TCSS 342 Data Structures
Assignment 2 – Evolved Names

Guidelines

This assignment consists of programming and written work. Solutions should be a complete working Java program including your original work or **cited contributions** from other sources. These files should be compressed in a .zip file for submission through the Canvas link.

This assignment is to be completed on your own or in a group of at most 3 students. If you choose to work in a group, this must be clear in your submission, and each group member must upload their own copy of the homework to Canvas. Please see the course syllabus or the course instructor for clarification on what is acceptable and unacceptable academic behavior regarding collaboration outside of a group of up to 3 students.

Assignment

Imagine a virtual world in which all that exists are strings of characters from the following set, namely, capital letters, a space, a hyphen, and an apostrophe:

{ A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, ␣, -, ’ }

(The ‘␣’ character stands for a blank space, and is displayed like this merely to be more easily visible). Strings in this world can reproduce new strings and die if they are not fit enough. You will evolve strings in this world until they spell your name. To do this you will create a Genome class which will contain a list of characters from the above set representing a string in your world, and you will create a Population class which will contain a list of Genomes representing all the strings in your world.

Your Genome class must:

- have some internal representation of the string of characters.
- initialize a new string to the default value ‘A’.
- be able to **mutate** by:
  - possibly **adding a new character** somewhere in the string.
  - possibly **deleting a randomly selected character** from the string.
  - possibly **changing a character** in the string to a different value.
- be able to **crossover** with another genome:
  - given two Genomes create a third that is a combination of the two.
- be able to measure Genome **fitness**:
  - using one of the two **zero-based fitness** methods listed here:
    - calculate how close the string in the Genome is from your name using the simple method detailed below.
    - **(Optional)** calculate how close the string in the Genome is from your name using the Levenshtein edit distance.
- display:
  - output the string and its fitness in an easy to read format.

Your Population class must:

- maintain a list of Genomes representing the current population.
- initialize the population with a fixed number of default Genomes.
- update the list of Genomes every breeding cycle by:
    - removing the least-fit members of the population.
    - mutating or breeding the most-fit members of the population.
    - **Note:** Because we are using a zero-based fitness the "most fit" member of the population has the lowest fitness score not the highest.
- display the entire population.
- display the most-fit individual in the population

The Main class is a controller and will:

- instantiate the Population class.
    - Use 100 genomes and a mutation rate of 0.05.
- call day() from the Population class until the fitness of the most fit genome is zero.
- output simulation progress.
- output runtime statistics.

Formal Specifications

Your simulation will implement the Genome class according to this interface:

- String *target* – a data element that is initialized to your name.
- Genome(double *mutationRate*) – a constructor that initializes a Genome with value 'A' and assigns the internal mutation rate. The *mutationRate* must be between zero and one.
- Genome(Genome *gene*) – a copy constructor that initializes a Genome with the same values as the input *gene*.
- void mutate() – this function mutates the string in this Genome using the following rules:
    - with *mutationRate* chance add a randomly selected character to a randomly selected position in the string.
    - with *mutationRate* chance delete a single character from a randomly selected position of the string but do this only if the string has length at least 2.
    - for each character in the string:
        - with *mutationRate* chance the character is replaced by a randomly selected character.
- void crossover(Genome *other*) – this function will update the current Genome by crossing it over with *other*.
    - Create the new list by following these steps for each index in the new string (which can be of any length between the minimum and the maximum of the two string lengths), starting at the first index:
        - Randomly choose one of the two parent strings.
        - If the parent string has a character at this index (i.e. it is long enough) copy that character into the new list. Otherwise end the new list here and replace the old list.
- Integer fitness() – returns the fitness of the Genome calculated using the following algorithm:
    - Let *n* be the length of the current string. Let *m* be the length of the target string.

    - Let *L* be the max(*n, m*).
    - Let *f* be initialized to |*m* – *n*|.
    - For each character position 0≤*i*<*L* add one to *f* if the character in the current string is different from the character in the target string (or if no character exists at that position in either string). Otherwise leave *f* unchanged.
    - Return *f*.
- **(Optional)** Integer fitness() – instead of the algorithm above use the Wagner-Fischer algorithm for calculating Levenshtein edit distance:
    - Let *n* be the length of the current string. Let *m* be the length of the target string.
    - Create an (n + 1) × (m + 1) matrix *D* initialized with 0s.
    - Fill the first row of the matrix with the column indices and fill the first column of the matrix with the row indices.
    - Implement this nested loop to fill in the rest of the matrix:

```
for i from 1 to n
    for j from 1 to m
        if (current[i-1] == target[j-1]) D[i, j] = D[i-1, j-1]
        else D[i, j] = min(D[i-1, j] + 1, D[i, j-1] + 1, D[i-1, j-1] + 1)
```

    - Return the value stored in D[n,m] + (abs(n-m)+ 1)/2. (Use integer arithmetic.)
- String toString() – this function will display the Genome's current character string and fitness in an easy to read format.

