3. EVOLUTIONARY ALGORITHMS

The algorithms presented until now are local search algorithms; these algorithms search by moving from one solution into one of its neighbors; the differences between the algorithms rely on the way the neighbor is chosen.

Evolutionary algorithms are different. They do not rely on single solution neighborhood steps to explore the whole space. They explore the space with a set (population) of solutions at the same time. So, they are not local search techniques; they are global optimization techniques.

Despite being global optimization techniques, they are not robust enough to guarantee finding the global optimum. However, they can be designed to find the global optimum solution with high probability. Designing such algorithms is the subject matter of this chapter.

3.1. Principles

Evolutionary algorithms (EA) are algorithms that manage the survival of a set of solutions based on the principles of natural selection (neo-Darwinian evolution). Such principles are:

- Generation sequencing (death/birth)
- Survival of the fittest (competition)
- Genetic exchange (mating)

How does neo-Darwinian evolution work?

Evolution takes place in time (generation sequence) if only the fittest get a chance (competition) to combine their genes (mating), and this way contributing to the subsequent generation. Such an evolutionary process can be presented as the following algorithm.

Evolutionary algorithm

Make t = 0;
Initialize a *population* of solutions P(0), at random.
Evaluate P(0)
Repeat step 1 to 4 (until saturation)
  Step 1 t ← t+1
  Step 2 Build P(t) by selecting the fittest from P(t-1)
  Step 3 Change P(t)
  Step 4 Evaluate P(t)
.

The "change" in step 3 is a crucial process of any evolutionary algorithm. Usually, the change is undertaken in two independent processes:

A solution-pair recombination process where information between solutions is exchanged (this process is usually denoted by *crossover*); the exchange should be designed to be a solution-pair neighborhood variation that explores the differences and (more important) keeps the likenesses of pairs; and

An individual solution arbitrary information change (usually denoted by *mutation*); these changes are usually designed to be local and rare.

Evolutionary algorithms that use these two processes are usually called Genetic Algorithms (GA). GAs can be written as the following procedure.

Genetic algorithm

Make t = 0;
Initialize the population P(0), at random.

Evaluate P(0)
Repeat step 1 to 5 (until close to saturation)
Step 1 t ← t+1
Step 2 Select the fittest from P(t-1) to build P(t)
Step 3 Cross P(t)
Step 4 Mutate some solution from P(t)
Step 5 Evaluate P(t)
.

Why do GAs work?

They work because:

The space is populated (randomly) by several solutions;

The better solutions are crossed-over, i.e., one explores the differences between these and exploits the likenesses, as one guarantees that the latest have a high probability of being represented in the next generation; and finally

One may explore the whole space even with a small population by randomly mutating some solutions

By exploring the differences and exploiting the likenesses of blocks of information, GAs perform an *experiment* and *abstraction* cycle which has been at the basis of many complex inventions.

Some famous inventions, like the airplane, have been conquered this way. According to Goldberg [2], the Wright brothers' approach to design the first airplane was somehow similar to a GA.

In the next sections we will go into each of the GA operators with more attention and define bounds for the GA parameters.

3.2. Selection

The first operation that one must address in evolutionary algorithms design (see Step 2) is building a new population based on the performance of the actual one (competition). The new population will have the individuals that will be crossed (next operation). This operation is called Selection.

Selection is the result of competition. But how should one implement a selection mechanism? How many should survive? And which ones will survive?

(How many) One very important aspect of selection is the selection pressure.

> High selection pressure will lead to premature genetic saturation, but on the contrary, low selection pressure will turn the process convergence very slow. A tradeoff is needed.

A high selection pressure could be easily undertaken by making many copies of the best, or 2nd and 3rd better individuals; that would simulate a very competitive mating, but it is easy to see that after a few iterations all the individuals would be very much the same. So, how could we design a selection mechanism with a good tradeoff between selection pressure and population diversity?

(Which) There are two different approaches to selection: numeric proportional selection and ordinal selection.

> The proportional selection can be implemented by a roulette wheel (of fortune). In the roulette, each individual solution gets a slice with size proportional to its fitness. To build a new population of n candidates, one just has to wheel the roulette n times and pick up the chosen ones. In Fig.3 we

illustrate that for a population of 6 solutions where solution 'a' happens to be a very good one.

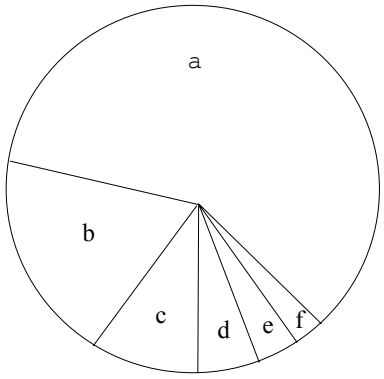

Fig.3. Selection pressure of the roulette wheel (of fortune) is very much dependent on objective function. In the figure, 'f' does not stand many chances of being selected; on the contrary, 'a' will be selected with (too) high probability.

