UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program design: Introduction

Assignment 1 Specifications

Release Date: 01-08-2022 at 06:00

Due Date: 22-08-2022 at 23:59

Total Marks: 96

Contents

1 General instructions

- This assignment should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline, as **no extension will be granted.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence of certain functions or classes).
- Failure of your program to successfully exit will result in a mark of 0.
- Read the entire assignment thoroughly before you start coding.
- **To ensure that you did not plagiarize, your code will be inspected with the help of dedicated software.**
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at http://www.ais.up.ac.za/plagiarism/index.htm.
- Unless otherwise stated, the usage of C++11 or additional libraries outside of those indicated in the assignment, will not be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements **Please ensure you use C++98**

2 Overview

For this assignment you will be implementing a Genetic Algorithm in C++. This algorithm will be used to demonstrate how to handle dynamic memory and how to correctly use classes. The algorithm will also be used to demonstrate the different ways function signatures can be designed in a class to allow for different functionality. In this assignment you will gain experience with a semi-complex class structures and pointers.

3 Background

3.1 Genetic Algorithm

Genetic Algorithms (GA) are a type of evolutionary algorithm developed by John H. Holland in the 1960's. GAs are based on the idea of survival of the fittest where a fitness value is assigned to possible solution. This value is an indicator as to how well the possible solution solves the given problem. A GA consists out of a population of possible solutions also referred to as chromosomes. Each chromosome consists of a set of genes, which is used to encode the actual value of the possible solution. Chromosomes are usually encoded as a binary string where each bit represents a gene. The main general algorithm of a GA is given below:

```
GeneticAlgorithm(int number_of_generations){
    Create the initial population P
    for(i from 0 to number_of_generations){
        Select the n number of fittest chromosomes.
        Add them to the mating pool.
        Change the population using genetic operators to P‘.
        Set P = P‘
    }
}
```

The two genetic operators that this assignment will use is cross-over and mutation. The exact implementations will be discussed later.

4 Your Task:

For this assignment, your GA will need to try and get all the chromosomes in the population to only contain 1's. Although this can be easily programmed explicitly the idea is to investigate how the GA performs and to gain experience with pointers.
You are required to implement the following classes. Pay close attention to the function signatures as the H files will be overwritten, thus failure to comply to the UML will result in a mark of 0.
Please read the red paragraph in section 7

