LAB Artificial Intelligence

Programming Assignment 3 - Genetic Algorithm (GA)

You are required to submit your solutions of this assignment on Moodle, during the next lab time. Submit your own work. Cheating will not be tolerated and will be penalized.

Genetic algorithms (GA) are inspired by natural evolution and are particularly useful in optimization and search problems with large state spaces. Given a problem, algorithms in the domain make use of a *population* of solutions (also called *states*), where each state represents a feasible solution. At each iteration (often called *generation*), the population gets updated using methods like *crossover*, *mutation* and *natural selection*.

A genetic algorithm works in the following way:

1. Initialize random population.
2. Calculate population fitness.
3. Select individuals for mating.
4. Mate selected individuals to produce new population.
5. Repeat from 2. until an individual is fit enough or the maximum number of iterations is reached.

Problem statement: Generating Phrases

In this problem, we use a genetic algorithm to generate a particular target phrase from a population of random strings. This is a classic example that helps build intuition about how to use this algorithm in other problems as well.

Glossary

Before we continue, let us lay the basic terminology of the algorithm.

◊ Individual/State: A list of elements (called *genes*) that represent possible solutions.
◊ Population: The list of all the individuals/states.
◊ Gene pool: The alphabet of possible values for an individual's genes.
◊ Generation/Iteration: The number of times the population will be updated.
◊ Fitness: An individual's score, calculated by a function specific to the problem.

We essentially want to evolve our population of random strings so that they better approximate the target phrase as the number of generations increase. Genetic algorithms work on the principle of Darwinian Natural Selection according to which, there are three key concepts that need to be in place for evolution to happen.

1. **Heredity**: There must be a process in place by which children receive the properties of their parents.

   For our problem, two strings from the population will be chosen as parents and will be split at a random index and recombined as described in the `recombine` function to create a child. This child string will then be added to the new generation.

2. **Variation**: We randomly change one or more characters of some strings in the population based on a predefined probability value called the mutation rate or mutation probability as described in the `mutate` function.

3. **Selection**: There must be some mechanism by which some members of the population have the opportunity to be parents and pass down their genetic information and some do not. This is typically referred to as "survival of the fittest".

   We need to determine which phrases in our population have a better chance of eventually evolving into the target phrase. This is done by introducing a fitness function `fitness_fn` that calculates how close the generated phrase is to the target phrase.

Genetic Algorithm implementation

Open a new Jupyter notebook document and name it **GA.ipynb**. In this file you are going to write the GA code for solving the problem of Generating Phrases.

1. Start by importing the GApsource.py module

   ```
   from GApsource import *
   ```

   Start by checking the functions written in this module. Every time you call a function in your code, you need to understand how this function operates.

2. Define `target` to the target phrase 'Genetic Algorithm'.

3. Then define the gene pool, i.e. the elements which an individual from the population might comprise of. Here, the gene pool contains all uppercase and lowercase letters of the English alphabet and the space character.

   Create a list `gene_pool` that contains
   - uppercase characters # Their ASCII values range from 65 to 91
   - lowercase characters # Their ASCII values range from 97 to 123
   - the space character.

   Hint: Use the `chr()` built-in function

4. Define the maximum size of each population. Larger populations have more variation but are computationally more expensive to run algorithms on.

   ```
   max_population = 100
   ```

5. Define also a mutation rate. The mutation rate is usually kept quite low. Since our population is not very large, we can afford to keep a mutation rate of 7%.

   ```
   mutation_rate = 0.07 # 7%
   ```

6. Initialize a random `population` by using the `init_population` function from the **GApsource** module. You need to pass in the maximum population size, the gene pool and the length of an individual, which in this case will be the same as the length of the target phrase.