Note that the roulette selection scheme is too much dependent on the objective function itself, which makes functions that are very abrupt (very flat) very hard to optimize because the selection pressure turns out to be very high (low). Moreover, roulette selection can not be used as a competition mechanism on ordinal functions.

The ordinal selection can be implemented by binary tournaments. The idea is to select (at random) two solutions from the population and choose the better of the two for the next generation (new population to be crossed). This kind of selection is very simple to program and is not sensitive to the abruptness of the objective function.

To increase the selection pressure in ordinal selection, one may use tournaments of three solutions instead of tournaments of two (binary).

To see comparisons between these and other selection schemes, read [3].

Another frequently addressed aspect of selection mechanisms is the so called elitism. Elitism stands for the attitude of not allowing losing the better individuals. Elitism is generally not necessary (if the GA is well designed), but still popular. About elitism one should note that:

> Roulette wheel selection is not elitist – the best might be lost
>
> Binary tournament is elitist only if every solution gets a chance of playing.

However, crossover and mutation may destroy the best individual. Loosing our best happens a lot and is not a problem. To guarantee that the best is never lost, one would have to copy it directly to the future generation. That is a little obsessive, but still ok.

3.3. Crossover

Crossover is a term used in biology to denote the interchange of sections between pairing homologous chromosomes during the phase of meiosis. So, to define crossover one has to define a chromosome as the representation of a solution.

Suppose that we are dealing with the SAT problem and that our solutions are easily represented by an array of binary values. A chromosome for the SAT can be defined as binary string, and the crossover operation defined as a string section exchange. The simplest exchange is defined by a single section defined by a single string position and is denoted by one-point crossover. The position

is chosen randomly. The operation is illustrated in the next two figures.

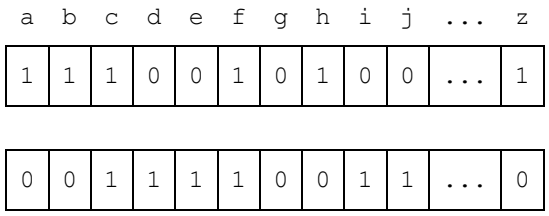

Fig.4 Solutions 1 and 2 before crossover

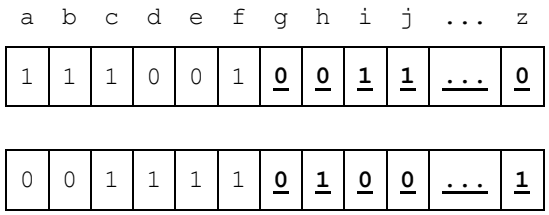

Fig.5 After one-point crossover in position f-g

Exercise 8
Represent a spanning tree as a set of n-1 order relationships between its nodes (a<b : where ab is an arc of A) and propose a crossover operation for the Minimum Spanning tree problem.
What is going to be exchanged between trees?

Crossover is a problem-specific operation. Remember that it was introduced as a solution-pair neighborhood variation, and neighborhoods (we saw that in local-search chapters) are problem specific.

3.4. Mutation

Mutation is a term used to denote any event that changes genetic structure. Mutation is used to create genetic diversity (to explore) and also to avoid the premature genetic saturation of the population.

Mutation is mandatory when the combination of individuals of the initial population cannot span the whole space (the usual case). E.g., take a SAT initial population P(0) and imagine that all individuals have g = 0; without the possibility of changing the genetic representation (mutation), one would never experiment g = 1: the combination of individuals by crossover can not do it.

Mutation is also problem-specific but usually it is easier to design than crossover. Note that mutation is a neighborhood variation and that high probability mutation turns the process into random search.

Suppose that we are dealing with the SAT problem and that our chromosome can be defined as a binary string. The mutation operation can be defined simply as a bit change. The bit position is chosen randomly. The operation is illustrated in Fig.6.

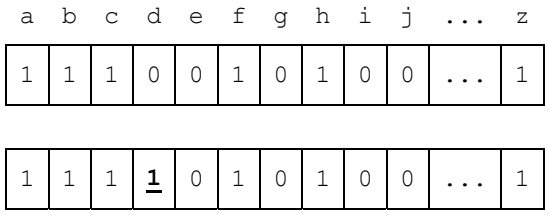

Fig.6 Before and after mutation by flipping position d.

3.5. Parameters

We have presented two genetic operations, namely, crossover and mutation, but we said nothing about how many solutions one should

cross or mutate. The importance of these and other GA parameters will be addressed in this section.