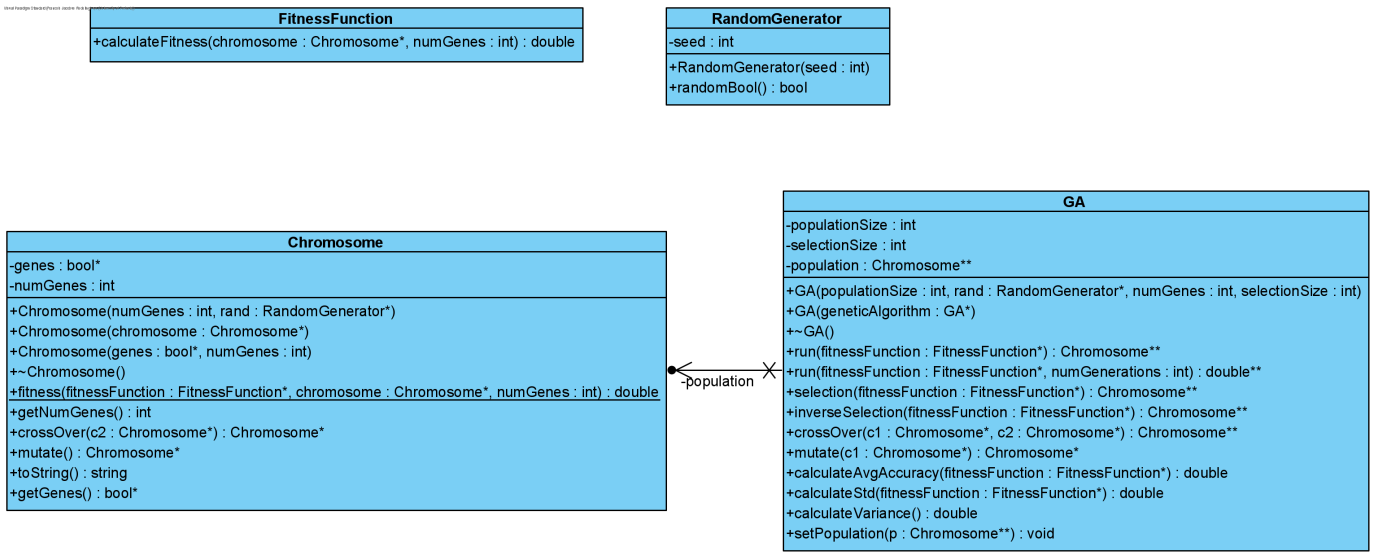

Figure 1: Class diagrams

4.1 RandomGenerator

- Members:
  - seed: int
    - This value will be used during the calculation of the random number.
- Functions:
  - RandomGenerator(seed: int)
    - This is the constructor for the RandomGenerator class.
    - This function should call the srand function and pass in the passed in seed as a parameter.
    - This function should also initialize the seed member with the passed in parameter.
  - randomBool(): bool
    - This function should return a random bool.
    - To achieve this the function call the rand() function and test if the returned value is even or odd.
    - If the value is even the function should return false.
    - If the value is odd the function should return true.

4.2 Chromosome

- Members:
  - genes: bool*
    - This is an array of bools that represents the genes.
    - Think of them as the bits in the binary string.
  - numGenes: int
    - This is the size of the genes array.
- Functions:
  - Chromosome(numGenes: int, rand: RandomGenerator*)
    - This is the constructor for the Chromosome class.
    - The constructor should initialize the genes array with numGenes as the size.
    - Using the appropriate function from the passed in rand object initialize each gene in the genes array.
    - Initialize the numGenes with the passed in parameter.
    - If the numGenes parameter is a negative number initialize numGenes member to 0 and the genes array to a size of 0.
    - If the rand parameter is null initialize the genes array to only consist out of false elements.

    - **Do not create a deep copy of the passed in rand object**
- Chromosome(chromosome: Chromosome*)
    - This is the copy constructor of the Chromosome class.
    - This function should make a deep copy of the genes array and use the passed in chromosome's genes array to populate the new genes array.
    - If the passed in chromosome is null then initialize the numGenes to 0 and initialize the genes array to a size of 0.
    - Remember to also initialize the numGenes member
- Chromosome(genes: bool*, numGenes: int)
    - This is another parameterized constructor. This constructor takes in two parameters, a bool array and an integer value.
    - Initialize the genes array and populate it with the passed in bool array.
    - Use the appropriate parameter to initialize the numGenes member.
    - If the numGenes is a negative number initialize numGenes to 0 and the genes array to a size of 0.
    - If the bool array is null initialize the genes array to only contain false.
- ~Chromosome()
    - This is the destructor for the Chromosome class.
    - The destructor should deallocate all the dynamically allocated memory assigned to the object.
- fitness(fitnessFunction: FitnessFunction*, chromosome: Chromosome*, numGenes: int): double
    - This function is a static function of the Chromosome class (indicated by the underlining).
    - This function will return the fitness of the passed in chromosome object by calling the appropriate function of the passed in fitnessFunction object.
    - If the passed in fitnessFunction is null the function should return 0.
    - If the passed in chromosome is null the function should return 0.
    - If numGenes is less than or equal to 0 return 0.
- getNumGenes(): int
    - This function will return the number of genes of the current chromosome object.

- crossOver(c2: Chromosome*): Chromosome*
  - This function should create a new chromosome by using the following pseudo code:

```
crossOver(c2){
    crossOverPoint := floor(c2->numGenes / 2);
    array of bools with size number of genes: nGenes;
    for(i: [0, crossOverPoint)){
        nGenes[i] := this->genes[i];
    }
    for(i: [crossOverPoint,numGenes)){
        nGenes[i] := c2->genes[i];
    }
    return a new Chromosome using nGenes array.
}
```

  - Remember to manage your memory correctly.
  - This function should not alter the current chromosome nor the passed in chromosome. Only the resulting chromosome may be altered.
  - If the passed in c2 is null the function should return a deep copy of the "this" chromosome.
- mutate(): Chromosome*
  - This function should create a new chromosome by using the following pseudo code:

```
mutate{
    array of bools with size number of genes: nGenes;
    for(i: [0, numGenes)){
        nGenes[i] := inverse of genes[i];
    }
    return a new Chromosome using nGenes array.
}
```

- toString(): string
  - This function should return a string representing the genes.
  - Trues should be represented by 1s and false should be represented by 0s.
  - For example:
    - Given the following genes array: true, true, false, false, true, true, false, true
    - The function should return: 11001101
  - If the numGenes is 0 then the function should return an empty string i.e. ""
- getGenes(): bool*
  - This function should return the genes array.

