Assignment 6

28 November 2012

Due Wednesday 5 December at 1am

This assignment will serve as lecture notes and a study guide, but it also contains numbered exercises for your assignment. This topic and its variants are known as *genetic algorithms, genetic programming,* or *evolutionary computation.*

Concepts from evolution & genetics

The basic idea is to apply concepts from biological evolution and genetics to artificial entities such as numerical configurations or computer programs. By applying these concepts, we can optimize for whatever 'fitness' measure we desire. The power of evolution will produce increasingly well-adapted individuals with each generation. Let's see how it works. There are two essential components to evolution: descent with modification, and selective survival.

Descent with modification

Descent just means that the *things* in our population reproduce to form more things. That is, they have *descendents.* The major way that descendents are created in biological evolution is by copying strands of DNA.

These descendents are not exact copies, but slight modifications. This occurs for two reasons: first, the copying process can have slight errors from time to time, known as *mutations.* Second, many organisms on this planet use *sexual* reproduction, which just means they have two parents instead of only one. So, their DNA – and therefore their traits – are *combinations* of the traits of their parents.

Here are some examples, using strings of nucleotides – adenine (A), cytosine (C), guanine (G), and thymine (T) – the components of DNA. First, a mutation:

```
AAGTAGCATAGACGAAGGTA → AAGTAGCAAAGACGAAGGTA
GGTCAGAGGTCGCGTGGGAC → GGTCAGCGGTCGCGTAGGAC
```

Can you spot the changes? In *recombination,* the DNA strands of two organisms (mom and dad) get combined using a technique called *cross-over.* A cross-over point is selected, and nucleotides are copied from one parent up to that point, and from the other parent thereafter.

```
mom AAGTAGCATAGACGAAGGTA
dad GGTCAGAGGTCGCGTGGGAC
cross-over^
kid AAGTAGCGGTCGCGTGGGAC
```

1. Now it's your turn. Using the two DNA strands shown below, pick a cross-over point (wherever you like) and then write down *both* of the strands that could result.

```
mom  TAGAGTAGCAGTGAGAGAGCG
dad  GGCAGATGAGGCACGATGCAG

kid1 _____________________________

kid2 _____________________________
```

Selective survival

The second essential component to evolution is selective survival – this is the brutal part! Inevitably, not all descendents will live long enough to reproduce themselves. Maybe some will get eaten by predators first; others might freeze to death because they lack features to keep them warm; others might simply be incapable of reproduction due to mutations.

So, of those that *do* live to reproduce, we can conclude that they are, on average, *more fit* (better adapted) for the tasks of living and reproducing in whatever environment they find themselves in. As the less fit individuals die off, the average fitness of the population increases. This is natural selection.

In genetic algorithms, we do something similar, but we don't necessarily measure the ability of our individuals to avoid predators and reproduce. In fact, we can apply any fitness measure we choose. So this could be called *artificial selection*.

2. Suppose we have a population of individuals whose fitness measures are as follows:

```
34  18  19  21  40   8  17  35  28  22
```

   a. Compute the *average* fitness of this population.
   b. Delete (kill off) the least-fit half of these numbers, and then compute the average fitness of those that remain.

Chance plays a huge role in evolution – mutations occur randomly, and cross-over points are chosen randomly – but survival is also due to chance. Sometimes relatively 'unfit' individuals will be lucky enough to reproduce, and sometimes extremely 'fit' ones will die young. The fitness has only a probabilistic influence on outcome. This makes question 2, where you killed off the bottom half, somewhat unrealistic. Let's try it again, with chance.

3. Here are six individuals and their fitness scores, in the initial generation.

```
Generation 1    Generation 2
  (1) 21          (1) _____
  (2) 17          (2) _____
  (3) 38          (3) _____
  (4)  9          (4) _____
  (5)  2          (5) _____
  (6) 25          (6) _____
AVG 18.67       AVG _____
```

   Roll two six-sided dice; this selects two individuals from generation 1. (Suppose, for example, that you rolled 2 and 4.) Compare the fitness of those two individuals (17 and 9 in our example) and write the larger one (17) in space #1 of generation 2. If you roll the same number twice, that's fine; just write its fitness in the next space. Repeat this to fill in all the spaces, and then compute the average fitness for generation 2. It still increased, didn't it? This process is called *tournament selection.*

Evolving bit strings – the knapsack problem

In genetic algorithms, we're not concerned with actual genes made of DNA but with solutions to problems. Here's a problem that is very well-suited to genetic algorithms. It's called the *knapsack* problem.

