Introduction to Computing Science

A Genetic Algorithm for the Partition Problem

Introduction

The *Partition Problem* is the following: Given a collection of n positive integer numbers, find a pair of disjoint subsets, such that the union of these sets is the original set, under the constraint that the sums of the two subsets or equal, or alternative, that the absolute difference between the sums is minimised. This kind of optimisation problem might occur in logistics, or in load balancing in parallel computers.

An example of the problem is a box full of blocks of different heights. The aim then becomes to build two towers of equal heights (or as nearly equal as possible), without leaving out a single block.

The problem seems simple enough, but it turns out the computational complexity is huge. To ensure you find the best solution you have to test *all* 2^n solutions, which becomes impractical for fairly modest n. In this assignment we will look at a special kind of algorithm which can give a very good approximation of the solution in reasonable time. The approach is unusual, because

a. it works in a very different way from anything you have written for Imperative Programming, and

b. it does not guarantee a correct result (and might not terminate if you are not very careful).

The algorithm works by imitating evolution. First, a group of random solution is guessed. We make a random selection from this population in such a way that better solutions (according to some criterion) are favoured. We then make some minor random changes to the population, and repeat the process. After many iterations, good solutions dominate the population. In this case we could terminate once two towers of identical heights are found.

Genetic Algorithms

Anybody with a background in evolutionary biology will see the principles of Darwinian evolution by survival of the fittest (a phrase not introduced by Darwin, and one with which he was uncomfortable). Algorithms that adhere to the outline sketched above are called *genetic algorithms*.

Genetic algorithms require potential solutions to be represented as *chromosomes*. A chromosome is a row of genes, each of which encodes a parameter of the solution using binary values (as a rule). A collection of chromosomes represents the population, and the state of the chromosomes during a specific iteration is called a *generation*. Evolution is the process by which one generation is replaced by a new generation in such a way that

- better solutions are favoured,

- most of the parameter settings are passed unchanged, and,
- new solutions are created in the form of
  - random changes to parameters (mutation), and
  - random recombination of existing parameters (cross-over).

In this way, a good solution might evolve from a series of random initial guesses.

Chromosomes, genes, mutations and cross-overs

New variations are introduced into the population in two ways, as mentioned above. We will demonstrate this with a few examples. Assume that a chromosome consists of six genes, each of which is either zero or one. In this case 01100 is a possible chromosome. The simplest way to introduce random variation is by randomly picking one or more genes, and flipping the values from zero to one. This process is known as *mutation*. Suppose we mutate the 1^{st} en 4^{th} gene of the example chromosome, we end up with **1**11**1**00. If genes are allowed to have more than two values, we typically add a random number to the genes selected for mutation.

The second way in which variability is introduced is through *cross-over*. Assume we have two chromosomes 110011 and **010101**. The cross-over process picks a random location between two genes, and creates two new chromosomes from the old by cutting each in half at the selected point, and attaching the "tail" of the first chromosome to the "head" of the second, and vice-versa. Assume the cut is made between the third and fourth genes, we would end up with chromosomes 110**101** and **010**011 in our case. The first one consists of the first half of 110011 combined with the second half of **010101** and vice-versa. Another outcome could be 1**10101** and **0**10011. In this case the cut was made after the first gene. Multiple cutting points are also possible.

The Data Structures

In our case we can encode each solution by storing a boolean per block, indicating whether the block is in the right (or left) tower. The constant `numBlocks` determines how many blocks are in the collection. The size of each of the block is stored in an array of integers. Element i represents the height of block i. During evolution, these heights stay constant. Each chromosome has a gene for each block, and can therefore be stored in an array of booleans of length `numBlocks`. for clarity, we define a constant `chromLength` representing the chromosome length. In our case `chromLength` = `numBlocks`. A generation consists of `popSize` (population size) chromosomes. Constant `popSize` is also defined in advance, and a generation is therefore an array of `popSize` chromosomes. Thus we arrive at the following declarations:

```
#define numBlocks 20
    // number of blocks in the input set
#define chromlength numBlocks
    //length of chromosome
#define popSize 10
```

```
    //number of chromosomes in population

int blocks[numBlocks];
bool generation[popSize][chromLength];
```

A chromosome of individual i in the population is indicated as `generation[i]`. The j-th gene of that chromosome is `generation[i][j]`. Note that for some C-compilers you will have to add a statement

```
#include <stdbool.h>
```

for the bool type to work. This also defines keywords `true` and `false`.

