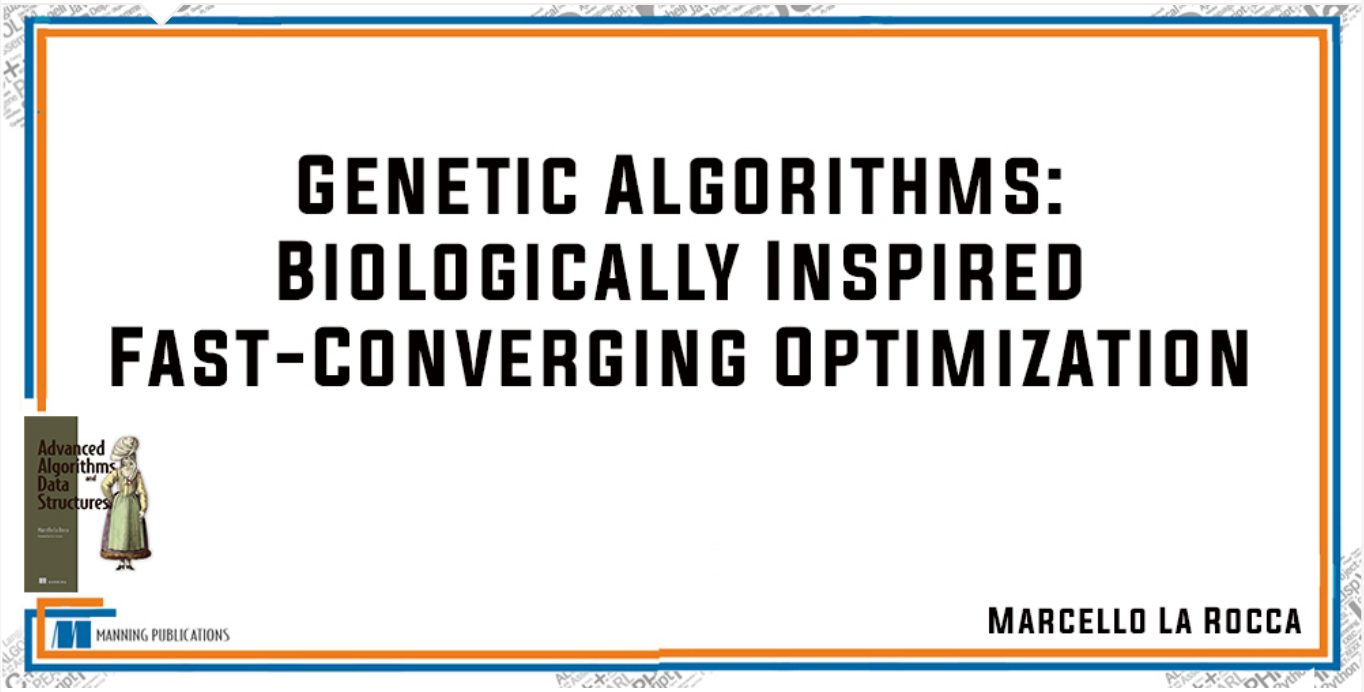
MANNING FREE CONTENT CENTER

Genetic algorithms: Biologically inspired, fast-converging optimization



This article covers

- Introducing the genetic algorithm

- Examining whether genetic algorithms are better than simulated annealing
- Solving the "Packing to Mars" problem with genetic algorithms

Genetic algorithms

When it comes to optimization algorithms, the gist is that we are trying to replace a brute-force search over the whole problem space with a local search that filters out as many points in the problem space as possible: finding the same result by searching a much smaller domain.

Optimization algorithms explore the neighborhood of current solutions, which are points in the search space, and do this either deterministically or randomly, trying to spot areas of interest, promising regions where we expect to find better solutions than what the search has previously found.

The way we perform this filtering, the range of the search and how "local" it actually is, if we move randomly or in a deterministic way.

In the previous chapter we discovered simulated annealing and discussed how it converges to near-optimal solutions by allowing transitions uphill, while gradient descent only moves downhill, toward better solutions, and as such it gets easily stuck in local minima. Simulated annealing can therefore work better than gradient descent when the cost function has many local (sub-)minima.

While powerful, simulated annealing can't be perfect, and indeed we have also seen that there are two main drawbacks to consider:

1. The algorithm is slow to converge. While gradient descent, which is deterministic, takes the fastest route downhill[1], simulated annealing (which instead is stochastic) randomly wanders across the cost function's landscape, and as such it might require many attempts before finding the right direction.
2. With certain shapes for the cost functions, where local/global minima are in narrow valleys, the probability that simulated annealing randomly "walks" into those valleys can be low, so low that it can fail to reach a near-optimal solution altogether.

Simulated annealing works well, and it creates this sort of dynamic filtering, indirectly restricting (based on their cost) the domain points that can be reached at a given stage. If you remember, initially the algorithm jumps from point to point, going uphill as likely as downhill, exploring the landscape at large.

We have also seen how it can happen that the optimization finds a near-optimal solution in the early stages, and then moves away from it, because in the initial phase

it's likely to accept transitions uphill. The problem is that simulated annealing has no memory, it doesn't keep track of the past solutions, and-especially if we allow long-range transitions-there is a concrete risk that the algorithm will never get back to a solution that was as good as.

An even more probable risk is that the algorithm does find the valley containing global minimum at some early-ish stage, not necessarily getting any close to the end of the valley (perhaps landing just somewhere close to the entrance of the valley) and then moves away and never manages to enter it again.

Figure 1 illustrates these situations. Moreover, *simulated annealing with restart* can keep track of past solutions and-when stuck-randomly restart from one of these past positions, to check if a solution better than current best is found.

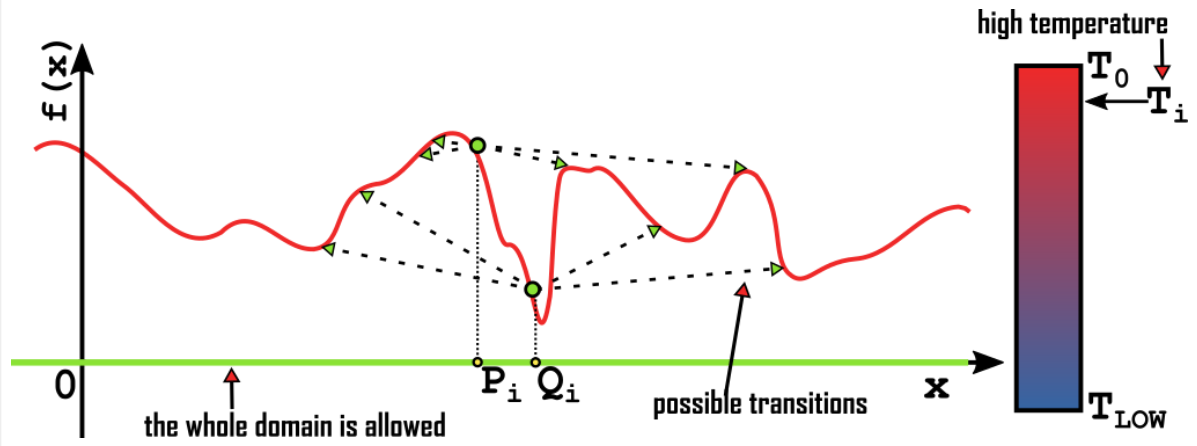


Figure 1. An example of a scenario where simulated annealing would find a promising (P i), or even good (Q i) solution in an early stage (as suggested by the temperature of the system, T i, still close to T 0): since all transitions, even uphill, are likely accepted when temperature is high, the algorithm might move away from the sweet spot in the cost function landscape, and possibly never find its way back.

This workaround can somehow help with the former situation, where we get to a great solution early, but it's unlikely to improve the latter (finding just a promising position at the entrance of a valley), because we only save at most a small number of the best previously found solutions. As such, even if the optimization had managed to find the beginning of a path to global minimum, we would forget about it, unless we were lucky enough to land close to the bottom of the valley.

Inspired by nature

Simulated annealing was inspired by metallurgy, mimicking its cooling process to drive a system from chaotic to ordered behavior. Nature, as a matter of fact, has often been a great source of inspiration for mathematics and computer science. Just think about neural networks, probably the most pervasive of these examples, at the time of writing.

For biological processes, such as neural networks, it's not hard to imagine why. They have been adapted and perfected over millions of years, and since their efficiency is tight with organisms' survival, we can find clever and efficient solutions everywhere in nature.

Genetic algorithms are yet another example of biologically inspired algorithms and, even more, they are based on this very principle of evolution in response to stimuli from the environment.

As shown in figure 2, a genetic algorithm is an optimization algorithm that maintains a pool of solutions at each iteration. Compared to simulated annealing, this allows maintaining a larger degree of diversity, probing different areas of the cost function's landscape at the same time.

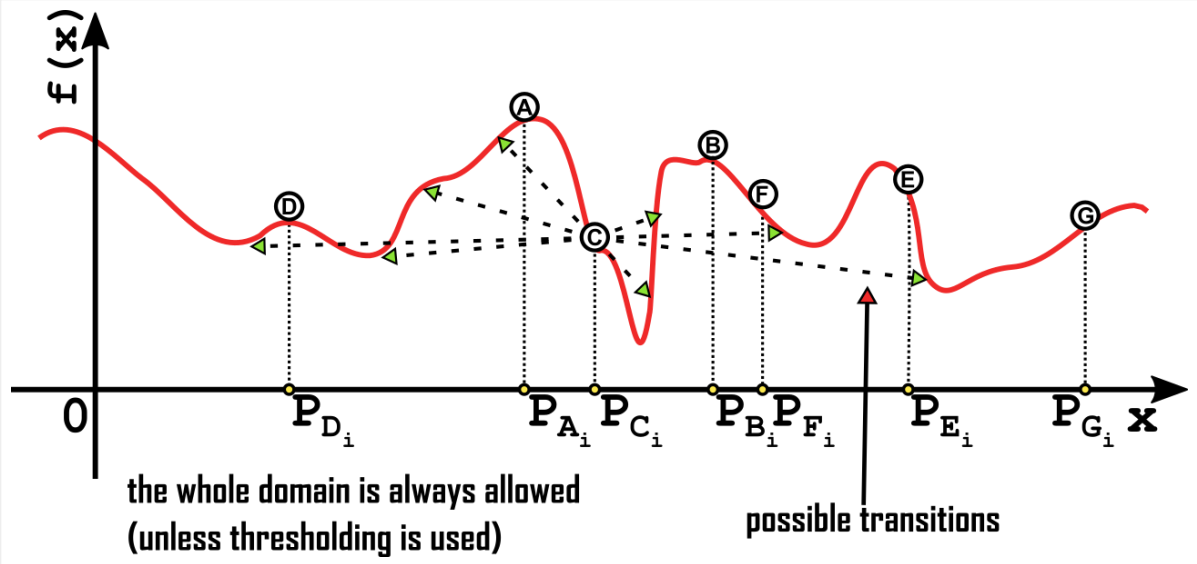


Figure 2. An example of how genetic algorithms would tackle the scenario in figure 1. At a generic iteration (in the figure, we assume we are at the `i` -th iteration), which could indifferently be toward the beginning or the end of the simulation, the algorithm maintains a pool of possible solutions. At the next iteration, the pool (as a whole) will

transition to a new set of candidate solutions (based on the current ones-we'll see how later in this section). Usually the new solutions can be anywhere in the domain.

