Feedback on GA Exercise

THE MAIN LESSON – you have to practise, tweak, reality-check, practise again and again, if you are ever going to use GAs properly in a real project of yours.

Lots of people say they found the solution – then fail to write it down!!! What was it? The sceptical reader needs to do a reality-check that it was indeed a correct answer, and you need to get into the habit of documenting results and how you got there.

You don't get to ride a cycle run a GA well just by reading the textbooks, you have to try them out, fall off, try a new problem etc etc.

There will be another (voluntary) GA+CTRNN exercise later, but ideally you should think up your own problems of interest to you, and practise writing GAs to solve/optimise.

Generic Feedback template

Some people write the code but then don't try it out for real! Or do not actually try it out thoroughly until they are sure they get satisfactory results. But the whole point of the exercise is to get practical results. The Wright brothers didn't get famous by sketching designs on the back of an envelope, they built and crashed, tweaked and rebuilt and crashed, and finally got off the ground. You have to do this for real if you are going to use GAs for real – and you need lots of practical experience.

Lots of people say they found the solution – then fail to write it down!!! What was it? The sceptical reader needs to do a reality-check that it was indeed a correct answer, and you need to get into the habit of documenting results and how you got there.

You don't get to ride a cycle run a GA well just by reading the textbooks, you have to try them out, fall off, try a new problem etc etc.

There will be another (voluntary) GA+CTRNN exercise later, but ideally you should think up your own problems of interest to you, and practise writing GAs to solve/optimise.

Fitness function sign: decide whether you calculate fitness as something you want as big as possible, or as small as possible (if the latter, probably best to call it 'error' rather than fitness) And then doublecheck you are doing it sensibly – eg if measuring 'error', then it is the ABSOLUTE size of the error that matters.

Fitness function 2-parter: how do you combine the fitness/error for each of 2 piles? Add them together is one way. But doublecheck the signs – eg an error of -5 plus an error of +5 might (stupidly) give a total-error of zero, unless you took the Absolute values first.

Fitness-function scaling: if you use rank-based (which includes tournament-based) GAs, you don't need to rescale the fitness/error to any particular scale, but where you have different component parts, as here, you may want to scale them relative to each other. Eg the product-pile typically has much bigger numbers than the sum-pile, so you might be well-advised to rescale them to reflect this. Targets were 36 and 360, so maybe rescale in this proportion 1:10, so that each pile-error has so-to-speak equal weighting on the final error-function.

Generations vs tournaments: GA textbooks typically talk of length of runs in terms of generations. With a population of size N, a generation creates N new offspring. With a Steady-state GA like a Micrornia one, a single tournament that produces 1 new offspring is "not" a generation – to use comparable language, one has to consider N tournaments (producing N new offspring) to be roughly equivalent to just 1 generation (producing N new offspring).

How many tournaments (or how many generations)? LOTS!! GAs are so-to-speak mindless, and produce good results through many many many repetitions of the mindless cycle. That is what computers are for, repeating things mindlessly millions of times. So for this simple example problem, you should certainly be contemplating making your GA runs last for tens of thousands of tournaments, ie potentially thousands of generations. If you only ran it for a few tournaments, it will not have worked, and you need to gain the practical experience coming from lots of experiments – think of the Wright brothers!
Efficiency: for this simple problem, running for tens of thousands of tournaments takes seconds, for bigger real problems it takes hours/days/weeks. So you need to get a feel for what efficiency gets results in less tournaments. Experiment (on small problems) with tweaking population size, mutation rates, recombination rates, use of demes etc etc -- as commented on in lectures. Get experience -- think of the Wright brothers!

Measuring efficiency: a single run might be lucky or unlucky, so you need to do multiple runs and gather statistics before you can be confident about what produces good results. Big variances in (time-to-achieve) results are typical of GAs.

Tweak parameters:

Adjust the parameters?
In my code I can adjust population size, mutation rate and recombination rate, while the rest of the code still works. Did you try doing the equivalent with your code? Adjust the population size to 2 - wow it runs faster with better results! (i.e. hits the jackpot with fewer trials) Adjust the mutation rate to 0.5 - wow it gets better results! What is happening here - mutation rate of 0.5 means random search? Play around with parameters, get a feel for what changes they make.

