BioComputation

Worksheet 1: Simple Genetic Algorithm Initial Steps

In a computer language of your choice, you need to implement a simple genetic algorithm (GA). That is, write the code for a generational GA which uses a binary encoding, tournament selection, and in the next worksheet single-point crossover and bit-wise mutation (refer to lecture 2). Each individual should be represented by a data structure consisting of an array of binary genes and a fitness value. For example, in C-like notation:

```
typedef struct {
        int gene[N];
        int fitness;
} individual;
```

Evolutionary algorithms use populations of individual candidate solutions, so you'll need an array of your data structure type:

```
individual population[P];
```

Or in Python to achieve the same thing:

```
class individual:
        gene = []
        fitness = 0

population = []
```

The initial population array of such individuals is usually created randomly to sample across the problem space. Make sure you know how to both seed and subsequently call a random number generator and then fill in the genes of your population, eg, say with N=10 and P=50:

```
for( i=0; i<P; i++) {
        for( j=0; j<N; j++ ) {
                population[i].gene[j] = random()%2;
        }
        population[i].fitness = 0;
}
```

Or in Python:

```
for x in range (0, P):
        tempgene=[]
        for x in range (0, N):
                tempgene.append( random.randint(0,1))
        newind = individual()
        newind.gene = tempgene.copy()
        population.append(newind)
```

A very simple fitness function is the well-known “counting ones” problem wherein the fitness of an individual is equal to the number of ‘1’s in its array of genes (genome). Use this to test your code – write a function that is passed an individual’s genes and returns the fitness value:

```
fitness=0;
for( i=0; i<N; i++ ) {
        if( ind.gene[i] == 1)
            fitness += 1;
}
```

Once the initial population is created and evaluated, selection must be implemented. In the lecture we discussed roulette-wheel selection. Here individuals have a chance to be selected as parents based upon their fitness relative to the others in the population. However, it has some drawbacks, eg, if the initial few generations have individuals much fitter than everything else, they will be selected almost all of the time and the population will quickly fill with *only* their offspring solutions. This means the population will prematurely converge into an area of the problem space which may not be useful. An easy way to reduce this problem is to make the chance of selection proportion to fitness rank rather than absolute value - tournament selection:

- Pick T individuals at random from the current population (1<T<P, typically 2).
- Find the individual with the highest fitness of the T.
- Add a copy of the fittest to a temporary offspring population.
- Repeat P times.

```
for( i=0; i<P; i++ ) {
    parent1 = random()%P;
    parent2 = random()%P;
    if( population[parent1].fitness > population[parent2].fitness)
        offspring[i] = population[parent1];
    else
        offspring[i] = population[parent2]
}
```

Or in Python:

```
for i in range (0, P):
    parent1 = random.randint( 0, P-1 )
    off1 = pop[parent1]
    parent2 = random.randint( 0, P-1 )
    off2 = pop[parent2]
    if off1.fitness > off2.fitness:
        offspring.append( off1 )
    else:
        offspring.append( off2 )
```

Once sufficient parents have been selected and copied into a temporary offspring array, a useful check is to see if the total fitness of the temporary population has increased from the original – typically, it should!