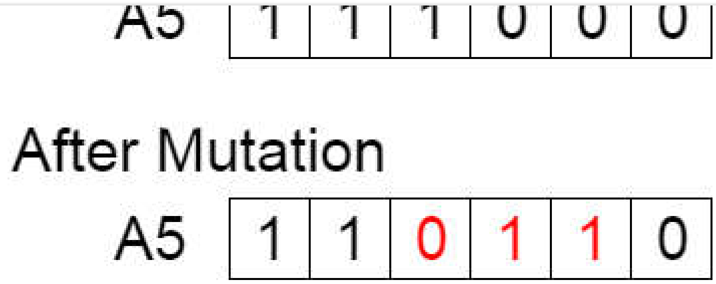
The new offspring are added to the population.



New offspring

Mutation

In certain new offspring formed, some of their genes can be subjected to a **mutation** with a low random probability. This implies that some of the bits in the bit string can be flipped.



Mutation: Before and After

Mutation occurs to maintain diversity within the population and prevent premature convergence.

Termination

The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.

Comments

The population has a fixed size. As new generations are formed, individuals with least fitness die, providing space for new offspring.

The sequence of phases is repeated to produce individuals in each new generation which are better than the previous generation.

Pseudocode

```
START
Generate the initial population
Compute fitness
REPEAT
    Selection
    Crossover
    Mutation
    Compute fitness
UNTIL population has converged
STOP
```

play around with the code.

Given a set of 5 genes, each gene can hold one of the binary values 0 and 1.

The fitness value is calculated as the number of 1s present in the genome. If there are five 1s, then it is having maximum fitness. If there are no 1s, then it has the minimum fitness.

This genetic algorithm tries to maximize the fitness function to provide a population consisting of the fittest individual, i.e. individuals with five 1s.

Note: In this example, after crossover and mutation, the least fit individual is replaced from the new fittest offspring.

```
import java.util.Random;

/**
 *
 * @author Vijini
 */

//Main class
public class SimpleDemoGA {

    Population population = new Population();
    Individual fittest;
    Individual secondFittest;
    int generationCount = 0;

    public static void main(String[] args) {

        Random rn = new Random();

        SimpleDemoGA demo = new SimpleDemoGA();

        //Initialize population
        demo.population.initializePopulation(10);

        //Calculate fitness of each individual
```

Get started Open in app

```java

        //While population gets an individual with maximum fitness
        while (demo.population.fittest < 5) {
            ++demo.generationCount;

            //Do selection
            demo.selection();

            //Do crossover
            demo.crossover();

            //Do mutation under a random probability
            if (rn.nextInt()%7 < 5) {
                demo.mutation();
            }

            //Add fittest offspring to population
            demo.addFittestOffspring();

            //Calculate new fitness value
            demo.population.calculateFitness();

            System.out.println("Generation: " + demo.generationCount + " Fittest: " + demo.popu
        }

        System.out.println("\nSolution found in generation " + demo.generationCount);
        System.out.println("Fitness: "+demo.population.getFittest().fitness);
        System.out.print("Genes: ");
        for (int i = 0; i < 5; i++) {
            System.out.print(demo.population.getFittest().genes[i]);
        }

        System.out.println("");

    }

    //Selection
    void selection() {

        //Select the most fittest individual
        fittest = population.getFittest();

```

```
    }

    //Crossover
    void crossover() {
        Random rn = new Random();

        //Select a random crossover point
        int crossOverPoint = rn.nextInt(population.individuals[0].geneLength);

        //Swap values among parents
        for (int i = 0; i < crossOverPoint; i++) {
            int temp = fittest.genes[i];
            fittest.genes[i] = secondFittest.genes[i];
            secondFittest.genes[i] = temp;

        }

    }

    //Mutation
    void mutation() {
        Random rn = new Random();

        //Select a random mutation point
        int mutationPoint = rn.nextInt(population.individuals[0].geneLength);

        //Flip values at the mutation point
        if (fittest.genes[mutationPoint] == 0) {
            fittest.genes[mutationPoint] = 1;
        } else {
            fittest.genes[mutationPoint] = 0;
        }

        mutationPoint = rn.nextInt(population.individuals[0].geneLength);

        if (secondFittest.genes[mutationPoint] == 0) {
            secondFittest.genes[mutationPoint] = 1;
        } else {
            secondFittest.genes[mutationPoint] = 0;
        }
    }

