SOLVING PROBLEM BY USING GENETIC ALGORITHM

This report was made to complete Main Assignment of Introduction to Artificial Intelligence

Universitas Telkom

Created by:
Muhammad Farhan Audianto1301180466

BACHELOR OF INFORMATICS
SCHOOL OF COMPUTING
TELKOM UNIVERSITY
BANDUNG
2020

Chromosome design and Decoding Technique

```
#generate random chromosome
def ch(nKrom):
  chr = []
  for i in range(nKrom):
    chr.append(random.randint(0,1))
  return chr
```

The Chromosome generator function, in this case I use binary as the component. The length of chromosome divine on the population function.

```
#dividing the chromosome
def decode(ind):
  x1 = form(ind[:len(ind)//2],1)
  x2 = form(ind[len(ind)//2:],2)
  return x1,x2
```

Decoding function is ask for a chromosome as input then call form, function that count the formula.

```
population = pop(8,10)
```

On this research, I'm using chromosome with length is 8 and 10 chromosome on each population. Screenshot is the part of code on main function.

Population Size

```
population = pop(8,10)
```

As I showed before, the population size that I use on this research is 10 chromosome for each population.

```
#generate random population
def pop(nKrom,nPop) :
  popu = []
  for i in range(nPop):
    popu.append(ch(nKrom))
  return popu
```

Function pop that generate population, the input is length of the chromosome and the size of the chromosome for each population

Parent Selection

```
#search parent
def parentSelection (pop):
  par = random.sample(pop,k=5)
  fitpar = evaluate(par)
  popandfit = sorted(zip(par,fitpar), key=lambda x: x[1],reverse=True )
  return popandfit[0][0]
```

Parent selection function that implement the Parent Selection Scheme on this research.

Genetic Operation Technique and Option and Each Probability

```
#do crossover
def crossover(parent1,parent2):
  if random.random()<0.7 :
    p = random.randint(0,len(parent1))
    return parent1[:p] + parent2[p:]

  return parent1
```

Here's the function that implement crossover for this research and use 70% probability.

```
#do mutation
def mutation(offspring) :
  if random.random()<0.1 :
    p = random.randint(0,len(offspring)-1)
    q = random.randint(0,len(offspring)-1)
    if p == q :
      while p == q :
        q = random.randint(0,len(offspring)-1)
    offspring[p] = random.randint(0,1)
    offspring[q] = random.randint(0,1)
  return offspring
```

Mutation implementation is written on the Mutation function. It have 10% probability and I make it 2 point of mutation (p&q on the code) but both must be on the different point but still there's a chance mutation happen but nothing change since the mutation is random between 0 and 1.

New Generation Selection Technique & The Stopping Criteria

```
def main() :
  nGeneration = 10
  population = pop(8,10)
  for i in range(nGeneration):
    fit = evaluate(population)
    newPop = elitism(population,fit)
    while len(newPop) <= len(population) :
      p1 = parentSelection(population)
      p2 = parentSelection(population)
      child = crossover(p1,p2)
      child = mutation(child)
      newPop.append(child)
    population = newPop.copy()
  return list(sorted(zip(population,fit), key=lambda x: x[1],reverse=True ))
```

For the research, I decide to use Generational Placement that implement/written on the main function then it will be stop on the 10th Generation.

Output

Here's some the best output from few run

```
[([1, 1, 1, 1, 1, 0, 0, 0], 2.018873019019574), [([1, 1, 1, 0, 0], 2.019247544966795),
 ([1, 1, 0, 1, 0], 2.018873019019574),           ([0, 1, 0, 0, 0], 2.019247544966795),
 ([1, 1, 1, 1, 1, 0, 0, 0], 2.018873019019574),  ([1, 1, 1, 0, 0], 2.019247544966795),
 ([1, 1, 1, 1, 1, 0, 0, 0], 2.018873019019574),  ([1, 1, 1, 0, 0], 2.019247544966795),
 ([1, 1, 1, 1, 1, 0, 0, 0], 2.018873019019574),  ([1, 1, 1, 0, 0], 2.019247544966795),
 ([1, 1, 0, 1, 1, 0, 0, 1], 2.018873019019574),  ([1, 1, 1, 0, 0], 2.019247544966795),
 ([1, 1, 1, 1, 0, 0, 1, 0], 1.900752455033205),  ([1, 1, 1, 0, 0], 2.019247544966795),
 ([1, 1, 1, 0, 1, 0, 0, 0], 1.900752455033205),  ([1, 1, 1, 0, 0], 2.019247544966795),
 ([1, 1, 1, 1, 1, 0, 0, 0], 1.8404013095476204), ([1, 1, 1, 0, 0], 1.900752455033205),
 ([1, 1, 1, 0, 1, 0, 0, 0], 1.8071709871794708), ([1, 1, 1, 0, 0], 1.900752455033205),
 ([1, 1, 1, 1, 1, 0, 0, 0], 1.8071709871794708), ([1, 1, 0, 1, 1], 1.900752455033205),
 ([1, 1, 1, 1, 1, 0, 0, 0], 1.7055624143776156), ([1, 1, 1, 0, 1], 1.900752455033205),
 ([1, 1, 1, 1, 1, 0, 0, 0], 1.594865547171068),  ([1, 1, 1, 0, 0], 1.900752455033205),
 ([1, 1, 1, 1, 1, 0, 0, 0], 1.4551063606050025), ([1, 1, 1, 0, 0], 1.900752455033205),
 ([1, 1, 1, 1, 1, 0, 0, 0], 1.451817364472744),  ([1, 1, 1, 0, 0], 1.900752455033205),
 ([1, 1, 1, 1, 1, 0, 0, 0], 1.451817364472744),  ([1, 1, 1, 0, 0], 1.8625940852271796),
 ([1, 1, 1, 1, 1, 0, 0, 0], 1.451817364472744),  ([1, 1, 1, 1, 1], 1.3501754883740147),
 ([1, 1, 1, 1, 1], 1.3501754883740147),          ([1, 1, 1, 0, 0], -0.14237172979226362),
 ([1, 1, 1, 1, 1, 0, 0, 0], 1.3225979170012623)] ([1, 1, 1, 0, 0], -0.14237172979226362)]
```