4.3 FitnessFunction

- calculateFitness(chromosome: Chromosome*, numGenes: int): double
  - This function will return the fitness of the passed in chromosome.
  - Fitness is defined as:

$$\frac{m}{n}$$

    where:
    - m: is the number of 1s (true) in the genes array
    - n: is the number of genes in the genes array
  - Example:
    - Given the following genes array: false, false, true, true, false, true
    - The fitness value would be:

$$= \frac{3}{6}$$
$$= 0.5$$

  - Remember to ensure that your code does **not** perform integer division.

4.4 GA

- Memebers:
  - populationSize: int
    - This is the size of the population array.
  - selectionSize: int
    - This is a number that will be used in the calculation of how many chromosomes will need to be replaced.
  - population: Chromosome**
    - This is an array of chromosomes representing the population.
- Functions:
  - GA(populationSize: int, rand: RandomGenerator*, numGenes: int, selectionSize: int)
    - This is the constructor for the GA class.
    - Use the passed in parameters to initialize the members of the object.
    - Populate the population array by creating new chromosomes and passing in the appropriate parameters.
    - Note: **Do not** make a deep copy of the passed in rand parameter.
    - It can be assumed that valid inputs will be passed to the function.

- GA(geneticAlgorithm: GA*)
  - This is the copy constructor for the GA class.
  - Use the passed in parameter to initialize the members of the object.
  - Populate the population array by performing deep copies of the passed in parameter's genes array. *Hint: Use the copy constructor of the Chromosome class to perform the deep copies*
  - It can be assumed that valid inputs will be passed to the function.
- ~GA()
  - This is destructor of the GA class.
  - The destructor should deallocate all the allocated memory in the GA object.
- selection(fitnessFunction: FitnessFunction*): Chromosome**
  - This function should return an array where the chromosomes are sorted from highest to lowest based on the chromosomes fitness value.
  - In other words the function should return an array where the fittest chromosome is at index 0 and the weakest chromosome is at the highest index.
  - Use the fitnessFunction as described above regardless if it is null or not.
  - If multiple chromosomes have the same fitness value sort them according to the order they were placed in the population array.
  - The resulting array should have a size equal to the population size.
  - **This function should not change the population array**
- inverseSelection(fitnessFunction: FitnessFunction*): Chromosome**
  - This function should return an array where the chromosomes are sorted from lowest to highest based on the chromosomes fitness value.
  - In other words the function should return an array where the fittest chromosome is at the highest index and the weakest chromosome is at index 0.
  - Use the fitnessFunction as described above regardless if it is null or not.
  - If multiple chromosomes have the same fitness value sort them according to the **reverse** order they were placed in the population array.
  - The resulting array should have a size equal to the population size.
  - **This function should not change the population array**
- crossOver(c1: Chromosome*, c2: Chromosome*): Chromosome**
  - This function will return a Chromosome array of size 2.
  - Index 0 of the array should be c1 crossed over with c2 by calling the appropriate function of c1 and passing c2.
  - Index 1 if the array should be c2 crossed over with c1 by calling the appropriate function of c2 and passing c1.
  - Note c1 and c2 should not be altered.

- mutate(c1: Chromsome*): Chromosome*
  - This function will return a mutated Chromosome of the passed in chromosome.
  - Call c1's appropriate function.
  - Do not alter c1.
- calculateAvgAccuracy(fitnessFunction: FitnessFunction*): double
  - This function should calculate the average fitness of the whole population using the passed in fitnessFunction.
  - The function should return the average fitness of the whole population.
- calculateStd(fitnessFunction: FitnessFunction*): double
  - This function should calculate the standard deviation of the fitness of the whole population using the passed in fitnessFunction.
  - From high-school statistics you should remember that the formula for the standard deviation is as follows:

$$STD(P) = \sqrt{\frac{\sum_{i=0}^{sizeOf(P)}(chromosome_i - AVG(P))^2}{sizeOf(P)}}$$

- calculateVariance(): double
  - This function should return the variance of the population.
  - Use the following formula to calculate the variance:

$$V(P) = \frac{numUniqueChromosomes(P)}{sizeOf(P)}$$

- setPopulation(p: Chromosome**): void
  - This function can be used to change the population to the passed in parameter.
  - A deep copy of each chromosome in p should be made when replacing the population.
  - It can be assumed that p won't be null and that p will have the same length as the current population.
- run(fitnessFunction: FitnessFunction*): Chromosome**
  - This function will represent a single generation run of the algorithm.
  - The function will return an array of Chromosomes that represent P‘ as described in the background section.
  - Use the pseudo code described in section 8.1 to implement this function
  - Remember to correctly manage your memory and deallocate any memory that was allocated and will not be used again.
  - Use the fitnessFunction as described in section 8.1 regardless if it is null or not.