Your simulation will implement the Population class. It must function according to this interface:

- Genome *mostFit* – a data element that is equal to the most-fit Genome in the population.
- Population(Integer *numGenomes*, Double *mutationRate*) – a constructor that initializes a Population with a number of default genomes (see above).
- void day() – this function is called every breeding cycle and carries out the following steps:
    - update *mostFit* variable to the most-fit Genome in the population. (Remember this is the genome with the **lowest fitness**!)
    - delete the least-fit half of the population.
    - create new genomes from the remaining (most-fit) population until the number of genomes is restored by doing either of the following with equal chance:
        - pick a remaining genome at random and clone it (with the copy constructor) and mutate the clone.
        - pick a remaining genome at random and clone it and then crossover the clone with another remaining genome selected at random and then mutate the result.

You will also provide a Main class for control and testing of your evolutionary algorithm.

- void main(String[] args) – this method should instantiate a population and call day() until the target string is part of the population.
    - The target string has fitness zero so the loop should repeat until the most-fit genome has fitness zero.
    - After each execution of day() output the most fit genome.
    - To measure performance output the number of generations (i.e times day() is called) and the execution time.
- void testGenome() – this method tests the Genome class.
- void testPopulation() – this method tests the Population class.

Include any other methods used to test components of your Genome and Population classes.

Also, none of those classes may be in a named package (just define them *without* a Java **package** clause).

Hint: *Even a slight deviation from the specifications can result in bugs that affect performance*. Make sure you implement these methods *exactly* as stated above.

Submission

The following files are provided for you:

- trace2.txt – a sample trace of a working solution.

You will submit a single .zip file containing:

- Genome.java – an implementation of the Genome class satisfying the above criteria.
- Population.java – an implementation of the Population class satisfying the above criteria.
- Main.java – a controller for your evolutionary system that serves to run the evolutionary simulation and test the components of your code.

Grading Rubric

This assignment is graded out of 25 points but there are 31 points available.

Correctness 15 points

- To get all 15 points your code must evolve my name "PAULO LICCIARDI BARRETO" in fewer than 1000 generations and in less than 2 seconds.
- Solutions that evolve my name in more than 1000 but at most 10000 generations will be deducted 4 points.
- Solutions that evolve my name in more than 10000 but less than 100000 generations will be deducted 8 points.
- Solutions that take more than 2 seconds will be deducted 3 points.
- A 0 will be awarded if your solution cannot evolve my name.
- One point is deducted if you do not output the fittest string and its fitness every generation.
- Solutions where the fitness of the fittest string is not non-increasing along the generations will be deducted 4 points.
- One point is deducted if you do not output the generation number every generation.
- One point is deducted if you do not output the runtime of your program.

Interface 5 points

- All Genome methods and properties match the interface provided above. One point is deducted for each mismatch.
- All Population methods match the interface provided above. One point is deducted for each mismatch.

Testing 4 points

- All Genome methods are tested. Points are deducted for missing tests or inadequate tests.

- All Population methods are tested. Points are deducted for missing tests or inadequate tests.

Miscellaneous 4 points

- All four points are awarded automatically with the following exceptions.
- One point is deducted if your submission is late.
- One point is deducted if you resubmit after your assignment is graded.
- Two points are deducted for each deviation from the correct format your solution has (i.e. not in a single .zip, and/or you submit .class or .jar files instead of only the .java sources, and/or you submit code that needs to be altered to work, etc.).

(Optional) Wagner-Fischer Algorithm 3 points

- All three points are awarded if you implement the optional fitness function using the WagnerFischer Algorithm.

Tips for maximizing your grade:

- Make sure your classes match the interface structure exactly. I will use my own controller (Main.java) and test files on your code and it should work without changes. Do not change the method name (even capitalization), return type, argument number, type, and order. Make sure all parts of the interface are implemented.
- Only zip up the .java files. If you use Eclipse or IntelliJ these are found in the "src" directory, not the "bin" directory. You will be docked points for including .class or .jar files.
- All program components must be in the default package. You will be docked points if you use a different (i.e. explicit) package.
- Place your name in the comments at the top of every file. If you are working in a group, make sure the names of all group members appear clearly in these comments, and that each group member uploads their own copy of the joint work to Canvas.