

genetic algorithm

Muneeb Ullah F2020376037



April 29, 2024

university of management and technology

SUBMITTED TO: Sir Amjad Khan

Contents

[Description 2](#_Toc165302200)

[Problem Statement 2](#_Toc165302201)

[Code Walkthrough 3](#_Toc165302202)

[Results: 7](#_Toc165302203)

[Analysis 8](#_Toc165302204)

[References 8](#_Toc165302205)

# Description

Genetic algorithms (gasoline) are optimization strategies inspired through the method of natural choice and genetics. They belong to the class of evolutionary algorithms and are extensively used to discover premiere solutions to complicated issues throughout various domains which includes engineering, economics, biology, and pc technology.

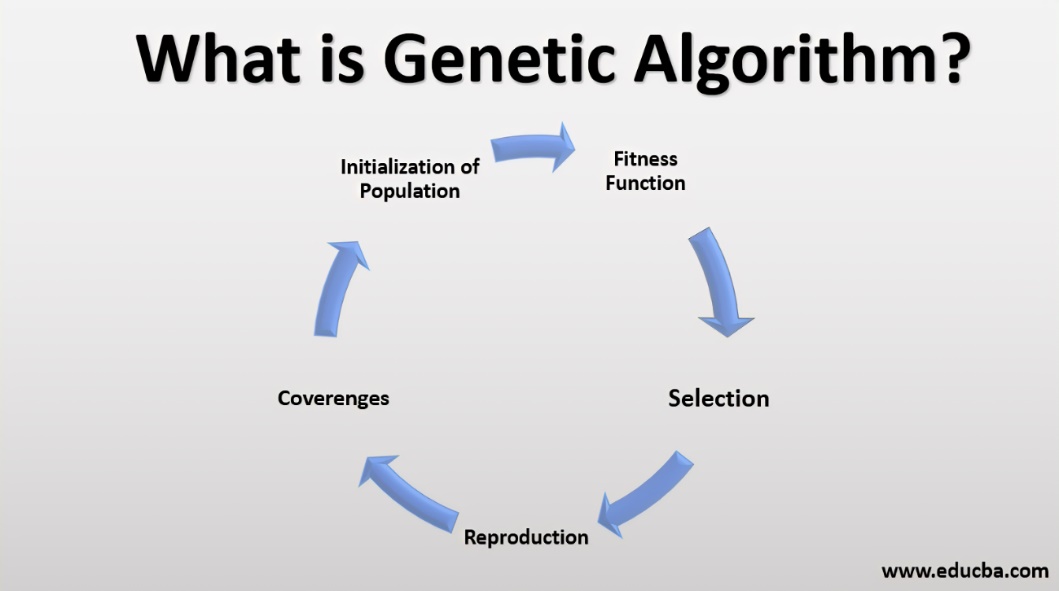
The primary idea in the back of genetic algorithms is to imitate the method of herbal choice, in which the fittest individuals are much more likely to continue to exist and skip on their genetic facts to the next generation. inside the context of optimization, answers to a trouble are represented as individuals in a population. these people are evaluated based totally on their health, which measures how close they may be to the choicest answer. through a technique of choice, crossover, and mutation, new generations of solutions are generated, regularly improving the general fitness of the population over successive iterations.

Genetic algorithms are characterised by way of their capacity to explore massive answer spaces efficaciously, making them especially appropriate for troubles with many possible answers and complicated search landscapes. they're also strong to noise and can cope with issues with non-linear and non-convex goal capabilities.

# Problem Statement

The problem addressed by way of genetic algorithms is that of finding an choicest or near-most advantageous way to a given optimization problem in which conventional optimization techniques may be impractical or useless because of the complexity of the hunt area or the character of the objective function.

Officially, given an optimization trouble with an objective function 𝑓(𝑥), where 𝑥 represents a candidate solution, the intention is to find an 𝑥\* such that 𝑓(𝑥\*) is minimized or maximized, difficulty to certain constraints. This problem may also involve a large seek area with multiple local optima, discontinuities, or constraints that make it hard to find the worldwide top-quality using conventional methods.



The genetic algorithm technique to solving this hassle includes the following steps:

1. Initialization: Generate a preliminary population of candidate answers randomly or the use of heuristics.
2. assessment: evaluate the fitness of every character within the populace based on the goal characteristic.
3. selection: select individuals from the modern populace to function parents for the subsequent generation, with chance proportional to their health.
4. Crossover: Generate offspring by combining genetic fabric from selected mother and father through crossover or recombination operations.
5. Mutation: Introduce random changes or mutations to the offspring to hold diversity in the population.
6. Replacement: update the current populace with the brand-new era of individuals.
7. Termination: Repeat steps 2-6 until a termination circumstance is met, which includes reaching a maximum wide variety of iterations or achieving a fine solution.

