Genetics Algorithm

Introduction

- Genetic Algorithm (GA) is a search-based optimization technique based on the principles of **Genetics and Natural Selection.**
- It is frequently used to find optimal or near-optimal solutions to difficult problems which otherwise would take a lifetime to solve.
- It is frequently used to solve optimization problems, in research, and in machine learning.

Introduction to Optimization

- Optimization is the process of **making something better**.
- In any process, we have a set of inputs and a set of outputs as shown in the following figure.

- Optimization refers to finding the values of inputs in such a way that we get the “best” output values.
- The definition of “best” varies from problem to problem, but in mathematical terms, it refers to maximizing or minimizing one or more objective functions, by varying the input parameters.
- The set of all possible solutions or values which the inputs can take make up the search space.
- In this search space, lies a point or a set of points which gives the optimal solution.
- The aim of optimization is to find that point or set of points in the search space.

What are Genetic Algorithms?

- Nature has always been a great source of inspiration to all mankind.
- Genetic Algorithms (GAs) are search based algorithms based on the concepts of natural selection and genetics.
- GAs are a subset of a much larger branch of computation known as **Evolutionary Computation**.
- GAs were developed by John Holland and his students and colleagues at the University of Michigan, most notably David E. Goldberg and has since been tried on various optimization problems with a high degree of success.
- In GAs, we have a **pool or a population of possible solutions** to the given problem.
- These solutions then undergo recombination and mutation (like in natural genetics), producing new children, and the process is repeated over various generations.
- Each individual (or candidate solution) is assigned a fitness value (based on its objective function value) and the fitter individuals are given a higher chance to mate and yield more “fitter” individuals.
- This is in line with the Darwinian Theory of “Survival of the Fittest”.

- In this way we keep “evolving” better individuals or solutions over generations, till we reach a stopping criterion.
- Genetic Algorithms are sufficiently randomized in nature, but they perform much better than random local search (in which we just try various random solutions, keeping track of the best so far), as they exploit historical information as well.

Advantages of GAs

- Does not require any derivative information (which may not be available for many real-world problems).
- Is faster and more efficient as compared to the traditional methods.
- Has very good parallel capabilities.
- Optimizes both continuous and discrete functions and also multi-objective problems.
- Provides a list of “good” solutions and not just a single solution.
- Always gets an answer to the problem, which gets better over the time.
- Useful when the search space is very large and there are a large number of parameters involved.

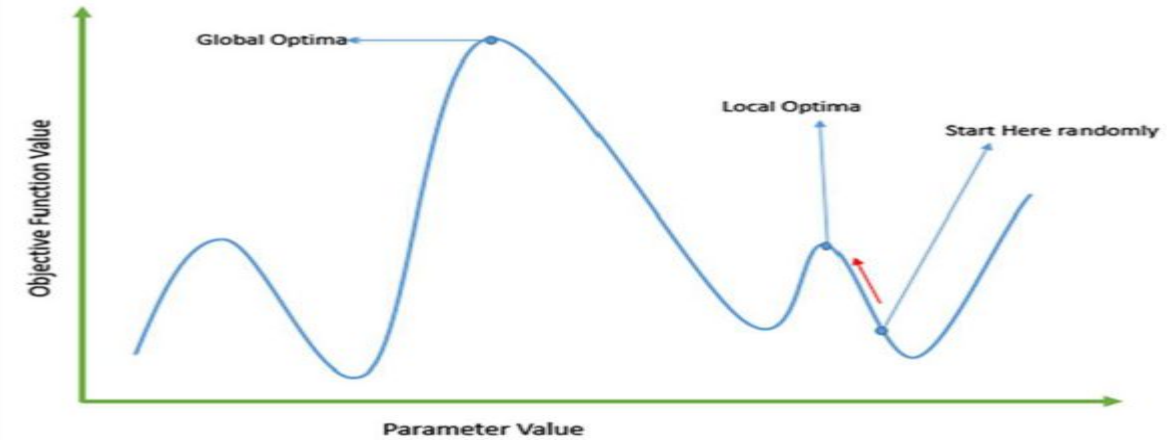
Limitations of GAs

- GAs are not suited for all problems, especially problems which are simple and for which derivative information is available.
- Fitness value is calculated repeatedly which might be computationally expensive for some problems.
- Being stochastic, there are no guarantees on the optimality or the quality of the solution.
- If not implemented properly, the GA may not converge to the optimal solution.

GA – Motivation

- Genetic Algorithms have the ability to deliver a “good-enough” solution “fast-enough”.
- This makes genetic algorithms attractive for use in solving optimization problems.
- The reasons why GAs are needed are as follows −
- **Solving Difficult Problems:**
  - In computer science, there is a large set of problems, which are **NP-Hard**.
  - What this essentially means is that, even the most powerful computing systems take a very long time (even years!) to solve that problem.
  - In such a scenario, GAs prove to be an efficient tool to provide **usable near-optimal solutions** in a short amount of time.

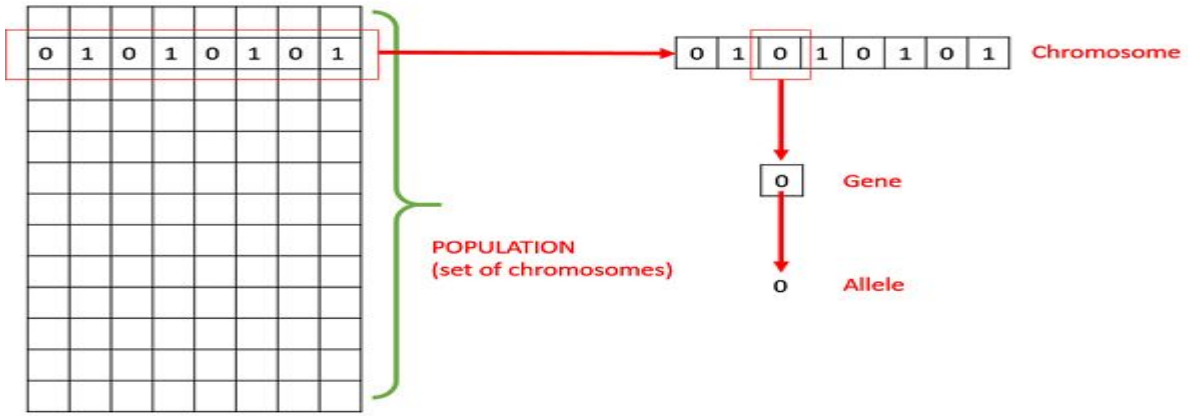
- **Failure of Gradient Based Methods:**
  - Traditional calculus based methods work by starting at a random point and by moving in the direction of the gradient, till we reach the top of the hill.
  - This technique is efficient and works very well for single-peaked objective functions like the cost function in linear regression.
  - But, in most real-world situations, we have a very complex problem called as landscapes, which are made of many peaks and many valleys, which causes such methods to fail, as they suffer from an inherent tendency of getting stuck at the local optima as shown in the following figure.



