CS256 — Exercise 9

March 7, 2016

Due: Wednesday, March 16, 2016 before midnight (60 points)

1 Preparing the Project

1. Go to https://codebank.xyz and you should see a project in the CS256 group named `CS256-EX9`. Fork this project for your own user.
2. Next, clone the repository so you have a local copy:

   ```
   $ git clone https://codebank.xyz/username/CS256-EX9.git
   ```

3. The repository should have three C++ files: `Chromosome.cpp`, `Chromosome.h`, and `GA.cpp`.

2 Genetic Algorithms

A genetic algorithm is a form of search algorithm used in artificial intelligence. The purpose of these algorithms is to mimic the process of natural selection to find the answer to a problem. They can be useful in search and optimization problems.

For example, the shape of the antenna pictured below was determined by an evolutionary algorithm to achieve the best result:

Image from: https://en.wikipedia.org/wiki/Genetic_algorithm

To implement a genetic algorithm, we will generate *candidate solutions* that we call *chromosomes*. We start with a randomly generated *population* of these chromosomes.

We need a function that determines the *fitness* of a given chromosome. That is, how good of an answer that specific candidate is.

We will also have the following two functions:

1. A *mutate* function that randomly modifies a chromosome.
2. A *crossover* function that takes two chromosomes and splits each one at the same spot, then combines them together.

Our genetic algorithm works by iterating over *generations* of chromosomes via the following process:

1. Generate random population.
2. Until we get an answer that is good enough, do the next steps in a loop:
   (a) Do the following in a loop for twice as many times as our population size:
      i. Choose a chromosome from our population. With probability *mr* mutate the chromosome.
      ii. With probability *cr*, choose another random chromosome from our population and perform crossover.
      iii. Add the chromosome over to the new population. If we didn't perform crossover or mutation this would copy over the chromosome from the previous generation.
   (b) Sort our new population by fitness. Keep the best half of it.
   (c) Replace our population by the new population.
3. Print out the best chromosome from the final population.

Note that because our mutate function changes a chromosome at random and our crossover function simply combines existing chromosomes, none of the above process explicitly directs our program toward a solution. However, by random modification over a period of time keeping only the most fit chromosomes we can end up at an answer.

3 Our Genetic Algorithm

For this exercise, we will implement a simple genetic algorithm. It will attempt to guess the content of a `std::string` passed as a command-line argument to our program.

We will limit our strings to only lower-case letters. Note: if you want, you may include other printable characters, however the below description assumes you are using only lower-case letters.

Our chromosomes will store a `std::string` as data. Mutation will randomly choose a character in the `std::string` and either increment or decrement it. Crossover will randomly choose an index and create a new string by taking the first part of one of the strings and concatenating it to the end of the other string.

For example, suppose one of our chromosomes has the following data: `abcdefg`. A mutation might result in: `abddefg` or `abcdeeg`.

Make sure that when mutating, we keep the result within the range for lower-case letters. For example, if the letter is `a` and we would decrement it, make it wrap to `z`. Similarly, if the letter is `z` and we would increment it, make it wrap to `a`. This makes sure our strings always have only lower-case letters in them.

Suppose we have two chromosomes with the following data: one has `abcdefg` and the other has `zyxwvut`. A crossover would first randomly choose a position, say 4. This means we are splitting at position 4 so we take the first 4 characters of the first string: `abcd` and concatenate them with the last 3 characters of the second string: `vut` to create the new chromosome `abcdvut`.

We will assume that all strings have the same size in our implementation.

We also have some *target* string that we are trying to guess. This is provided as `argv[1]` to our program.

We can generate random chromosomes for our initial population by:

```
Chromosome randomChromosome(int size)
{
    std::ostringstream sout;
    for (int i = 0; i < size; i++)
    {
        sout << (char) ((rand() % 26) + 97);
    }
    return Chromosome(sout.str());
}
```

where `size` is the length of our target string. Note that when we pick a random number, we first make sure it is in the range of [0, 26) then we shift it up by 97 because 97 is `a` in ASCII. This gives us a random lower-case letter.

Our fitness function should return the sum of the differences between each character of the chromosome and the target.

For example, given `abcd` and target string `bbcd`, the fitness function returns 1 because the first characters are 1 apart and the others are all the same. If we had `abcd` and target string `bcde`, our fitness function would return 4 because all four characters are 1 off from the target. Therefore, a fitness of 0 indicates a perfect match. Again, we assume all strings are the same size as the target. If you want to allow strings of different lengths, think about how you can change the fitness function so that it would give strings of the wrong length a poor fitness score.

You must implement the `mutate` and `crossover` functions of the `Chromosome` class and the following function to run the algorithm:

```
Chromosome runGA(std::string target, int popSize, double mr, double cr);
```

where `target` is the target string, `popSize` is the size of the population we keep on each generation, `mr` is the mutation rate, and `cr` is the crossover rate.

