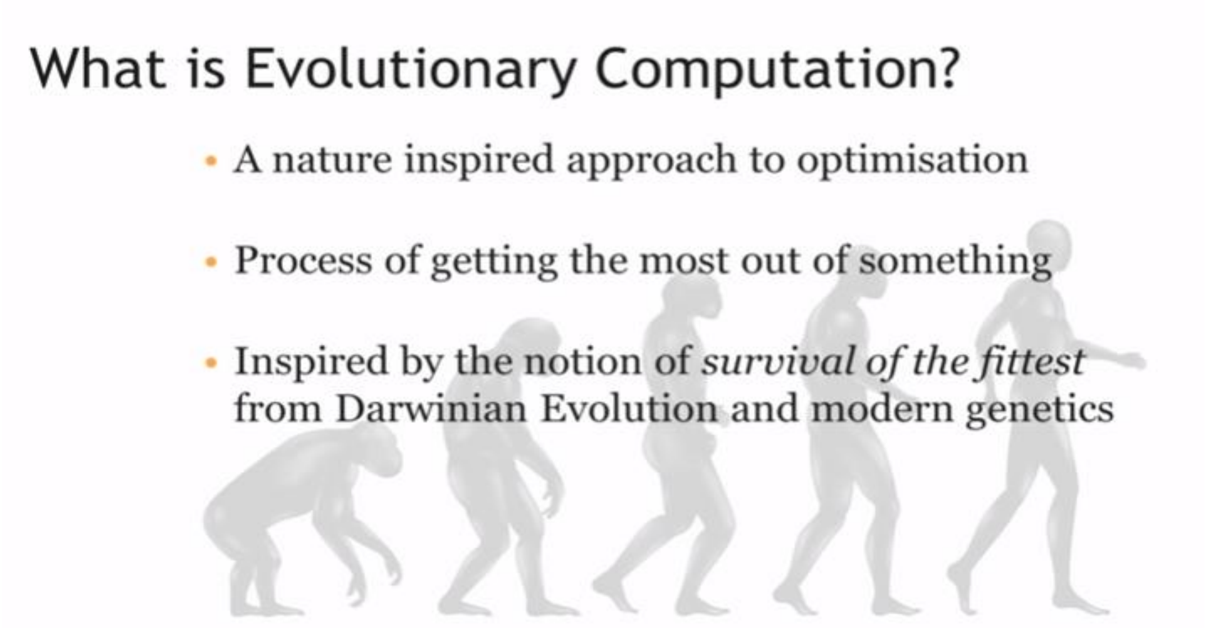
National University of Computer and Emerging Sciences CL 461 Artificial Intelligence

Lab Manual 08



What is Evolutionary Computation?

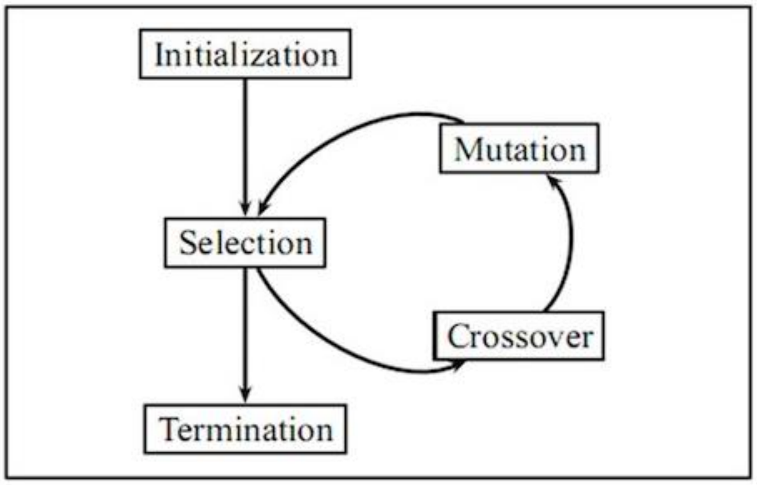
- A nature inspired approach to optimisation
- Process of getting the most out of something
- Inspired by the notion of *survival of the fittest* from Darwinian Evolution and modern genetics

Evolutionary algorithm:

Evolutionary algorithms are a heuristic-based approach to solving problems that cannot be easily solved in polynomial time, such as classically NP-Hard problems, and anything else that would take far too long to exhaustively process.