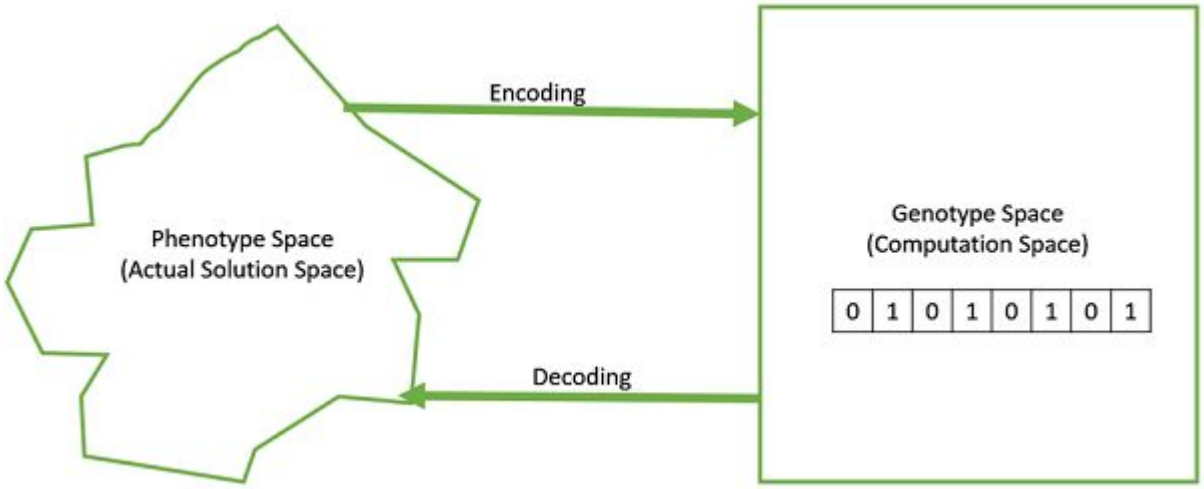
- **Getting a Good Solution Fast:**
  - Some difficult problems like the Travelling Salesperson Problem (TSP), have real-world applications like path finding and VLSI Design.
  - Now imagine that you are using your GPS Navigation system, and it takes a few minutes (or even a few hours) to compute the “optimal” path from the source to destination.
  - Delay in such real world applications is not acceptable and therefore a “good-enough” solution, which is delivered “fast” is what is required.

Basic Terminology

- **Population** –
  - It is a subset of all the possible (encoded) solutions to the given problem.
  - The population for a GA is analogous to the population for human beings except that instead of human beings, we have Candidate Solutions representing human beings.
- **Chromosomes** –
  - A chromosome is one such solution to the given problem.
- **Gene** –
  - A gene is one element position of a chromosome.
- **Allele** –
  - It is the value a gene takes for a particular chromosome.



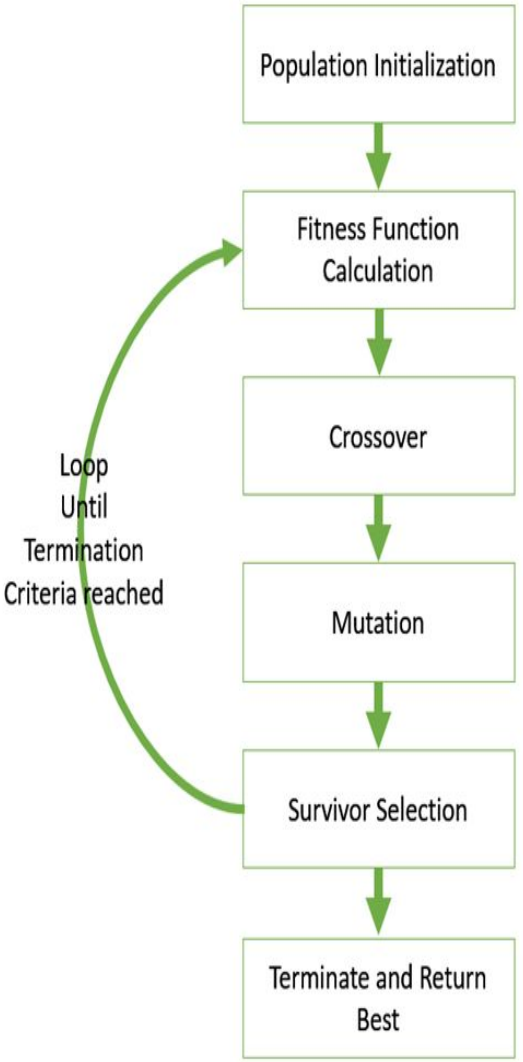
- **Genotype** –
  - Genotype is the population in the computation space.
  - In the computation space, the solutions are represented in a way which can be easily understood and manipulated using a computing system.
- **Phenotype** –
  - Phenotype is the population in the actual real world solution space in which solutions are represented in a way they are represented in real world situations.
- **Decoding and Encoding** –
  - For simple problems, the **phenotype and genotype** spaces are the same.
  - However, in most of the cases, the phenotype and genotype spaces are different. Decoding is a process of transforming a solution from the genotype to the phenotype space, while encoding is a process of transforming from the phenotype to genotype space.
  - Decoding should be fast as it is carried out repeatedly in a GA during the fitness value calculation.
  - For example, consider the 0/1 Knapsack Problem. The Phenotype space consists of solutions which just contain the item numbers of the items to be picked.
  - However, in the genotype space it can be represented as a binary string of length n (where n is the number of items).
  - A **0 at position x** represents that **x^{th}** item is picked while a 1 represents the reverse.
  - This is a case where genotype and phenotype spaces are different.



- **Fitness Function** −
  - A fitness function simply defined is a function which takes the solution as input and produces the suitability of the solution as the output.
  - In some cases, the fitness function and the objective function may be the same, while in others it might be different based on the problem.
- **Genetic Operators** −
  - These alter the genetic composition of the offspring. These include crossover, mutation, selection, etc.

Basic Structure

- We start with an initial population (which may be generated at random or seeded by other heuristics), select parents from this population for mating.
- Apply crossover and mutation operators on the parents to generate new off-springs.
- And finally these off-springs replace the existing individuals in the population and the process repeats.
- In this way genetic algorithms actually try to mimic the human evolution to some extent.

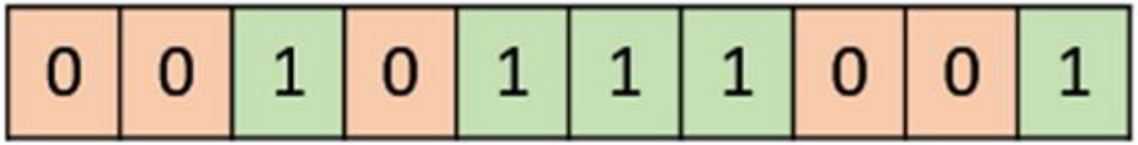


Genotype Representation

- One of the most important decisions to make while implementing a genetic algorithm is deciding the representation that we will use to represent our solutions.
- It has been observed that improper representation can lead to poor performance of the GA.
- Therefore, choosing a proper representation, having a proper definition of the mappings between the phenotype and genotype spaces is essential for the success of a GA.
- In this section, we present some of the most commonly used representations for genetic algorithms.
- However, representation is highly problem specific and the reader might find that another representation or a mix of the representations mentioned here might suit his/her problem better.

Binary Representation

- This is one of the simplest and most widely used representation in GAs.
- In this type of representation the genotype consists of bit strings.
- For some problems when the solution space consists of Boolean decision variables – yes or no, the binary representation is natural.
- Take for example the 0/1 Knapsack Problem.
- If there are n items, we can represent a solution by a binary string of n elements, where the x^{th} element tells whether the item x is picked (1) or not (0).



- For other problems, specifically those dealing with numbers, we can represent the numbers with their binary representation.
- The problem with this kind of encoding is that different bits have different significance and therefore mutation and crossover operators can have undesired consequences.
- This can be resolved to some extent by using **Gray Coding,** as a change in one bit does not have a massive effect on the solution.

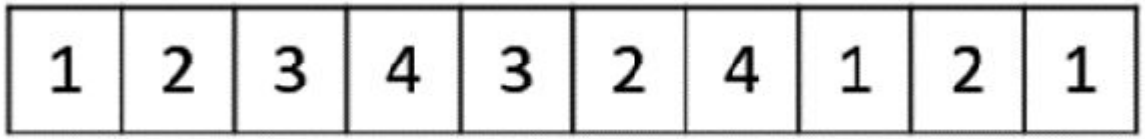
Real Valued Representation

- For problems where we want to define the genes using continuous rather than discrete variables, the real valued representation is the most natural.
- The precision of these real valued or floating point numbers is however limited to the computer.
-

0.5	0.2	0.6	0.8	0.7	0.4	0.3	0.2	0.1	0.9

Integer Representation

- For discrete valued genes, we cannot always limit the solution space to binary ‘yes’ or ‘no’.
- For example, if we want to encode the four distances – North, South, East and West, we can encode them as **{0,1,2,3}**.
- In such cases, integer representation is desirable.