A `main` function has already been provided for testing purposes. It assumes the target string, pop size, mutation rate, and crossover rate have been passed as command-line arguments.

The mutation rate and crossover rate values tell us how often we perform each operation. They will be in the range [0, 1] and act as probabilities.

For example, suppose we have a mutation rate of 0.02 (that is, we mutate 2% of the time). To test if we should perform mutation on this chromosome we generate a random number in the range [0, 1] and check if the number we generated is `<=` the mutation rate. Because our number was generated at at random, it will fall in the range [0, 0.02] 2% of the time. When it does, we perform mutation. Apply the same logic for `cr`.

Changing the population size will allow more chances for change per generation but make each generation take longer to compute.

Sample Output

Suppose we are trying to find string `abcdefghijklmnopqrstuvwxyz` with population size 1000, mutation rate 0.02, and crossover rate 0.90. In my program I am printing the best chromosome on each generation:

```
$ ./ga abcdefghijklmnopqrstuvwxyz 1000 0.02 0.90
Iteration 1
Best: "azciugfopjrbrjqrrnlocxrqua"
Iteration 2
Best: "azciugfopjrbrjqrrnlocxrqua"
Iteration 3
Best: "yywhaoefwdmfkplrzsvpstwuzv"
Iteration 4
Best: "zcyfhbylnplimsrbpstvwsmsyd"
Iteration 5
Best: "ajjmfddgjgzllifqprruxrxauz"
Iteration 6
Best: "dfhffiylihngnqsqprruvwrtez"
Iteration 7
Best: "ybhffiylihngnqsqprruvwrtez"
Iteration 8
Best: "bbvddgebnhlhpnporzuoxrxaza"
Iteration 9
Best: "zcfhfcddenmkmnloprruptwuzv"
Iteration 10
Best: "caubcedhikolmrrpnryrwtywwa"
Iteration 11
Best: "bbvddgelihngnqsqprruvwvswz"
Iteration 12
Best: "azwhfedhizlimnokprstrvvxaz"
Iteration 13
Best: "azwhfedhizlimnokprstrvvxaz"
Iteration 14
Best: "azcddedhenmkmnsqprruvwwuzv"
Iteration 15
Best: "accceaekfjlhmnokprsuxvwuzz"
Iteration 16
Best: "azccdfglilmlinmqpssrvtwuzz"
Iteration 17
Best: "bbbddedhilmhmnkpnrruvwvxxz"
Iteration 18
Best: "azcedfglilmlmnloprrptuwwwz"
Iteration 19
Best: "bbecbhghijgkmnoqprruvwvxyz"
Iteration 20
Best: "bbbddedhilkkmnmnprruvwwxxz"
Iteration 21
Best: "azcecgghikolmnoqprruvwvxyz"
Iteration 22
Best: "abbfeighihklmnnoqssuvwwyaz"
Iteration 23
Best: "abcdefghihklmnnpprspvwwyyz"
Iteration 24
Best: "azcdefdiijmkmnopprsttwvxyz"
Iteration 25
Best: "abcdcfgiijmkmnopprstttwxyz"
```

```
Iteration 26
Best: "abbccfgiijmkmnopprstttwxyz"
Iteration 27
Best: "bbbddgghijklmnopprsrvwwxzz"
Iteration 28
Best: "abcdefghgjllmnopprruvwwyyz"
Iteration 29
Best: "bbbddgghijklmnopprstvwwxzz"
Iteration 30
Best: "abbeefghijlkmnopprstvwwxyz"
Iteration 31
Best: "abcdefghgjllmnopprsuuvvxyz"
Iteration 32
Best: "abcdefghijllmnoqqrsuuwwxzz"
Iteration 33
Best: "abcdefghijklmnopprstvwwxyz"
Iteration 34
Best: "abcdefghijklmnopprstvvwxyz"
Iteration 35
Best: "abcdefghijklmnopprstvvwxyz"
Iteration 36
Best: "abcdefghijklmnopprstvvwxyz"
Iteration 37
Best: "abcdefghijklmnopprstvvwxyz"
Iteration 38
Best: "abcdefghijklmnopqrsuuwwxyz"
Iteration 39
Best: "abcdefghijklmnopprstuvwxyz"
Iteration 40
Best: "abcdefghijklmnopqrstvvwxyz"
Iteration 41
Best: "abcdefghijklmnopqrstvvwxyz"
Iteration 42
Best: "abcdefghijklmnopqrstuvwxyz"
Answer found after 42 iterations.
```

4 Submission

Once you have completed the program, you can use `git add`, `git commit`, and `git push` to push the changes to https://codebank.xyz. You can make as many commits and push as many times as desired until the deadline.