

## CS 205

### Evolutionary Computation Notes

#### Introduction

Evolutionary computation was conceived and articulated in the 1960's as a method of solving otherwise intractable problems. Computer programs based on evolutionary techniques typically consume lots of computer resources and until the 1990's the technique was implemented only by the few who had access to those resources. Over the last ten years, computers have become fast enough and enough memory has become cheaply available that evolutionary computation is now a workable technique for a typical desktop computer running a program for several hours or over a weekend.

Evolutionary computation is a powerful technique and has been applied to a variety of problems including electric circuit design, time-optimal control circuits, quantum computer algorithms, robotics and many others. The algorithms used to implement evolutionary computation are not complex and are accessible to the typical undergraduate.

#### Terminology

Evolutionary computation is based on Darwinian evolution and much of the terminology originates with terms defined in biological systems. The field is still very new and many of the definitions used here do not rigidly apply but are instead "consensus" definitions gleaned from other sources.

*Evolutionary Algorithm* – (EA) a generic term used for all programming methods which rely on evolution to achieve a stated goal.

*Evolutionary Computation* – (EC) any computation that makes use of evolutionary algorithms.

*Evolutionary Programming* – (EP) an evolutionary algorithm which uses mutation but not crossover to achieve a goal.

*Evolutionary Strategy* – (ES) an evolutionary algorithm which uses mutation and crossover to achieve a goal.

*Genetic Algorithm* – (GA) a generic term often used interchangeably with evolutionary algorithm. Some use the term "genetic algorithm" to mean those evolutionary algorithms which rely most heavily on crossover, and in which mutation is not used or plays a minor role.

*Genetic Programming* – (GP) evolutionary techniques are applied to a problem to create a program. In genetic programming the output is a program, typically in Lisp, that has evolved to solve a particular problem.

*Fitness* – a measure or assessment of the quality of an individual item in a population as compared to an ideal. In an evolutionary algorithm, only the fittest individuals survive to the next generation.

*Mutation* – a random change in some portion of an individual's make up.

*Growth* – a small change in an individual directed toward a better fit to a goal.

*Crossover* – similar to biological reproduction. Characteristics of two or more parent individuals are combined by some algorithm to form a new individual.

## How It Works

An evolutionary algorithm works in the following manner:

- An original population is selected. This is typically done in a random fashion so that each member of this population represents a solution to the problem. This original population is said to be the first generation.
- The population is assessed to determine each member's fitness. In a typical program the population is sorted with the best at the top.
- After sorting, parents are selected from the population. The parents are those members which are most fit. For example the top 30% may be chosen as parents.
- Crossover is performed using the parents to generate children. For example, two randomly selected parents may be chosen with parts of each going together to form a child. This process continues until all of the desired children are produced. If, say 30% of the population are parents then the remaining 70% may be replaced with new children.
- Mutation is performed. Mutation involves small random changes to certain elements of a member of the population without regard to whether or not those changes are beneficial. Typically some small percentage of the population is involved.
- Growth operations may be performed. Growth allows some small group of the population to change in a positive fashion. Growth is similar to mutation except that for growth, members are assessed immediately after a change and only positive changes are allowed.
- The population with the survivors (typically the parents) and the new children become the next generation.
- The process is repeated many times until some time limit expires or until some member of the population achieves a desired assessed value.

## A Simple Example: Generating Phrases

As a purely academic example, consider the problem of using an evolutionary algorithm to generate a phrase such as "Twas brillig, and the slithy toves". We begin by selecting an original population of say 1,000 randomly generated phrases. For simplicity we will specify that each randomly generated phrase is the same length as the ideal phrase or 33 characters (counting the spaces as characters but not the comma). To assess the fitness of each phrase we sum up the differences between each character of a given phrase and the corresponding character in the ideal phrase using ASCII values. A partial list of the population and the error numbers might look like that shown in Figure 1.