Random numbers

Before we can start thinking about implementation, we need to consider how to generate (pseudo-)random numbers and bits. We use a standard library functions to do this. First declare

```
    #include<stdlib.h>
```

You can then obtain random numbers with the function `rand()`. Getting a new random number and storing it in n is achieved by

```
    n =  rand();
```

Function `rand` computes a pseudo-random number based on the previous number it generated. To initialise the sequence, we call

```
    srand(seed);
```

with `seed` some integer value. For a given seed, the sequence of repeated calls to `rand` is completely fixed (but hard to predict for mere mortals). This allows you to generate the same "random" sequence in repeated experiments, if desired. By using e.g. the time when `srand` is called as seed, the sequence becomes really unpredictable.

Note that `rand` produces number between 0 and `RAND_MAX`, a predefined constant, guaranteed to be at least 32767. What you need is random numbers between 0 and some smaller number $n - 1$. In the smallest case (for booleans) $n = 2$. One way to achieve this is using

```
  r = rand() % n;
```

Though valid, it is not a good idea to do this for small values of n. Many random-number generators will yield a sequence

0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 .....

for $n = 2$ (try it). For random booleans you can use:

```
  r = ( rand() > (RAND_MAX / 2) );
```

For $n \geq 10$ the first form is usually OK.

Assignment 1
Write a function that reads the heights of the blocks. You may assume that *precisely* `numBlocks` integers are input. This function is used to fill `blocks`. The result must be stored in this array. Note that the heights must be positive!

Assignment 2
Write a function that fills a chromosome with random boolean values. Use the method for random boolean given above.

Assignment 3
Write a function that mutates a random gene in a chromosome, using the given random-number generator. Write a second function that performs cross-over.

As said before, a chromosome represents two subsets of blocks. Let us call these the T and F sets. The T set is the set that contains those blocks with a corresponding gene value of *true* (or 1), whereas the F set contains those blocks with the gene set to *false* (or 1).

Assignment 4
Write a function:

```
int heightOfTower(bool *chromosome, bool b){
/* This function returns the height of the tower of the T set of
   chromosome if b == true, and that of the F set otherwise
*/
}
```

Assignment 5
To implement "natural selection" we will need a measure of fitness. A natural choice is the difference in the height of the towers. Write a function to compute this difference, either using the function from the previous assignment, or develop a more efficient way to do it.

Assignment 6
A chromosome for which the difference between the towers is zero is the fittest possible solution. In general, small values of difference indicate high fitness. Can you write a function to compute a good measure of fitness based on the height. Is it strictly necessary?

We have now made a large series of building blocks for our final solution. What we still need to do is write code that

- Makes a new generation from a previous one
- Add a selection mechanism to eliminate weak chromosomes in favour of string ones.

- Simulates evolution.
- Find the strongest chromosome to check whether an optimal solution has been found.
- Output the final result.

To build a new generation we must be careful that we do not introduce too much change, or have a selection which is too strict. If too much random change takes place, we will not converge easily, as each generation "forgets" previous good solutions. Selection which is too strong leads to premature convergence by reducing variability too much. The number of cross-overs and mutations must be kept low, and the number of chromosomes eliminated by selection must also be kept down.

One way to implement selection is to find the strongest two and the weakest two in the current generation, and replace the weakest two with the strongest two, adding cross-over and mutation into the mix. Alternatively, you can use "tournament selection" in which two arbitrary pairs are chosen, and of each pair, the weaker is replaced by the stronger one. Again, add cross-over and mutation to taste.

Some people favour evolution which is called "elitist", in which the best chromosome so far is *always* preserved unchanged. The advantage is that whenever you stop, the best chromosome in the current generation is also the best found in *all* generations.

After initialisation, the main process now looks like this:

1. Make a new generation.
2. Perform selection.
3. Check for convergence (e.g. do we have a perfect solution).
4. If not, back to the first step, else print result.

We have to be very careful about the third step in this list. If you only check for a zero difference your program may never terminate on certain input (try 19 blocks of height 1 and one of height 20). You must introduce criteria to ensure that the program does terminate.

Finally, you must produce output. This must not simply print the chromosome, but show how the blocks are divided neatly, and show the heights of each block. It is also interesting to show the number of iterations needed.

Assignment 7 Write a program to implement a full genetic algorithm for the partition problem, using the building blocks you have developed. Test your program on a range of input, and experiment with the number of cross-overs and mutations you use per iteration.