10. Evolutionary learning

Evolution works on a population through an imaginary *fitness landscape*, which has an implicit bias towards animals that are ”fitter”, in other words, those animals that live long enough to reproduce and generate more and healthier offspring.

The genetic algorithm models the genetic process that gives rise to evolution. In particular, it models sexual reproduction, where both parents give some genetic information to their offspring.

10.1. The genetic algorithm (GA)

The genetic algorithm is a computational approximation to how evolution performs search, which is by producing modifications of the parent genomes in their offspring and thus producing new individuals with different fitness. It attempts to abstract away everything except the important parts that are needed to understand what evolution does. From this principle, the things that are needed to model simple genetics inside a computer and solve problems with it are the following.

- A method for representing problems as chromosomes
- A way to calculate the fitness of a solution
- A selection method to choose parents
- A way to generate offspring by breeding the parents

String representation

The first that needed is some way to describe the individual solutions, in analogy to the chromosome. The genetic algorithm use a string, with each element of the string, equivalent to the gene, being chosen from some alphabet. The different values in the alphabet, which is often just binary, are analogous to the alleles. For the problem to be solved, a way of encoding the description of a solution as a string has to be worked out. Then a set of random strings is created to be the initial population.

Evaluating fitness and population

The *fitness function* takes a string as an argument and returns a value for that string. Together with the string encoding the fitness function forms the problem-specific part of GA. The best string should have the highest fitness, and the fitness should decrease as the strings do less well on the problem.

GA works on a population of strings, with the first generation usually being created randomly. The fitness of each string is then evaluated, and that generation is bred together to make a second generation and so on. The algorithm tries to get fitter individuals as going on.

Generating offspring: parent selection

For the current generation we need to select such strings that will be used to generate new offspring. The idea here is that average fitness will improve if we select strings already relatively fit compared to the other members of the population. However, it is also good to allow some exploration in there, which means that we have to allow some possibility of weak strings being considered. There are three employed ways to handle this, although the last tends to produce better results.

Tournament selection: Repeatedly pick four strings from the population, with replacement and put the fittest two of them into the ”mating” pool.

Truncation selection: Pick some fraction f of the best strings and ignore the rest. For example, f=0.5 is often used, so the best 50% of the strings are put into the mating pool, each twice so that the pool is the right size. The pool is randomly shuffled to make the pairs.

Fitness proportional selection: The better option is to select strings probabilistically, with the probability of a string being selected being proportional to its fitness.

The function that is usually applied is for string α

$$p^{\alpha} = \frac{F^{\alpha}}{\sum_{\alpha'} F^{\alpha'}}$$

where F^{α} is the fitness. If the fitness is not positive, then F needs to be replaced by $\exp(sF)$ throughout, where s is the *selection strength*, a parameter.

$$p^{\alpha} = \frac{\exp\left(sF^{\alpha}\right)}{\sum_{\alpha'} \exp\left(sF^{\alpha'}\right)}$$

We want to pick each string with probability proportional to its fitness, but if there is only one copy of each string, then the probability of picking each string is the same. One way around this is to add more copies of the fitter strings, so that they are more likely to get chosen. This is called ”roulette selection”, because while imagining that each string gets an area on a roulette wheel, then the larger area associated with one number, the more likely it is that the ball will land there. Strings from this larger set can then be picked randomly.

10.2 Generating offspring: genetic operators

Having selected breeding pairs, we have to decide how to combine their two strings to generate the offspring. Two genetic operator mostly applied are *crossover* and *mutation*.

Members of the GA population only have one chromosome-equivalent, the string. Thus the new string is generated as part of the first parent and part of the second. The most common way for this is to pick one point at random in the string, and to use parent 1 for the part of the string, up to the *crossover point* and parent 2 for the rest.

Two offspring are generated with the second one containing the first part of parent 2 and the second part of parent 1. This is *single point crossover*, and the extension to *multi-point crossover* is obvious. The third version is *uniform crossover.* Three types of crossover are depicted in Fig. 10.1.

Mutation effectively performs local random search. The value of any element of the string can be changed, governed by some, usually low probability p such as 0.01 or 0.001. For the binary alphabet, mutation causes a bit-flip as in Fig. 10.2. For real values, some random number is added or subtracted from the current value. Often, probability $p \approx 1/L$ where L is the string length so that this is around one mutation in a string.