What about random search?

With this particular problem, there only are $2^{10} = 1024$ possible ways to arrange into 2 piles. If you search systematically through all combinations, you are "bound" to find it within 1024 trials, and random search on average around half that, 512 trials. So how come a GA took longer? Did you do a reality check on your version, and ask yourself whether it was sensible? Actually on this problem, random search is better than a GA!

So are GAs useless?
On this problem, GAs are not a very sensible method! So the purpose of the exercise was to get you to practice writing GA code, to see how simple it "should" be, and gain experience generally. This is not a GA-friendly problem, because the fitness landscape is very rugged - high epistasis. Nevertheless, there "are" many GA-friendly problems, far too big for random search -- and you can use the same GA on these more interesting problems, far more effectively. The parameter settings that work best may well be different on typical GA-friendly problems.

Scaling up the problem

10 cards, $2^{10} = 1024$ possible genotypes, random or indeed exhaustive search is fast, probably faster than a GA.

100 cards, $2^{100} = \text{more than number of atoms in the universe}$, impossible to search exhaustively – GAs still work to get reasonable results in a reasonable time.

Rule of thumb: for an n-bit problem, you probably need $\sim 100n$ generations – and multiply this by pop size for tournaments. 10-card problem, pop size 30, be prepared to go 30000 tournaments if necessary.

Get that CPU hot!

CODE

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

#define POP 30 //pop size
#define DEME_WIDTH 10 //optional extra, if using demes
#define LEN 10 //binary genotypes of length 10
#define MUT 0.1 //mutn prob at each locus
#define REC 0.5 //recomb prob at each locus
#define END 100000000 //max number of tournaments
#define SUMTARG 36.0 //target value for the sum pile
#define PRODTARG 360.0 //target value for the product pile
#define NUMBIGLOOPS 100 //do many runs from scratch
#define PROD TARG 360.0 //target value for the product pile
#define NUMBIGLOOPS 100 //do many runs from scratch
#define SUM TARG 36.0 //target value for the sum pile

int gene[POP][LEN]; //global array containing the binary genotypes of popn

void init_pop() { //initialise them all 50/50 chance of 0/1
    int i,j;
    for (i=0;i<POP;i++)
        for (j=0;j<LEN;j++)
            gene[i][j]=(drand48()<0.5 ? 0 : 1);
}

double evaluate (int n) { //evaluate the n-th genotype
    double sum=0.0, prod=1.0; //add card values to sum, multiply prod by card values
    int i;
    for (i=0;i<LEN;i++)
        if (gene[n][i]==0)
            sum += (1.0 + (double)i); //code counts 0-9, whereas numbers run 1-10
        else
            prod *= (1.0 + (double)i);
    return(fabs(sum-SUMTARG)/SUMTARG + fabs(prod-PRODTARG)/PRODTARG);
}
```
```c
int main(void) { //this has the BigLoop for many runs, GA within each run
    int i, t, a, b, W, L, BigLoop, Tot=0;
    double r;
    srand48(time(0)); //different random seed each time I run
    for (BigLoop=0; BigLoop < NUMBIGLOOPS; BigLoop++) { //many runs
        if (evaluate(a) < evaluate(b)) // looking for *smallest* error, here
            {W=a; L=b;} //identify W Winner L Loser
        else
            {W=b; L=a;}
        for (i=0; i<LEN; i++) { //now work along genotype
            if (drand48()<REC) //prob of copying from Winner
                gene[L][i]=gene[W][i];
            if (drand48()<MUT) //prob of mutating 0 <-> 1
                gene[L][i]=1-gene[L][i];
        }
        if ( evaluate(L)==0.0) { //extra little recording routine
            display(t,L); Tot+=t; break; }
    }
    print_out_statistics_routine();
}
```

**Main GA Loop ....**

**...end of Code...**

---

**Feedback on GA Exercise**

**THE MAIN LESSON** — you have to practise, tweak, reality-check, practise again and again, if you are ever going to use GAs properly in a real project of yours.

---

**THIS IS YOU NOW!!**

**YOU NEED TO BE LIKE THIS!!**