Nathan Shummoogum

ID: 40004336

COEN-244: Final Project

Genetic Algorithm: Optimization and Curve fitting program

Concordia Winter 2016

Problem Description

As the final project of this course we have been asked to create a genetic algorithm code. This code will have two applications, the first will be the optimization of a specific function, the second is a curve fitting of sets of points. The optimization application consists of selecting a specific equation and graph. We must optimize the x and y values of the graph to find the global minima of the graph. The curve fitting application will take twenty sets of coordinates and find the best fitting curve for these points based on the mean square error equation. The lower the mean square error the better fit the curve is for chosen set of points.

The genetic algorithm used to apply the optimization and curve fitting will consist of similar components and must follow a specific method. First the population will be created and initialized, where every individual in the population will be given specific distinctions at random. Now that every individual is given their own traits, they can each be given a fitness score. Using a tournament selection method, a portion of the population will be selected to become parents to create the next population. Using the newly created parent pool a new pool of offspring will be created. The variables of the parent pool will be crossed-over using intermediate recombination between two parents to create two offspring. Once the offspring are created they will pass a mutation algorithm generating mutated children. The individuals with the top fitness value in the joint pool of parents and offspring will create the next generation population. This next generation will then be re-evaluated and this process will continue until an individual with an ideal fitness score is found.

Alternatives & Recommendations

Design a)

Template <class T>
Class Algorithm
private:
vector<T> population
vector<T> nextPop
vector<T> parent
vector<T> pool
vector<double> fitness_val
public:
application()
virtual void initialize() = 0;
virtual void tournament();
virtual void generate_next();
virtual void mutate(T&);
virtual void crossover();
void fitness(vector<T>&)
	Class Optimize
	Private:
	Double x,y;
	Double a,b,c // constants
	
	Public:
	Ackleyf()
	Ackleyf(double,double)
	
	Initialize()
	Set_variables();
	Score(); //return fitness
	Solve(); //find fitness
	GetY();
	SetY();
	GetX();
	SetX();
		Class Curve
		Private:
		Double x,y;
		Double a,b,c,d;
		
		Public:
		Curve()
		curve(double,double,double,
		double)
		set_coordinates();
		initialize()
		Score(); //return fitness
		Solve(); //find fitness
		Getters and setters for –
		- a,b,c,d.

a) Pseudo-Code

- create object application declaring template class T as either an object optimize or curve.
Algorithm<optimize> Ackley; ||

- create population initialization algorithm for both optimize and curve to overwrite the pure virtual method in application.

- Initialize a population of a specific object from the application object created beforehand.

while(generations < counter && solution == false)
{
- test population fitness
- create parents using tournament selection algorithm.
- apply cross over intermediate recombination algorithm onto parents to create offspring pool.
- mutate offspring pool using real valued mutation algorithm.
- evaluate parent pool + mutated offspring pool, order this pool using fitness values.
- populate the next generation of individuals with the top percentage of the joint pool.
-re apply previous steps to next generation population.
- evaluate the next generation fitness.
- if the lowest fitness possible is found set solution to true, if not update counter.
}

Design b)

Class Optimize	Class Curve	Main

Private:
Double x,y;
Double a,b,c // constants
 	
Public:
Ackleyf()
Ackleyf(double,double)
 	
Set_variables();
Score(); //return fitness
Solve(); //find fitness
GetY();
SetY();
GetX();
SetX();
	Private:
	Double x,y;
	Double a,b,c,d;
	
	Public:
	Curve()
	curve(double,double,double,
	double)
	set_coordinates();
	Score(); //return fitness
	Solve(); //find fitness
	Getters and setters for –
	- a,b,c,d.
		void initialize();
		void tournament();
		void generate_next();
		void mutate(vector<optimize>&);
		void mutate(vector<curve>&);
		void crossover();
		void fitness(vector< optimize >&)
		void fitness(vector< curve >&)
		
		int generation, counter;
		double fitness, score;
		char condition;

Design B pseudo-code

Int main()
{
- Set a condition char, one char will represent the optimization, one will represent the curve.

While(generations < counter && solution == false)
{
- Initialize both populations with an initialize method in main. The initialize method will use two different algorithms one for the optimization one for the curve fitting. We can decide between these two algorithms in this method using the condition char.

For(int i=0; i<population.size(); i++)
{
- Use the fitness method to call a vector of optimize or curve objects. In their respective methods the object will be called and given a fitness based on their own method “solve”.

- call tournament selection on the desired population to create parent pool, both optimize and curve fitting have the same algorithm.
}

- evaluate parent pool fitness. – fitness(parents);

- apply intermediate recombination cross over algorithm to parents, offspring pool is now created.