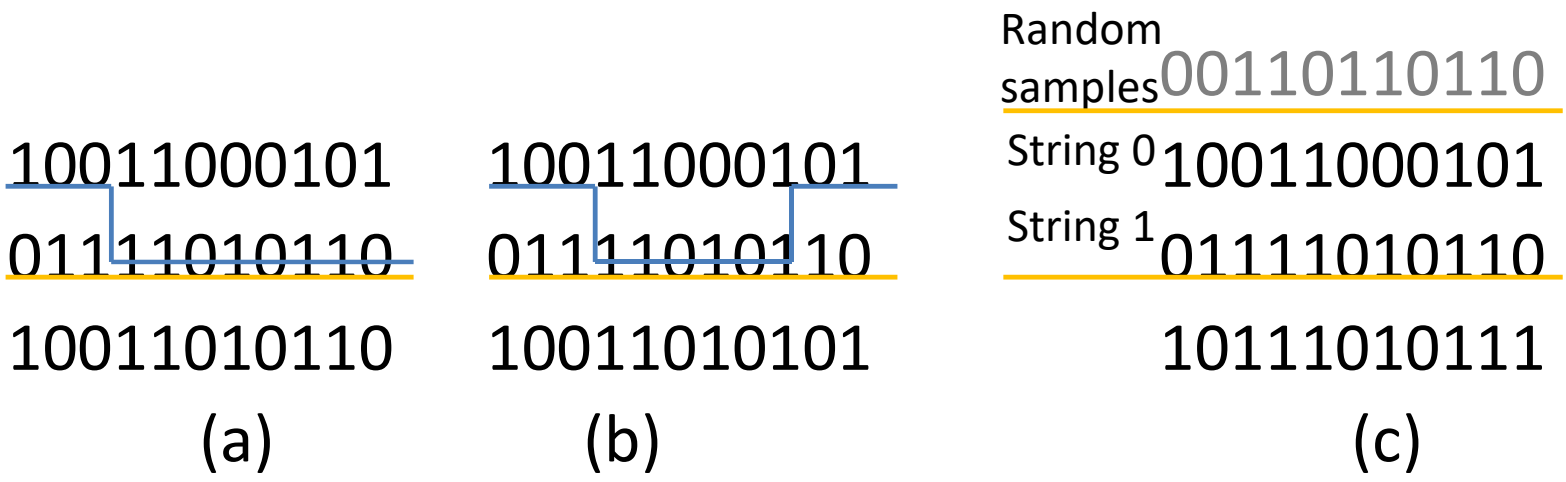

Fig. 10.1 Different forms of crossover. (a) Single point crossover. A position in the string is selected randomly, and the offspring is made up of the first part of parent 1 and the second part of parent 2. (b) Multi-point crossover. Multiple points are selected, with the offspring being made in the same way. (c) Uniform crossover. Random numbers are used to select which parent to take each element from.

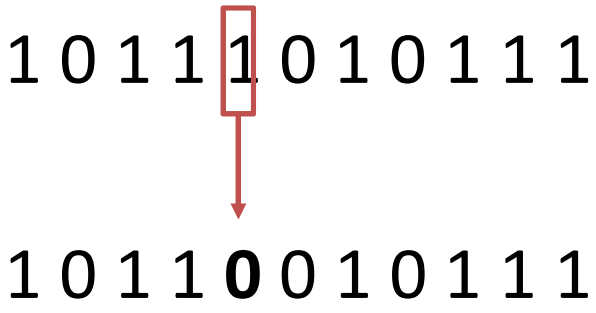

Fig. 10.2 The effect of mutation on a string.

Elitism, tournament and niching

So far offspring have been produced. The simplest next option is to replace the parents by their offspring to make a completely new population, and carry on from there. Nevertheless, the maximum fitness in each generation can decrease, at least temporarily. This seems a bit risky, since the ”best” solution is pursued.

There are three ways to avoid the above problem: elitism, tournament and niching.

In *elitism* some number of the fittest strings are taken from one generation and put them directly into the next population, replacing strings already there either at random or by choosing the least fit to replace.

In *tournament* two parents and their offspring compete, with the two fittest out of the four being put into the new population.

While elitism and tournament both ensure that good solutions are not lost, they both have the problem that they can encourage *premature convergence,* where the algorithm settles down to a constant population that never changes even though it has not found an optimum. This occurs since GA favors fitter members of the population, which means that a solution that reaches a local maximum will generally be favored and this solution will be exploited. Tournaments and elitism encourage this, since they reduce diversity in the population.

One way to solve the problem of premature convergence is through *niching* (or *island populations*), where the population is divided into several subpopulations to evolve independently for some number of iterations, so that they are likely to have converged to different local maxima. A few members of one subpopulation are occasionally injected as ”immigrants” into another subpopulation.

The following algorithm consists of the previous properties. Its stopping criterion is often a fixed number of generations run.

10.3 The basic genetic algorithm

Initialization

Generate *N* random strings of length *L* within the chosen alphabet

Learning

Repeat

(1) Create an (initially empty) new population

(2) Repeat

(2.1) Select two strings from current population, preferably using fitness-proportional selection

(2.2) Recombine them in pairs to produce two new strings

(2.3) Mutate the offspring

(2.4) Either add two offspring to the population, or use tournaments to put two strings from the four of parents and offspring into the population

(2.5) Until *N* strings for the new population are generated

(3) Optionally, use elitism to take the fittest strings from the parent generation and replace some others from the offspring generation

(4) Keep track of the best string in the new population

(5) Replace the old population with the new one

Until stopping criteria met.