- run(fitnessFunction: FitnessFunction*, numGenerations: int): double: **
  - This is the main run of the algorithm.
  - This function will return a 2d double array of size [numGenerations][3].
  - Use the pseudo code described in section 8.1 to implement this function.
  - Remember to deallocate any memory that will not be used again.
  - Use the fitnessFunction as described in section 8.2 regardless if it is null or not.

5 Implementation Details

- You must implement the functions in the header files exactly as stipulated in this specification. Failure to do so will result in compilation errors on fitch fork.
- You may only use **C++98**.
- You may only utilize the specified libraries. Failure to do so will result in compilation errors on fitch fork.
- Do not include using namespace std in the h files
- You may only use the following libraries:
  - String
  - CMath
  - IOStream
- You are supplied with a trivial main demonstrating the basic functionality of the assignment. You are also provided with two textfiles depicting the expected output generated by VS2022 and WSL-Debain for the same main.

6 Upload checklist

- GA.h,cpp
- Chromosome.h,cpp
- FitnessFunction.h,cpp
- RandomGenerator.h,cpp

7 Benchmark

Benchmarks are used to determine how well a program performed with respect to performance. On the next page you will find some results of a benchmark program for the required task. **Note if you implemented the tasks correctly your GAs should give relatively the same results. The reasons for minor variations may be attributed to C++'s rand function's implementation differs depending on the platform and the version of the compiler.**Execution times should only be considered for curiosity.

Note execution time will not effect your mark on fitchfork.

For the students that would like to try and optimize their code use the execution times as a guide as to how well your code performs. Also remember you may use valgrind to check if your code has memory leaks.

Run	Seed	PopulationSize	NumGenerations	NumGenes	Selection size	Avg avg	Avg std	Avg var	Execution Time (ms)
1	1	100	100	10	5	0.8451	0.862927	0.0914	15.625
2	12345	1000	1000	5	10	0.98082	0.986831	0.002926	7859.38
3	8888	50	50	10	2	0.75292	0.763151	0.284	0
4	5050	1000	1000	100	100	0.683209	0.696919	0.007908	17187.5
5	1	200	75	10	10	0.84044	0.858301	0.1136	78.125
6	44	10	10	100	1	0.5751	0.578271	0.41	0
7	54321	700	50	5	70	0.886583	0.934461	0.00505714	218.75

The above table was generated using WSL-Debain GCC compiler with C++ 98 standard.

For students that would like to play around with this project above that which is required for the assignment, try and find the optimal parameters for the benchmark functions such that the GA will converge on a population that consists just out of 1s in the least amount of generations and the quickest execution time.

8 Pseudo code

8.1 run(fitnessFunction: FitnessFunction*): Chromosome**

```
run(fitnessFunction): Chromosome[]{
    winners := use the select function to obtain an array of chromosomes.

    losers := use the inverse select function to obtain an array of chromosomes.

    offsprings := an array of size: 3* selection size of type chromosome.

    P‘ := an array of chromosomes with the same size as the population array.

    for(i: [0,2*selectionSize)){
        nChromosomes := call the appropriate function that will return an array of crossed over chromosomes passing
           winners[i] and winners[i+1] as parameters.
        place the chromosomes in nChromosomes into offsprings[i] and offsprings[i+1] respectively.
        increment i with 1.
    }
    for(i: [0,selectionSize)){
        use the appropriate function to mutate winners[i+2*selectionSize] and place this newly created chromosome in
           offspring[i+2*selectionSize]
    }

    Using a shallow copy method populate P‘ with the contents of the population array.

    for(i: [0, size of offsprings)){
        dyingChromosome := losers[i]
        u := the index of dyingChromsome in P‘
        P‘[u] := offsprings[i]
    }
    return P‘
}
```

8.2 run(fitnessFunction: FitnessFunction*, numGenerations: int): double: **

```
run(fitnessFunction, numGenerations){
    results := double array of size [numGenerations][3].
    for(i in [0,numGenerations)){
        P‘ := using the other overloaded run function formulate a new population.
        population := P‘.
        results[i][0] := call the appropriate function that will return the average of the population passing in the
           appropriate parameter.
        results[i][1] := call the appropriate function that will return the standard deviation of the population
           passing in the appropriate parameter.
        results[i][2] := call the appropriate function that will return the variance of the population.
    }
    return results

}
```