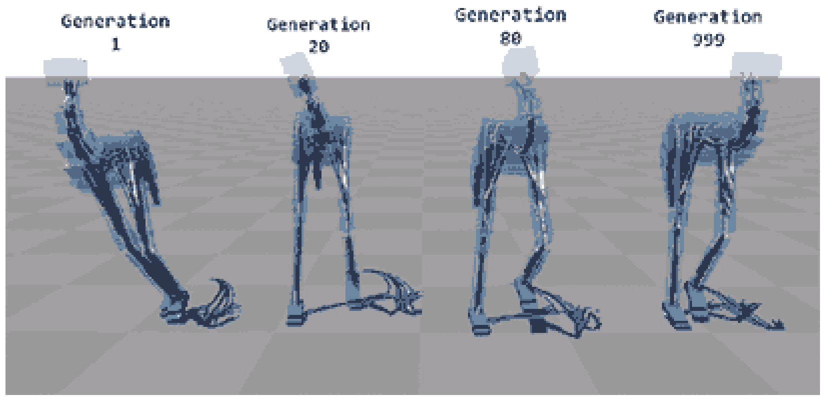
An EA contains four overall steps: initialization, selection, genetic operators, and termination. These steps each correspond, roughly, to a particular facet of natural selection, and provide easy ways to modularize implementations of this algorithm category.

Simply put, in an EA, fitter members will survive and proliferate, while unfit members will die off and not contribute to the gene pool of further generations, much like in natural selection.



Example:

Now, just to illustrate the result of this process I will show an example of an EA in action. The following gif shows several generations of dinosaurs learning to walk by optimizing their body structure and applied muscular forces. From left to right the generation increases, so the further right, the more optimized the walking process is. Despite the fact that the early generation dinosaurs were unable to walk, the EA was able to evolve the dinosaurs over time through mutation and crossover into a form that was able to walk.



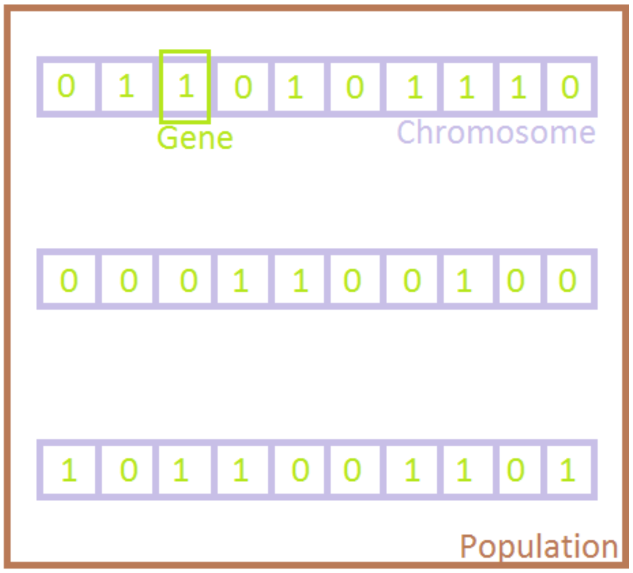
Genetic Algorithm:

A Genetic Algorithm (GA) is a meta-heuristic inspired by natural selection and is a part of the class of Evolutionary Algorithms (EA). We use these to generate high-quality solutions to optimization and search problems, for which, these use bio-inspired operators like mutation, crossover, and selection. In other words, using these, we hope to achieve optimal or near-optimal solutions to difficult problems.

This algorithms work in four steps:

1. Individuals in population compete for resources, mate.
2. Fittest individuals mate to create more off-springs than others.
3. Fittest parent propagates genes through generation; parents may produce off-springs better than either parent.
4. Each successive generation evolves to suit its ambience.

In optimization, we try to find within this search space the point or set of points that gives us the optimal solution. Each individual is like a string of characters/integers/floats and the strings are like chromosomes.



Phase in Genetic Algorithms:

Five phases are considered in a genetic algorithm.

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

Example Of GA:

1. Initialization & Fitness

We toss a fair coin 10 times and get the following initial population:

s1 = 1111010101	f (s1) = 7
s2 = 0111000101	f (s2) = 5
s3 = 1110110101	f (s3) = 7
s4 = 0100010011	f (s4) = 4
s5 = 1110111101	f (s5) = 8
s6 = 0100110000	f (s6) = 3

2. Selection:

We randomly (using a biased coin) select a subset of the individuals based on their fitness.

Individual *i* will have a probability to be chosen $\frac{f(i)}{\sum_i f(i)}$

Suppose that, after performing selection, we get the following population:

s1 ` = 1111010101 (s1)

s2 ` = 1110110101 (s3)

s3 ` = 1110111101 (s5)

s4 ` = 0111000101 (s2)

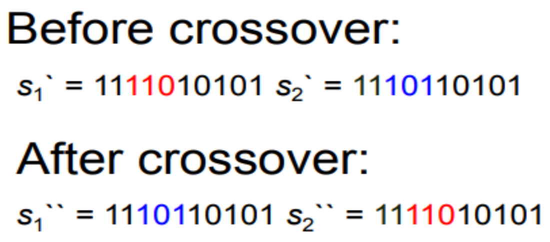
s5 ` = 0100010011 (s4)

s6 ` = 0100110000 (s6)

You can analyze here the fi values 7,7,8 for respective populations have the highest and nearest fitness function calculations we select the starting two pairs **S1' and S2'** for crossover process according to their fitness calculation.