One thing that can't be shown in this "cost function" kind of graphic is that genetic algorithms also evolve the solutions by recombining them. This allows us to take the strengths of different solutions and merge them in a better one. Imagine that we have two poor solutions, each having the best possible sequence for a different half of the vertices, but each with terrible solutions for the remaining half. By combining these two solutions in the right[2] way, we can take the good halves from each and get a great candidate solution, possibly (if you are lucky!) even the best possible.

This idea of combining solutions is something that we don't find neither in gradient descent (obviously!) nor in simulated annealing, where a single solution is kept and "studied" at any time.

We'll see that this new idea is embodied in a new operator, the *crossover operator*, that derives from the biological analogy used for this technique. But to avoid getting ahead of ourselves, we still need to reveal what inspired genetic algorithms; in doing so, we will also make clear where the name comes from.

A genetic algorithm is an optimization meta-heuristic based on the principles of genetics and natural selection. We mentioned that this optimization technique maintains a pool of solutions. What we hadn't said is that these solutions will mimic a population that has evolved through several generations: each organism in the population is defined by a chromosome (or, sometimes, a pair of chromosomes), that encodes a single solution to the problem that's optimized.

Biologically inspired algorithms

Most living organisms on this planet are made by cells[3], each of which carries the same[4] genetic material in a set of chromosomes (usually more than one, for advanced organisms such as animals or plants). Each chromosome is a *DNA* (*deoxyribonucleic acid*) molecule that can be broken down in a sequence of nucleotides, each of which is, in turn, a sequence of *nitrogen-containing nucleobases* encoding information. The information contained in each cell's DNA is used by cells to drive their behavior and synthesize proteins. Figure 3 succinctly illustrate these concepts.

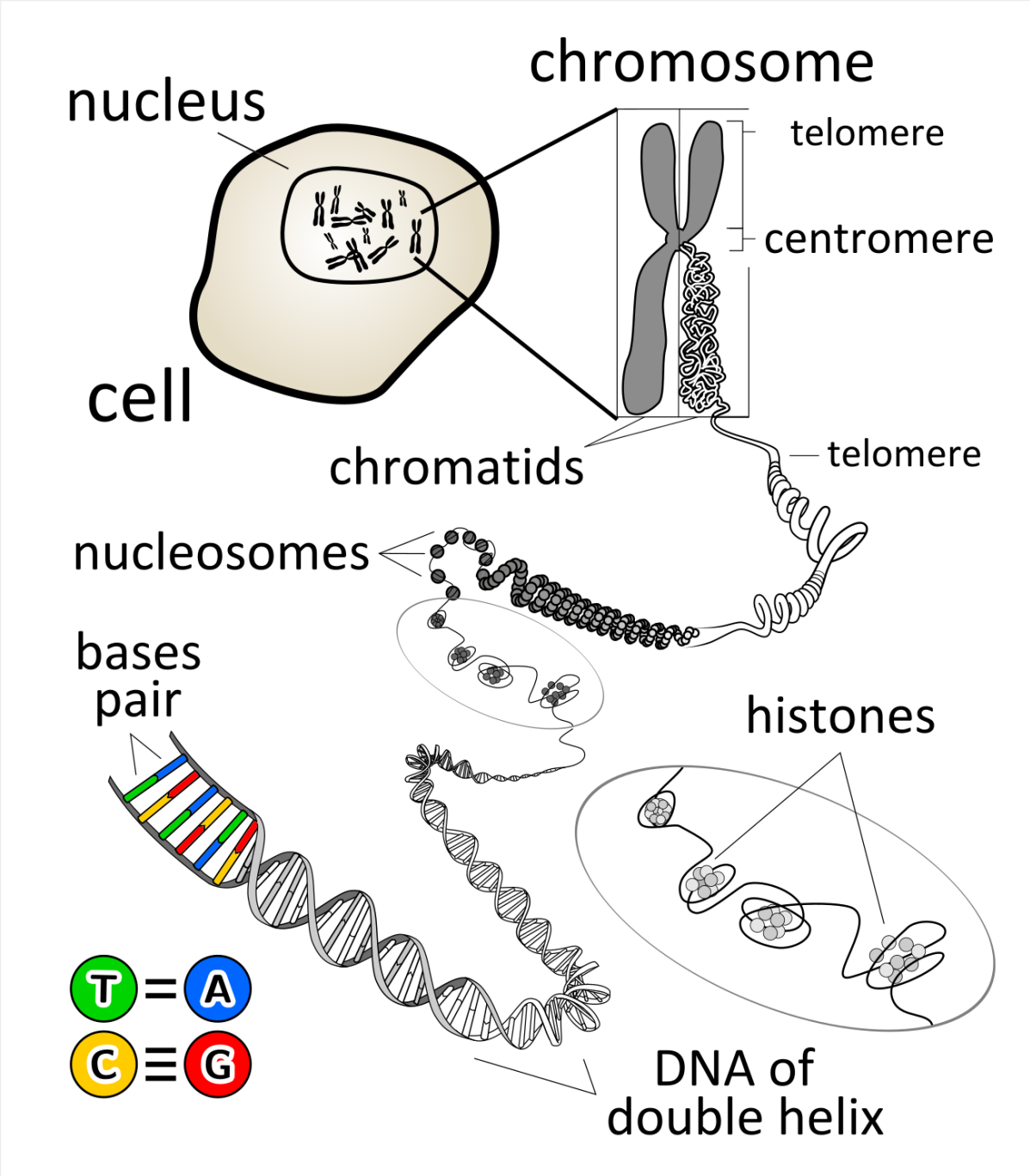


Figure 3. From cells to DNA: A cell's genome is contained in its nucleus, inside structures called chromosomes, whose telomeres, when unraveled, reveal double helixes of DNA, a sequence made of just four nucleobases (hence we can consider DNA a base-4 encoding). Source: https://commons.wikimedia.org/w/index.php?curid=10856406, author: KES47.

So, in a nutshell, chromosomes encode information determining an organism's behavior (*genetics*), and in turn how well an organism can adapt to its environment, survive, and generate an offspring (*natural selection*). Information such as this may be

especially useful for people who require paternity testing, gender revelation, or even forensics. In the past few years, there have even been companies like Health Street (which does DNA testing in Kansas City), where they aim towards helping these individuals and organizations encrypt information.

Computer scientist John Holland, inspired by this mechanism, in early 1970s devised[5] a way to use these two principles to evolve artificial systems. One of his students, David Goldberg (considered for a long time the expert of reference on this topic), popularized this approach through its research and its book at the end of 1980s[6].

The core idea, shown in figure 4 in the next section, is to encode a solution as a chromosome and assign each chromosome to an organism. Then, naturally, it follows that based on how good a solution is, we derive how fit the corresponding organism is for its environment. Fitness, in turn, is a proxy for the chances of an organism to reproduce and pass its genetic material to the next generation through its offspring. Like in nature, the alpha-individuals of a population, the stronger (or faster, or yet smarter) ones, tend to have a higher chance of surviving longer and finding a mate.

But perhaps the best way to understand genetic algorithms is by seeing them in action. To that extent, and to keep things concrete, while we describe all the building blocks of this meta-algorithm, we'll develop an example along the way to help readers visualize how each component works. Do you remember the 0-1 knapsack problem? Check out this article, when we introduced this problem to model our "packing to Mars" situation. In it, we mentioned there is a pseudo-polynomial dynamic algorithm that can provide the absolute best solution, but it can be too slow when the capacity of the knapsack is large. Branch-and-bound approximated algorithms exist that can compute near-optimal solutions (with some guarantees on the upper bound and in reasonable time), although the efficient ones are usually quite complex[7].

To get a fast, clear, simple optimization algorithm, we will implement a genetic algorithm that can find approximated, near-optimal solutions to the knapsack problem.

Before we delve into the details of genetic algorithms, I suppose you could use a refresher. Let's quickly review the problem definition, and the instance we were using.

NOTE **In the generic 0-1 knapsack problem, we have a container with a limited capacity `M`, and a set of goods, each characterized by a weight (or any other measure of capacity) `w`, and a value `v`. If we add an item to our knapsack (or any generic container we decide to use!) it must be all of it, we can't add fractions of items. The**

sum of the weights of all available items exceeds the capacity of the knapsack, so we need to choose a subset of items, and the goal, therefore, is to choose the particular subset that maximizes the value of the goods carried.

The 0-1 knapsack article, we tackle a more concrete problem: given the list of goods shown (for your convenience) in table 1, we'd like to fill a crate that can carry at most 1 ton, not with as much food as possible, but with the largest total calories count possible.

As such, we could easily check that wheat flour, rice, and tomatoes could sum up to the maximum carriable weight, but they wouldn't give the maximum number of calories.

In chapter 1, we briefly described the key consideration that is used by the branch-and-bound heuristics to approximate this problem. It computes the value by weight (in this case, calories by weight) for each of the goods, and prefers food with higher ratios; although that's used as a starting point by the Martello-Toth algorithm, just choosing products in descending order of per-kilo values won't give us the best possible solution.

And, in fact, the dynamic programming algorithm that exactly solves 0-1 knapsack won't even compute this ratio; we will not use it here either, so it's not even shown in table 1.

Table 1. A Recap of the Available Goods for the Mission to Mars, with Their Weight and Calories

Food	Weight (kgs)	Total calories
Potatoes	800	1,502,000
Wheat Flour	400	1,444,000
Rice	300	1,122,000
Beans (can)	300	690,000
Tomatoes (can)	300	237,000
Strawberry jam	50	130,000
Peanut butter	20	117,800

If you want to read the article here it is again.

Now move on to discussing the main components of genetic algorithms (*GA*).