7. Now, define a function that returns the a `new_population` list, by using the following steps.

```
def new_population(population):

        # For every individual in the current population:
            # Use the imported select function on the population to
            select two individuals with high fitness values. These will be the
            parents.
            # Recombine the parents using the recombine function to generate the
            child. Note that, the recombine function takes two parents as
            arguments, so you need to unpack the parents variable.
            # Next, apply mutation according to the mutation rate using the
            imported mutate function, on the child with the gene pool and mutation
            rate as additional arguments.
        # Return the new population
```

8. Now, you need to define the most important metric for the genetic algorithm, i.e. the fitness function. It calculates how close the generated phrase is to the target phrase. This function should return the number of matching characters.

```
def fitness_fn(sample):
        # initialize fitness to 0
        # increment fitness by 1 for every matching character
        return fitness
```

9. Now we define the main steps of the genetic algorithm. The algorithm takes the following input:

   - `population`: The initial population.
   - `fitness_fn`: The problem's fitness function.
   - `gene_pool`: The gene pool of the states/individuals. By default 0 and 1.
   - `f_thres`: The fitness threshold. If an individual reaches that score, iteration stops. By default, 'None', which means the algorithm will not halt until the generations are ran.
   - `ngen`: The number of iterations/generations.
   - `pmut`: The probability of mutation.

   The algorithm gives as output the state with the largest score.

   For each generation, the algorithm updates the population. First it calculates the fitnesses of the individuals, then it selects the most fit ones and finally crosses them over to produce offsprings. There is a chance that the offspring will be mutated, given by `pmut`. If at the end of the generation an individual meets the fitness threshold, the algorithm halts and returns that individual.

   To do this, write the following function:

```
def genetic_algorithm(population, fitness_fn, gene_pool=[0, 1],
 f_thres=None, ngen=1200, pmut=0.1):
```

# Run a for loop for `ngen` number of times

# generate a new population using the `new_population` function and assign it to `population`

# Find the individual `current_best` with the highest fitness in the current population. If you print it, you should get a list of characters similar to the following:

```
['J', 'y', 'O', 'e', ' ', 'h', 'c', 'r', 'C', 'W', 'H', 'o', 'r', 'R', 'y', 'P', 'U']
```

# Convert the previous list of characters (`current_best`) into a string using the join function. If you print it, you should get something similar to the following:

```
JyOe hcrCWHorRyPU
```

P.S. After verifying the outputs, you can remove the previous two prints from your code.

# Print out the best individual of the generation and its corresponding fitness value:

```
print(f'Current best: {current_best}\t\tGeneration: {str(generation)}\t\t Fitness: {fitness_fn(current_best)}\r', end='')
```

# compare the fitness of the `current best` individual to `f_thres` using the `fitness_threshold` imported function

# if fitness is greater than or equal to `f_thres`, terminate the algorithm

```
return max(population, key=fitness_fn) , generation
```

The algorithm stops when it gets all the characters correct; or when it has completed the number of generations

10. After having defined all the required functions and variables. Create now a new population and test the functions you have wrote, using the following code

# set the maximum number of generations to 1200

# set the threshold fitness `f_thres` equal to the length of the target phrase

```
population = init_population(max_population, gene_pool, len(target))
solution, generations = genetic_algorithm(population, fitness_fn, gene_pool, f_thres, ngen, mutation_rate)
```

You should get an output similar to the following:

```
Current best: GenetiZ AHgorithm        Generation: 1199                 Fitness: 15
```

Questions

Answer the following questions:

1. What is the maximum value of the fitness?

2. What type of crossover is used?

3. What type of selection is used?

4. Vary the number `ngen` and write the fitness obtained in the following table. **Comment on the result**.

`ngen`	fitness
500	
800	
1000	

`ngen`	fitness
1500	
2000	
2500	

5. Keep on changing `ngen` until your algorithm gets the target phrase. What is in this case? Do you think you can get the target phrase with a smaller (say half of it) `ngen`, if you do not change anything else in your code?

6. For a same number of `ngen` repeat your algorithm 4 times and write the fitness obtained in the following table. **Comment on the result**.

`ngen = 1000`	fitness
1st run	
2nd run	
3rd run	

7. The function `new_population` can be condensed into 1 line of code that returns directly the new population into variable `population`. Write this line of code:

```
population =
```

8. Is it possible for the fitness to have a lower value when passing from a generation to the next?