3. Crossover:

Next we mate strings for crossover. For each couple we first decide whether to actually perform the crossover or not. If we decide to actually perform crossover, we randomly extract the crossover points, for instance from point 2 and 5.



Before crossover:

s_1` = 1111010101 $s_2$` = 1110110101

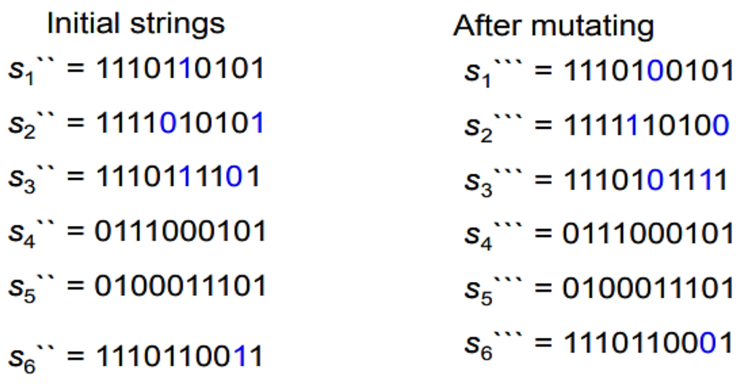
After crossover:

s_1`` = 1110110101 $s_2$`` = 1111010101

4. Mutation:

The final step is to apply random mutations: for each bit that we are to copy to the new population we allow a small probability of error (for instance 0.1).

Here, we also perform the crossover between the remaining pairs at a certain instance points.



Initial strings	After mutating
s_1`` = 1110110101	s_1``` = 1110100101
s_2`` = 1111010101	s_2``` = 1111110100
s_3`` = 1110111101	s_3``` = 1110101111
s_4`` = 0111000101	s_4``` = 0111000101
s_5`` = 0100011101	s_5``` = 0100011101
s_6`` = 1110110011	s_6``` = 1110110001

In one generation, the total population fitness changed from 34 to 37, thus improved by ~9%. At this point, we go through the same process all over again, until a stopping criterion is met.

Benefits of Genetic Algorithms:

- Concept is easy to understand.
- Modular, separate from application.
- Supports multi-objective optimization.
- Always an answer; answer gets better with time.
- Easy to exploit previous or alternate solutions.
- Flexible building blocks for hybrid applications.

GA Applications:

Domain	Application Type
Control	Gas pipeline, missile evasion
Design	Aircraft design, keyboard configuration, communication networks
Game playing	Poker, checkers
Security	Encryption and Decryption
Robotics	Trajectory planning

Limitations of Genetic Algorithms

- Not suitable for simple problems with available derivative information
- Stochastic; no guarantee of the result solution being optimal
- Frequent calculation of fitness value is computationally expensive for some problems
- No guarantee of convergence to the optimal solution if not implemented properly

Pseudo Code of Genetic Algorithm:

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

Lab Task

Task#1:

Given a target string, the goal is to produce target string starting from a random string of the same length. In the following implementation, following analogies are made:

- Characters A-Z, a-z, 0-9 and other special symbols are considered as genes
- A string generated by these character is considered as chromosome/solution/Individual
- Population size= 70
- Target string to be generated: TARGET = "Artificial Intelligence Lab"

Fitness score is the number of characters which differ from characters in target string at a particular index. So individual having lower fitness value is given more preference.

Task#2:

Implement genetic algorithm to the Traveling Salesman Problem.

We have a set of four cities A, B, C, and D. The distances between the cities are also given to us. Here (4-1)! That is 3! Route can be generated. The tour with A→B→C→D→A will be the optimal route for given problem.

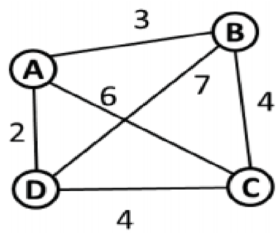


Fig. 1: Traveling salesman problem

Initial Population (set of solutions) = 6

Fitness (Quality of solution) = each solution is generally represented as a string of binary numbers, known as a **chromosome.** The most common fitness function for TSP is the length of the route. However, the 'shorter' the route is - the better.

Crossover point, exchange data after '1' instances of the list.

Mutation point perform between 2nd and 4th item.

The pseudo code for genetic algorithm for implementing the Traveling salesman problem :

```
GeneticAlgorithm ( )
{
    Initialize population of routes of cities randomly with a function Random ( )
    Evaluate the fitness of each individual route using function Fitness ( )
    While the fitness criteria is not satisfied
    do
    {
        Selection of two routes for reproduction using select function
        (Select (parent_route1, parent_route2))
        Perform crossover on the selected parent routes with crossover function
        (child_route = Crossover (parent_route1, parent_route2))
        Perform mutation on the newly generated child routes with mutation function
        (Mutation (child_route) )
        Evaluate the fitness of child_route and replace the parent population with
    child_route
    } }
```