To completely define an optimization algorithm, we need to specify the following components:

- *How to encode a solution*-Chromosomes, for *GA*
- *Transitional Operators*-Crossover and mutations
- *A way to measure cost*-Fitness function
- *How system evolves*-Generations and natural selection

Chromosomes

As we mentioned, chromosomes encode a solution to the problem we need to solve. In the original work by Holland, chromosomes were only supposed to be strings of bits, sequences of zeros and ones. Not all problems can be encoded with a binary string, and so a more generic version was derived later, allowing *genes* (each of the values in a chromosome) to be continuous, real values.

Conceptually, those versions can be considered equivalent, but whenever we don't use binary strings for the chromosomes, we will likely require restrictions on the possible values they can store: the operators acting on organisms will have to be adjusted in order to check and maintain these constraints.

Luckily for us, the 0-1 knapsack is the perfect example to discuss the original genetic algorithm, because a solution to this problem can be encoded exactly as a binary string. For each item, a zero means we leave it on Earth, while a 1 means we add it to the crate going to Mars.

Figure 4 shows a bit-string representation for a chromosome that can be used in our example: two different solutions (or ***phenotypes***, in biological terms) can be represented by two chromosomes (or ***genotypes***). Their difference boils down to the difference between the two bit-strings.

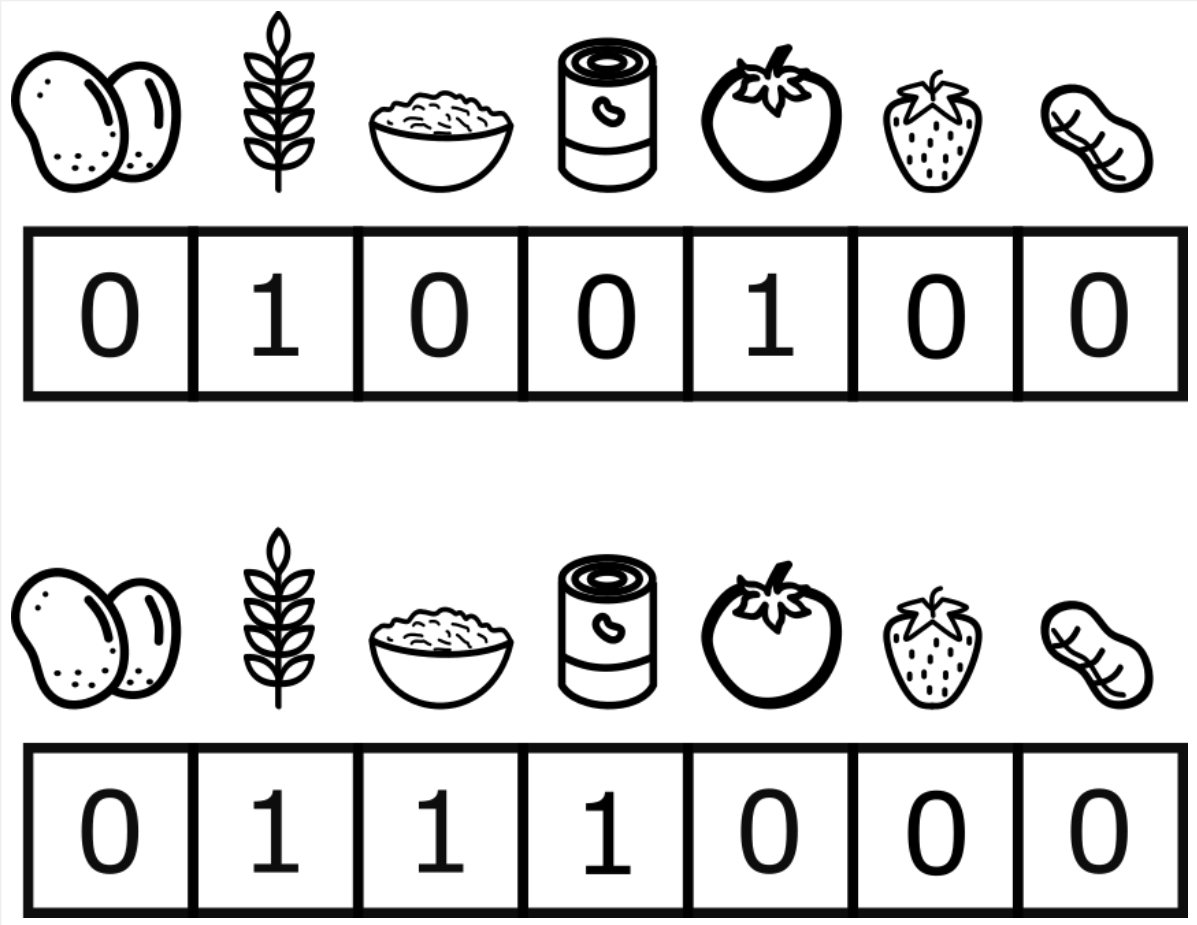


Figure 4. Examples of encoding chromosomes for the knapsack problem. Two figures are shown: the differences in the genotypes (the two strings) reflect in the phenotypes (the goods taken to Mars: wheat flour and tomatoes versus wheat flour, rice, and beans). The latter chromosome encodes the best solution for the instance of the problem summed up in table 1).

It's worth noting that for this example we have a 1:1 mapping between *genotype* and *phenotype*: a solution (the *phenotype*) is uniquely identified by its representation (the *genotype*) and so two organisms with the same genotype will translate to the same solution and in turn to the same fitness value. In the context of our 0-1 knapsack example, if two organisms have the same genotype, both solutions will add the same set of goods added to the crate to Mars.

The equivalence between genotype and phenotype is, however, not observed in nature, where the difference between them is clear, because genome only encodes developmental rules, not precise outcomes, and an organism is also determined by its interaction with the environment[8].

Likewise, in some simulated problems fitness is not always fully determined by genotype. One of the first examples of an application of genetic algorithms I studied was the work[9] of Floreano and Mondada at EPFL, where they evolved the neural networks of small two-wheel robots, simulating the evolution of gatherers, and later of predators and preys populations (in a sort of hide and seek game).

In this experiment, the genotype was the set of weights for a robot's neural network, which, incidentally, also completely identify the phenotype. Each organism's fitness, though, was later determined by its interaction with the environment and with other robots.

If we were to extend their original work by allowing online learning for the robots (today, it wouldn't hard to imagine applying *stochastic* or *mini-batch backpropagation* to evolve the NN's weight based on the input from the environment), then neither would the phenotype be completely determined by the genotype, at least not since the first update, because after that the way weights change is heavily influenced by the interaction of the robot with the environment.

With respect to figure 4, we also need to make one more thing clear: the actual representation of chromosomes depends on the actual, specific problem we are solving. In this case, on the specific instance of 0-1 knapsack (although a generic representation for all 0-1 knapsack can and should be designed).

If we want to design code for the genetic algorithm main method, instead, we should be concerned in designing organisms, for example modeling them with a class that will be provided the actual chromosome (and, as we'll see, the methods doing modifications on them) either at runtime through composition, as arguments (using *strategy* design pattern), or at compile time through inheritance (*template* design pattern).

NOTE **The genetic algorithm is a meta-algorithm, a template that is completed by the code specific to each problem tackled. As such, there is part of the code that is generic-the structural code defining the optimization technique-and another part that is specific to the problems. At this point, while describing the components of genetic algorithms, we'll provide the pseudo-code for the template, and since we are using the knapsack problem as an example, we'll also provide some pseudocode to implement examples of the specific methods for 0-1 knapsack, with these snippets clearly marked.**

Listing 1 shows a possible implementation of the `Individual` class, modeling each individual organism in the population evolved.

Listing 1. Class Individual

```
  class Individual
#1
     #type array
     chromosome
#2

     function Individual(chromosome)
#3

     function fitness()
#4
```

#1 Class `Individual` models each organism in the population to evolve.

#2 Each individual has a genome made of a chromosome (multi-chromosome organisms are, in theory, also possible, though not as common).

#3 We assume the chromosomes are provided as arguments to the constructor (implementation, which is trivial, isn't shown). Alternatively, there can be a static generator method on this class.

#4 Each individual can be queried for its fitness (possibly this needs to take arguments if the fitness depends on the environment).

Population

The most apparent difference between simulated annealing and genetic algorithms is that, instead of tuning a single solution, genetic algorithms will evolve a population of individuals.

Listing 2 shows a possible implementation for a method that initializes the population that will be used for the genetic algorithm. This is also a templated method, since it will have to be provided with the actual code for initializing the single chromosomes (which, again, will depend on the actual problem tackled).

Listing 2. Method initPopulation

```
  function initPopulation(size, chromosomeInitializer)
#1
     population ← []
#2
```

```
    for i in {1,...,size} do
        population.add(new Individual(chromosomeInitializer())
#3
    return population
```

#1 Method `initPopulation` takes the size of the population to create (assumed, or checked. to be positive) and a function to initialize individual chromosomes. It returns the newly created population.

#2 Init `population` as an empty list.

#3 Adds as many new individuals as necessary, each with a newly created genome.

There are a few different strategies that can be used for initialization. You can rely on random initialization to provide diversity in your initial population. This is the original and still most common approach, but in certain situations you might also add constraints on the chromosomes (to avoid invalid solutions) or even decide to externally provide an initial population (that can, for instance, be the output of a different algorithm or of a previous iteration).

For a generic instance of the 0-1 knapsack problem, since chromosomes are just bit strings, we are good with a random generator, as shown in listing 3.

Listing 3. | 0-1 Knapsack: chromosome generation

```
  function knapsackChromosomeInitializer(genesNumber)
#1
    chromosome ← []
#2
    for i in {0,...,genesNumber-1} do
        chromosome[i] ← randomBool().toInt()
#3
    return chromosome
```

#1 Method `knapsackChromosomeInitializer` takes the size of the chromosome to create (assumed, or checked, to be positive) and returns a random bit string (or, in this case, a bit array for the sake of simplicity).

#2 Init `chromosome` as an empty list.

#3 Generates as many random bits as necessary. Here we used a Java-like pattern, but it's also possible to directly use a method that returns random integers between `0`

and `1`.

There is a caveat: not all randomly generated strings are acceptable solutions for a given instance of the knapsack problem. For example, a string with all values set to 1 would certainly mean (for a non-trivial instance) that the weight constraint has been violated. As we'll see next, we can assign a low fitness to solutions that violates this constraint; alternatively, when we generate chromosomes, we can add a check to avoid violating it in the first place.

Fitness

Connected to the definitions of chromosome and organism, there is the notion of ***fitness*** of an individual. As you can see in listing 1, the `Individual` class has a fitness method returning a value that measures how well an organism adapts to its environment.