Permutation Representation

- In many problems, the solution is represented by an order of elements.
- In such cases permutation representation is the most suited.
- A classic example of this representation is the travelling salesman problem (TSP).
- In this the salesman has to take a tour of all the cities, visiting each city exactly once and come back to the starting city.
- The total distance of the tour has to be minimized.
- The solution to this TSP is naturally an ordering or permutation of all the cities and therefore using a permutation representation makes sense for this problem.

Genetic Algorithms - Population

- Population is a subset of solutions in the current generation. It can also be defined as a set of chromosomes.
- There are several things to be kept in mind when dealing with GA population −
  - The diversity of the population should be maintained otherwise it might lead to premature convergence.
  - The population size should not be kept very large as it can cause a GA to slow down, while a smaller population might not be enough for a good mating pool. Therefore, an optimal population size needs to be decided by trial and error.
  - The population is usually defined as a two dimensional array of – **size population, size x, chromosome size**.

Population Initialization

- There are two primary methods to initialize a population in a GA. They are −
  - **Random Initialization** − Populate the initial population with completely random solutions.
  - **Heuristic initialization** − Populate the initial population using a known heuristic for the problem.
- It has been observed that the entire population should not be initialized using a heuristic, as it can result in the population having similar solutions and very little diversity.
- It has been experimentally observed that the random solutions are the ones to drive the population to optimality.
- Therefore, with heuristic initialization, we just seed the population with a couple of good solutions, filling up the rest with random solutions rather than filling the entire population with heuristic based solutions.
- It has also been observed that heuristic initialization in some cases, only effects the initial fitness of the population, but in the end, it is the diversity of the solutions which lead to optimality.

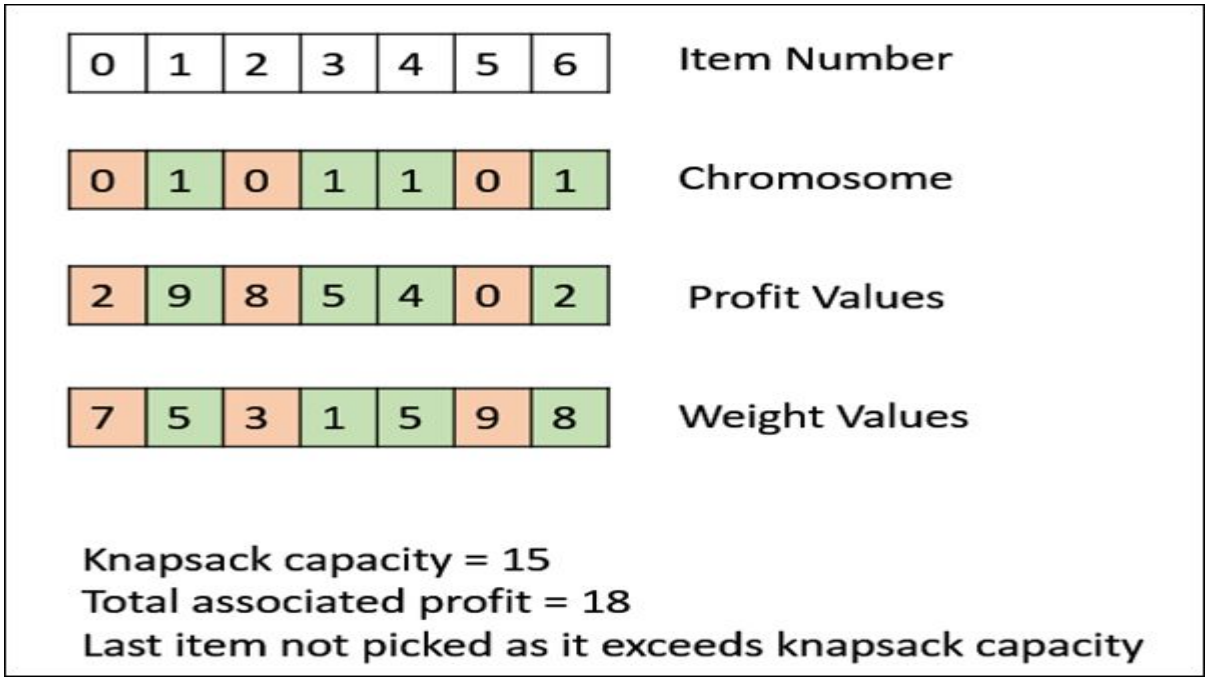
Population Models

- There are two population models widely in use −
- **Steady State:**
  - In steady state GA, we generate one or two off-springs in each iteration and they replace one or two individuals from the population. A steady state GA is also known as **Incremental GA**.
- Generational:
  - In a generational model, we generate ‘n’ off-springs, where n is the population size, and the entire population is replaced by the new one at the end of the iteration.

Genetic Algorithms - Fitness Function

- The fitness function simply defined is a function which takes a **candidate solution to the problem as input and produces as output** how "fit" our how "good" the solution is with respect to the problem in consideration.
- Calculation of fitness value is done repeatedly in a GA and therefore it should be sufficiently fast.
- A slow computation of the fitness value can adversely affect a GA and make it exceptionally slow.
- In most cases the fitness function and the objective function are the same as the objective is to either maximize or minimize the given objective function.
- However, for more complex problems with multiple objectives and constraints, an **Algorithm Designer** might choose to have a different fitness function.
- A fitness function should possess the following characteristics −
  - The fitness function should be sufficiently fast to compute.
  - It must quantitatively measure how fit a given solution is or how fit individuals can be produced from the given solution.

- In some cases, calculating the fitness function directly might not be possible due to the inherent complexities of the problem at hand.
- In such cases, we do fitness approximation to suit our needs.
- The following image shows the fitness calculation for a solution of the 0/1 Knapsack.
- It is a simple fitness function which just sums the profit values of the items being picked (which have a 1), scanning the elements from left to right till the knapsack is full.



Genetic Algorithms - Parent Selection

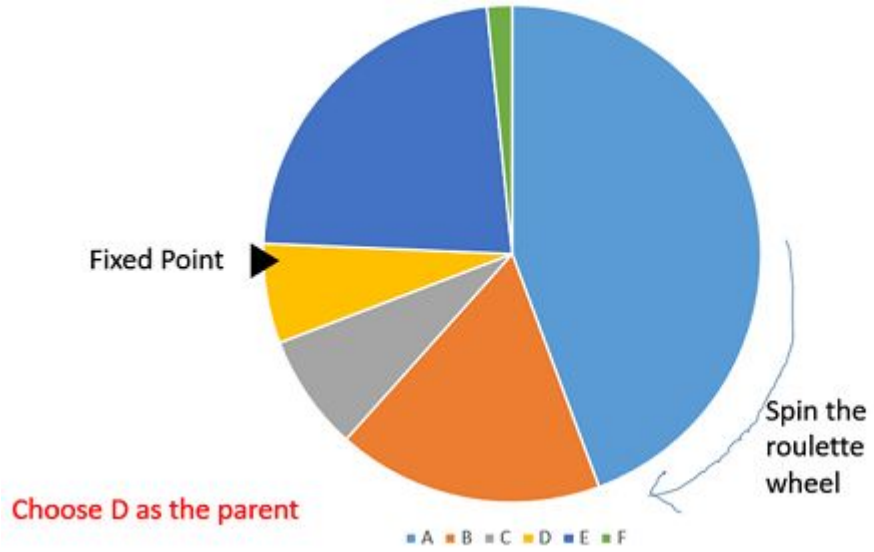
- Parent Selection is the process of selecting parents which mate and recombine to create off-springs for the next generation.
- Parent selection is very crucial to the convergence rate of the GA as good parents drive individuals to a better and fitter solutions.
- However, care should be taken to prevent one extremely fit solution from taking over the entire population in a few generations, as this leads to the solutions being close to one another in the solution space thereby leading to a loss of diversity.
- **Maintaining good diversity** in the population is extremely crucial for the success of a GA.
- This taking up of the entire population by one extremely fit solution is known as **premature convergence** and is an undesirable condition in a GA.