The genetic algorithm iteratively refines the populace, steadily improving the nice of solutions till a stopping criterion is glad, preferably converging toward the worldwide best or a quality solution inside the seek area.

# Code Walkthrough

This document provides an overview of each function in the provided Python code for a genetic algorithm implementation. It explains the functionality for each function and its brief description of implementation:

**1. generate\_genome(length: int) -> Genome**

def generate\_genome(length: int) -> Genome:

    return choices([0, 1], k=length)

This function generates a random genome of binary digits (0s and 1s) with the specified length.

**2. generate\_population(size: int, genome\_length: int) -> Population**

def generate\_population(size: int, genome\_length: int) -> Population:

    return [generate\_genome(genome\_length) for \_ in range(size)]

This function generates a population of genomes with the specified size, where each genome has the specified length.

**3. single\_point\_crossover(a: Genome, b: Genome) -> Tuple[Genome, Genome]**

def single\_point\_crossover(a: Genome, b: Genome) -> Tuple[Genome, Genome]:

    if len(a) != len(b):

        raise ValueError("Genomes a and b must be of same length")

    length = len(a)

    if length < 2:

        return a, b

    p = randint(1, length - 1)

    return a[0:p] + b[p:], b[0:p] + a[p:]

This function performs single-point crossover between two parent genomes **a** and **b**, producing two offspring genomes.

**4. mutation(genome: Genome, num: int = 1, probability: float = 0.5) -> Genome**

def mutation(genome: Genome, num: int = 1, probability: float = 0.5) -> Genome:

    for \_ in range(num):

        index = randrange(len(genome))

        genome[index] = genome[index] if random() > probability else abs(genome[index] - 1)

    return genome

This function introduces mutations into a genome by randomly flipping bits with a given probability.

**5. population\_fitness(population: Population, fitness\_func: FitnessFunc) -> int**

def population\_fitness(population: Population, fitness\_func: FitnessFunc) -> int:

    return sum([fitness\_func(genome) for genome in population])

This function calculates the total fitness of the entire population by summing the fitness values of all genomes in the population.

**6. selection\_pair(population: Population, fitness\_func: FitnessFunc) -> Population**

def selection\_pair(population: Population, fitness\_func: FitnessFunc) -> Population:

    return choices(

        population=population,

        weights=[fitness\_func(gene) for gene in population],

        k=2

    )

This function selects a pair of parent genomes from the population based on their fitness values using the roulette wheel selection method.

**7. sort\_population(population: Population, fitness\_func: FitnessFunc) -> Population**

def sort\_population(population: Population, fitness\_func: FitnessFunc) -> Population:

    return sorted(population, key=fitness\_func, reverse=True)

This function sorts the population based on the fitness values of the genomes in descending order.

**8. genome\_to\_string(genome: Genome) -> str**

def genome\_to\_string(genome: Genome) -> str:

    return "".join(map(str, genome))

This function converts a genome (list of integers) into a string representation for printing purposes.

**9. print\_stats(population: Population, generation\_id: int, fitness\_func: FitnessFunc)**

def print\_stats(population: Population, generation\_id: int, fitness\_func: FitnessFunc):

    print("GENERATION %02d" % generation\_id)

    print("=============")

    print("Population: [%s]" % ", ".join([genome\_to\_string(gene) for gene in population]))

    print("Avg. Fitness: %f" % (population\_fitness(population, fitness\_func) / len(population)))

    sorted\_population = sort\_population(population, fitness\_func)

    print(

        "Best: %s (%f)" % (genome\_to\_string(sorted\_population[0]), fitness\_func(sorted\_population[0])))

    print("Worst: %s (%f)" % (genome\_to\_string(sorted\_population[-1]),

                              fitness\_func(sorted\_population[-1])))

    print("")

    return sorted\_population[0]

This function prints statistics about the current population, including the average fitness, best and worst genomes, and the generation number.