The term *fitness* naturally implies a maximization problem, because higher fitness is usually associated with better performance in an environment. Conversely, for many problems we usually want to express our problems' goals as *cost functions* to be minimized, so we have two choices: either implement the genetic algorithm template in such a way that a lower value for fitness means better performance or reformulate our cost functions according to the choice we made. For instance, if we need to find the point of highest elevation in a map, we can either implement a maximization algorithm or set the fitness function to `f(P)=-elevation(P)`, for any given point on the map, and stick with our usual minimization strategy.

For instance, take our example with the 0-1 knapsack problem. The goal is naturally expressed by a value function that we want to maximize, the sum of the nutritional values of the goods added to the crate.

The first chromosome in figure 4, for example, is `0100100`, which means we sum the calories of rice and tomatoes, for a total of `1,359,000` calories. What happens when we exceed the weight that the crate can carry, for instance with the chromosome `1111111`? To spot these edge cases, we need to check the total weight while computing the fitness and assign a special value (for instance, in this case, `0`) to those solutions that violate the constraint on the weight.

An alternative can be making sure that this situation never occurs, by carefully checking it in the operators that create and modify individuals (initialization, crossover, and mutations, as we'll see). This solution, however, may have consequences

on the optimization process, artificially reducing the plurality of features in the population.

At this point, if the template method that we have implemented for our genetic algorithms tries to maximize the fitness of individuals, we are golden. What if, instead, our implementation strives for lower fitness? For this particular case, we might have an easy solution: we sum the values of the goods discarded, corresponding to zeroes in a chromosome. For any given solution, the lower this sum is, the highest will be the sum of the calories of the goods in the crate. Of course, if we go this way, we have to assign a very high value (even infinity) to the solutions where the weight threshold is exceeded.

Natural selection

We now have the tools to create a single organism, generate a whole population, and check individual fitness. In turn, this allows us to mimic a basic natural process that guides the evolution of populations in the wild: ***natural selection***.

In nature, we see it everywhere: the strongest, fastest individuals, the ones that are best at hunting or at hiding tend to survive longer, and in turn (or in addition to that) they have greater chances of mating and transmitting their genes to the next generation.

The goal of genetic algorithms is to use a similar mechanism to have the population converge toward good solutions. In this analogy, the genes of individuals with high fitness[10] encode features that give them an advantage over the competitors.

In our knapsack example, a good feature could be including food with high calories-per-weight ratio. Solutions including potatoes will be individuals with low fitness, and that therefore will struggle to survive to the next generations. As an ulterior example, in the predator-prey simulation, the prey that developed a strategy to hide behind objects were more efficient in escaping predators.

Listing 4 shows, through some pseudocode, how natural selection (in genetic algorithms) works. This method regulates how a population transitions between current generation and the next one, taking the old population as input, and returning a new population, randomly created from the original organisms through a set of operators.

Listing 4. Natural Selection

```
  function naturalSelection(
        population, elitism, selectForMating, crossover, mutations)
#1
     newPopulation ← elitism(population)
#2
     while |newPopulation| < |population| do
#3
         p1 ← selectForMating(population)
#4
         p2 ← selectForMating(population)
         newIndividual ← crossover.apply(p1, p2)
#5
         for mutation in mutations do
             mutation.apply(newIndividual)
#6
         newPopulation.add(newIndividual)
#7
     return newPopulation
```

#1 Method `naturalSelection` takes the current population and returns a new population evolved from the input one. The function also needs to be provided the operators that will change the population, either in the arguments list or, for instance, through inheritance.

#2 The first thing to do is to initialize the new population. This can be done by simply creating a new, empty list. A more generic alternative, however, is to allow passing a method for this initialization, which could, for instance, be used to include elitism.

#3 Add new individuals, one by one, until the new population matches in size the old one.

#4 Select two individuals from the old population. The selection method is passed as an argument (equivalently, as for the others, it can be provided through inheritance).

#5 Combines the two organisms using crossover; details of this and the other operators are, as discussed, for the most part specific to the problem, and will be discussed later in the section.
In order for this template to work, however, we require that crossovers and mutations abide by a common interface, and be implemented as objects providing an `apply` method (we'll also discuss wrappers, to make this easy).

#6 For each possible mutation, apply it to the result of crossover (as we'll see, each mutation is characterized by a probability of happening, so it will be randomly decided

if any mutation is applied to organisms).

#7 Adds the newly generated individual to the output list.

This natural selection process is applied at each iteration of the algorithm. When it starts, it's always with a fully formed population outputted by the initialization method (its size is determined at runtime through a method argument, as we have seen). Individuals in the input population are evaluated for their fitness and then go through a process of selection that determines which ones are going to either pass to the next generation unaltered or propagate their genes through mating.

In genetic algorithm's simplest form, the new population is just initialized as an empty list (line #2), a list which is then *populated* (pun intended) with the new individuals resulting from selection and mating (#3-#5).

For mating, there are, obviously, differences with respect to the biological analogous. First and foremost, in genetic algorithms sexual reproduction between two ***asexual*** organisms is generally[11] used. Moreover, the recombination of genetic material between the two parents doesn't follow any biologically meaningful rules and the actual details of how crossover is performed are for the most part left to the specific problem that is solved.

Finally, we add mutations to the genetic material after crossover, modeling crossover and mutations as separate phases.

Figure 5 illustrates the general mechanism with an analogy to the biological process.

We start with an initial population where individuals have a common basis, but also some peculiar characteristics. The diversity and variance in the population largely depends on which stage of the simulated evolution we are in. As we'll see, our aim is having greater diversity at the beginning of the simulation and then have the population converge toward a few homogenous, high-fitness groups.

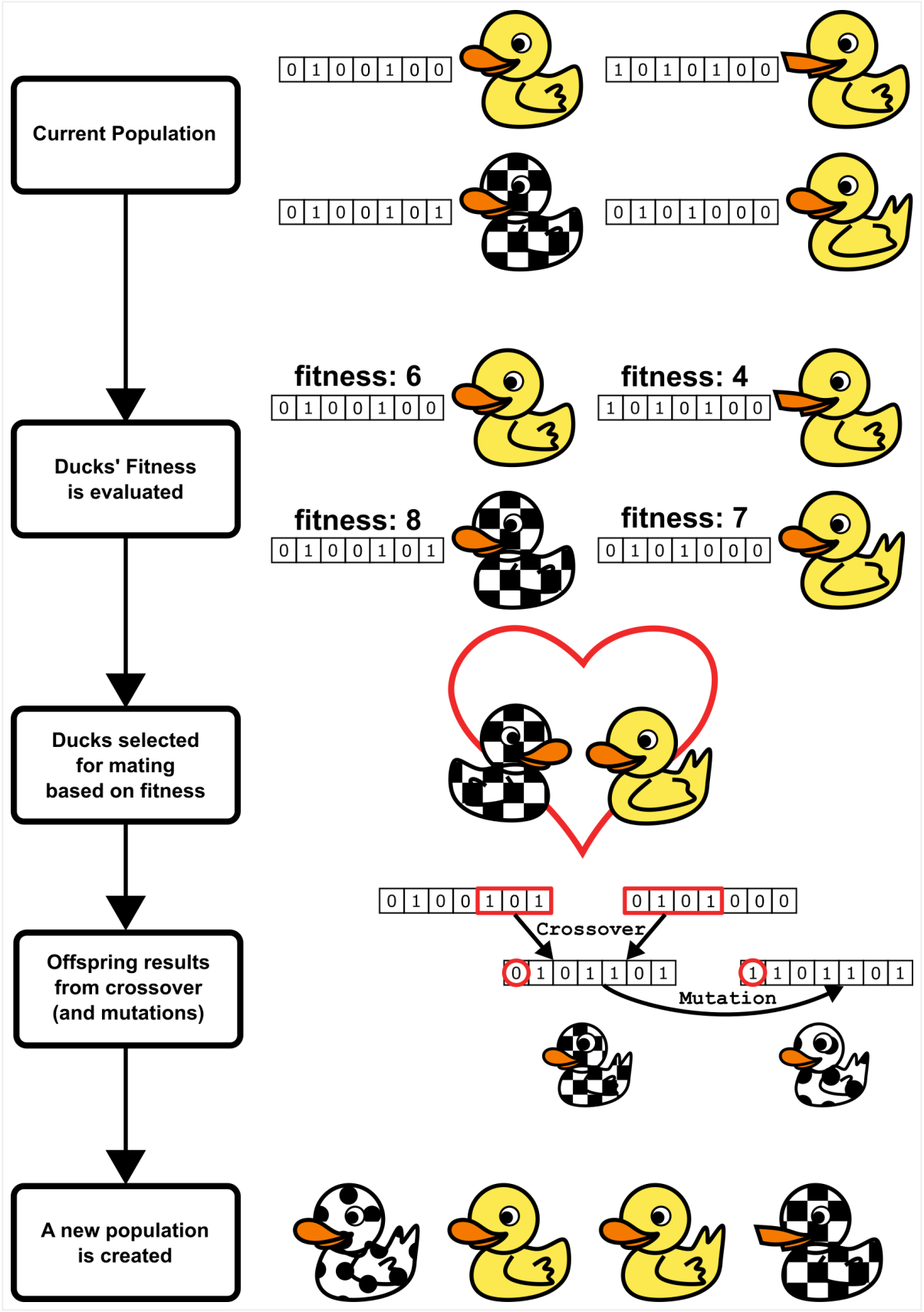


Figure 5. Natural selection in genetic algorithms, in an analogy with the biological process.

In our example, our population of ducks shows a few unique features, from the shape of their beak to their color/pattern, to the shape of wings and tails. If you look closely, you should be able to spot them and to figure out how these changes are encoded in their chromosomes.

Yes, because, as we have discussed, each feature in individuals' phenotype is determined by a difference in their genotype, a `1` that becomes a `0`, or vice versa (or maybe a few bits that need to flip together, or even, if we move past binary strings, a gene assigned with a different real number).

In order to perform selection, we need to evaluate each *duck*'s fitness. This can be as easy as feeding each chromosome to a function or as complicated as running a simulation for hours (as in the predator-prey setting we had described) and seeing which individuals survived longer or performed a real-world task better.

We'll only talk about selection for mating in the next subsection, but one thing is generally true regardless of the details of the selection mechanism chosen: organisms with higher fitness will have greater chances of being chosen to pass their genes to the next generation. This is crucial, maybe the only really crucial part, because without this "meritocracy", no optimization would be performed.

Once we have selected two individuals, we need to recombine their genetic material. We'll also tackle crossover and mutations in their own section, but one thing that we have already mentioned is that these operators, acting on chromosomes, are largely determined by the specific problem that is being solved.

In our example in figure 5, our little duckling takes its feathering pattern and color from one parent and its tail shape from the other. Then a further mutation kicks in changing the feathering again from a checkerboard pattern to a polka dots pattern.