Fitness Proportionate Selection

- Fitness Proportionate Selection is one of the most popular ways of parent selection.
- In this every individual can become a parent with a probability which is proportional to its fitness.
- Therefore, fitter individuals have a higher chance of mating and propagating their features to the next generation.
- Therefore, such a selection strategy applies a selection pressure to the more fit individuals in the population, evolving better individuals over time.
- Consider a circular wheel. The wheel is divided into **n pies**, where n is the number of individuals in the population.
- Each individual gets a portion of the circle which is proportional to its fitness value.
- Two implementations of fitness proportionate selection are possible −
  - Roulette Wheel Selection
  - Stochastic Universal Sampling (SUS)

Roulette Wheel Selection

- In a roulette wheel selection, the circular wheel is divided as described before.
- A fixed point is chosen on the wheel circumference as shown and the wheel is rotated.
- The region of the wheel which comes in front of the fixed point is chosen as the parent.
- For the second parent, the same process is repeated.
-

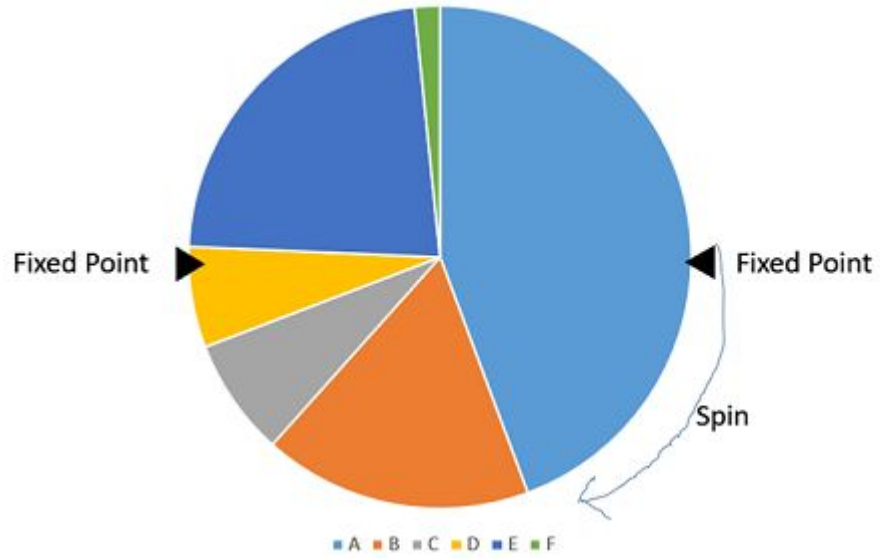


Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

- It is clear that a fitter individual has a greater pie on the wheel and therefore a greater chance of landing in front of the fixed point when the wheel is rotated.
- Therefore, the probability of choosing an individual depends directly on its fitness.
- Implementation wise, we use the following steps −
  - Calculate S = the sum of a finesses.
  - Generate a random number between 0 and S.
  - Starting from the top of the population, keep adding the finesses to the partial sum P, till P<S.
  - The individual for which P exceeds S is the chosen individual.

Stochastic Universal Sampling (SUS)

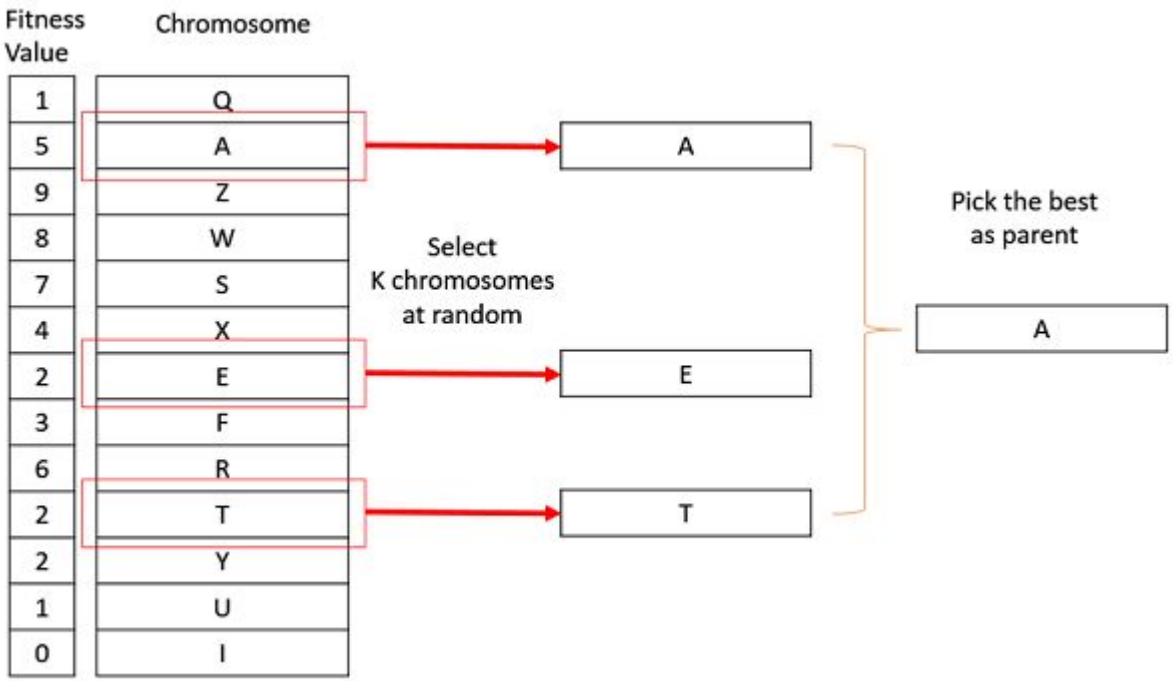
- Stochastic Universal Sampling is quite similar to Roulette wheel selection, however instead of having just one fixed point, we have multiple fixed points as shown in the following image.
- Therefore, all the parents are chosen in just one spin of the wheel. Also, such a setup encourages the highly fit individuals to be chosen at least once.
- It is to be noted that fitness proportionate selection methods don't work for cases where the fitness can take a negative value.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

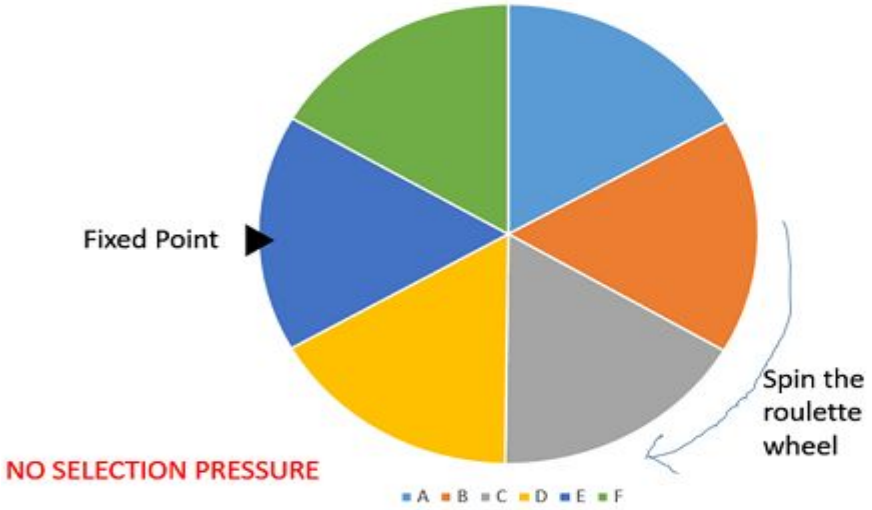
Tournament Selection

- In K-Way tournament selection, we select K individuals from the population at random and select the best out of these to become a parent.
- The same process is repeated for selecting the next parent.
- Tournament Selection is also extremely popular in literature as it can even work with negative fitness values.



