Assignment 1 CS472-F16 Due: Wed Feb 4, 2015 at 5pm PT

Points: 120

Reference Files (these should be live links):

- makefile template
- rand.cpp A simple fast random number generator
- rand.h
- bitHelpers.cpp Helpful example bit functions
- bitHelpers.h
- bitOps.html Tutorial on bit operations
- make.html Tutorial on make and makefiles
- basicUnix.html Tutorial on basic UNIX commands including the tar command
- mapFunctions.cpp functions for mapping a number from one range to another. See comments.

WARNING: It is surprisingly easy to miscode this, so read carefully, code carefully.

This assignment will get us familiar with navigating fitness spaces, the basics of local search, and what can go right and wrong during a search. The code isn't very complex, but it is parameterized so when run on the commandline you can test several approaches to maximizing the fitness. Bit manipulation might be new to you so you might practice writing some simple expressions and seeing what you get before you write the bit extractions etc. just to be sure you understand what is going on.

Few of our assignments use bits directly like this so don't worry.

Here are the details:

1 Genotype to Phenotype Mapping

The genotype, representing our chromosome, will be the least significant 20 bits of an `unsigned long long int`. In this case, by changing the genotype/phenotype mapping, we will be altering the meaning of the bits in the chromosome.

In general, the genotype to phenotype mapping will take the 20 least significant bits (LSB) and use them to generate an x and y. We will try the two genotype/phenotype mappings and three mutation functions that follow.

1.1 The Two Genotype/Phenotype Mappings

1.1.1 Plain Mapping

Extract the two 10 bit strings from the 20 LSB breaking it into two 10 bit numbers. The numbers should be considered to just be a binary number in the range 0 to $2^{10} - 1$. Hint: you can use a simple bit mask to get the integer value of x and a shift to get the value of y.

1.1.2 Gray Code Mapping

In this mapping we will extract the two 10 bit strings as above, but we deGray each of the 10 bit strings. Then we map them to the real number ranges as described below.

2 The Fitness

The chromosome is decoded as either two 10 bit binary numbers directly or as two Gray coded binary numbers by deGraying them. The resulting numbers are in the range 0 to $2^{10} - 1$. Next, scale that numbers to fit into the ranges for real values x' and y'. Specifically, We take the 10 bits and linearly convert that to a double in the range 0.0 to 10.0 called x'. This means that if all 10 bits are zeros will be $x' = 0.0$ and all 1's will be $x' = 10.0$. Take the next 10 most least significant bits and linearly convert that to a double in the range -10.0 to 10.0 called y' . All 10 bits zeros will be -10.0 and all 1's will be +10.0. We talk about scaling in class.

The fitness will be used to evaluate the quality of the x and y will be this "simple" single peak fitness function:

$$f(x', y') = \frac{1.0}{(x' - 1.0)^2 + (y' - 3.0)^2 + 1.0} \quad \text{for } x \in [0, 10] \text{ and } y \in [-10, 10]$$

A test point is $f(5, 1) = .047619$.

2.1 The Three Mutation Operators

2.1.1 Random Jump

This is the random jump mutation operator. Create a mutation operator that takes the last chromosome (unsigned long long int) and the size of the chromosome in bits and **ignores the chromosome input** and returns a random bit string of the given size.

2.1.2 Bit Flip

This is a mutation operator that randomly flips exactly one bit in the 20 bit chromosome.

2.1.3 Single Dimension Inc/Dec

This is a mutation operator that either increments or decrements **one of the fields** of 10 bits fields for x or for y. That is one of four possible changes. This must be done **very carefully** since the field must **wrap within each 10 bit field** and must not carry over into the adjacent field. This might be best done by extracting the two 10 bit fields and then inc or dec one of them and wrapping the result. Then reassemble the 20 bit chromosome using shift and or operators.

3 The Experiments

The experiments will use a simple local search algorithm but vary the genotype/phenotype mappings and mutation operators from experiment to experiment.

