Assignment 1 CS472-F20 DUE: Sun Sep 13, 2020 at 5pm PT

REWARD: 150 points ¡img src=../../Media/Icons/status2.jpg¿

Reference Files (these should be live links):

- makefile template
- rand.tar rand.zip Simple fast portable random number generators (see makefile)
- bitHelpers.cpp A Few 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. Check your work through the test system early.

This assignment will get us familiar with navigating fitness spaces, and the concept of landscape and neighborhood. We will also understand the basics of local search, and what can go right and wrong during a search. The code is parameterized via 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 programs to be sure you understand the operators before you write the program for the assignment.

We will look at the interaction with a local search between the genotype to phenotype mapping and the fitness function.

$$genotype \rightarrow phenotype \rightarrow fitness$$

1 Genotype to Phenotype Mapping

The genotype is the genetic representation for a solution to a problem. It must be able to be mapped to a phenotype representing a solution in the problem space.

The chromosome or value in our local search is the genotype. The genotype, is the least significant 20 bits (LSB) of an `unsigned long long int`. We will see that by changing the genotype to 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 real numbers x and y. We will try the two genotype/phenotype mappings and three mutation functions that follow.

You will use these two genotype to phenotype mappings in this assignment.

1.1 Binary Mapping

Extract the two 10 bit strings from the 20 bit genotype. The least significant 10 bits will represent y. The most significant 10 bits of the 20 bits will represent x. The 10 bit 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 y and a shift to get the value of x.

1.2 Gray Code Mapping

In this mapping we will extract the two 10 bit strings same as above, but then we will deGray each number. This will still result in a number in the range 0 to $2^{10} - 1$. This forces the 10 bit numbers to be considered as if they are in Gray code.

2 The Three Mutation Operators

These are operators on the genotype, although inc/dec mutate operator knows where the x and y values are. The mutate operator takes a chromosome and returns a mutated one. Be careful that you don't damage the input chromosome in computing the new one!! Common mistake.

2.1 Random Jump

This is the random jump operator. This "mutation" operator isn't. It just chromosome in bits and **ignores** it returning a random 20 bit string.

2.2 Bit Flip

This mutation operator takes a chromosome (unsigned long long int) and randomly flips exactly one bit in the chromosome. This can be done with a single bit mask and the xor operator in C++. This will create a neighborhood of 20 chromosome.

2.3 Single Dimension Inc/Dec

This is mutation operator is the trickiest. It either increments or decrements **one** of the 10 bit fields for x or for y. **It does this without Graying or deGraying!**. It is working the 10 bit fields as if they are all binary numbers that can be incremented or decremented. The increment will roll over to 0 if the increment would create 2^{10} and decrement rolls over to $2^{10} - 1$ if you decrement 0. After one of the 10 bit fields is incremented or decremented the 20 bit chromosome is reassembled. Be sure to only randomly pick one of the x or y bit fields.

3 The Phenotype to Fitness Mapping

The resulting numbers from either Binary or Gray Code process are in the range 0 to $2^{10} - 1$. Next, scale that numbers to fit into the domain ranges for real values x' and y'. Specifically, We take the bits in x 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 y 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 using the map functions.

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 to verify you have coded the function correctly is $f(5, 1) = .047619$.

4 The Experiments

The experiments will use a simple local search algorithm but vary the genotype/phenotype mappings and mutation operators from experiment to experiment. Here are the comments actually extracted from my code:

```
// init random number generator
// arg list processing
// Loop: do 1000 experiments
    // init newgene as random first point
    // set the best gene and its fitness
    // Loop: do one experiment of 10000 passes through this loop
        // extract bits from newgene
        // translate bits as binary or Gray
        // convert to doubles: x, y
        // compute fitness from x, y
        // keep the newgene if fit(newgene) is better than best
        // mutate to create newgene and loop to extract and test
    // print out stats
```

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. Note: don't compute the fitness of the best so far. Just remember its value. This is just a casual description. Later in the semester, fitness computations may be very expensive or not be a fixed value.

When your algorithm has looped to the limit it will print the following 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 the best 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. This is done 1000 times so that we can see an average behavior. There will be 1000 lines of output each with 5 numbers in it.

In testing your code will be run for three different mutation operators and two genotype/phenotype mappings controlled by the command line arguments. You will turn in your code and a report on what you find. When you turn in your code you will need code for the local search 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.tar , rand.zip. 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.

4.1 Command Line Arguments

Your make should produce a program named `localsearch`. The command should take two command line arguments: The first is whether binary (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 use the binary encoding and will use the inc/dec mutation operator.

5 Turning in Your Programs

5.1 The Code

Please tar up your code as an uncompressed tar containing 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 below.

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.

5.2 The Report

Your report should be in a pdf file named exactly "name".pdf with "name" replaced by your University of Idaho email address. It should say:

In an Introduction section: What is the point of the experiments? What questions are you trying to answer?

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. Try to get this all on one page. Be sure to comment on each of the six experiments.

5.3 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. Code that doesn't run gets almost no points. Please stick to the example format. I will take off if the format is different because it makes it easy to understand what your results are. I will read the report to make sure it is clear and that you understand

what happened when you ran the experiments. Your report needs to be concise, to the point, and readable.

5.4 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.