The task of finding good parameters for the GA is a very important one in algorithm design. Parameters are difficult to define based on theoretical considerations. Frequently, good parameters are only possible to be found by experience. However, it is easy to understand GAs at the following two extreme conditions:

If we don't mutate, populations do rapidly saturate

If we don't cross, GAs get to be like a SHC

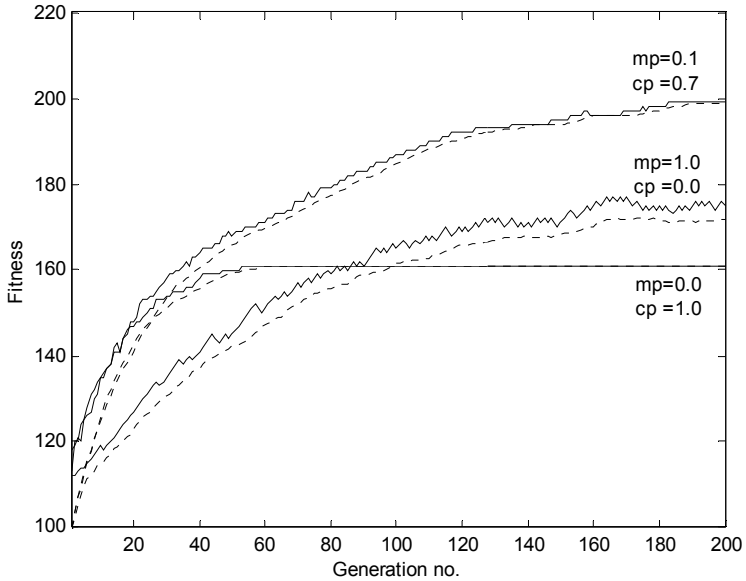

Fig.7. GA performance for different crossover and mutation probabilities. 100 solutions population evolution for the one-max problem of string size 200: population average in dashed line, best individual in solid line.

In Fig. 7 we show three different GA process evolutions for different crossover probabilities (cp) and mutation probabilities (mp). The problem solved is the *one-max* problem, a very simple but well known problem. The problem consists in maximizing the number of 'ones' in a bit string of size n. We solve the problem for n = 200 with a population size of 100 solutions (the space size $2^{200} \approx 10^{60}$ solutions). The figure shows that:

> If we use a lot of crossover and little mutation, good solutions can be found; we got very close to the optimum (f = 199/200) with cp = 0.7 and mp = 0.1.
>
> If we don't use mutation (mp = 0), the process is very similar to a 'good' evolutionary process at the beginning, but quickly stops evolving. That is because the population saturates without mutation.
>
> If we don't use crossover (cp = 0), the process evolves all along (it doesn't stop) although it evolves much slower than the other two processes, and seems never to be able to yield good solutions, no matter how many generations one takes.

So, the question is: should one always cross with probability 0.7 and mutate with 0.1?

A quick experience shows that there are bounds for the crossover probability cp and mutation probabilities mp, and not such thing as the right value for each. Many authors tend to state that ideal values are $0.7<cp<0.8$ and $0.05<mp<0.10$, but that is generally not true. In Fig.8, we plot the GA valid parameter region for the problem just solved.

Besides crossover and mutation probabilities, population size plays an important role on GA performance. If the size is too small, the GA will converge to sub-optimal solutions; if too large, the process

will spend unnecessary time and resources to yield good solutions. The issue of sizing populations is a complex one because it relies in building-block cost variance (noise) statistical effects [4].

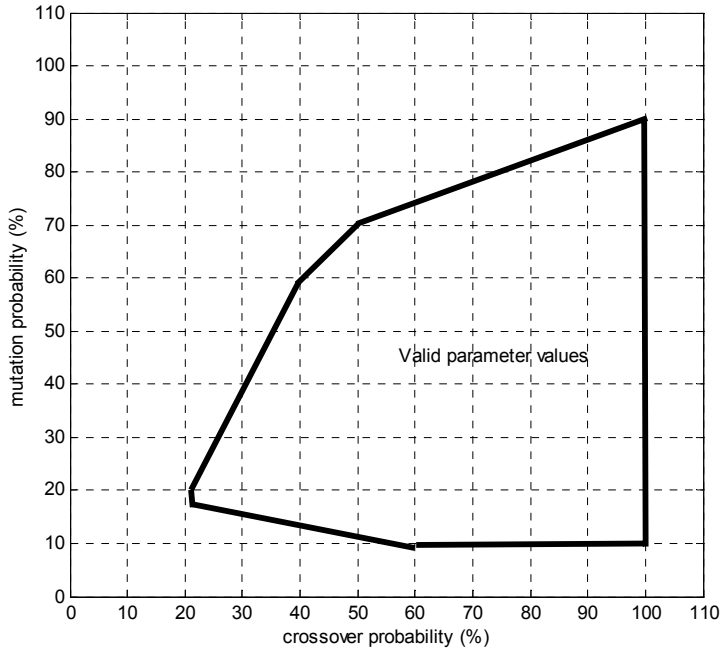

Fig.8. Valid parameter values for the one-max GA

GAs exploit the likenesses of good solution-blocks to build better solutions. However, "good blocks" are not always easy to spot among a set of evaluated solutions (solutions are evaluated as a whole, thus a good block can be unnoticed just because a bad one also exists, and together the solution does not perform well). The way "good blocks" influence the evaluation of solutions importantly affects the population size necessary for the GA to perform well.

There are formulas to size populations based on chromosome length and building-block signal-to-noise ratios, but these are generally very difficult to apply in practice.