The algorithm will be a loop waiting until the number of fitness evaluations exceeds 10000. This is an important limit since some of these simple algorithms will run forever without it!!! Having a limit on the number of evaluations is also a good thing to do in any stochastic algorithm.

Each time through the loop it will mutate and evaluate the fitness. If the fitness is strictly greater than the old best fitness, the algorithm will remember the new fitness as best fitness and the new chromosome as best seen so far. If it is not strictly greater than the old best fitness, it does not replace the best fitness. Your algorithm must keep track of the number of fitness evaluations and the number of times an improved chromosome is found. It must also remember the number of fitness evaluations when it found the last best chromosome. The algorithm starts with a random 20 bit string. The algorithm is sketched below. Note: don't recompute the fitness of X in the if statement. This is just a casual description. Later in the semester, fitness computations may be very expensive.

```
initialize random number generator
initialize X
for (i=1; i<10000; i++) {
    X' = mutate(X)
    if (fitness(X')>fitness(X)) X=X'
}
```

When your algorithm has looped to the limit it will print on one line in this order:

- The number of fitness evaluations when it found the best fitness it found in the number of evaluations allowed.
- The number of improving moves made. Number of times it assigned to X
- The x' and y' that gave that fitness.
- The best fitness found.

That is 5 numbers: 2 integers and 3 doubles. For example:

```
152 23 0.997067 3.000978 0.999990
```

For each experiment your program should reinitialize and run again from a random starting point 1000 times so that we can see an average behavior.

For the experiments, you will run essentially the same program except for three different mutation operators and two genotype/phenotype mappings. You will turn in your code and a report on what you find. More on the report in a moment. When you turn in your code you will need code for each of the four experiments and a single makefile named exactly "makefile".

Here is a makefile template for assignment 1 . Here is a 64bit portable random number generator that is reasonably fast: rand.cpp , rand.h . Please use the 64 bit generator and **don't forget to initialize the generator before use.** Here are some helpful bit utility functions which you may or may not need: bitHelpers.cpp, bitHelpers.h. We will cover these in class.

3.1 Command Line Arguments

Your make should produce a program named `localsearch`. The command should take two command line arguments: The first is whether identity (0) or Gray encoding (1). The second is the mutation: random jump (0), bit flip (1), inc/dec (2). For example the call `localsearch 0 2` will not use Gray code and will use the inc/dec mutation operator.

4 Turning in Your Programs

4.1 The Code

Please tar up your code as a set of files and not a directory of a set of files. You should submit as described below:

- all of the source code necessary to build the required program. Hopefully just one or two files.
- a makefile that will build code.
- a report in pdf format named as described.

My scripts will automatically explode your tar file into a fresh directory and then execute your code with the following parameters:

```
localsearch 0 0
```

```
localsearch 1 0
localsearch 0 1
localsearch 1 1
localsearch 0 2
localsearch 1 2
```

to build the 6 experiments.

4.2 The Report

Your report should be in a pdf file named exactly yourlastname.pdf with "yourlastname" replaced by your lastname in all lower case and all special characters removed. It should say:

In an Introduction section: What is the point of the experiments? What questions are you trying to answer? In a Methods section: Briefly, what software you wrote, as required above, to answer your questions. What experiments you ran and what you observed, being sure to supply data and statistics if any. In a Conclusions section: What are your conclusions based on your observations and that answers the questions posed. [Hint: each experiment has a purpose.] Your report should include the summary statistics for your data. This should all fit on one to two pages. Be sure to comment on each of the six experiments.

4.3 Grading this Assignment

I will grade this based first be seeing that your code compiles, runs and returns a sensible answer. Please turn in code that runs. I will read the report to make sure it is clear and that you understand what happened when you ran the experiments. 50be concise and to the point.

4.4 Submission

Homework will be submitted as an uncompressed tar file to the homework submission page linked from the main class page. You can submit as many times as you like. The LAST file you submit BEFORE the deadline will be the one graded. There are no late papers. For all submissions you will receive email giving you some automated feedback on the unpacking and compiling and running of code and possibly some other things that can be autotested. I will read the results of the runs and the reports you submit.

Have fun.