同濟大學
TONGJI UNIVERSITY

Term Paper for Academic English Writing Course

Interpretation of the Principles of Genetic Algorithms and Algorithm Optimization

By

Jiahao Xu (2251xxx)

Xiaokun Chen (2251xxx)

Zan Chen (2252xxx)

Xinran Yu (2253xxx)

Under the Supervision of Ms.Hu

Contents

Related Work

Procedure

Three Genetic Operators

How are the three operators related?

4.3.4 A summary

Abstract: In the realm of complex optimization problems such as combinatorial optimization and scheduling problems, a need exists for an intuitive and effective strategy to obtain optimal solutions. Genetic algorithms, rooted in the biological evolution process, offer an optimization approach by simulating natural selection, crossover, and mutation mechanisms to explore the solution space of a given problem. This paper emphasizes the correspondence between various components of genetic algorithms and biological principles, striving to provide a deeper understanding of genetic algorithm principles through analogies and associations. Practical demonstrations of genetic algorithm code implementations are presented using examples. We contend that for newcomers venturing into the field of machine learning, genetic algorithms serve as a suitable "gateway" due to their easy-to-comprehend and readily associable characteristics.

Keywords: genetic algorithm,machine learning

1. Introduction

In the field of computer science, genetic algorithms are optimization algorithms inspired by the biological evolution process. This study aims to explore the principles of genetic algorithms and delve into their correspondence with biology.

The paper divides the principles of genetic algorithms into three parts. The first part focuses on initializing the initial population. It introduces the transition from biological concepts to specific genetic algorithm concepts, emphasizing the explanation of terms such as genes and chromosomes. The second part analyzes fitness, briefly providing examples in different scenarios , given its close relation to real-world problems. The third part delves into genetic operators, including selection, crossover, and mutation. In various genetic algorithm designs, genetic operators often have different implementations. For instance, selection operators may include methods such as roulette wheel selection and tournament selection. Similarly, crossover operators may involve single-point crossover, multi-point crossover, and so on.Each is introduced with two typical methods to clarify differences and characteristics.

Following the theoretical interpretation is the second module, which includes code examples and analysis. The paper uses the OneMax problem as an example, a task involving maximizing the sum of binary digits in a given binary string of a specified length. The goal is to present a code implementation of the genetic algorithm as simply as possible.

The third part covers related algorithms and optimization analysis of genetic algorithms. It begins by introducing the pros and cons of genetic algorithms, leading to an explanation of the principles of the simulated annealing algorithm.

Finally, a hybrid algorithm combining genetic algorithms and simulated annealing is presented as an optimization approach to address the shortcomings of genetic algorithms.

This paper aims to theoretically understand the fundamental concepts of genetic algorithms. It leans towards explaining the principles of genetic algorithms in a thorough and simple manner through analogies and examples, emphasizing the theoretical understanding of genetic algorithm principles rather than the code implementation.

2. Overview of Genetic Algorithms

The Genetic Algorithm (GA) is a global optimization probabilistic algorithm that draws

inspiration from the principles of natural selection in evolution and the chromosome crossover and mutation mechanisms. It aims to retain individuals or populations with the strongest adaptability to the current environment.

For a specific problem, solutions in the solution space are represented through encoding. An fitness function is employed to measure the quality of solutions. The introduction of random factors through crossover and mutation helps prevent the optimization process from getting stuck in local optima. During each evolution step, weaker individuals in terms of fitness are eliminated, ultimately leading to the identification of the global optimum. The traditional flowchart of a genetic algorithm is illustrated in Figure 2-1.