Imagine you're a traveler or a merchant, and you need to take some valuables with you on your travels. You have many things you may want to take, and they have varying worth, but they won't all fit in your suitcase (or aboard your ship, or whatever). To make this simple, let's give each item a value (in dollars) and a weight (in kilograms or pounds). Like this:

item	value($)	weight(kg)
A	50	20
B	25	5
C	35	10
D	30	15

Now, if your bag can carry only 20kg, which items would you select, to maximize the value of the items you are carrying? Here are some possibilities:

- Carry just item A, and nothing else. Value is $50.
- Carry B and D. Value is $55.
- Carry C and D… oops, that puts you over the weight limit.
- Carry B and C. Value is $60. That's the best choice, right?

The only way to get the absolute best answer is to check all the possibilities. But this can be a problem because, if there are n items, then there are 2^n combinations to check, which can be enormous. Let's see how to set this up as a genetic algorithm.

Representing individuals

We can represent solutions to the problem just as strings of bits. For example, the string `1011` would represent an attempt to pack A,C,D and omit B. If there are 10 items to consider, then a string like `1101011001` would tell me what items to take (the `1`s) and which to leave (the `0`s).

Next we need to define what we mean by mutation and cross-over on these representations. For strings like these, it is very easy because they're already similar to DNA. Mutation means we just randomly flip a bit, so `1011` could become `1001`. For cross-over, we choose a random point and combine them, so `1011` and `1100` could cross-over to become `1000`, for example.

4. List *all* the possible bit strings that could result from a cross-over of `1011` and `1100`. Remember that cross-over can occur anywhere.

Defining fitness

Next, we need a way to measure how fit (how good) a candidate is. For the knapsack problem this is pretty easy: just add up the total value of the items chosen, but if it exceeds the weight limit then reset the score to zero. Here is what I mean, using the same 4 items from the table on the previous page.

candidate	total value	total weight	fitness
1000	50	20	50
0101	55	20	55
0011	65	25	0

5. Add the candidates `0010`, `0111`, and `1001` to the table above, and fill out the rest of the entries for them.
6. Using the same four items, suppose the weight limit is increased to 30kg; now what is the best solution to the problem?

Applying GA to a larger problem

Open any file in your `cs102` folder and Sync with Git. Now you should find `cs102/a6/knapsack.cpp`. Open that file. Starting around line 15, you see an array of `items`:

```
item items[] = {
    {5 , 10},    {8 , 20},    {4 ,  5},    {7 , 30},
    {5 , 10},    {8 , 13},    {4 , 10},    {5 , 10},
    {4 ,  5},    {7 , 30},    {5 , 10},    {8 , 13},
// etc.
```

In each pair, the first number is the weight, and the second is the monetary value. There are 32 items here, which means more than *4 billion* possible combinations! The constant `CAPACITY` below gives the weight limit on the items we can take. Our GA will find a solution to this knapsack problem.

Run the program. It should give some details of the first generation, which was chosen randomly (your values may differ):

```
Generation 0
  average fitness 16.31 (min 0, max 179)
  {0 1 3 6 7 9 11 12 14 22 24 28 } weight 68
```

We have initialized the GA with a population of 200 (`POPULATION`) individuals, and measured their average, minimum, and maximum fitness. No mutation or cross-over or selective survival have yet been applied.

The third line in the output shows the best individual found so far: it includes the items listed, has the max monetary value of $178, and weight 58 kg.

Now, ask the GA to evolve by 1 generation:

```
How many generations (0 to quit)? 1
Generation 1
  average fitness 61.415 (min 0, max 209)
  {0 3 7 9 10 20 23 31 } weight 71
```

The best candidate has improved a bit, and the *average* fitness of the whole population increased substantially. Try a few more generations:

```
How many generations (0 to quit)? 3
Generation 2
  average fitness 123.125 (min 0, max 214)
  {0 2 3 7 9 10 20 23 31 } weight 75
Generation 3
  average fitness 134.8 (min 0, max 214)
  {0 2 3 7 9 10 20 23 31 } weight 75
Generation 4
  average fitness 122.24 (min 0, max 214)
  {0 2 3 7 9 10 20 23 31 } weight 75
```

In these, the best candidate has stabilized, to the value of $214. Is that the best we can do? Keep running the GA and find out!

7. Continue running the program, and as the generations go by, fitness will continue to improve. Eventually, you should see the best score stay the same for several consecutive generations; this probably means we have hit the optimum solution to this problem. What is the solution? How many generations did it take to get there?

8. Take a look at the best solution found, correlating it with the values in the `items` array and verify by hand that the value of this set of items is indeed equal to the best fitness score achieved. Then, compute the total weight of the selected items. How close did it come to the weight limit?
9. Change the `CAPACITY` in `knapsack.cpp` to something larger, such as 80 or 100. Then re-run all of the above. This time, record the average and best fitness at each generation in a spreadsheet. Create a line chart showing them, with generation number across the horizontal (x) axis, and fitness on the vertical (y) axis. How many generations did it take to find the solution?