ARTIFICIAL INTELLIGENCE PROGRAMMING ASSIGNMENT 01 REPORT IMPLEMENTATION OF GENETIC ALGORITHM IN DISCOVERING MAXIMUM VALUE OF A FUNCTION

Report

Created to fulfill the assignment of the Artificial Intelligence course

By:

Akmal Rafiid	1301192218
Rizqi Khoir Y	1301194255
Muhammad Furqon Fahlevi	1301194214

INFORMATICS

MAJOR SCHOOL OF

COMPUTING

TELKOM

UNIVERSITY

BANDUNG

2021

1. INTRODUCTION

Genetic Algorithms is a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover. (Holland, John. 1975).

Genetic algorithms (GAs) have become popular as a means of solving hard combinatorial optimization problems. The first part of this chapter briefly traces their history, explains the basic concepts and discusses some of their theoretical aspects. It also references a number of sources for further research into their applications. The second part concentrates on the detailed implementation of a GA. It discusses the fundamentals of encoding a 'genotype' in different circumstances and describes the mechanics of population selection and management and the choice of genetic 'operators' for generating new populations. In closing, some specific guidelines for using GAs in practice are provided (Reeves, Colin. *Genetic Algorithms*, 2010).

There are several properties of Genetic Algorithms:

- Individual – any possible solution
- Population – group of all individuals
- Search Space – all possible solutions to the problem
- Chromosome – blueprint for an individual
- Fitness – quality of solution
- Recombination – decomposes two distinct solutions and then randomly mixes their parts to form novel solutions
- Mutation – randomly perturbs a candidate solution

2. OBSERVATION ANALYSIS

2.1. Problem Solving Strategies

Strategy we use to solving genetic algorithms problem that we are using Python 3 programming language. We use Python3 to solve the problem because all of the group members understood the Python language very well. We also compare current generation with (current generation - 10) to get the maximal value or the best result. We use several methods to observe genetic algorithms.

2.1.1.1. Chromosome Design and Decoding Method

[screenshot code]

```
[ ]  def Decode(popul):
       # -1 <= x <=2 && -1 <= y <= 1 *Formula*
       populDecoded = [] # list of decoded population will save here
       x, y = 0.0, 0.0
       i, sigma = 0, 0

       while i < len(popul):
         decoded = []
         temp = popul[i]
         z, b, a = 0, 0, 0
         while z < len(popul[i]):
           if z < len(popul[i]) / 2:
             a = a + temp[z] * 2** - (z + 1)
             sigma += 2 ** - (z + 1)
           else:
               b = b + temp[z] * 2** - (z + 1)
               z = z + 1
           z += 1
         x = -1 + (2 - (-1) / sigma) *a # Formula to get the x
         y = -1 + (1 - (-1) / sigma) *b # Formula to get the x
         decoded.append(x)
         decoded.append(y)
         populDecoded.append(decoded)
         i += 1

       return populDecoded
```

From chromosome design we are using binary encoding with length of 6 which consist of 3 bits for the x value and another 3 bits for the y value. Inside the function we have 2 loops. For the first loop, we create a temporary list that holds the binary string from the population list. For the second loop, we create if-else condition to decode the value from the first 3 binary and last 3 binary and we compute with the formula:

$$g_1 * 2^{-1} + g_2 * 2^{-2} + \cdots + g_n * 2^{-n}$$

After that we compute the x and y value using this formula:

$$x = r_{min} + \frac{r_{max} - r_{min}}{\Sigma_{i=1}^{N} 2^{-i}} (g_1 * 2^{-1} + g_1 * 2^{-2} + \cdots + g_N * 2^{-N})$$

The function should return x and y for every set inside the population list.

2.1.1.2. Population Size

[screenshot code]

```
[ ]  def Population(popSize, ChromoSize, popul):
       i = len(popul) # i == popul size

       while i < popSize:
         chromo = []
         j = 0
         while j < ChromoSize:
           chromo.append(random.randint(0,1)) # random gen and append it to list of chromosome
           j += 1
         popul.append(chromo) # result of the 1 i loop will append 1 chromosome to population
         i += 1

       return popul
```

From population size we are using random import to randomize the population. We create a population function with 2 parameter such as population size and chromosome size. The function will randomly choose a number either 0 or 1. This is an example if we input 5 population size and 6 chromosome size, there will be 5 chromosomes with 6 gens.

```
population(5,6)
```

```
[[1, 0, 0, 0, 0, 1],
 [1, 1, 0, 1, 1, 0],
 [0, 0, 1, 0, 0, 1],
 [1, 1, 0, 1, 0, 0],
 [1, 1, 1, 0, 0, 0]]
```

The function should return binary strings inside a list that we call population.