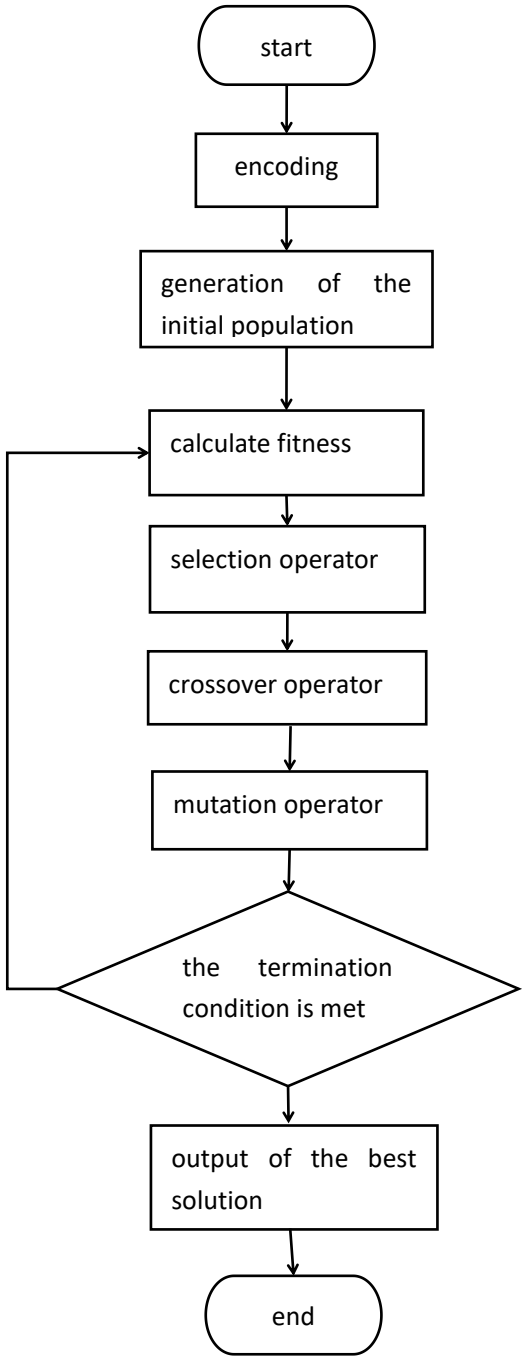

Figure 2-1

3.Biological Knowledge Referenced

3.1 Natural Selection Mechanism

Natural selection, a fundamental concept in biology proposed by Charles Darwin, emphasizes the principles of adaptability and survival. This principle posits that in nature, individuals with better adaptability have a greater chance of surviving, reproducing, and passing their genetic information to the next generation. This mechanism of natural selection leads to species gradually adapting to their environment, progressively refining their traits.

The natural selection mechanism finds simulation and application in genetic algorithms. In genetic algorithms, each individual represents a potential solution. The value of each individual for the given problem is assessed by a fitness function. Individuals with higher fitness are more likely to be selected for reproduction, passing their recorded solutions to the next generation of individuals. This process mirrors the phenomenon in natural selection where individuals with stronger adaptability have a greater chance of passing on their genes. The natural selection mechanism of genetic algorithms propels the population towards the evolution of better solutions in a similar fashion.

3.2 Chromosome Crossover and Mutation Mechanisms

In biology, a chromosome is the genetic material within a cell nucleus, encompassing all genes and genetic information of an organism. Genes, the units controlling individual traits, are located on chromosomes. Chromosome crossover and mutation are fundamental mechanisms in biological evolution, contributing to the introduction of genetic diversity.

In genetic algorithms, chromosomes simulate this biological structure, where each chromosome represents a potential solution, and genes denote specific measures within that solution. A chromosome comprises multiple genes. For example, if a chromosome represents a solution to the question "which ingredients to buy for hotpot," its genes could be "buy bok choy," "buy lamb slices," and so on.

The crossover operation corresponds to genetic recombination in the biological realm, involving the exchange of a portion of information between two chromosomes to create new offspring chromosomes. This facilitates the fusion of favorable traits from different individuals, thereby promoting the search process.

On the other hand, the mutation operation introduces randomness in genetic algorithms, simulating genetic mutations in the biological world. It randomly alters some genes on a chromosome, introducing new features or traits. This helps maintain population diversity and has the potential to uncover novel solutions.

4. Detailed Steps Explanation

4.1 Initializing the Initial Population

Genetic algorithms commence with an initial population representing a set of potential

solutions to the problem. This population is comprised of a specific number of individuals encoded with genes. Each individual is essentially a tangible entity with features, consisting of both a chromosome and a fitness value.

The chromosome, serving as the primary carrier of genetic material, is a collection of multiple genes. Its internal representation (genotype) is a combination of various genes, determining the external manifestation of an individual's traits. For example, the characteristic of having black hair is governed by a specific gene combination within the chromosome. Thus, at the outset, there is a need to establish a mapping from phenotype to genotype, known as encoding. This involves using a particular transformation function to represent an individual's actual features in code. In genetic algorithms, the term "gene" refers to the encoding that reflects the actual features of each individual. A chromosome is a class containing multiple genes, with each individual corresponding to a chromosome, and the chromosome containing various genes. For instance, an individual with black hair and of Asian descent would have a chromosome representing these two genes, denoted as (Black Hair, Asian).