**10. run\_evolution(...) -> Tuple[Population, int]**

def run\_evolution(

        populate\_func: PopulateFunc,

        fitness\_func: FitnessFunc,

        fitness\_limit: int,

        selection\_func: SelectionFunc = selection\_pair,

        crossover\_func: CrossoverFunc = single\_point\_crossover,

        mutation\_func: MutationFunc = mutation,

        generation\_limit: int = 100,

        printer: Optional[PrinterFunc] = None) \

        -> Tuple[Population, int]:

    population = populate\_func()

    for i in range(generation\_limit):

        population = sorted(population, key=lambda genome: fitness\_func(genome), reverse=True)

        if printer is not None:

            printer(population, i, fitness\_func)

        if fitness\_func(population[0]) >= fitness\_limit:

            break

        next\_generation = population[0:2]

        for j in range(int(len(population) / 2) - 1):

            parents = selection\_func(population, fitness\_func)

            offspring\_a, offspring\_b = crossover\_func(parents[0], parents[1])

            offspring\_a = mutation\_func(offspring\_a)

            offspring\_b = mutation\_func(offspring\_b)

            next\_generation += [offspring\_a, offspring\_b]

        population = next\_generation

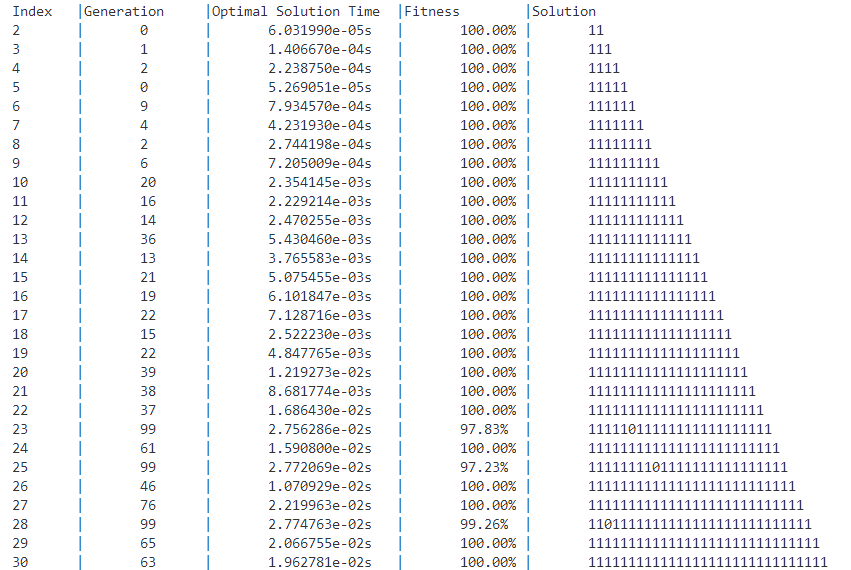
    return population, i

This function orchestrates the genetic algorithm evolution process. It iterates through generations, performing selection, crossover, and mutation to evolve the population towards an optimal solution. The function takes various parameters such as population initialization method, fitness function, fitness limit, selection method, crossover method, mutation method, and generation limit. It returns the final evolved population and the generation number at which the algorithm terminates.

**How Functions Work Together:**

1. The **run\_evolution** function initializes the population using the **populate\_func**.
2. It iterates through generations, performing selection, crossover, and mutation operations on the population.
3. At each generation, it evaluates the fitness of individuals using the **fitness\_func**.
4. The best individuals are selected as parents for the next generation using the **selection\_func**.
5. Crossover and mutation are applied to create offspring genomes.
6. The process continues until a termination condition (fitness limit or generation limit) is met.
7. Throughout the process, statistics are printed using the **print\_stats** function.

# Results:



Run the above code of Genetic Algorithm on Knapsack problem and achieved the above results. In the above table index represents the number of objects being used for the optimization problem. Here is the information on other variables used:

1. **Index (Number of Objects):** The first column indicates the number of objects considered in each instance of the knapsack problem. As the index increases, the complexity of the problem also increases.
2. **Generation:** This column represents the number of generations required for the genetic algorithm to converge to a solution. It indicates how many iterations were needed for the algorithm to find an optimal or near-optimal solution.
3. **Optimal Solution Time:** The third column shows the time taken to find the optimal solution. This time is measured in seconds and represents the computational effort required by the genetic algorithm to converge to the solution.
4. **Fitness:** The fitness column displays the fitness of the best solution found by the genetic algorithm. In the context of the knapsack problem, fitness typically represents the total value of items selected for the knapsack, normalized to the target value.
5. **Solution:** The last column presents the binary representation of the solution, where each digit corresponds to whether the corresponding object is selected (1) or not (0) for inclusion in the knapsack.

## Analysis

* For smaller problem instances (e.g., 2-6 objects), the algorithm converges quickly within a few generations, achieving 100% fitness.
* As the number of objects increases, the algorithm requires more generations to converge to the optimal solution. This is expected due to the increased complexity of the search space.
* Despite the increasing complexity, the algorithm is still able to find optimal solutions within a reasonable time frame for larger problem instances (e.g., 10 objects), demonstrating its effectiveness in solving the knapsack problem.

# References

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley.

Koza, J. R. (1992). Genetic programming: On the programming of computers by means of natural selection. MIT press.