The new population, formed repeating this process many times, should show a larger presence of those genes/features associated with high-fitness individuals in the initial population.

Now that we have seen how the broad natural selection process works, it's time to dive into the specifics of its steps.

Select individuals for mating

Let's start with selection: we'll go back to our knapsack problem example to explain it better.

There are several techniques that can possibly be used for selecting, at each iteration, the individuals that will mate or progress; some of the most successful alternatives are:

- Elitism
- Thresholding
- Tournament selection
- Roulette wheel selection

But there are also many other possible techniques used. Considering our restricted list, the first two techniques are filters on the old population, while the remaining two are methods to select organisms for mating.

Elitism and thresholding, illustrated in figure 6, are used to decide what organisms can or can't transmit their genomes to the next generation.

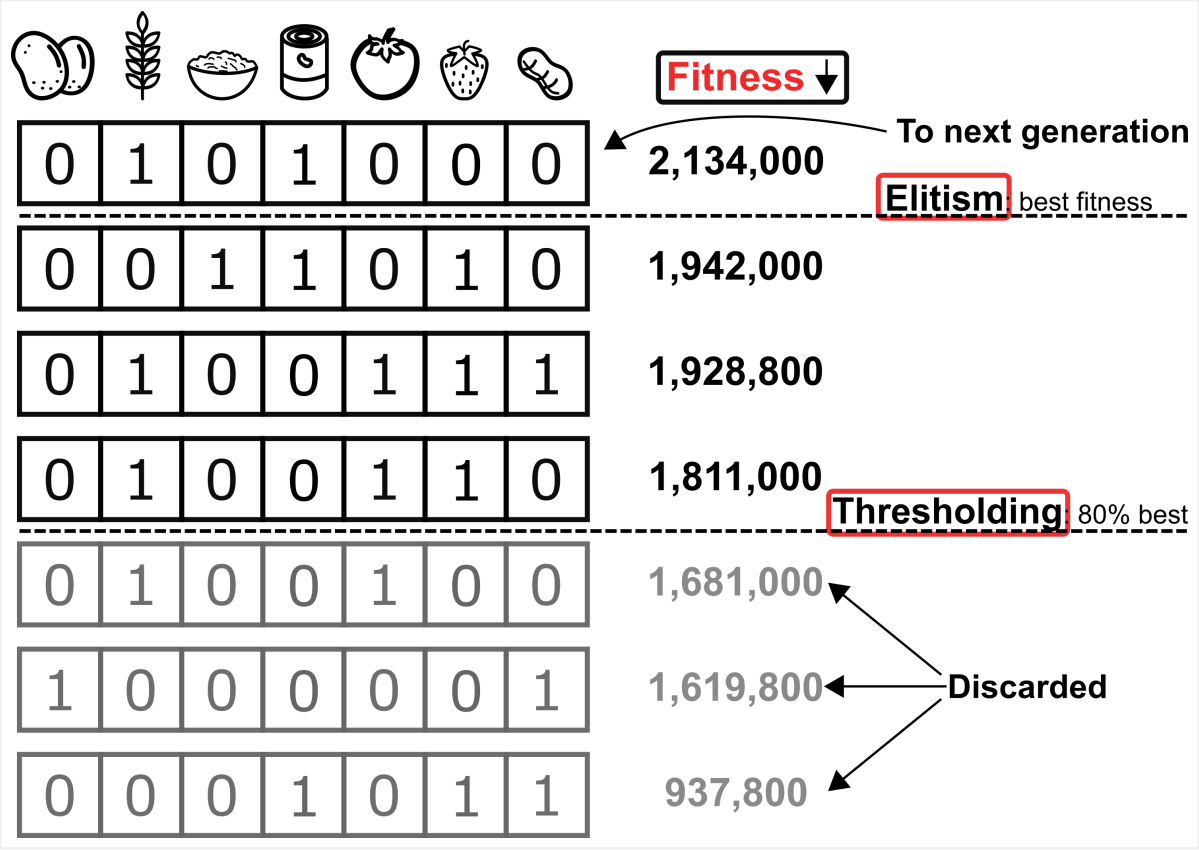


Figure 6. Elitism and thresholding illustrated with an example (0-1 knapsack problem). Elitism takes the best individual(s) (one or a few of the highest-fitness entries) and directly brings them to the next generation without mating or mutations. Thresholding takes care of the opposite problem. It's a hard barrier that stops low-fitness individuals

from passing their genes to the next generations. It's possible to discard a certain fixed number of individuals, or all individuals below some threshold. For instance, in this example (where we maximize fitness value), all individuals below 80% of the fitness value of the best individual will be discarded and never selected for mating.

Elitism allows the best individuals to pass completely unaltered to the next generation. Applying or not this principle is up to the algorithm designer; as always, it might work better with some problems and worse with others.

Without elitism, all organisms in current iteration would be replaced in the next one, so that all organisms would "live" exactly one generation; with this workaround, we can somehow instead simulate a longer lifespan for particularly well-fitting individuals-the better their fitness in comparison to the average population's, the longer they will be kept.

In turn, this will also ensure that the genes of the *alpha organisms* will be present, untouched, in the next generation too (although it's not certain they will still be the top-fitness individuals in the next generations . . .).

One notable effect of using elitism is that the fitness of the best element in the population is monotonically improving over generations (that, however, might not be true for the average fitness).

Thresholding has the opposite goal: it will prevent lowest-fitness organisms from transmitting their genes to the next generations, which in turn reduces the probability that the algorithm explores less-promising areas of the fitness function's landscape.

The way it works is simple. We set a threshold for the fitness of the individuals that are allowed to be selected for mating and ignore the ones outside the requirements. We can discard a fixed number of organisms, such as the five individuals with the worst fitness value or discard a variable number of individuals based on their fitness value.

In the latter scenario, the threshold on the fitness value is usually set dynamically (changing each generation) and based on the fitness value of the best individual for that generation. With a static, absolute value (which would require domain knowledge to be set), the risk is that it would be ineffective (resulting in the whole population filtered in) or even worse, fatal for the simulation when it's too stringent, resulting in all, or almost all, individuals being filtered out in the early stages.

In the example in figure 6 we suggested to set the threshold to 80% of the best individual's fitness value[12]. Whether or not thresholding should be applied and what the threshold ratio should be, completely depend on the actual problem to be solved.

After filtering the initial population, and possibly escorting the best individuals to the next generation, we are still tasked with recreating the remaining individuals in the new population. To do so, we implement *crossover*, which will be discussed in the next sub-section, as a way to generate a new organism from two parents, and *mutations*, to provide genetic diversity to the new generations. The necessary step before being able to apply crossover is, therefore, selecting those two parents. As you can see in listing 4, we are going to perform this selection several times at each iteration.

There are many possible ways to select individuals for mating. We are going to discuss two of the most common here.

Tournament selection is one of the simplest (and easiest-to-implement) selection techniques used in genetic algorithms; it's illustrated through an example in figure 7. Conceptually, we randomly select a handful of organisms and then have them "compete" in a tournament where only the winner gets the right to mate. This is similar to what we see everywhere in wildlife, where (usually) male adults fight for the right to mate with females (ever watched a documentary on deer crossing horns during mating season?).

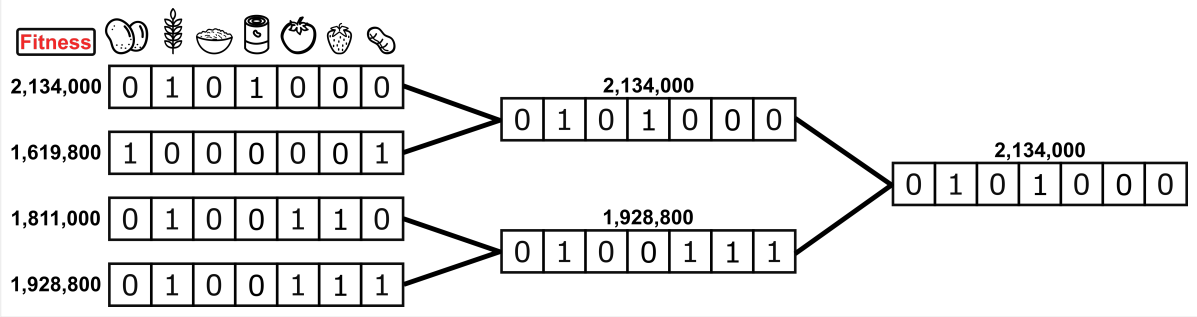


Figure 7. Tournament selection applied to our instance of the 0-1 knapsack problem.

Now, in reality we don't actually code a tournament, unless we are running a tournament simulation[13]! As shown in listing 5, we just randomly select k individuals and take the one with the best fitness. The exact number, k, can vary depending on the problem and on the size of the population, but usually somewhere

around 3 to 5 elements could be a good choice. Consider that the larger is the pool of individuals participating in the tournaments, the lower is the chance for low-fitness individuals to be chosen.

Listing 5. Tournament Selection

```
  function tournamentSelection(population, k)
#1
     bestFitness ← lowestPossibleFitness()
#2
     for i in {1,..,k} do
#3
         j ← randomInt(|population|)
#4
         if isHigher(population[j].fitness, bestFitness) do
#5
             bestElement ← population[j]
             bestFitness ← bestElement.fitness
#6
     return bestElement
```

#1 Method `tournamentSelection` takes current population and the number of elements that should participate in each tour, and returns an individual, the best of the ones selected for the "tournament".

#2 Init the temporary variable storing the best fitness found. Since this is a generic method and we don't know if, in concrete problems, the best fitness will mean highest or lowest values, we use a helper function returning the "lowest" possible fitness, `0` for functions that will be maximized, infinity for the ones to be minimized.

#3 Repeat `k` times.

#4 Randomly selects an index between `0` and the population's size. Indirectly, this selects an individual. In this implementation there are no measures to avoid duplicated selections, which can be fine when the population size is large (and consequently probability of a duplicates low), but still, you need to be aware of this.

#5 Checks if this element is better than the current best; again, we use a helper function to abstract on the meaning of good/high fitness.

#6 If we found a new best, just updates the temporary variables.

For instance, for the third-highest individual to be chosen, neither the best nor the second-best fitness organisms can be chosen (and, of course, the third-best must be), so the probability this happens with a pool of k individuals is[14]:

```
1/n * [(n-2)/n]k-1
```

assuming `n` is the size of the population.

For the lowest fitness individual (after, possibly, applying thresholding), it becomes:

```
1/n * [1/n]k-1 = 1/nk
```

The generic formula for the `m` -th best fitness becomes:

```
1/n * [(n-m+1)/n]k-1
```

As you can see, beyond the details and the actual exact probability, the chances of any individual (but the first) are decreasing exponentially with `k` (while polynomially with `m`).