To simplify the intricate task of mimicking genetic coding, often binary encoding is employed as a representation method.

4.1.1 Binary Encoding Method

Similar to the four types of base sequences (AGCT) in human genes, in genetic algorithms, we use only two bases, 0 and 1, forming a chain to create a chromosome. Since a single bit can represent two states of information, a sufficiently long binary chromosome can represent all features. This is the essence of binary encoding.

If we want to represent x belonging to [0, 1023] as an integer with a precision of 1 (as it is an integer), and m represents the length of the binary encoding string, due to the requirements of the solution space, $1024 \leq 2^{(m-1)}$. Therefore, the minimum value for m is 10. For example, the chromosome 0010101111 can represent an individual with a phenotype of x = 175.

Of course, this is the simplest form of encoding, where the representation of the object can be directly obtained by converting binary to decimal. In general, binary encoding is often transformed through complex functions to yield results.

As the most straightforward encoding method, binary encoding has the following advantages:

1.Simple and feasible encoding and decoding operations.

2.Facilitates implementation of genetic operations such as crossover and mutation.

However, binary encoding has its drawbacks, particularly in optimizing continuous functions. Due to its randomness, it exhibits poor local search capabilities. For precision-oriented problems, when a solution approaches the optimal value, the large phenotypic changes caused by mutations, being discontinuous, can lead the solution far from the optimum. For instance, in the example mentioned earlier, if the second bit of 0010101111 mutates, the phenotype changes from x = 175 to x = 175 + 256 = 431. Such mutations result in significant and unstable phenotypic changes, making it less stable.

4.2 Fitness

In genetic algorithms, calculating individual fitness is a crucial step in the optimization process. It determines which individuals will be selected for reproduction, thereby influencing the evolutionary direction of the population. The quality of a chromosome is measured using fitness.

By selecting a certain number of individuals from the previous and current generations based on fitness, they form the next generation, continuing the evolution. After several generations, the algorithm converges towards the best chromosome, likely representing the optimal or suboptimal solution to the problem.

As for the fitness function, it is designed to quantify the advantages and disadvantages of individuals in the problem domain. It evaluates how close each individual is to the optimal solution for the specified problem. Different types of problems may require different metrics for the fitness function, such as minimizing error, maximizing profit, or minimizing cost. The choice of the fitness function depends on the characteristics of the problem and the goals of the algorithm.

The general process of evaluating individual fitness is as follows:

After decoding the chromosome string, the phenotype of the individual is obtained.

The features of the individual's phenotype are scored, calculating the value of the individual's objective function.

Using the fitness function, the objective function value is converted into the individual's fitness.

4.3 Genetic Operators

After obtaining the initial population, genetic operations are utilized iteratively to generate high-quality individuals for the next generation. These operations mainly include selection, crossover, and mutation.

4.3.1 Selection Operator

After calculating the fitness of each individual in the population, the selection process determines which individuals will be used for reproduction and contribute to the next generation. Individuals with higher fitness values are more likely to be selected, passing their genetic material to the next generation.

4.3.1.1 Roulette Wheel Selection

The essence of the roulette wheel selection is to metaphorically represent the overall fitness as a length on a numerical axis ranging from [0,1]. The individual fitness proportions within the overall fitness are depicted as small line segments on this numerical axis. Subsequently, a random value within the range [0,1] is generated to represent a point on this numerical axis. The individual whose fitness segment encompasses this point is selected as the outcome for the current iteration.

This selection strategy simulates the concept of a roulette wheel, where each individual is assigned a portion on the wheel that is proportional to its fitness (or evaluation function value). The selection process resembles spinning the roulette wheel, with individuals of higher fitness having a greater probability of being chosen.

The basic steps of the roulette wheel selection are outlined below:

Compute Fitness Values: For each individual, calculate a fitness value based on its fitness function. The fitness value reflects the quality of the individual in the solution space.

Calculate Fitness Sum: Sum up all the fitness values to obtain the total fitness sum. This sum is used to determine the probability of an individual being selected.

Compute Selection Probabilities: For each individual, its selection probability is calculated as

the ratio of its fitness value to the total fitness sum. This yields the relative probability of each individual being selected.

