

The Cards Problem: A Genetic Approach

EDUARDO IZQUIERDO
 University of Sussex, Brighton BN1 9QG, U.K.
 E.J.Izquierdo-Torres@sussex.ac.uk

The present work studies a genetic approach to the solving of “the cards problem”. There are 10 cards numbered from 1 to 10 and the goal is to divide them into 2 piles so that (i) the sum of the first pile is as close as possible to 36 and (ii) the product of the rest is as close as possible to 360.

A genetic algorithm with the following characteristics was used: (a) an individual is encoded with a binary genotype of length 10; (b) the size of the population is constant and comprised of 30 individuals, i.e. steady-state genetic algorithm; (c) linear rank is used in parenting selection, implemented via tournament selection; (d) new individuals are generated from 2 selected parents via uniform crossover (50/50 at each locus) plus mutation at each locus with a probability of 1/size of the genotype; (e) for the new individual to become part of the population, an individual is arbitrarily chosen to die; (f) one generation is considered as the production of n individuals where n is the size of the population; and (g) one individual is fitter than another if its fitness score is closer to 0 (more in section 2).

1. SUMMARY OF RESULTS

First of all, the genetic algorithm was tried with a simpler problem in order to be validated. The problem used was the minimization of ones in a binary genotype of the same size. This problem, just as the cards problem, has 1025 possible solutions and one optimal answer. A graph for the average fitness of the population and the scores for the individual with the best and worst fitness every generation are shown in (Figure 1) over several trials. As it’s expected at start off the population has an average

fitness of 5 (50%) due to random initialization of the population and decrements until certain stabilization at a minima. The algorithm was run 1000 times to estimate the mean at which it arrived at the optimal solution. As it was expected the GA got there after generating 114 individuals. This result by far improved the chances of just arriving at the best solution by chance, since randomly it would take on average 512 individuals.

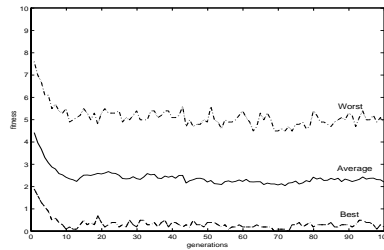


Fig. 1: Minimize Ones Problem: Average (solid line) \ Best (dashed) \ Worst (dashdot) individual at every generation

For the card problems, the genetic algorithm also managed to arrive at the optimal solution (i.e. pile_1 is comprised of cards 1, 3, 4, 5, and 6, and pile_0 the rest) for most of the times. Nevertheless, the algorithm does not perform as well as it does with the simpler validation problem shown above. Similar graphs are used to summarize the results (Figure 2). Figure 2a shows the mean fitness scores for one particular run and over several runs. This graph shows that the population improves rapidly at the beginning, but for the reason that the range in fitness is much more varied the population remains less stable throughout the remaining generations. Also, this graph shows that the population stabilizes at around a fitness score of 3000 which represents the upper 0.1% of the overall fitness. Figure 2b shows the fitness

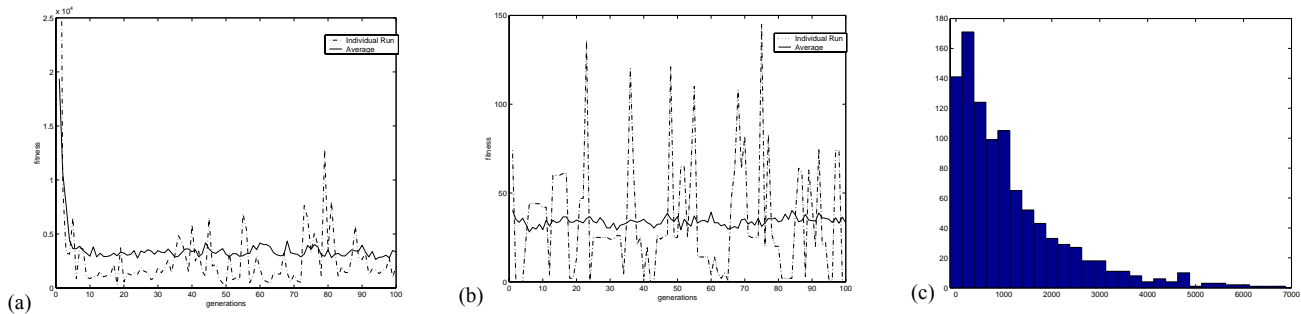


Fig. 2: Cards Problem (a) mean fitness of the population for one particular run (dotted line) and an average for 200 runs (solid line) (b) best fitness of the population for one particular run (dotted line) and an average for 200 runs (solid line) (c) histogram of the number of individuals it took to arrive at an optimal solution for 1000 runs.

of the best individuals at each generation for one particular run and for the average after several runs. The best individual's fitness score is on average 35, and as seen in the particular run varies between the optimal (0) and 150. A third graph shows a histogram for the distribution of the number of individuals it takes to arrive at the optimal solution after 1000 trials (Figure 2c), which depicts an uneven distribution. This time however, in comparison with the simpler validation problem, the performance is highly decremented with an average of 1169. Nonetheless, based on the distribution a much more valid value is expressed by the median, in this case 772. Furthermore, as portrayed in this last figure for a good proportion (39%) of the times the algorithm manages to arrive before generating the 512th individual. Still overall, the performance seems to be superseded by the theoretical average that would be obtained by random search (512).