It goes without saying that we need to apply tournament selection twice to get the pair of parents we need to generate a single element in the new population.

*Roulette wheel **selection*** is definitely more complicated to implement than tournament selection, but the high-level idea is the same: higher-fitness individuals must have more chances to be selected.

As we have seen, in tournament selection the probability that an element with low fitness is chosen decreases polynomially with the rank of the element (its position in the list of organisms sorted by fitness); in particular, since the probability will be `O([(n-m)/n]` k `)` the decrease will be super-linear, because `k` is certainly greater than `1`.

If, instead, we would like for lower-fitness elements to get a real chance of being selected, we could resort to a fairer selection method. One option is assigning the same probability to all organisms, but then we wouldn't really reward high-fitness individuals anymore[15]. A good balance would be, for example, making sure that the probability of selecting each individual is proportional to its fitness.

If we decide for the latter, we can use the roulette wheel selection. Each organism is assigned a section on a "roulette wheel" whose angle (and in turn the length of the arc subtended) is proportional to the fitness of the organism.

One way to obtain this is to compute the sum of the fitness of all individuals and then, for each one, see the percentage of the cumulative fitness for the whole population it accounts for[16]. For instance, figure 8 shows how to apply this to our example population for the knapsack problem. The angles of each organism's sector are computed with the formula $\theta=2\pi*f$, where f is the normalized fitness of the given individual, aka the fraction of the individual's fitness over the total sum for the whole population.

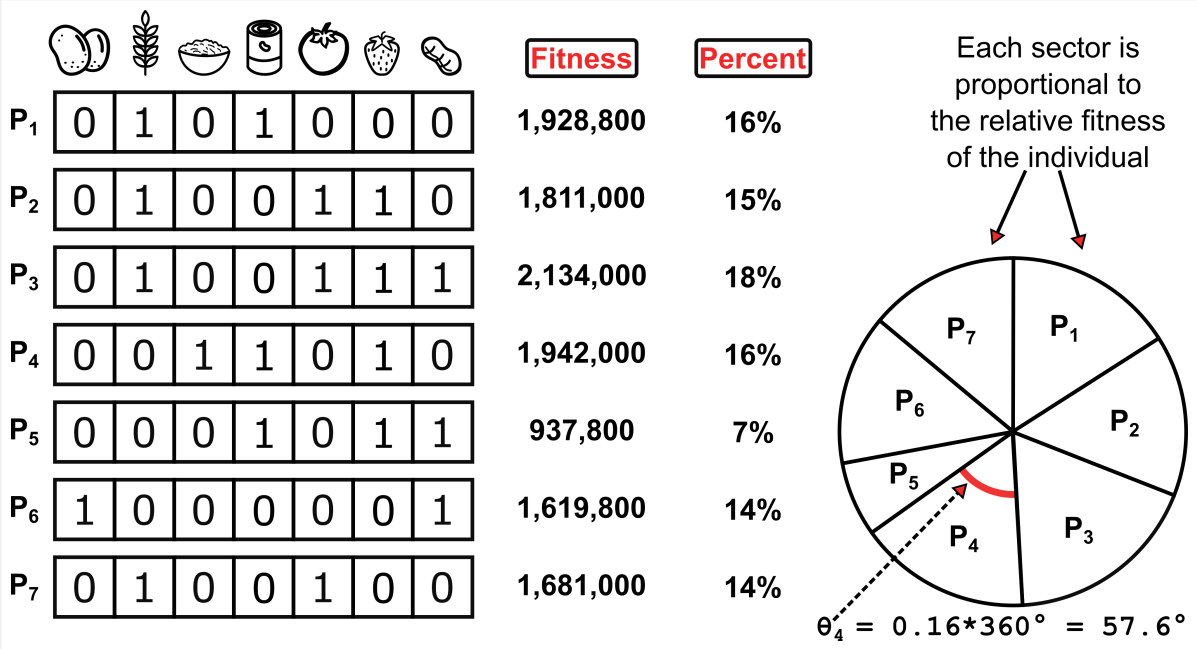


Figure 8. Roulette wheel selection once again applied to the example of 0-1 knapsack problem we are describing in this section.

Then each time we need to select a new parent for mating, we spin the wheel and see where it stops. Higher fitness will have a larger probability of being picked, but lower ones still have a shot.

NOTE Notice that if we choose to use rank instead of fitness, we'll need to sort the population first (time required: O(n*log(n))). Sticking to fitness, instead, we only need a linear number of operations to compute the total and the percentages, and since this

is computed at each iteration, for large populations we can have a relevant saving. Likewise, tournament selection as well doesn't require prior sorting of the population.

When it comes to implementing this technique, however, we are clearly not going to bother with building an actual wheel (not even a simulated one)!

The easiest way we can achieve the same result is by building an array like the one shown in figure 9, where the `i` -th element contains the sum of the normalized[17] fitness of all the individuals in current population (taken in no particular order).

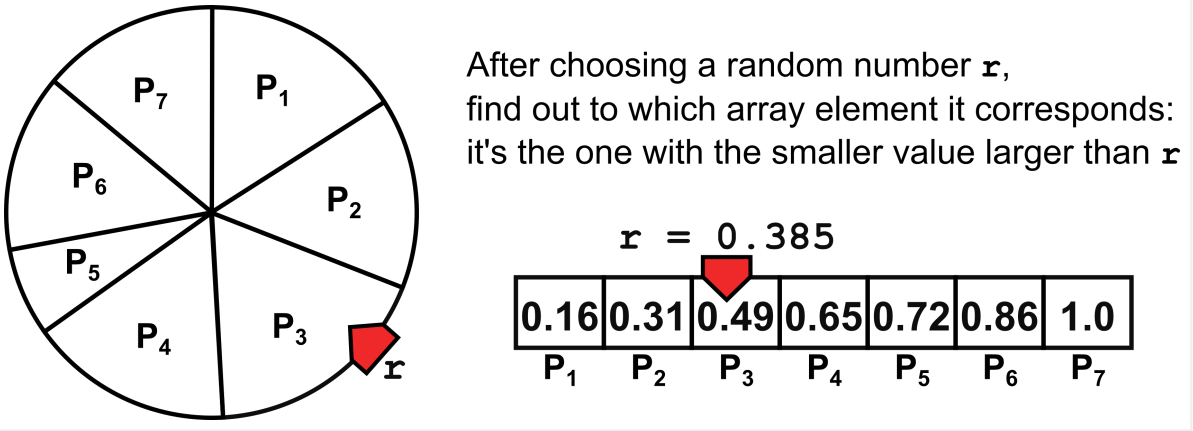


Figure 9. Roulette wheel selection in practice. We use an array with cumulative percentages, the difference between `A[i]` and `A[i-1]` is exactly the ratio between the `i` -th element's fitness and the sum of all elements' fitness.

For example, if we refer to the population shown in figure 8, the first element of the wheel-array holds the `population[0].normalizedFitness = 0.16` , the second element holds `population[0].normalizedFitness + population[1].normalizedFitness = 0.16 + 0.15` , and so on, as summarized in figure 9. Notice that the last element sums up exactly to 1. Ideally, we can imagine a hidden first element, not shown in the figure, whose value is 0 (we'll see that this helps writing cleaner, succinct code).

To select an element, we draw a random number `r` from a regular distribution or real numbers between `0` and `1` (not included). Equivalently, in the wheel analogy, we would draw a random angle `θ` between 0 and 360 (not included) to find how much we should spin the wheel. The relation between the two numbers would be `θ=r*360` .

Listing 6 shows the method to generate a wheel-array like the one in figure 9.

To find where our wheel's pin is pointing, we need to run a search on the array. We look for the smallest element that's larger than r, the random number drawn by the selection routine. We can modify the *binary search* algorithm to perform this task and limit the runtime to `O(log(n))` for each element selected; therefore, for each iteration, we will have to perform `O(n*log(n))` operations to select parents and apply crossover.

Alternatively, a linear search is also possible. It's a lot easier to write and less likely to be buggy, but that makes the runtimes grow to `O(n)` and $O(n^2)$ respectively.

Listing 6. Creating a selection roulette wheel

```
  function createWheel(population)
#1
     totalFitness ← sum({population[i].fitness, i=0,...,|population|-1})          #2
     wheel ← [0]
#3
     for i in {1,..,|population|} do
#4
         wheel[i] ← wheel[i-1] + population[i-1].fitness / totalFitness
#5
     pop(wheel, 0)
#6
     return wheel
```

#1 Method `createWheel` takes current population and returns an array whose generic `i`-th elements is the cumulative fitness of all population's individuals, from the first to the `i`-th.

#2 Computes the total sum of all individuals' fitness values (here we use the simplified notation discussed in appendix A).

#3 Initializes the array for the "roulette wheel". We set the first element to `0` for convenience, so that we don't have to handle a special case for the first individual outside the next `for` loop.

#4 Cycles through all individuals in the population.

#5 Each element in the roulette wheel's array is the sum of the normalized fitness of all individuals before it (already stored in `wheel[i-1]`) plus the ratio between current individual's fitness and the sum of all organisms' fitness.

#6 Optionally, we can now drop the first element of the array, containing a `0`. In many languages, this operation requires `O(n)` assignments, so we might want to avoid it. If

we keep the first value, the search method can easily take that into account, for instance by starting search from element at index `1` and subtracting one from the index found before returning.

Choose wisely depending on the time-constraints you have, and your domain knowledge. We leave the implementation of your preferred method of search as an exercise, and I strongly suggest you start with the simplest method, test it thoroughly, and then, if you really need the speedup, attempt to write the binary search method and compare their results.

Crossover

Once we have selected a pair of parents, we are ready for the next step. As we outlined in listing 4, it's time for crossover.

We mentioned this already, but just to give you a quick recap: crossover simulates mating and sexual reproduction of animal[18] populations in nature, stimulating diversity in the offspring by allowing the recombination of subsets of features from both parents in each child.

In the analogous algorithmic process, crossover corresponds to wide-range search of the fitness function landscape, since we generally recombine large sections of the genome (and hence of the solutions) carried by each of the parents. This is equivalent to a long leap in the problem space (and cost function landscape).

For instance, take our example problem: 0-1 knapsack and packing goods for a journey in space. Figure 10 shows a possible definition for the crossover method for this problem: we choose a single crossover point, an index at which we cut both chromosomes; then one part of each chromosome (at the opposite sides of the crossover point) will be used for recombination. In this case, we just glue the two parts together.