2.1.1.3. Fitness Calculation

[screenshot code]

```
[ ]  def fitnessRule(x,y): #Formula funtion that will return 1 float number

       return (x*x)*math.sin(math.sin(y*y))+(x+y)
```

```
[ ]  def fitnessScore(popDecoded):
       temp, fitness = [], []
       i = 0

       while i < len(popDecoded):
         j, z, x, y = 0, 0, 0, 0
         temp = popDecoded[i]
         while j < len(temp):
           if j < len(temp)/2:
             x = temp[j]
           else:
             y = temp[j]
           j+=1
         z = fitnessRule(x,y) # the result x and y will be calculated by fitnessRule and output the result of 1 float number
         fitness.append(z)
         i+=1

       return fitness
```

In fitness calculation we compute the value of x and y from the decode function to find a fitness score. In this function we use the math library to compute the formula, we use math.sin to calculate sin in formula:

$$h(x,y) = x^2 * \sin(\sin y^2) + (x+y)$$

2.1.1.4. Parent Selection

[screenshot code]

```
[ ]  def parentSelection(fitnessResult):
       #use Turnament method, the best and the second best value will be the parent
       #this func will return int index of the parent
       parent = []
       sort = sorted(fitnessResult) #sorting ascending
       best = sort[len(sort)-1]
       secBest =  sort[len(sort)-2]

       counterIndex = 0
       for i in fitnessResult:
         if i == best or i == secBest:
           parent.append(counterIndex)
         counterIndex += 1
       # print("sort", sort)
       # print("real list", fitnessResult)

       return parent
```

In the parent selection function, we use tournament selection as a method to find the best and second-best value that will be the parent. This function will return the index of the parent.

2.1.1.5. Crossover

[screenshot code]

```
[ ]  def CrossOver(parent, popul):
       # len(popul) > parent
       # parent must be index
       # popul is normal list of chromo
       coResult = [] # result of the crossover

       [0,1,1,1,0,0] [0,1,1,1,0,0]

       s1 = popul[parent[0]]
       s2 = popul[parent[1]]
       coResult.append(s1[0:3] + s2[3:6])
       coResult.append(s2[0:3] + s1[3:6])
       # print("parent", s1, "--", s2)
       # print("coResult", coResult)

       return coResult
```

After we found the best and the second-best value from parent selection as a parent, we crossover the best parent with the second-best parent.

2.1.1.6. Mutation

[screenshot code]

```
[ ]  def mutation(coResult):
       # coResult cannot be 0
       # bit string mutation
       # flip bit(**not yet**)
       # probability = 0.01
       prob = 0.01
       mut = coResult
       i = 0
       changeCounter = 0

       while i < len(mut):
         j = 0
         while j < len(mut[i]):
           randProb = round(random.uniform(0.00, 1.00), 2)
           if randProb < float(prob):
             changeCounter += 1
             if mut[i][j] == 1:
               mut[i][j] = 0
             else:
               mut[i][j] = 1
           j += 1
         i += 1
       print("got", changeCounter, "Mutation")

       return mut
```

After we found the value from crossover function, we mutation the value with probability 0.01 or 1%. We use bit string mutation because it's easy to use.