Rank Selection

- Rank Selection also works with negative fitness values and is mostly used when the individuals in the population have very close fitness values (this happens usually at the end of the run).
- This leads to each individual having an almost equal share of the pie (like in case of fitness proportionate selection) as shown in the following image and hence each individual no matter how fit relative to each other has an approximately same probability of getting selected as a parent.
- This in turn leads to a loss in the selection pressure towards fitter individuals, making the GA to make poor parent selections in such situations.



Chromosome	Fitness Value
A	8.1
B	8.0
C	8.05
D	7.95
E	8.02
F	7.99

- In this, we remove the concept of a fitness value while selecting a parent.
- However, every individual in the population is ranked according to their fitness.
- The selection of the parents depends on the rank of each individual and not the fitness.
- The higher ranked individuals are preferred more than the lower ranked ones.

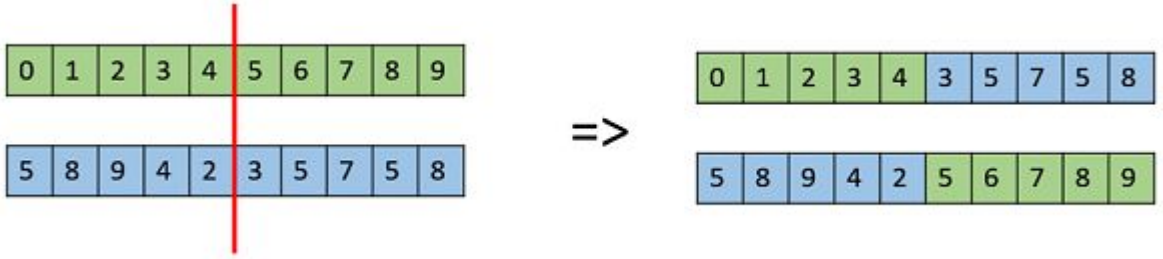
Chromosome	Fitness Value	Rank
A	8.1	1
B	8.0	4
C	8.05	2
D	7.95	6
E	8.02	3
F	7.99	5

Random Selection

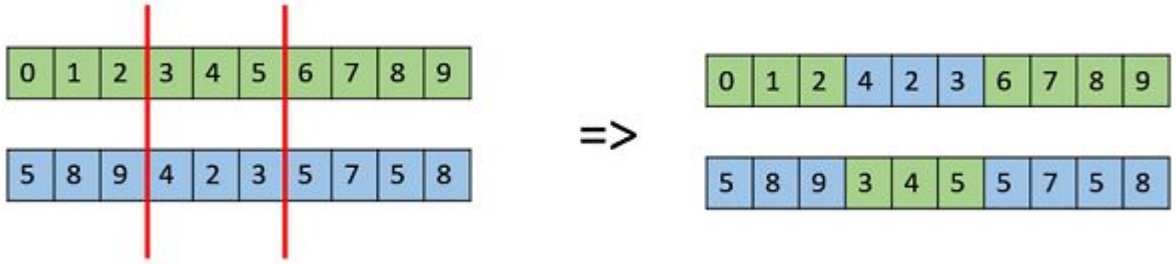
- In this strategy we randomly select parents from the existing population.
- There is no selection pressure towards fitter individuals and therefore this strategy is usually avoided.

Genetic Algorithms - Crossover

- The crossover operator is analogous to reproduction and biological crossover.
- In this more than one parent is selected and one or more off-springs are produced using the genetic material of the parents.
- Crossover is usually applied in a GA with a high probability – p_c
- **Crossover Operators:**
- One Point Crossover
  - In this one-point crossover, a random crossover point is selected and the tails of its two parents are swapped to get new off-springs.
-

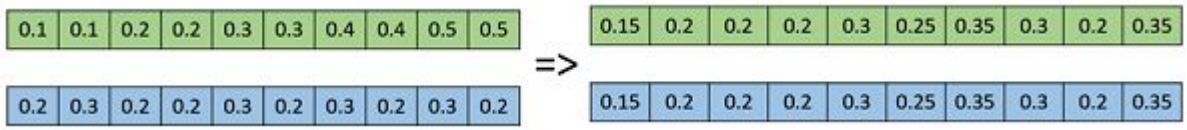


- **Multi Point Crossover:**
  - Multi point crossover is a generalization of the one-point crossover wherein alternating segments are swapped to get new off-springs.

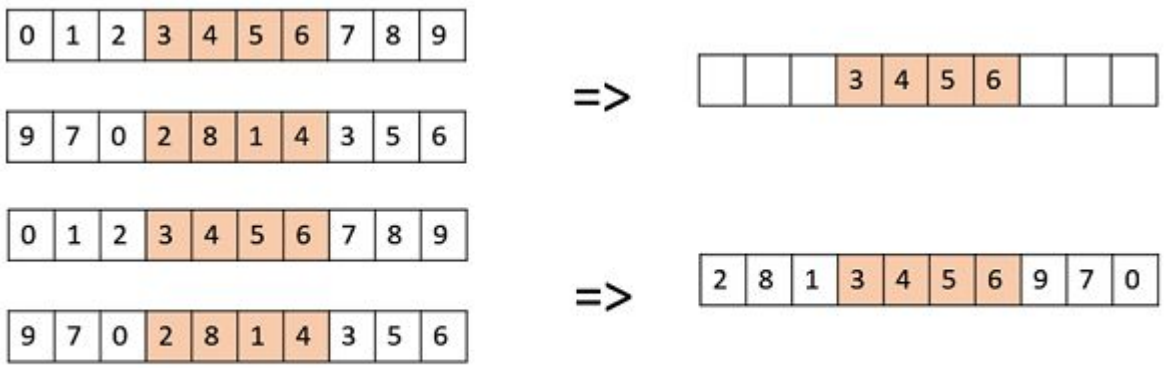


- **Uniform Crossover:**
  - In a uniform crossover, we don't divide the chromosome into segments, rather we treat each gene separately.
  - In this, we essentially flip a coin for each chromosome to decide whether or not it'll be included in the off-spring.
  - We can also bias the coin to one parent, to have more genetic material in the child from that parent.

- **Whole Arithmetic Recombination**
  - This is commonly used for integer representations and works by taking the weighted average of the two parents by using the following formulae −
    - Child1 = α.x + (1-α).y
    - Child2 = α.x + (1-α).y
  - Obviously, if α = 0.5, then both the children will be identical as shown in the following image.



- **Davis' Order Crossover (OX1)"**
  - OX1 is used for permutation based crossovers with the intention of transmitting information about relative ordering to the off-springs. It works as follows −
    - Create two random crossover points in the parent and copy the segment between them from the first parent to the first offspring.
    - Now, starting from the second crossover point in the second parent, copy the remaining unused numbers from the second parent to the first child, wrapping around the list.
    - Repeat for the second child with the parent's role reversed.



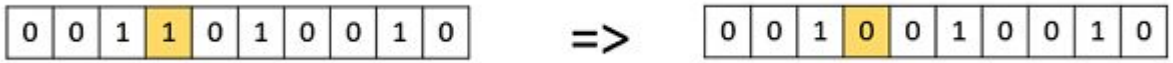
Repeat the same procedure to get the second child

Genetic Algorithms - Mutation

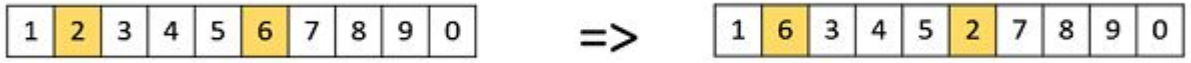
- In simple terms, mutation may be defined as a small random tweak in the chromosome, to get a new solution.
- It is used to maintain and introduce diversity in the genetic population and is usually applied with a low probability – $\boldsymbol{p_m}$.
- If the probability is very high, the GA gets reduced to a random search.
- Mutation is the part of the GA which is related to the “exploration” of the search space.
- It has been observed that mutation is essential to the convergence of the GA while crossover is not.