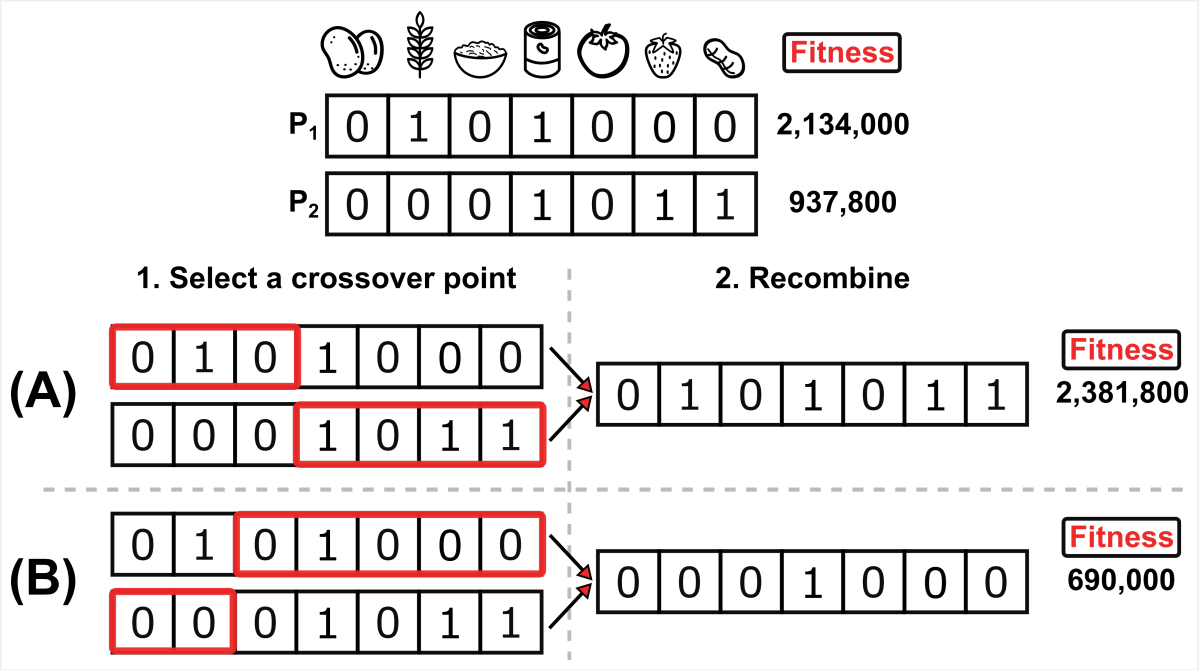


Figure 10. Examples of crossover for the knapsack problem. Once the two parents are selected, we need to choose a crossover point, and then recombine the two genomes. Depending on this choice, we can have an improvement (A) or a worsening (B) in fitness for the children.

Randomness is and should be a huge part of crossover. In our example we can see it in the choice of the crossover point. Some choices (as in figure 10.A) will lead to improvements in the child's fitness, while other crossover points (as in figure 10.B) will lead to poorer or sometimes even disastrous results.

This single-crossover-point technique, if you noticed, actually produces two pairs of opposite sub-sequences (left-right and right-left of the crossover point). While in our example we only used one of the pairs and discarded the other, some implementations of the genetic algorithm use both pairs to produce two children with all the possible combinations[19] (either keeping them both or just keeping the one with better fitness).

But nobody says we need to stick with the single-point crossover, that's just one of many alternatives. For instance, we could have a two-point crossover, selecting a segment of the first chromosome (and, implicitly, one or two remaining segments of the other), or we could even randomly select each value from one of the parents' chromosomes. Both examples are shown in figure 11.

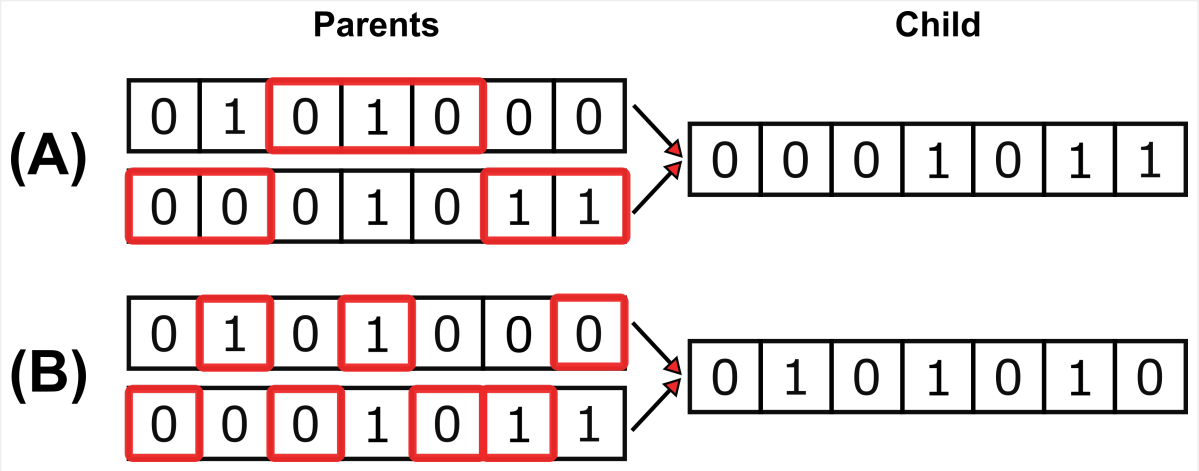


Figure 11. More examples of crossover for the knapsack problem. (A) Two-point recombination: select a segment from the first chromosome, and the rest from the other. (B) For each gene (aka value), randomly decide from which parent it will be copied.

And many more alternatives are possible. As we said before, crossover can only be defined on the actual chromosomes, and depending on their structure and constraints (it ultimately depends on the structure of the candidate solution), different way of recombining chromosomes are possible. We'll see plenty of examples later in this chapter.

One last thing: it is customary to associate a *crossover chance*, aka crossover ratio, to our crossover operators. This represents the probability that crossover/mating actually happens between the selected individuals. For instance with a crossover chance of 0.7, there is a 70% chance that crossover does happen for any pair of organisms selected for mating.

As shown in listing 7, every time we apply crossover, we randomly decide what to do, based on the crossover chance. If the choice is not to apply crossover, then there are a few alternatives; most commonly we randomly select one of the parents that move to the next step. Notice that this is different from *elitism* because the organism outputted by crossover will still undergo mutations, while any individual selected by elitism will be copied completely unaltered to the new population.

Listing 7. A wrapper class for Crossover operator

```
  class CrossoverOperator
#1
```

```
    #type function
    method
#2
    #type float
    chance
#3

    function CrossoverOperator(crossoverMethod, crossoverChance)
#4

    function apply(p1, p2)
#5
        if random() < this.chance then
            return new Individual(method(p1.chromosome, p2.chromosome))
#6
        else
            return choose(p1, p2)
#7
```

#1 Class `CrossoverOperator` models a wrapper for the actual crossover method specific to individual problems. While the crossover method changes with the problem definition, it's good to have a uniform, stable API that can be used in the main method for the genetic algorithm.

#2 We need to store a reference/pointer to the actual method to run.

#3 We also need to store the probability that this method is applied to the parents.

#4 The constructor (body omitted) takes two arguments to initialize the two class' attributes.

#5 This method will take two organisms, the two parents, and output the organism that will be passed to the next generation.

#6 With probability proportional to `crossoverChance`, it will apply the crossover method passed at construction, and then return a new individual created recombining (in some, problem-dependent way) its parents' chromosomes.

#7 If the crossover is not to be applied, then we have a choice of what to return: in this case, we just return one of the two parents, randomly choosing it.

Remember that listing 7 shows a wrapper for the actual crossover operator, compatible with the template we provided in listing 4 for natural selection's generic template. The actual method will be crafter and passed every time a specific problem instance is addressed.

Listing 8, instead, shows the single-point crossover operator that we discussed for the 0-1 knapsack problem.

Listing 8. | 0-1 Knapsack: crossover

```
function knapsackCossover(chromosome1, chromosome2) #1
i ← randomInt(1, |chromosome1|-1) #2
return chromosome1[0:i] + chromosome2[i:|chromosome2|] #3
```

#1 Method `knapsackCossover` takes two chromosomes (as bit strings) and returns a new chromosome obtained by recombining the head of the first one, with the tail of the second one.

#2 Chooses a cut point, making sure at least one gene is taken from each parent.

#3 Returns the combination of the head of `chromosome1` (up to index `i`, excluded) and the tail of `chromosome2` (from the index `i` to the end).

Mutations

After a new organism is created through crossover, the genetic algorithm applies mutations to its newly recombined genome. In nature, as we have seen, the recombination of parents' genomes and mutations happens simultaneously[20] during sexual reproduction. Conceptually, this works analogously in genetic algorithms, although the two operators are implemented separately. If crossover is regarded as wide-range search, mutations are often used for small-range, local search.

We have seen that crossover promotes diversity to the new population, so why do we need mutations?

The answer is simple and best provided with an example (shown in figure 12). The population considered in this example shares the same values for three of the genes. This means that, no matter how you recombine the organisms' genes during crossover, the resulting children will all have beans and strawberries included in the solution and potatoes excluded from it. This is a real risk for problems where chromosomes are large (carrying a lot of information), especially if the length of the chromosome is larger or comparable to the size of the population. The danger is that, no matter how long we run our simulation, or how many iterations and crossover we perform, we won't be able to flip the values for some genes, which in turn means that the areas of the problem space that we can explore are limited by the choice of the initial population. And this, of course, would be a terrible limitation.

Figure 12. An example of a population where, without mutations, there wouldn't be enough genetic diversity. All individuals have the same value for three of their genes.

To prevent this issue and increase the diversity in the genomic pool of our population, we add a mechanism that can change each gene independently and for any organism.

In Holland's original work, as we had mentioned, chromosomes were bit strings, and the mutation was also conceived as domain-agnostic. The mutation operator would be applied to each organism's chromosome, and for each gene (that is, each bit) it would toss a coin[21] and decide if the bit should be flipped. This is shown in figure 13.