- mutate offspring pool – mutate(pool);

- followed by re-evaluation of the pool. – fitness(pool);

- add parents to the pool of offspring using a join method. – join();

- re-evaluate pool again, proceed to create next generation and apply previous steps until loop condition is met.

}

Pros and Cons of each design

Design A

Pros:

- Using templates allows easier access to both class optimize and curve in the main.
- The virtual functions show clearly the similarities between the classes.
- The pure virtual function shows the similar functions that are used between both classes however demonstrate how they are implemented differently.
- Organized structure.

Cons:

- Limits the diversification in the class. The non-virtual functions must be the same therefore it must account for both classes. Limits the design of the method, and does not allow for class specific functions and variables to be used.
- Template classes may not be inherited, this is not too large of a problem since the class T can be set to a specific object thus giving it access to all members and methods in that class.

- Templates are unnecessary as in this method you will have two separate applications, one for optimization and one for curves. Instead of having “algorithm optimize” and “algorithm curve” we can simply have optimize O; && curve C;

Design B

Pros:

- Having your methods in the main allows you to manipulate the variables found in the main as well as the variables found within your classes.
- Easier to implement individual methods. (methods specific to either optimize or curve).
- The classes are distinct therefore the genetic algorithm may be uniquely implemented onto each class.
- Capability of manipulating variables in ways that you may not in a class. For example, creating an initial population of objects simply by setting the size of a vector holding those objects.
  Ex: vector <optimize> population(size);

Cons:

- In this method variables are general, not unique. Meaning every object has it's specific set of variables however if everything is main we must only have general variables, or variables that may be used by all.
- The methods are unnecessarily repeated, for example to fitness functions in the main just taking in different variables.
- The classes do not have access to the variables in the main therefore the class cannot directly manipulate some variables.