Mutation Operators

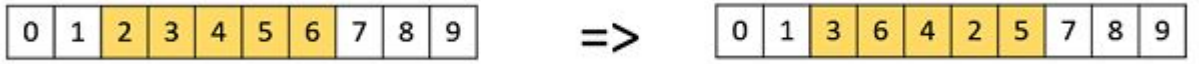
- **Bit Flip Mutation**
  - In this bit flip mutation, we select one or more random bits and flip them. This is used for binary encoded GAs.



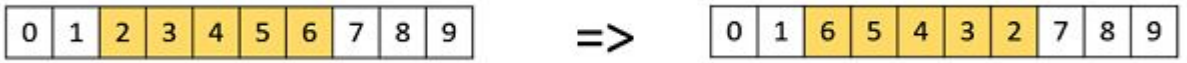
- **Random Resetting**
  - Random Resetting is an extension of the bit flip for the integer representation.
  - In this, a random value from the set of permissible values is assigned to a randomly chosen gene.
- **Swap Mutation**
  - In swap mutation, we select two positions on the chromosome at random, and interchange the values. This is common in permutation based encodings.



- **Scramble Mutation**
  - Scramble mutation is also popular with permutation representations. In this, from the entire chromosome, a subset of genes is chosen and their values are scrambled or shuffled randomly.



- **Inversion Mutation**
  - In inversion mutation, we select a subset of genes like in scramble mutation, but instead of shuffling the subset, we merely invert the entire string in the subset.

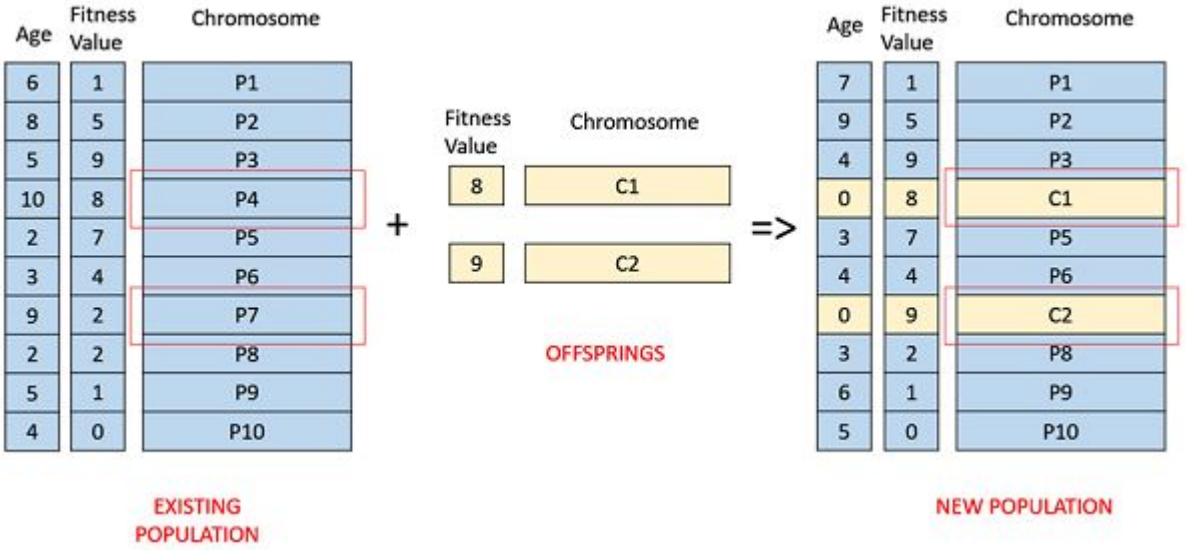


Genetic Algorithms - Survivor Selection

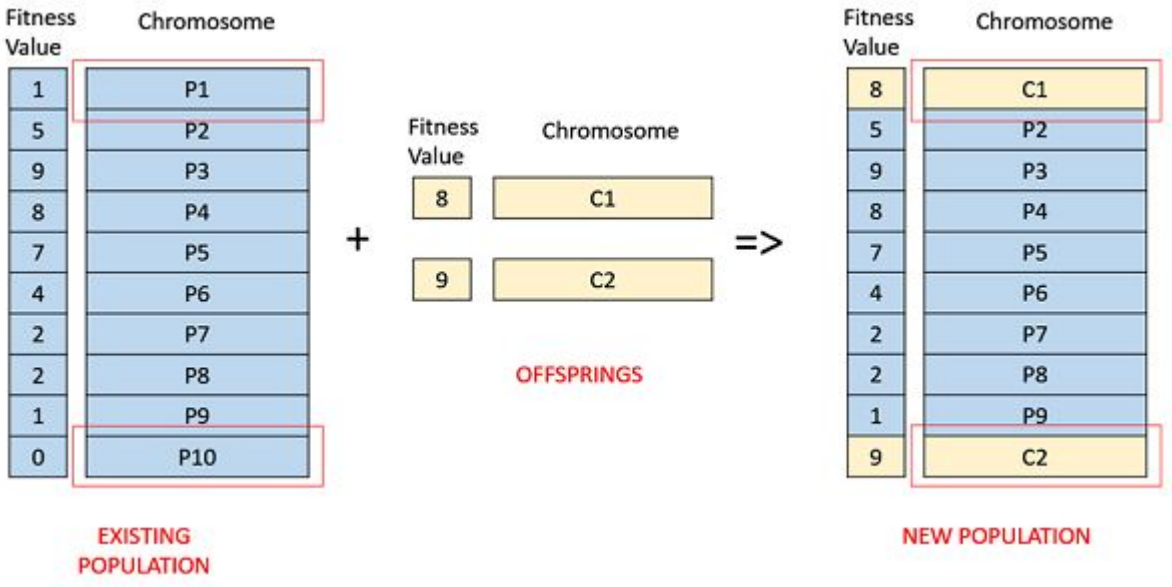
- The Survivor Selection Policy determines which individuals are to be kicked out and which are to be kept in the next generation.
- It is crucial as it should ensure that the fitter individuals are not kicked out of the population, while at the same time diversity should be maintained in the population.
- Some GAs employ **Elitism**. In simple terms, it means the current fittest member of the population is always propagated to the next generation. Therefore, under no circumstance can the fittest member of the current population be replaced.
- The easiest policy is to kick random members out of the population, but such an approach frequently has convergence issues, therefore the following strategies are widely used.

Age Based Selection

- In Age-Based Selection, we don't have a notion of a fitness.
- It is based on the premise that each individual is allowed in the population for a finite generation where it is allowed to reproduce, after that, it is kicked out of the population no matter how good its fitness is.
- For instance, in the following example, the age is the number of generations for which the individual has been in the population.
- The oldest members of the population i.e. P4 and P7 are kicked out of the population and the ages of the rest of the members are incremented by one.



- **Fitness Based Selection**
  - In this fitness based selection, the children tend to replace the least fit individuals in the population.
  - The selection of the least fit individuals may be done using a variation of any of the selection policies described before – tournament selection, fitness proportionate selection, etc.
  - For example, in the following image, the children replace the least fit individuals P1 and P10 of the population.
  - It is to be noted that since P1 and P9 have the same fitness value, the decision to remove which individual from the population is arbitrary.



Genetic Algorithms - Termination Condition

- The termination condition of a Genetic Algorithm is important in determining when a GA run will end.
- It has been observed that initially, the GA progresses very fast with better solutions coming in every few iterations, but this tends to saturate in the later stages where the improvements are very small.
- We usually want a termination condition such that our solution is close to the optimal, at the end of the run.
- Usually, we keep one of the following termination conditions −
  - When there has been no improvement in the population for X iterations.
  - When we reach an absolute number of generations.
  - When the objective function value has reached a certain pre-defined value.