Construct the Roulette Wheel: Map the selection probabilities onto the roulette wheel, creating a roulette wheel with sectors proportional to individual selection probabilities.

Select an Individual: Generate a random number, and based on where it falls on the roulette wheel, determine the selected individual. Since the sector sizes on the wheel are proportional to the selection probabilities, individuals with higher fitness have a greater chance of being chosen.

One of the advantages of the roulette wheel selection is its effective ability to choose individuals based on their fitness, increasing the likelihood of higher-fitness individuals being passed on to the next generation. However, it also has some drawbacks, such as pronounced differences in selection probabilities when fitness disparities are significant, potentially leading to premature convergence or premature convergence issues in the algorithm.

4.3.1.2 Tournament Selection

Tournament selection is a commonly employed mechanism in optimization algorithms, characterized by its core principle of competitively selecting individuals with higher fitness from a pool of candidate solutions. This selection method emulates the process of natural selection, where competitive interactions among individuals lead to the emergence of those with superior fitness as candidates for the next generation of solutions. This process is repeated until a sufficient number of winners are chosen.

The advantages of tournament selection lie in its relative simplicity and the absence of a requirement to rank individuals by fitness. Furthermore, by adjusting the size of the tournament, one can control the selection pressure during the process, thereby striking a balance between the demands of exploration and exploitation. Tournament selection helps maintain population diversity while providing more opportunities for outstanding individuals to reproduce, contributing to the overall enhancement of the algorithm's global search and convergence performance..

4.3.2 Crossover Operator

The crossover operator involves randomly selecting two individuals from the population as parents. Then, a crossover point is chosen within the encoding strings of these two parents. The process involves swapping segments of the encoding strings of the two parents to create new individuals. The method employed for crossover varies depending on the encoding method used in the genetic algorithm.

One of the most commonly used crossover methods in traditional genetic algorithms is:

1.Single-Point Crossover: In this method, a crossover point 'k' is selected in the range [1, N-1], where N is the number of individuals. The crossover is performed by swapping the segments of the encoding strings at and after the crossover point. The result is two new individuals with mixed genetic material from the parents.

For example, in the case of single-point crossover, if the crossover point is chosen as indicated in Figure 4-1, the crossover is executed, and the new individuals are formed as illustrated in Figure 4-1.

0	0	0	1	1	1

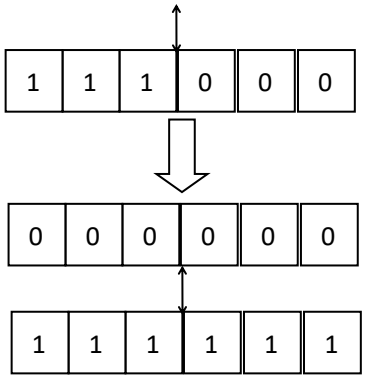

Figure 4-1

2.Multi-Point Crossover: This method allows for the selection of multiple crossover points. Figure 4-2 illustrates the selection of two crossover points. After selecting the points, the segments between these points are swapped between the parents, creating new individuals. The result of two-point crossover is depicted in Figure 4-2.

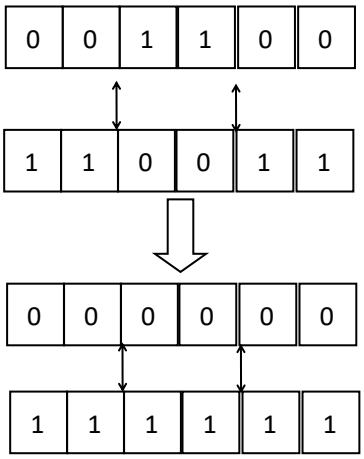

Figure 4-2

4.3.3 Mutation Operator

The mutation operator is applied to the offspring generated through selection and crossover operations. Mutation is a probabilistic operation and usually occurs with a very low probability, as it carries the risk of damaging the individual's performance. In some versions of genetic algorithms, the mutation probability gradually increases with generations to prevent stagnation and ensure population diversity. On the other hand, if the mutation rate increases excessively, the genetic algorithm may become equivalent to random search.

The simplest approach is as follows: for each bit of each offspring in the crossover set, generate a random number between 0 and 1. If it is less than the preset mutation probability, invert that bit; otherwise, leave it unchanged.

Basic Bit Mutation: Perform a mutation operation on a randomly specified bit or bits in an individual's encoding string, based on the mutation probability.