2.1.1.7. Generation Replacement

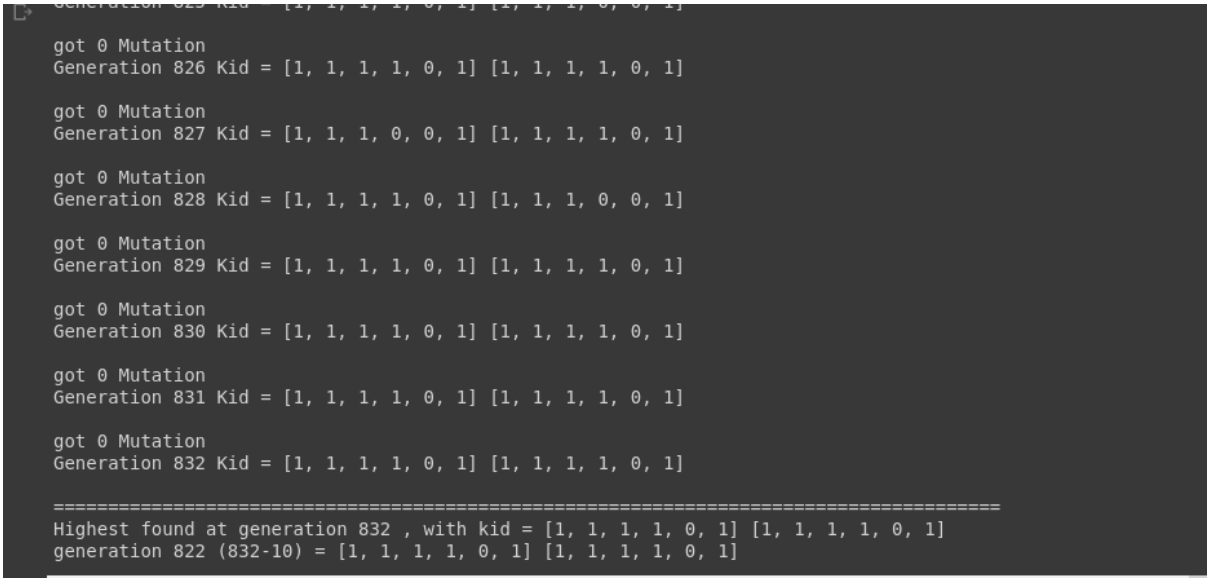
[screenshot code]

```
[ ]  def GeneralReplacement(popul, mutated, kidList, generation):
       # by Utilizing recursive in this func to run all untill get the highest value
       # by utilizing recursive the program will be more light-weight and can increase the running speed
       popul = []

       kidList.append(mutated[0])
       kidList.append(mutated[1])
       print("Generation", generation, "Kid =", kidList[generation], kidList[generation-1], "\n")
       popul.append(mutated[0])
       popul.append(mutated[1])
       popul = Population(10, 6, popul)

       if generation >= 10:
         if kidList[generation] == kidList[generation-10] and kidList[generation-1] == kidList[generation-10-1]:
           print("=============================================================================================")
           print("Highest found at generation", generation, ", with kid =", kidList[generation], kidList[generation-1])
           print("generation", generation-10,"("+str(generation)+"-10"+")", "=", kidList[generation-10], kidList[generation-10-1])
         else:
           generation += 1
           GeneralReplacement(popul, mutation(CrossOver(parentSelection(fitnessScore(Decode(popul))), popul)), kidList, generation)
       else:
         generation += 1
         GeneralReplacement(popul, mutation(CrossOver(parentSelection(fitnessScore(Decode(popul))), popul)), kidList, generation)
```

After we found the kid (child) and we have done the mutation and we got the perfect kid generation. We create a new list population for kid generation, because there is only 2 population from kid generation, we add more randomized population from population function. After that by utilizing

recursion, we call the Generation replacement function itself and loop until the current generation kid, are similar to last 10 generation child, we can take conclusion that if pass 10 generation the kids are the same, it means we found the best solution.

3. OBSERVATION RESULT



4. CONCLUSION

In summary, this code is intended to find the best solution for this problem

$$h(x,y) = x^2 * \sin(\sin y^2) + (x+y)$$

With a boundary, -1≤x ≤2 and -1≤y ≤1 by creating random population (a set of solution) with the chromosomes (solution) consist of zeroes and ones and then comparing it each other using tournament selection to create a best solution and second-best solution for the generation and crossing it over to create a new “perfect” child. We do this cycle until we find the maximum score.

5. VIDEO PRESENTATION LINK

Here is the google drive link for the video:
https://drive.google.com/drive/folders/1PD2HauHSybQcnBlw29-9hfLG7Md7kJ41?usp=sharing