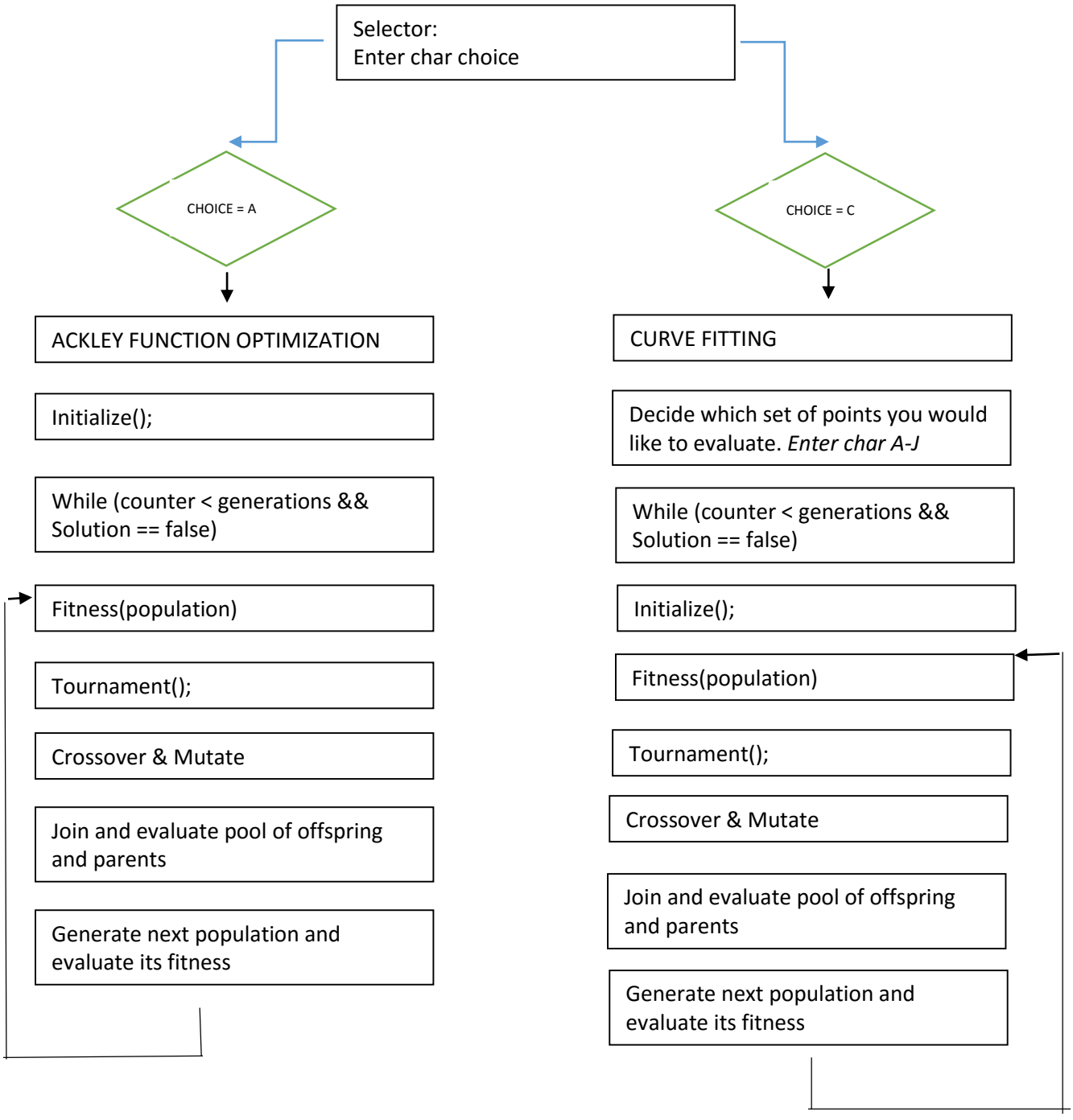
Chosen method

The method that I have chosen to implement my code is not the best, nor it is my preferred however I feel that it simply gets the job done and would be even better when trying to implement a third or fourth application. Instead of having a common class between the optimization and curve fitting class I decided to just separate the two completely. Having the two separate classes with each their own functions as they are technically different, in the way they are solved, modified and implemented with the genetic algorithm. Although we are using a single method of genetic algorithm for both classes, my implementation of the genetic algorithm requires many specifications of a class. This makes it hard to implement a virtual or pure virtual method because I wanted things done a certain way.

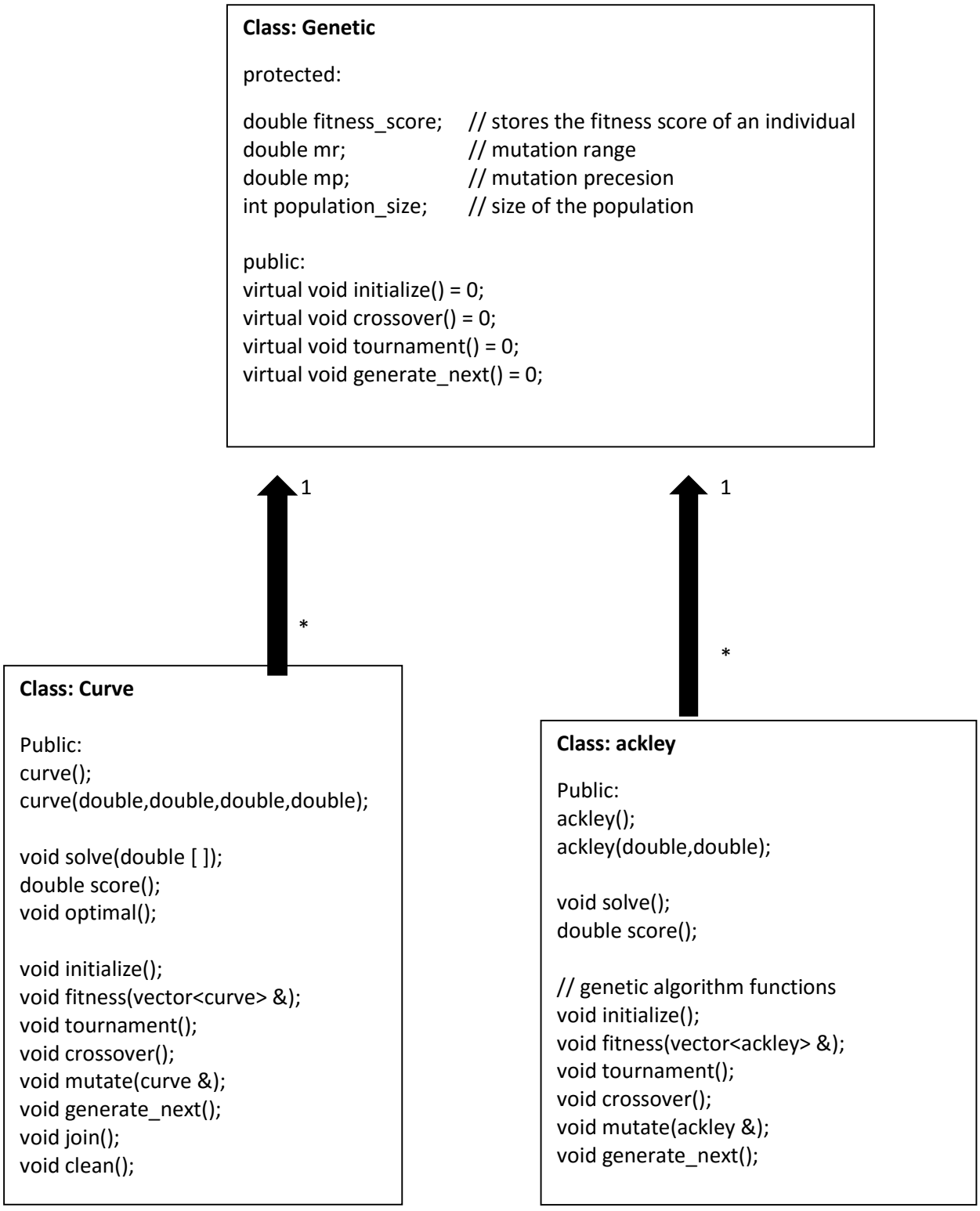
Another large reason for removing the template base class was due to the fact that it was presenting errors. This was apparently due to the compiler I was using yet I was unaware at the time. The issue I encountered was setting an initial size to a vector in the class. Since I could not implement initialized vectors I had to improvise and dynamically create arrays. However these arrays would lose information or cause memory leak else where. I realized that even though I could implement templates and abstract classes they were not necessary and developing the classes separately and in the main would distinguish them from the genetic algorithm. By being completely apart from the genetic algorithm, I can easily implement more applications without making changes to the currently existing code. The method chosen allowed for some vector, array, and memory manipulation that I was not able to get away with while working with abstract classes and templates. My method takes a bit from both design a and b. The functions for the genetic algorithm are declared in the classes themselves, in the main I included variables that should remain constant. I am simply using the main to call each class one a time to complete the full genetic algorithm.