Uniform Mutation: Replace the original gene values at each gene locus in an individual's encoding string with random numbers uniformly distributed within a certain range, with a small

probability. (Especially suitable for the early stages of the algorithm.)

Boundary Mutation: Randomly select one of the two boundary gene values on a gene locus to replace the original gene value. Particularly useful for problems where the optimal point is on or near the boundary of feasible solutions.

Non-Uniform Mutation: Introduce a random perturbation to the original gene value, with the perturbed result serving as the new gene value. After applying the mutation operation to each gene locus with the same probability, the entire solution vector undergoes a slight change in the solution space.

Gaussian Approximation Mutation: Use a random number from a normal distribution with a mean of 'μ' and a variance of 'σ^2' to replace the original gene value during the mutation operation.

5. Example Code for OneMax

5.1 Analysis of OneMax Task

The OneMax task involves finding a binary string of a given length that maximizes the sum of its digits. For example, for a OneMax problem with a length of 5, the digit sum of 10010 is 2, and the digit sum of 01110 is 3.

Clearly, the optimal solution to this problem is a binary string where each digit is 1. However, the genetic algorithm does not possess this knowledge, so its genetic operators are used to search for the optimal solution. The algorithm aims to find the optimal solution in a reasonable amount of time or at least find an approximate optimal solution.

5.2 Approach Combining Genetic Algorithm

Before diving into the practical implementation, it's essential to define the elements used in the genetic algorithm.

Chromosome Selection: Since the OneMax problem involves binary strings, a natural choice is to represent each individual's chromosome directly using a binary string. In Python, this can be implemented as a list containing only 0/1 integer values. The length of the chromosome matches the size of the OneMax problem. For example, for a OneMax problem with a size of 5, the individual 10010 is represented by the list [1,0,0,1,0].

Fitness Calculation: To maximize the sum of digits in the binary string, and considering that each individual is represented by a list of 0/1 integer values, the fitness can be designed as the sum of the elements in the list. For example: sum([1,0,0,1,0]) = 2.

Genetic Operators Selection:

Selection Operator: Tournament selection is chosen.

Crossover Operator: Single-point crossover is selected.

Mutation Operator: Bit-flip mutation is chosen.

Stop Condition Setting: Limiting the number of generations is a common stop condition to ensure the algorithm doesn't run indefinitely. Additionally, since the optimal solution for the OneMax problem is known (a binary string with all 1s, which is equivalent to the length of the list representing the individual), this can be used as another stop condition.

5.3 Configuration of Genetic Algorithm Elements

Import Packages: As shown in Figure 5-1, import the required packages.

Declare Constants: As shown in Figure 5-2, declare constants to set parameters for the OneMax problem and control the behavior of the genetic algorithm.

```
from deap import base
from deap import creator
from deap import tools
import random
import matplotlib.pyplot as plt
```

```
ONE_MAX_LENGTH = 100     //length of bit string to be optimized
POPULATION_SIZE = 200    //number of individuals in population
P_CROSSOVER = 0.9        //probability for crossover
P_MUTATION = 0.1         //probability for mutating an individual
MAX_GENERATION = 50      //max number of generations for stopping condition
```

Next, use the Toolbox class to create the zeroOrOne operation, which is designed to customize the random.randomint(a, b) function. By fixing the parameters a and b to the values 0 and 1, when this operation is called, the zeroOrOne operator will randomly return 0 or 1.

```
toolbox = base.Toolbox()
toolbox.register("zeroOrOne", random.randint, 0, 1)
```

Next, it is necessary to create the Fitness class. Since there is only one objective here - maximizing the total sum of numbers, FitnessMax strategy is chosen. Use a weight tuple with a single positive weight for this purpose:

```
creator.create("FitnessMax", base.Fitness, weights = (1.0, ))
```

In DEAP, the Individual class is commonly employed to represent each individual in a population. This class is created using the creator tool, utilizing a list as the base class to represent the individual's chromosome. Additionally, a Fitness attribute is added to this class, initialized with the previously defined FitnessMax class from the preceding step:

```
creator.create("Individual", list, fitness = creator.FitnessMax)
```

