Project C: Genetic Algorithms

Due Wednesday April 13th 2005 at 8pm.

A genetic algorithm (GA) is an "evolutionary programming" technique used to solve complex minimization/maximization problems. The technique has been used in the fields of bioinformatics (protein folding), industrial engineering (factory floor layout, scheduling), computer engineering (test vector generation, circuit delay minimization) and physics (minimum energy state computations) as well as many many other fields. This university boasts having one of the creators of GAs on its faculty (not in Computer Science interestingly enough) and a research group in Mechanical Engineering that has used them on a regular basis.

Background

The idea behind genetic algorithms, and in fact the whole class of evolutionary programming techniques, is to model computation around the way that nature finds "fit" species. In the case of genetic algorithms this is done by:

1. Creating a set of "chromosomes" (a series of bits) each of which encodes *a* solution to a problem.
2. Creating an "evaluation function" which measures the "fitness" of a specific solution.
3. Creating a new generation of chromosomes from the old generation. This is done by:
   a) Selecting a pair of chromosomes from the old generation. The higher the fitness rating of the chromosome the more likely it is to be selected. These two chromosomes will be the "parents" of a new chromosome.
   b) Causing "crossover" to occur between the two chromosomes. In general this means taking some of the bits from both parents and using them to create a new chromosome (the child).
4. Allowing some chance (usually very small) of mutation in the children. That is, some chance a bit or bits will be flipped in a child. After this we go back to step 2.
5. Steps 3 and 4 are done until the new generation is as big as the old.

Because parents are chosen on the basis of "fitness," the general trend should be improved fitness over time.

Example

Consider the (trivial) problem of finding the maximum value of x for the following formula: $y=200-(x-6)^2$ where x can be any integer from 0 to 31. Obviously the maximum is at x=6, where y=300. Even if we couldn't do the math by hand, it would be easy to try each value of x from 0 to 31. None-the-less, we will use this example to illustrate how a GA might work.

First we create a set of chromosomes that encodes all possible answers. In this case we can represent all possible answers using a 5 bit value and treating it as an unsigned binary number. So let's say we start with the following chromosomes:

```
10100
01000
01001
00010
```

These are 20, 8, 9 and 2 respectively. We then use our evaluation function, which in this case is simply the formula $200-(x-6)^2$ to arrive at fitness values of 4, 196, 191, and 184. We will then create 4 children to replace this generation. To create a child we select two parents (chromosomes with higher fitness values are more likely to be selected). Say for the first child we chose parents 01000 and 00010. We then (randomly) choose a crossover point. In this case we are generating a number from 0 to 5 (the number of bits in the chromosomes). Say we choose 2. In that case we will take the first 2 bits from the first parent (01) and the rest of the bits from the second parent (010) giving us a child of 01010. As it turns out, this child will be worse (have a lower evaluation function) than either of its parents. The same algorithm is carried out to generate the rest of the new generation. For each child in this new generation we select two parents and perform crossover as described above. Say we end up with the following chromosomes.

```
01010
00010
00001
01000
```

Then we consider mutation, where we *may* flip one or more bits randomly. In this case let's say we flip the 4th bit of the 3rd string. We now have:

```
01010
00010
00011
01000
```

We then perform the evaluation function on each bit string and start over. The following is a real run by a GA working on this problem:

01101 score=151
00000 score=164
11111 score=0
01001 score=191
Round 1:
00000 score=164
01000 score=196
01001 score=191
01001 score=191
Round 2:
01001 score=191
01000 score=196
01000 score=196
00000 score=164
Round 3:
00000 score=164
01000 score=196

00000 score=164
00001 score=175
Round 4:
00000 score=164
00000 score=164
00001 score=175
00101 score=199
Round 5:
00001 score=175
00000 score=164
00101 score=199
00010 score=184

While it hadn't solved the problem exactly, it did find a fairly good solution (x=5 for a score of 199). In general, GAs cannot be relied upon to solve a problem exactly. In general you use a GA when you are looking for a good (but not perfect) solution over a large search space (where the chromosomes are at least 40 bits and usually 100's for real problems).

Algorithm in detail

There are three main steps: selection, crossover and mutation. While there are many ways to do this in general, we provide a specific algorithm for each step. You are to use the algorithms provided.

Selection

When doing crossover, two parents need to be selected. Say there are N chromosomes in a given generation and each chromosome has a its fitness evaluated as F_i. Let $F = \sum_{i=1}^{N} F_i$ (in other words, F is the sum of all the fitness scores for a given generation). Then the probability that the *i*th chromosome will be selected as a parent at any given time is F_i/F. As each time a parent is needed a new one is randomly selected, a given chromosome may be used as both parents for a given child.

In the original example where the fitness values were 4, 196, 191, and 184; we would state that

Name	**Chromosome**	**Fitness**	**Probability of Selection**
F_1	10100	4	4/575
F_2	01000	196	196/575
F_3	01001	191	191/575
F_4	00010	184	184/575
F	--------	**575**	**------**

Note, that in effect this requires that no fitness value be negative. As such any negative fitness value should be treated as some small constant (zero or one) in the evaluation function.

Crossover

Once both parents are selected, the next step is to perform the crossover. Assume there are M bits in each chromosome. When doing this a uniformly random (that is each possibility is equally likely) value from 0 to M should be selected. Call that number X. The first parent should contribute the first X bits to the child, while the second contributes the remaining bits. Note that bits don't move. So if a bit was in the 3rd position in a parent it should either not be included or be in the 3rd position in the child.