```

```
                return fittest;
            }
            return secondFittest;
        }


        //Replace least fittest individual from most fittest offspring
        void addFittestOffspring() {

            //Update fitness values of offspring
            fittest.calcFitness();
            secondFittest.calcFitness();

            //Get index of least fit individual
            int leastFittestIndex = population.getLeastFittestIndex();

            //Replace least fittest individual from most fittest offspring
            population.individuals[leastFittestIndex] = getFittestOffspring();
        }

    }


    //Individual class
    class Individual {

        int fitness = 0;
        int[] genes = new int[5];
        int geneLength = 5;

        public Individual() {
            Random rn = new Random();

            //Set genes randomly for each individual
            for (int i = 0; i < genes.length; i++) {
                genes[i] = Math.abs(rn.nextInt() % 2);
            }

            fitness = 0;
        }

        //Calculate fitness
```

```
        fitness = 0;
        for (int i = 0; i < 5; i++) {
            if (genes[i] == 1) {
                ++fitness;
            }
        }
    }

}

//Population class
class Population {

    int popSize = 10;
    Individual[] individuals = new Individual[10];
    int fittest = 0;

    //Initialize population
    public void initializePopulation(int size) {
        for (int i = 0; i < individuals.length; i++) {
            individuals[i] = new Individual();
        }
    }

    //Get the fittest individual
    public Individual getFittest() {
        int maxFit = Integer.MIN_VALUE;
        int maxFitIndex = 0;
        for (int i = 0; i < individuals.length; i++) {
            if (maxFit <= individuals[i].fitness) {
                maxFit = individuals[i].fitness;
                maxFitIndex = i;
            }
        }
        fittest = individuals[maxFitIndex].fitness;
        return individuals[maxFitIndex];
    }

    //Get the second most fittest individual
    public Individual getSecondFittest() {
        int maxFit1 = 0;
        int maxFit2 = 0;
        for (int i = 0; i < individuals.length; i++) {
```

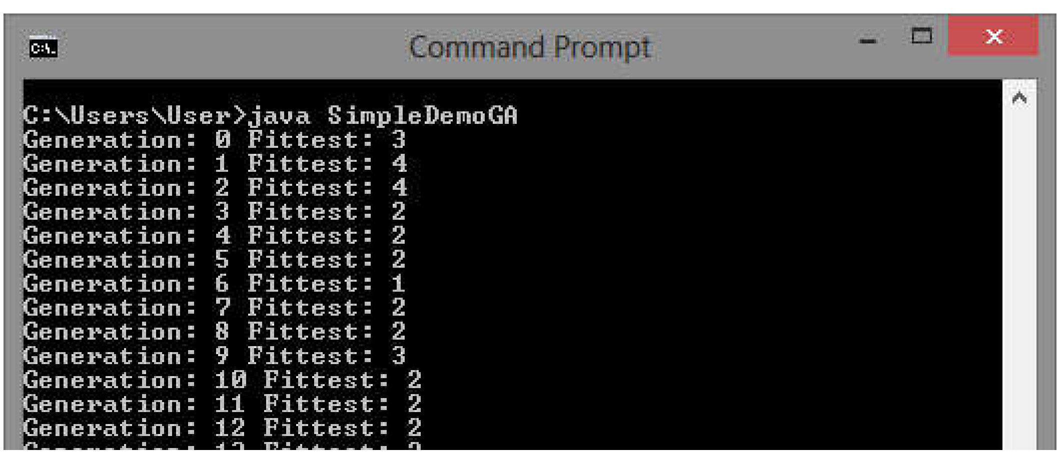
```java
                maxFit1 = i;
            } else if (individuals[i].fitness > individuals[maxFit2].fitness) {
                maxFit2 = i;
            }
        }
        return individuals[maxFit2];
    }

    //Get index of least fittest individual
    public int getLeastFittestIndex() {
        int minFitVal = Integer.MAX_VALUE;
        int minFitIndex = 0;
        for (int i = 0; i < individuals.length; i++) {
            if (minFitVal >= individuals[i].fitness) {
                minFitVal = individuals[i].fitness;
                minFitIndex = i;
            }
        }
        return minFitIndex;
    }

    //Calculate fitness of each individual
    public void calculateFitness() {

        for (int i = 0; i < individuals.length; i++) {
            individuals[i].calcFitness();
        }
        getFittest();
    }

```



```
Generation: 19 Fittest: 4
Generation: 20 Fittest: 2
Generation: 21 Fittest: 2
Generation: 22 Fittest: 2
Generation: 23 Fittest: 1
Generation: 24 Fittest: 1
Generation: 25 Fittest: 1
Generation: 26 Fittest: 3
Generation: 27 Fittest: 3
Generation: 28 Fittest: 3
Generation: 29 Fittest: 1
Generation: 30 Fittest: 1
Generation: 31 Fittest: 3
Generation: 32 Fittest: 5

Solution found in generation 32
Fitness: 5
Genes: 11111

C:\Users\User>_
```

Sample output where the fittest solution is found in the 32nd generation

Edit

Check out this awesome implementation of genetic algorithms with visualizations of the gene pool in each generation at **https://github.com/memento/GeneticAlgorithm** by mem ento.

Thank you very much mem ento for sharing this repo with me and letting me add the link to the article.

Genetic Algorithm Machine Learning Evolutionary Algorithms Data Science Computer Science