- For example, in a genetic algorithm we keep a counter which keeps track of the generations for which there has been no improvement in the population. Initially, we set this counter to zero. Each time we don't generate off-springs which are better than the individuals in the population, we increment the counter.
- However, if the fitness any of the off-springs is better, then we reset the counter to zero. The algorithm terminates when the counter reaches a predetermined value.
- Like other parameters of a GA, the termination condition is also highly problem specific and the GA designer should try out various options to see what suits his particular problem the best.

Genetic Algorithms - Application Areas

- **Optimization** − Genetic Algorithms are most commonly used in optimization problems wherein we have to maximize or minimize a given objective function value under a given set of constraints. The approach to solve Optimization problems has been highlighted throughout the tutorial.
- **Economics** − GAs are also used to characterize various economic models like the cobweb model, game theory equilibrium resolution, asset pricing, etc.
- **Neural Networks** − GAs are also used to train neural networks, particularly recurrent neural networks.
- **Parallelization** − GAs also have very good parallel capabilities, and prove to be very effective means in solving certain problems, and also provide a good area for research.
- **Image Processing** − GAs are used for various digital image processing (DIP) tasks as well like dense pixel matching.
- **Vehicle routing problems** − With multiple soft time windows, multiple depots and a heterogeneous fleet.
- **Scheduling applications** − GAs are used to solve various scheduling problems as well, particularly the time tabling problem.

- **Machine Learning** − as already discussed, genetics based machine learning (GBML) is a niche area in machine learning.
- **Robot Trajectory Generation** − GAs have been used to plan the path which a robot arm takes by moving from one point to another.
- **Parametric Design of Aircraft** − GAs have been used to design aircrafts by varying the parameters and evolving better solutions.
- **DNA Analysis** − GAs have been used to determine the structure of DNA using spectrometric data about the sample.
- **Multimodal Optimization** − GAs are obviously very good approaches for multimodal optimization in which we have to find multiple optimum solutions.
- **Traveling salesman problem and its applications** − GAs have been used to solve the TSP, which is a well-known combinatorial problem using novel crossover and packing strategies.

Numerical Example

- Here are examples of applications that use genetic algorithms to solve the problem of combination.
- Suppose there is equality a + 2b + 3c + 4d = 30, genetic algorithm will be used to find the value of a, b, c, and d that satisfy the above equation.
- First we should formulate objective function, for this problem the objective is minimizing the value of function f(x)

  where f(x) = ((a + 2b + 3c + 4d) - 30).
- Since there are four variables in the equation, namely a, b, c, and d, we can compose the chromosome as follow: To speed up the computation, we can restrict that the values of variables a, b, c, and d are integers between 0 and 30.

Step 1. Initialization

- For example we define the number of chromosomes in population are 6, then we generate random value of gene a, b, c, d for 6 chromosomes
- Chromosome[1] = [a;b;c;d] = [12;05;23;08]
- Chromosome[2] = [a;b;c;d] = [02;21;18;03]
- Chromosome[3] = [a;b;c;d] = [10;04;13;14]
- Chromosome[4] = [a;b;c;d] = [20;01;10;06]
- Chromosome[5] = [a;b;c;d] = [01;04;13;19]
- Chromosome[6] = [a;b;c;d] = [20;05;17;01]

Step 2. Evaluation

- We compute the objective function value for each chromosome produced in initialization step:
- F_obj[1] = Abs((12 + 2*05 + 3*23 + 4*08) - 30) = Abs((12 + 10 + 69 + 32) - 30) = Abs(123 - 30) = 93
- F_obj[2] = Abs((02 + 2*21 + 3*18 + 4*03) - 30) = Abs((02 + 42 + 54 + 12) - 30) = Abs(110 - 30) = 80
- F_obj[3] = Abs((10 + 2*04 + 3*13 + 4*14) - 30) = Abs((10 + 08 + 39 + 56) - 30) = Abs(113 - 30) = 83
- F_obj[4] = Abs((20 + 2*01 + 3*10 + 4*06) - 30) = Abs((20 + 02 + 30 + 24) - 30) = Abs(76 - 30) = 46
- F_obj[5] = Abs((01 + 2*04 + 3*13 + 4*19) - 30) = Abs((01 + 08 + 39 + 76) - 30) = Abs(124 - 30) = 94
- F_obj[6] = Abs((20 + 2*05 + 3*17 + 4*01) - 30) = Abs((20 + 10 + 51 + 04) - 30) = Abs(85 - 30) = 55

Step 3. Selection

- The fittest chromosomes have higher probability to be selected for the next generation.
- To compute fitness probability we must compute the fitness of each chromosome.
- To avoid divide by zero problem, the value of F_obj is added by 1.
- Fitness[1] = 1 / (1+F_obj[1]) = 1 / 94 = 0.0106
- Fitness[2] = 1 / (1+F_obj[2]) = 1 / 81 = 0.0123
- Fitness[3] = 1 / (1+F_obj[3]) = 1 / 84 = 0.0119
- Fitness[4] = 1 / (1+F_obj[4]) = 1 / 47 = 0.0213
- Fitness[5] = 1 / (1+F_obj[5]) = 1 / 95 = 0.0105
- Fitness[6] = 1 / (1+F_obj[6]) = 1 / 56 = 0.0179
- Total = 0.0106 + 0.0123 + 0.0119 + 0.0213 + 0.0105 + 0.0179 = 0.0845

- The probability for each chromosomes is formulated by:
  - P[i] = Fitness[i] / Total
- P[1] = 0.0106 / 0.0845 = 0.1254
- P[2] = 0.0123 / 0.0845 = 0.1456
- P[3] = 0.0119 / 0.0845 = 0.1408
- P[4] = 0.0213 / 0.0845 = 0.2521
- P[5] = 0.0105 / 0.0845 = 0.1243
- P[6] = 0.0179 / 0.0845 = 0.2118

- From the probabilities above we can see that Chromosome 4 that has the highest fitness, this chromosome has highest probability to be selected for next generation chromosomes.

- For the selection process we use roulette wheel, for that we should compute the cumulative probability values:
- C[1] = 0.1254
- C[2] = 0.1254 + 0.1456 = 0.2710
- C[3] = 0.1254 + 0.1456 + 0.1408 = 0.4118
- C[4] = 0.1254 + 0.1456 + 0.1408 + 0.2521 = 0.6639
- C[5] = 0.1254 + 0.1456 + 0.1408 + 0.2521 + 0.1243 = 0.7882
- C[6] = 0.1254 + 0.1456 + 0.1408 + 0.2521 + 0.1243 + 0.2118 = 1.0

- Having calculated the cumulative probability of selection process using roulette-wheel can be done. The process is to generate random number R in the range 0-1 as follows.
- R[1] = 0.201
- R[2] = 0.284
- R[3] = 0.099
- R[4] = 0.822
- R[5] = 0.398
- R[6] = 0.501

- If random number R[1] is greater than C[1] and smaller than C[2] then select Chromosome[2] as a chromosome in the new population for next generation:
- NewChromosome[1] = Chromosome[2]
- NewChromosome[2] = Chromosome[3]
- NewChromosome[3] = Chromosome[1]
- NewChromosome[4] = Chromosome[6]
- NewChromosome[5] = Chromosome[3]
- NewChromosome[6] = Chromosome[4]

- Chromosomes in the population thus became:
- Chromosome[1] = [02;21;18;03]
- Chromosome[2] = [10;04;13;14]
- Chromosome[3] = [12;05;23;08]
- Chromosome[4] = [20;05;17;01]
- Chromosome[5] = [10;04;13;14]
- Chromosome[6] = [20;01;10;06]