Next, register the individualCreator operation, which creates instances of the Individual class and fills them with random 0s and 1s using the custom zeroOrOne operation defined in step 1. Register the individualCreator operation using the base class initRepeat operation, instantiated with the following parameters: (1) creator.Individual class as the container type for the resulting objects. (2) zeroOrOne operation as the function to generate the objects. (3) Constant ONE_MAX_LENGTH as the number of objects to generate. (4) Since the zeroOrOne operator generates random 0s or 1s, the resulting individualCreator operator will fill a single instance with 100 randomly generated 0s or 1s:

```
toolbox.register("individualCreator", tools.initRepeat, creator.Individual, toolbox.zeroOrOne, ONE_MAX_LENGTH)
```

Finally, register the populationCreator operation for creating a list of individuals. This definition uses the initRepeat base class operation with the following parameters: (1) List class as the container type. (2) The function for generating objects in the list - the personalCreator operator. (3) The last parameter for initRepeat—the number of objects to generate—is not passed here. This

means that when using the populationCreator operation, it will be necessary to specify this parameter to determine the number of individuals to create:

```
toolbox.register("populationCreator", tools.initRepeat, list, toolbox.individualCreator)
```

5.4 Implementation of Genetic Algorithm

Create the initial population by using the previously defined populationCreator operation with the POPULATION_SIZE constant as its parameter. Initialize the generationCounter variable for tracking the generation count:

```
population = toolbox.populationCreator(n = POPULATION_SIZE)
generationCounter = 0
```

To calculate the fitness of each individual in the initial population, apply the evaluate operation to each individual using the map() function. Since the evaluate operation is an alias for the oneMaxFitness() function, the result of the iteration consists of tuples containing the computed fitness for each individual. Convert this result into a list of tuples.

As the items in fitnessValues correspond to the items in the population (the list of individuals), use the zip() function to combine them into tuples, assigning the respective fitness tuples to each individual.

Next, since the fitness tuple has only one value, extract the first value from the fitness of each individual to obtain the statistical data. Simultaneously, create two lists, maxFitnessValues and meanFitnessValues, to store the maximum and mean fitness for each generation of the population:

```
fitnessValues = list(map(toolbox.evaluate, population)
for individual, fitnessValue in zip(population, fitnessValues) :
    individual.fitness.values = fitnessValue;
fitnessValues = [indivalual.fitness.values[0] for individual in population]
```

The core of the genetic algorithm lies in the genetic operators. The first one is the selection operator, utilizing the tournament selection defined earlier with toolbox.select. Since the tournament size has been set during the definition of the operator, passing the population and its length as parameters is sufficient for the selection operator.

The selected individuals are assigned to the offspring variable. Subsequently, they are cloned so that we can apply genetic operators without affecting the original population. It's important to note that even though the selected individuals are named "offspring," they are still clones of individuals from the previous generation. We still need to use the crossover operator to pair them and create actual offspring:

```
offspring = toolbox.select(population, len(population))
offspring = list(map(toolbox.clone, offspring)
```

The next genetic operator is crossover, previously defined as the toolbox.mate operator, which is simply an alias for single-point crossover. Use Python slicing to pair every even-indexed item with the subsequent odd-indexed item in the offspring list as parents. Then, perform crossover with the crossover probability set by the constant P_CROSSOVER. This determines whether this pair of individuals will undergo crossover or remain unchanged. Finally, remove the fitness values of the offspring because their existing fitness is no longer valid:

```
for child1, child2 in zip(offspring[::2], offspring[1::2]) :
    if random.random() < P_CROSSOVER :
        toolbox.mate(child1, child2)
        del child1.fitness.values
        del child2.fitness.values
```

The final genetic operator is mutation, previously registered as the toolbox.mutate operator, set to perform a bitwise flip mutation operation. Iterate through all offspring items and apply the mutation operator with the probability set by the mutation probability constant P_MUTATION. If an individual undergoes mutation, ensure to remove its fitness value. Since the fitness value might be inherited from the previous generation and is no longer valid after mutation, it needs to be recalculated:

```
for mutant in offspring :
if random.random() < P_MUTATION :
    toolbox.mutate(mutant)
    del mutant.fitness.values
```

Individuals that undergo neither crossover nor mutation remain unchanged, so their existing fitness values (calculated in the previous generation) do not need to be recomputed. The fitness values for the remaining individuals are set to None. Use the valid attribute of the Fitness class to identify these new individuals and then compute their new fitness in the same way as the original fitness values were calculated:

```
freshIndividuals = [ind for ind in offspring if not ind.fitness.valid]
freshFitnessValues = list(map(toolbox.evaluate, freshIndividuals))
for individual, fitnessValue in zip(freshIndividuals, freshFitnessValues) :
    individual.fitness.values = fitnessValue
```