Design Description:

Flow Chart



My design : Class Diagram



Pseudo-code for member functions

Solve()
{
Depending on the application this function will be created accordingly.

Optimization : solve the function f(x,y) given the individual's variables. Store f(x,y) as fitness score.

Curve Fitting: solve the 3rd degree polynomial get value for f(x). Use f(x) in the MSE equation. Store the MSE as fitness score.
}

```
Double score(…)
{
        Return fitness score;
}

Void initialize()
{

Optimization: randomly generate x and y. store them in an optimization population
Curve: randomly generate a,b,c,d. store them in a curve population.
}

Void fitness(..)
{
For entire population ( optimization or curve)
        Solve();
}


Void tournament()
{
        //Create parents, certain portion of initial population.
        Randomly create 3 ids, each id corresponds to a parent.
        See which parent of the three have the best fitness score.
        If id1.score > id2.score && id1.score > id2.score
        Return id1; etc…
}

Void crossover()
{
        Take two parents at random.
        Create crossover coefficients for each variable.
        Apply intermediate recombination algorithm onto each variable of two parents.

        For optimization return new x and new y value
```

```
        For Curve fitting return a new a,b,c,d values.
}

Void mutate()
{
        Optimization: Apply real valued mutation onto X and Y.  set new x and y values.
        Curve fitting: apply real valued mutation onto a,b,c,d. set new a,b,c,d.
}

Void generate_next()
{
        Join the parent pool and offspring pool.
        Evaluate fitness of every individual in newly created pool.
        Create a new vector storing the fitness value of every individual in the pool.
        Sort this vector so that the first index will point to the individual with the greatest score.

        While(next population size < initial population size)
        {
                Algorithm to match fitness value with individual.
                Once individual is found, push_back into next population.
        }
}

Void clean()
{
        Erase all vectors, since I used push back and only vectors I am not reassigned memory but asking
for more memory each time. After each generation I should clean the vectors as the information in the
past generation will no longer be needed.

Vector.erase(begin,end);
}
```

TESTING

A →  generation:
Fitness, average fitness ….

This code does not require much user input. The user must first simply decide between two things, whether they would like to solve the Ackley optimization equation or perform a curve fitting, this is taken cared of using a simple character, ‘a’ for Ackley, ‘c’ for curve. The Ackley function does not

change much each time as only two variables change x and y, it always gives back a solution. All methods inside the Ackley class has been tested in independence and unison, with different mutation rates and a larger domain of x and y. The equation will always find a correct answer within a few generations.

The curve fitting has a lot more variety to it, fitting a perfect 3rd degree polynomial to a set of points is rather difficult. Having a single variable off with cause another variable to be off as well. The mean square error also has it's issues as some a,b,c,d values will give a low MSE but might be slightly different then the a,b,c,d values desired. To ensure that the curve fitting is fully functional it is tested with 10 different sets of points chosen by the user. The sets range from having negative numbers, fractions, same numbers everything has been tested. Sometimes the genetic algorithm will be unlucky and generate numbers that are slightly off. If it were to continue for much longer it would eventually find a solution however I have limited the generations for marking purposes. It may take to or three tries for the more complex polynomial equations to be solved.