2. THE FITNESS FUNCTION

Several different fitness functions were tried, yet regardless of how intrinsic the function was made, the performance of the algorithm did not appear to improved significantly. Thus, in an attempt to comply with Occam's razor, the simplest fitness function was kept.

$$f = |\text{pile}_0 - 36| + |\text{pile}_1 - 360| \quad (1)$$

Where pile_0 is the sum of the cards in one of the piles and pile_1 is the product of the cards in the other. As it follows from the definition of such fitness function, it is said that genotype of individual a is better than that of individual b if fitness of individual a is lower than the fitness of individual b . Consequently, the problem is one of minimization with a range between 0 and 3.628.836; the best individual being the one that accumulates 36 and 360 on pile_0 and pile_1 respectively. It also follows from (1) that two different individuals may have the same fitness function if their piles lack or exceed in the same number of integers from the goals (e.g. individual: 1110000000, pile_0 : 49, pile_1 : 6, and individual: 0000010000, pile_0 : 49, pile_1 : 6, fitness in both cases: 367).

3. DISCUSSION OF RESULTS

On the cards problem, the genetic algorithm seems to work better than chance only on a proportion of the trials; this behavior is best explained by a combination of two factors: (i) the reduced dimension of the search space and (ii) the non-smoothness of the fitness landscape.

In the simpler problem of minimizing the number of ones, the genetic algorithm performs very well regardless

of the reduced dimension of search space because of the smoothness of its landscape. Here, changing a 1 to a 0 is good regardless of the value of the rest and therefore positive improvements are easily accumulated. The fitness landscape described by this problem is easily climbed by the individuals in the population.

The fact that the genes in the cards problem are intrinsically related to each other makes the fitness landscape much more rugged (perhaps if I had designed a superior fitness function this wouldn't have happened, nevertheless I argue this is the nature of this problem). Here, the genes form a network such that for improvements in fitness a combination of changes are needed (e.g. having one gene turned on may require several others to be turned off or vice versa). This trait makes accumulation of positive improvements a much more improbable event, for real improvement in fitness will depend not on independent well directed mutations but on several changes occurring concurrently in the same individual.

In conclusion, the ruggedness reduces the chances of a serious attempt of the individuals climbing up through the landscape. On the other hand, the reduced search space makes it so that very few changes in the genome drive the fitness of the individual from good to bad or vice versa. Therefore the genetic algorithm is a poor choice for this optimization problem. However, a genetic algorithm has the ability to perform better on either a simpler problem like the one shown in this work, or on problems whose search space is so big that common search strategies don't prove efficient.

Is this a sensible use of Genetic Algorithm?

Genetic Algorithms are not originally intended as function optimizers, but rather as adaptive improvers and for this reason alone is hard to precise the sensibleness of this approach. Nonetheless, given the reduced space of the fitness landscape (i.e. 1025) in addition to the intrinsic relation between the genes, it seems more prudent to use a simple search algorithm. For this particular problem I have shown that even a random generator of solutions will prove at least as useful.

Is it efficient?

Even though this approach is in a number of cases (i.e. 39%) more efficient than a random search, this approach is not *reliably efficient*, on the account that on average the solution takes as much as two times as many individuals as would take a random search. Furthermore, at times the solution is found at absurdly late generations.

Appendix A. CODE IN C [ga.c]

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

#define BN 10 //number of genes in genotype
#define MUTPR 0.1 //mutation probability
#define RMBPR 0.5 //recombination probability
#define PN 20 //population number
#define GN 100 //number of generations

int randInt(int max){ // random integer between 0 and max
    float n = 0;
    int d = 0;
    n = (rand()/32767.0) * max;
    d = (int) n;
    return(d);
}

float calcFit(int pop[PN][BN],int ind){ //Fitness Function
    int i, pile_0=0, pile_1=1;
    float a,b;
    for (i=0;i<BN;++i)
        if (pop[ind][i]==0)
            pile_0 = pile_0 + (i+1);
        else
            pile_1 = pile_1 * (i+1);
    a = (pile_0-36<0?36-pile_0:pile_0-36);
    b = (pile_1-360<0?360-pile_1:pile_1-360);
    return(a+b);
}

int main(){
    int pop[PN][BN]; //population
    int newInd[BN]; //new individual
    int i,j,equal; //counters
    int m1,m2,d1,d2,mom,dad,dead; //variables
    srand(time(0)); // random seed
    // instantiate population
    for (i=0;i<PN;++i) for (j=0;j<BN;++j) pop[i][j]=((rand()/32767.0)>0.5?0:1);

    for (i=0;i<GN*PN;++i) { // generation
        equal = 1; // pick 2 moms and 2 dads to compete
        // they can not repeat!
        while (equal==1) {
            m1 = randInt(PN);m2 = randInt(PN);d1 = randInt(PN);d2 = randInt(PN);
            equal=( (m1==m2|m1==d1|m1==d2|m2==d1|m2==d2|d1==d2)?1:0);
        }
        mom = (calcFit(pop,m1)<calcFit(pop,m2)?m1:m2); // tournament selection
        dad = (calcFit(pop,d1)<calcFit(pop,d2)?d1:d2);
        for (j=0;j<BN;++j) { // generate new individual by recombination
            newInd[j]=((rand()/32767.0)<RMBPR?pop[mom][j]:pop[dad][j]);
            if ((rand()/32767.0)<MUTPR) newInd[j]=((newInd[j]==1)?0:1);
        }
        dead = randInt(PN); // eliminate individual at random
        for (j=0;j<BN;++j) pop[dead][j]=newInd[j]; //replace by new individual
        if (calcFit(pop,dead)==0){ // check if optimal solution found
            for(j=0;j<BN;++j) printf("%d ",newInd[j]);
            printf("\n%d\n",i); // print solution and end program
            return(0);
        }
    }
    printf("++%d\n",PN*GN);
    return(0);
}
```