After completing the genetic operators, you can replace the old population with the new population. Obtain the maximum and mean fitness values and add their values to the statistics lists. Additionally, use the obtained maximum fitness value to find the index of the best individual and print that individual:

```
population[:] = offspring

maxFitness = max(fitnessValues)
meanFitness = sum(fitnessValues) / len(population)
maxFitnessValues.append(maxFItness)
meanFItnessValues.append(meanFItness)
print("- Generation {}: Max Fitness = {}, Avg Fitness = {}".format(generationCounter,
    maxFitness, meanFitness)

best_index = fitnessValues.index(max(fitnessValues))
print("Best Individual = ", *population[best_index], "\n")
```

When running the program, you can observe the output of the program. It is evident that after 30 generations, the algorithm discovers a solution consisting of all 1s, resulting in a fitness score of 100, prompting the termination of the genetic process. The average fitness starts around 53 initially and approaches 100 towards the end.

```
Generation 27: Max Fitness = 99.0, Avg Fitness = 96.805
Best Indivadual = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

- Generation 28 : Max Fitness = 99.0, Avg Fitness = 97.235
Best Indivadual = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

- Generation 29 : Max Fitness = 99.0, Avg Fitness = 97.625
Best Indivadual = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

- Generation 30 : Max Fitness = 100.0, Avg Fitness = 98.1
Best Indivadual = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

6.Conclusion

In this paper, we transitioned from an explanation of relevant biological concepts to an elucidation of the principles behind genetic algorithms, emphasizing the corresponding relationships between the two. We provided accessible explanations for concepts in genetic algorithms such as chromosomes, genes, and populations, and introduced the most commonly used methods for the three genetic operators: selection, crossover, and mutation.

As a non-deterministic heuristic algorithm inspired by natural processes, genetic algorithms offer a solution for optimizing complex systems, and their effectiveness has been demonstrated through practical applications. Genetic algorithms serve as a global optimization method, requiring no prior knowledge of the system's structure but capable of finding global optimum points among many local optima, effectively handling complex nonlinear problems. Further theoretical research into genetic algorithms is warranted, and their applications are poised for expansion. It is anticipated that genetic algorithms will find even broader applications in the future.

Through our research, we hope to provide readers with a clearer understanding of the implementation principles of genetic algorithms. We aim for this understanding to serve as a medium for deeper exploration into the field of machine learning.

7. References

[1]Sajad Jafari;Tomasz Kapitaniak;Karthikeyan Rajagopal;Viet-Thanh Pham;Fawaz E. Alsaadi.Effect of epistasis on the performance of genetic algorithms[J].Journal of Zhejiang University-SCIENCE A,2019,Vol.20(2): 109-116

[2]Tang You. Research on Data Optimization Based on the Principles of Genetic Algorithms[J]. Da Dong Fang, 2016, (7): 153.

[3] Zhao Yipeng, Meng Lei, Peng Chengjing. Overview of the Principles and Development Directions of Genetic Algorithms[J]. Heilongjiang Science and Technology Information, 2010, (13): 79-80.

[4]Long Fuhai. Research on Feature Selection Method Based on Improved Genetic Algorithm Optimization [D]. Guizhou Minzu University, 2022.

[5] Jin Tiankun, Gao Yang. A Brief Analysis of the Principles and Components of Genetic Algorithms[J]. Science and Technology Horizon, 2014, (4): 19, 6.

[6] Dong Liyan, Li Yongli, Shi Jingrong, Yao Jianhui. Factor Analysis Based on Genetic Algorithm and Neural Network[J]. Journal of Jilin University of Technology, 2001, Vol.31(3): 70.

8. Contributions

Jiahao Xu:Participate in project discussion;Study on the development history, principles, application examples and key code interpretation of genetic algorithms;Allocate the tasks;Integrate the paper

Xiaokun Chen:Participate in project discussion;Write the report summary;Tease out the principles of genetic algorithms

Zan Chen: Participate in project discussion;Make PPT of all the sections;Tease out the structure of the content

Xinran Yu: Participate in project discussion;Study on the functions and advantages of genetic algorithms,optimize algorithms and expand applications;Typeset the paper