Crossover

- In this example, we use one-cut point, i.e. randomly select a position in the parent chromosome then exchanging sub-chromosome.
- Parent chromosome which will mate is randomly selected and the number of mate Chromosomes is controlled using crossover_rate (ρc) parameters.
- Chromosome k will be selected as a parent if R[k]< ρc.
- Suppose we set that the crossover rate is 25%, then Chromosome number k will be selected for crossover if random generated value for Chromosome k below 0.25.
- The process is as follows:
- First we generate a random number R as the number of population.
- R[1] = 0.191
- R[2] = 0.259
- R[3] = 0.760
- R[4] = 0.006
- R[5] = 0.159
- R[6] = 0.340

- For random number R above, parents are Chromosome[1], Chromosome[4] and Chromosome[5] will be selected for crossover.
- Chromosome[1] >< Chromosome[4]
- Chromosome[4] >< Chromosome[5]
- Chromosome[5] >< Chromosome[1]
- After chromosome selection, the next process is determining the position of the crossover point.
- This is done by generating random numbers between 1 to (length of Chromosome – 1).
- In this case, generated random numbers should be between 1 and 3.
- After we get the crossover point, parents Chromosome will be cut at crossover point and its gens will be interchanged.
- For example we generated 3 random number and we get:
- C[1] = 1
- C[2] = 1
- C[3] = 2

- Then for first crossover, second crossover and third crossover, parent's gens will be cut at gen number 1, gen number 1 and gen number 3 respectively, e.g.
- Chromosome[1] = Chromosome[1] >< Chromosome[4]
- = [02;21;18;03] >< [20;05;17;01]
- = [02;05;17;01]
- Chromosome[4] = Chromosome[4] >< Chromosome[5]
- = [20;05;17;01] >< [10;04;13;14]
- = [20;04;13;14]
- Chromosome[5] = Chromosome[5] >< Chromosome[1]
- = [10;04;13;14] >< [02;21;18;03]
- = [10;04;13;03]

- Thus Chromosome population after experiencing a crossover process: Chromosome[1] = [02;05;17;01]
- Chromosome[2] = [10;04;13;14]
- Chromosome[3] = [12;05;23;08]
- Chromosome[4] = [20;04;13;14]
- Chromosome[5] = [10;04;13;03]
- Chromosome[6] = [20;01;10;06]

Step 5. Mutation

- Number of chromosomes that have mutations in a population is determined by the mutation_rate parameter.
- Mutation process is done by replacing the gen at random position with a new value.
- The process is as follows.
- First we must calculate the total length of gen in the population.
- In this case the total length of gen is
  - total_gen = number_of_gen_in_Chromosome * number of population
  - = 4 * 6 = 24
- Mutation process is done by generating a random integer between 1 and total_gen (1 to 24).
- If generated random number is smaller than mutation_rate(ρm) variable then marked the position of gen in chromosomes.
- Suppose we define ρm 10%, it is expected that 10% (0.1) of total_gen in the population that will be mutated:
- number of mutations = 0.1 * 24 = 2.4 ≈ 2

- Suppose generation of random number yield 12 and 18 then the chromosome which have mutation are Chromosome number 3 gen number 4 and Chromosome 5 gen number 2.
- The value of mutated gens at mutation point is replaced by random number between 0-30.
- Suppose generated random number are 2 and 5 then Chromosome composition after mutation are:
- Chromosome[1] = [02;05;17;01]
- Chromosome[2] = [10;04;13;14]
- Chromosome[3] = [12;05;23;02]
- Chromosome[4] = [20;04;13;14]
- Chromosome[5] = [10;05;13;03]
- Chromosome[6] = [20;01;10;06]

- Finishing mutation process then we have one iteration or one generation of the genetic algorithm.
- We can now evaluate the objective function after one generation:

- Chromosome[1] = [02;05;17;01] F_obj[1] = Abs((02 + 2*05 + 3*17 + 4*01) - 30) = Abs((2 + 10 + 51 + 4) - 30) = Abs(67 - 30) = 37
- Chromosome[2] = [10;04;13;14] F_obj[2] = Abs((10 + 2*04 + 3*13 + 4*14) - 30) = Abs((10 + 8 + 33 + 56) - 30) = Abs(107 - 30) = 77
- Chromosome[3] = [12;05;23;02] F_obj[3] = Abs((12 + 2*05 + 3*23 + 4*02) - 30) = Abs((12 + 10 + 69 + 8) - 30) = Abs(87 - 30) = 47
- Chromosome[4] = [20;04;13;14] F_obj[4] = Abs((20 + 2*04 + 3*13 + 4*14) - 30) = Abs((20 + 8 + 39 + 56) - 30) = Abs(123 - 30) = 93
- Chromosome[5] = [10;05;14;03] F_obj[5] = Abs((10 + 2*05 + 3*14 + 4*03) - 30) = Abs((10 + 10 + 42 + 12) - 30) = Abs(74 - 30) = 44
- Chromosome[6] = [20;01;10;06] F_obj[6] = Abs((20 + 2*01 + 3*10 + 4*06) - 30) = Abs((20 + 2 + 30 + 24) - 30) = Abs(76 - 30) = 46

- From the evaluation of new Chromosome we can see that the objective function is decreasing, this means that we have better Chromosome or solution compared with previous Chromosome generation.
- New Chromosomes for next iteration are:
- Chromosome[1] = [02;05;17;01]
- Chromosome[2] = [10;04;13;14]
- Chromosome[3] = [12;05;23;02]
- Chromosome[4] = [20;04;13;14]
- Chromosome[5] = [10;05;14;03]
- Chromosome[6] = [20;01;10;06]

- These new Chromosomes will undergo the same process as the previous generation of Chromosomes such as evaluation, selection, crossover and mutation and at the end it produce new generation of Chromosome for the next iteration.
- This process will be repeated until a predetermined number of generations.
- For this example, after running 50 generations, best chromosome is obtained:
- Chromosome = [07; 05; 03; 01]
- This means that: a = 7, b = 5, c = 3, d = 1

Practice problem: 0-1 Knapsack

- Suppose we have a knapsack that has a capacity of 13 cubic inches and several items of different sizes and different benefits.
- We want to include in the knapsack only these items that will have the greatest total benefit within the constraint of the knapsack's capacity.
- There are three potential items (labeled 'A,' 'B,' 'C'). Their volumes and benefits are as follows:

Item #	A	B	C
Benefit	4	3	5
Volume	6	7	8

We seek to maximize the total benefit:

$$\sum_{i=1}^{3} B_i X_i = 4XI + 3X_2 + 5X_3$$

Subject to the constraints:

$$\sum_{i=1}^{3} V_i X_i = 6X_I + 7X_2 + 8X_3 \leq 13$$

And

$$X_i \in \{0,1\}, \text{ for } i = 1, 2, \ldots, n.$$

For this problem there are 2^3 possible subsets of items:

A	*B*	*C*	*Volume of the set*	*Benefit of the set*
0	*0*	*0*	*0*	0
0	*0*	*1*	*8*	5
0	*1*	*0*	*7*	3
0	1	1	15	-
1	*0*	*0*	*6*	4
1	0	1	14	-
1	***1***	***0***	***13***	**7**
1	1	1	21	-

- In order to find the best solution we have to identify a subset that meets the constraint and has the maximum total benefit.
- Hence, the optimal benefit for the given constraint (V = 13) can only be obtained with one quantity of A, one quantity of B, and zero quantity of C, and it is 7.

NP problems and the 0-1 KP

- NP (non-deterministic polynomial) problems are ones for which there are no known algorithms that would guarantee to run in a polynomial time.
- However, it is possible to “guess” a solution and check it, both in polynomial time.
- Some of the most well-known NP problems are the traveling salesman, Hamilton circuit, bin packing, knapsack, and clique.
- GAs have shown to be well suited for high-quality solutions to larger NP problems and currently they are the most efficient ways for finding an approximately optimal solution for optimization problems.
- They do not involve extensive search algorithms and do not try to find the best solution, but they simply generate a candidate for a solution, check in polynomial time whether it is a solution or not and how good a solution it is.
- GAs do not always give the optimal solution, but a solution that is close enough to the optimal one.

Thanks