Say that the chromosome 01000 is selected as the first parent and 10001 is selected as the second parent. If X=2 the child would be 01001. If X=1 the child would be 00001.

Mutation

Mutation is very important to getting a GA to work correctly. At the same time, it is vital that the amount of mutation be very small. On the average mutation should be occurring no more than once or perhaps twice a generation. Averages of less than one tend to work best. Each bit in the entire generation should have an equal probability of being flipped. Details on how to implement mutation are found below.

Coding

The following lists a number of requirements your code must satisfy:

1. You are to represent your chromosomes as an array of characters, where each character is either a zero or one (*not* the ASCII values for 0 and 1).

2. You are to have a file named "eval.h". It should consist of *only* the following (i.e. don't add any other constants, function declarations or anything else to this header file). Your program is to include this header file (**`#include"eval.h"`**):

```
const int POP_SIZE=30;
const int CHROM_LENGTH=58;
const double MUTE_PROB=.05;
const int MUTE_TRIES=10;
const int ROUNDS=200;
const int INIT=0;

double evalfunction(char chrom[]);
void initChrom(char c[POP_SIZE][CHROM_LENGTH]);
void printResult(char chrom[]);
```

   You may change the values of the constants as desired. In fact your program should work for a reasonable range of values. The constants have the following meanings:

   - POP_SIZE is the number of chromosomes in a given generation.
   - CHROM_LENGTH is the number of bits in each chromosome.

- ROUNDS is the number of generations your program is to run.
- When doing mutation you will attempt to mutate a number of times equal to MUTE_TRIES. For each try you will only mutate with a probability equal to MUTE_PROB. So for the given values you are to consider mutating 10 times per generation, but there is only a 5% chance a mutation will occur for each try. (So the average number of mutations per generation will be 0.5 using these numbers).
- If INIT is 1 you are to initialize your chromosomes by calling initChrom(). Otherwise you are to randomly initialize each character to either a 1 or a 0.

3. You are to have a file named "eval.cc" which acts as the "driver" for your program. This means that it is the code in eval.cc that is changed in order to solve a certain problem using your GA. Your main code, once correctly written, should not need to be changed at all to solve a different problem. *We have provided a couple of sample eval.cc files for you to use.* This file is to include the following functions:

   a) To evaluate the fitness of a chromosome you are to use the function:
      double evalfunction(char chrom[]);
      It takes a chromosome as input and returns a fitness value (higher is better).

   b) If INIT is 1 you are to use the following function to initialize your chromosomes
      void initChrom(char c[POP_SIZE][CHROM_LENGTH]);
      Otherwise your are to randomly initialize each bit of each chromosome to zero or one.

   c) When your program finishes it should display the best result found (highest fitness) *during the entirety of the program*. The function
      void printResult(char chrom[])
      should nicely print the result in a readable format. *This function must be called at the end of your program*, printing the best chromosome found ***during the whole run of the program***. That is to say you will need to keep track of the most fit chromosome encountered during the run of the program, not just those in the last generation.

   Again, we have provided a couple of sample eval.cc files for your use (and to provide guidance for writing your own).

4. Your code should take a single, optional, command line argument. A single integer value should be used which is to be used as an argument to seed the function srand(). This can be accomplished as follows:

```
if(argc>1)
    srand(atoi(argv[1]));
```

5. You are to use the rand() function for all your random numbers.

Evaluation code

On the class webpage you will find a number of eval.cc files you can use to solve various problems. You are to write two additional ones.

The first should find the maximum of the following equation:

(100-X)*(X-50)+(Y*Z/X)-(Y*Y)-(Z*Z)+min(X*X, Y*Z).

Where X is an integer in the range of 1 to 256 while Y and Z are integers in the range of 1 to 64.

The second should evaluate the following:
A certain small urban airport can have only 2 planes unloading at the same time. There are 28 potential incoming flights that could be scheduled in a 90-minute block on a given morning (any flights not scheduled will be sent to a nearby airport). Each flight has an amount of time it will take to unload its passengers as well as an amount of money that will be made by the airport for accepting the flight. You are to write an evaluation function that uses as its fitness function how much money the airport would make by accepting a certain set of flights. If that set of flights would take more then 90 minutes you are to provide a fitness function of 1. Otherwise the fitness should be the amount of money made.
The values you should use are:

```
const int value[28]={1,  1,  2,  3,  5,  5,  5,  5,  6,  7,  7,  8,  8,  8,
                     9,  9,  10, 11, 11, 12, 12, 12, 12, 13, 15, 15, 20, 21};
const int time[28] ={5,  7,  6,  15, 16, 10, 12, 14, 15, 20, 22, 18, 30, 22,
                     23, 23, 30, 22, 18, 10, 19, 35, 28, 10, 11, 19, 34, 35};
```

Where this indicates the first flight has a value of 1 and takes 5 minutes to unload.

Handing it in

You are to create a directory called PC in your engin101 space. That directory should include the following files:

- evalA.cc (The eval file for solving the equation provided above)
- evalB.cc (The eval file for solving the plane scheduling problem)
- eval.h with a CHROME_SIZE large enough for both evalA.cc and evalB.cc (it is okay to use too large of a chromosome, the extra bits are just ignored).
- ga.cc which is to be the rest of your program.

Note that things should properly compile by using “g++ ga.cc eval.cc” where eval.cc could be either of your evaluation functions or one of the ones provided by us.