If we choose the top 30% to be parents then we have 300 parents and 700 children. We choose two random parents to generate each child phrase. For example, referring to Figure 1, if we take strings 1 and 5 to be parents the two parents are

```
1 OYBV DWRTFJGLRRMFQEM AIDEMZMNUGCY
5 VCJEHISELMDA GYASODKHIBOFIUMXRJHP
```

No.	Err	String
1	295	OYBV DWRTFJGLRRMFQEM AIDEMZMNUGCY
2	337	BUMNKWELEEMGNENJATGGXWVGWHT NRPRU
3	338	XABP HPFZIBSGBDQYVAXJRFKYIF QOFUU
4	343	ULXSAPQWGLCMHNFU VKIDITVXHXJSNQVW
5	350	VCJEHISELMDA GYASODKHIBOFIUMXRJHP
..	..	...
996	649	EE QTLEDBHDLKPH CDNHF M NYGHEH
997	650	UZOZNTFRYM FDS REETOYYK PQNOO OE
998	662	VKHQUSDUGUYIKGHBH V YNPR YTFD E L
999	674	BUBLPFQX EZWRUOUEJDHASXYPURVC SB
1000	678	V WMGM BDAHLKPKXRWHVN VSCWKUSU U

**Figure 1**

The original population of 1,000 randomly generated strings. The error number is generated by subtracting ASCII values of each string from those of the ideal string and summing up the absolute value of the differences. The character set consists only of the capital letters plus the space character.

Next we randomly choose a substring from each parent. Suppose we take the substring YBV DWRTFJGLRR from parent 1 and the substring BOFIUM from parent 5. We then choose an existing child, say string 999 in Figure 1, and substitute the two substrings from the parents in a corresponding positions as shown in Figure 2. This forms a new child string. This process is continued until all 700 children are generated.

999	674	BUBLPFQX EZWRUOUEJDHASXYPURVC SB	<b>Original child string</b>
		↑	
1	295	OYBV DWRTFJGLRRMFQEM AIDEMZMNUGCY	<b>Parent string 1</b>
		↑	
5	350	VCJEHISELMDA GYASODKHIBOFIUMXRJHP	<b>Parent string 5</b>
		YBV DWRTFJGLRR OUEJDHABOFIUMVC SB	<b>New child string</b>

**Figure 2**

Substrings from parent strings 1 and 5 are chosen to fit into child string 999 to form a new child string.

After all of the children are generated by the crossover operation we select a small population for mutation. In mutation we change a small subset of letters in a child in a random fashion. The mutation may result in the child having a better or worse fit and there is no attempt made to do assessment of the mutation results at this time. Typically mutation is done to less than 5% of the population.

The process of growth is similar to mutation. Mutation is done to children but growth is applied to the surviving parents (the parents that have reproduced and will go on to the next generation). In growth we make a random change but we immediately assess its impact by determining whether the error became larger or smaller. The result of the growth is kept only if it leads to a smaller error value for the parent string.

The operations of fitness sorting, crossover, mutation, and growth are applied to the population in a given generation to form the next generation. The process is repeated for thousands (or perhaps millions) of generations. The operations stop when a given amount of time has passed or when the error is sufficiently small.

***Class project: Genetic Phrase Generator***

Download the C# project file *GeneticPhrase.zip* from the web site. Unzip this file, load it into Visual Studio and verify that it works. Note that you can enter your own phrase which it will attempt to duplicate, or, if you don't enter a phrase the program will choose one for you at random.

GeneticPhrase has both crossover and mutation but it does not have growth. We are going to modify the crossover, mutation, and error methods.

**Mutation modification:**

There is an ArrayList structure named *phArr*, that holds all of the phrases for one generation. This list is sorted before the Mutation method runs so that the phrases with the lowest error numbers are at the lowest index in *phArr*. The phrases can be broken into two groups called "parents" and "children" where parents have the smallest error numbers. The user sets the percent parents and this is available as the variable called *parents*. The number of children is the total number of phrases minus the parents. Mutation mutates only the children and it chooses these at random.

Modify this routine so that after it does a mutation it checks to see if the mutation decreased the error. If so, keep the mutation. Otherwise, don't keep the mutated string but retain the original.

**Crossover modification:**

The current crossover method choose two random parents and picks a substring out of the first half of the first parent and the second half of the second parent. It then places these substrings into the child. Alter the crossover program to do the following:

- Instead of having the first parent randomly chosen alter the program so that the first parent is the one with the lowest error. The second parent may be randomly chosen but should not be the same as the first parent.
- After a new child string is formed get its error and replace the current child with the new child string only if the new child has a smaller error.

**Error:**

The error method simply adds up the absolute value of the difference between corresponding characters. Modify this method so that the error is square root of the sum of the squares of the distance between characters.