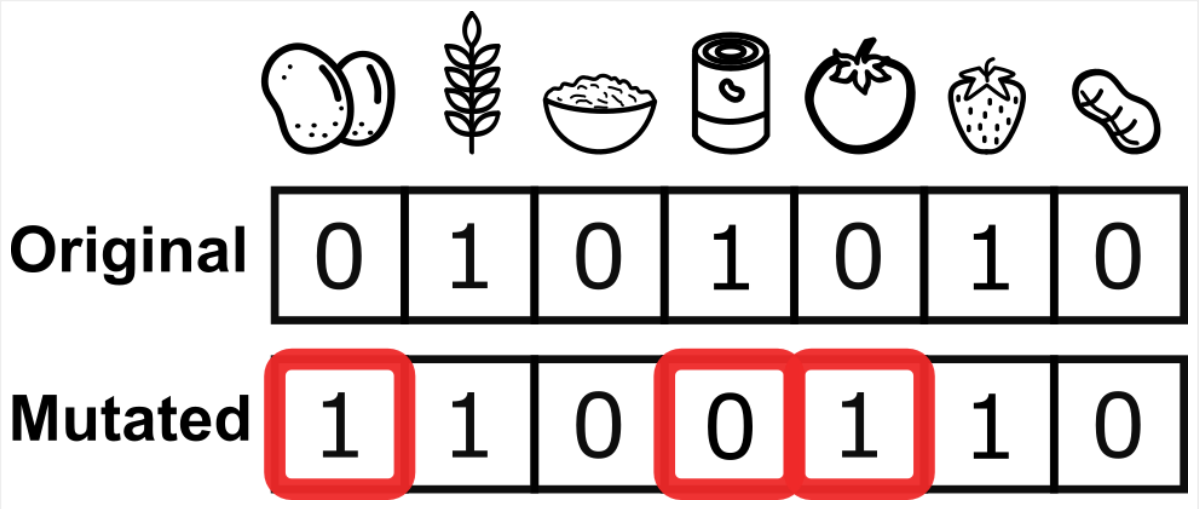


Figure 13. An example of mutation for the 0-1 knapsack problem. The mutation rate here is overly boosted (for the sake of presentation). In a real scenario it would likely be set somewhere between 1% and 0.1% (also depending on the size of the chromosomes). A too-large mutation ratio can hinder the stability of the algorithm's convergence toward the minimum, voiding the benefits of crossover and selection.

In modern genetic algorithms, however, chromosomes can take different shapes and have constraints, so mutations can become much more complex or are applied to the chromosome as a whole, rather than to single genes. As we'll see for TSP, for instance, a mutation operator can be applied (with a certain probability) just once to the whole chromosome (instead that to each of its genes) and swap two vertices in the tour.

The wrapper class for the mutation operator is identical to the one for crossover shown in listing 7, apart from minor differences in handling the arguments, especially the ones to the wrapped methods. We only have one organism to pass, but it's a good idea to also forward the mutation chance to the wrapped method. This is necessary for bit-wise mutations, such as the one we discussed for the knapsack problem, implemented in listing 9.

Listing 9. 0-1 knapsack: Bit-wise mutation

```
  function knapsackMutation(chromosome, mutationChance)
#1
     for i in {0,..,|chromosome|} do
#2
         if random() < mutationChance then
             chromosome[i] ← 1 – chromosome[i]
#3
```

```
    return chromosome
```

#1 Method `knapsackMutation` takes a chromosome (as a bit string) and a probability of mutation (as a real number between `0` and `1`) and applies mutations to the chromosome. The method, in this implementation, alters the argument. It's possibly and often cleaner to clone the input and return a new object as output, unless this cloning becomes a bottleneck. Weigh the pros and cons and run some profiling before making a choice.

#2 Cycles through each gene (that is, bit) in the chromosome.

#3 With probability `mutationChance(*100)` flip current bit.

The genetic algorithm template

Listing 10 provides an implementation of the main method for the genetic algorithm template, and also concludes our discussion on the 0-1 knapsack problem. We have all we need now to actually run the algorithm on our instance and find out that the best possible solution is bringing wheat flour, rice, and beans. To see a solution in action, and possibly apply it to larger problem instances with hundreds of possible items to pack, you can use the implementation of genetic algorithm provided by JsGraphs on GitHub[22] and implement the methods for crossover, mutations and random initialization that we discussed in this section. It will be a great exercise to test your understanding of this technique and delve deeper into its details.

Listing 10. A generic implementation of the genetic algorithm

```
  function geneticAlgorithm(
        populationSize, chromosomeInitilizer, elitism, selectForMating,
        crossover, mutations, maxSteps)
#1
    population ← initPopulation(populationSize, chromosomeInitilizer)          #2
    for k in {1..maxSteps} do
#3
        population ← naturalSelection(
            population, elitism, selectForMating, crossover, mutations)
#4
    return findBest(population).chromosome
#5
```

#1 Method `geneticAlgorithm` is a template method that implements the backbone of a genetic algorithm and can be adapted to run on several problems, by passing the

specialized methods for the concrete problems' instances.

#2 Initializes the population.

#3 Repeats `maxSteps` times. Each iteration is a new generation in the simulation.

#4 Let natural selection take its course . . .

#5 Finally, finds the best individual in the population and returns its chromosome (which is the best solution found). Method `findBest` can also be supplied as a parameter, if needed.

When does the genetic algorithm work best?

In the book, I present a few techniques that can be used in optimization problems to find near-optimal solutions without exploring the whole problem space. Each of them comes with some strengths and some pain points:

- ***Gradient descent*** converges fast but tends to get stuck in local minima. It also requires the cost function to be differentiable (as such, it couldn't be used in experimental settings or game theory, for instance in the predator-prey robotic evolution experiment described earlier in this section).
- ***Random sampling*** overcomes the need for differentiability and the issue with local minima, but it's (unbearably) slow in converging (*when* it manages to converge).
- ***Simulated annealing*** has the same advantages of random sampling but evolves in a more controlled and steady way toward minima. Nevertheless, convergence can still be slow, and it has a hard time in finding minima when they lie in narrow valleys.

At the beginning of the article we discussed how the genetic algorithm can overcome many of the issues of these heuristics; for instance, it can speed up convergence, compared to simulated annealing, by keeping a pool of solutions and evolving them together.

To conclude, I'd like to provide yet another criteria that can help you choosing which optimization technique you should use. It involves another term, ***epistasis***, that is borrowed from biology. Its mean is *gene interaction*, and in our algorithmic analogy expresses the presence of dependent variables; in other words, variables whose value depends on other variables.

Let's first go through an example to explain this better.

Each gene on a chromosome can be considered a separate variable that can assume values within a certain domain. For the 0-1 knapsack problem, each gene is an independent variable, because the value it assumes won't directly influence other

variables. (If we enforce the constraint on the weight of each solution, however, flipping a gene to `1` might force us to flip to `0` one or more other genes. That's a loose indirect dependency, anyway.)

For the TSP, we assign a vertex to each gene with the constraint that there can't be any duplicate. Assigning a variable will impose constraints on all other variables (that won't be able to be assigned the same value), so we have a more direct, although still loose, dependency.

If our problem was optimizing the energy efficiency of an house that is being designed, and some variables were its area, the number of rooms, and the amount of wood needed for the floors, the latter would be dependent on the other two, because the area to cover with wood floor would depend on the square feet and rooms of the house. Changing the size of the house would immediately force us to change the amount of wood used, if the floor's thickness remains fixed (the possibility to change its thickness makes it meaningful having a separate variable for this).

For 0-1 knapsack, we say that the problem has low variables interaction and so low epistasis. The house optimization has high epistasis, while for the TSP, its epistasis is somewhere in the middle.

Now, the interesting thing is that the degree of variables interaction contributes to the shape of the landscape of the objective function that we want to optimize. The higher the epistasis, the wavier the landscape will likely look.

But above all, when the degree of interaction is high, varying one variable also changes the value of the other variable, and this means making a bigger leap in the cost function's landscape and problem domain, making it harder to explore the surroundings of solutions and fine-tune the algorithm's results.

As such, knowing the epistasis of a problem can guide our choice of the best algorithm to apply:

- With low epistasis, minimum seeking algorithms like gradient-descent works best.
- With high epistasis, it's best to prefer random search, so simulated annealing, or for very high variables interaction, random sampling.
- What about genetic algorithms? Turns out they work best in a wide range of medium to high epistasis.

Therefore, when we design the cost/fitness function for a problem (which, let me state this clearly once again, is one of the most crucial steps to a good solution), we need to be careful about the degree of interaction in order to choose the best technique that we can apply.

Even more, during the design phase we can try to reduce the dependent variables, whenever it's possible. This will allow us to use more powerful techniques and ultimately get better solutions.

Now that we have concluded our discussion on the components and theory of genetic algorithms, we are ready to delve into a couple of practical applications, to see how powerful this technique is.

If you want to learn more about the book, you can check it out on Manning's liveBook platform here.

[1]Technically, it follows the path of steepest descent, with a *locally-optimal*, greedy choice at each step. These choices usually aren't *globally optimal*, and hence gradient descent is not guaranteed either to reach global optimum, unless the cost function has a particular, convex shape with a single minimum point.
[2]
[3]With the notable exception of viruses, which are just DNA or RNA encapsulated in a protein coat.
[4]Approximately the same, as there can be small local variations, for various reason including copy errors.
[5]Holland's book was originally published in 1975. You can currently find the 1992 MIT press edition: *Holland, John Henry. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. MIT press, 1992.*
[6]Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA: Addison-Wesley.
[7]For example, the Martello-Toth algorithm which is one of the state-of-the-art solutions: Martello, Silvano, and Paolo Toth. "A bound and bound algorithm for the zero-one multiple knapsack problem." Discrete Applied Mathematics 3.4 (1981): 275-288.
[8]Consider, for example, a couple of twins, who share their DNA, but each of them is a different and unique being.
[9]For example, check out "Evolutionary neurocontrollers for autonomous mobile robots", Neural Networks

Volume 11, Issues 7–8, October–November 1998, Pages 1461-1478
[10]In the rest of the article, we'll talk about "high fitness" as a generic term, decoupling it from the actual implementation. It will be mean large values for those

problems maximizing a function and small values when the goal of the optimization is to minimize a cost.

[11]It is possible, and it has been attempted, to include the notions of sex-specific chromosomes and sexual subgroups, but, to the best of my knowledge, the possible improvements in the efficiency or effectiveness of the algorithm are in the order of magnitude of optimization. As an example, the dedicated reader can check: ZHANG, Ming-ming, Shu-guang ZHAO, and Xu WANG. "Sexual Reproduction Adaptive Genetic Algorithm Based on Baldwin Effect and Simulation Study [J]." Journal of System Simulation 10 (2010).

[12]Because we are trying to maximize the fitness function for 0-1 knapsack; otherwise, with a function to be minimized, we could have set it to 120%, or 105%, etc.

[13]For the predator-prey robots example, for instance, or for any setup where we evolve systems that runs tasks in the physical world, we might actually do that and score individuals based on how they perform on a real task.

[14]Approximately, with a few simplifications: the actual value depends on coding details like if the same individual can be chosen multiple times. This is fine, though, because we are not really interested in the exact values for these probabilities, but just to understand its order of magnitude and get an idea of how it changes with k.

[15]For some problems, however, using elitism aggressively can compensate pure randomness in selection.

[16]This works when higher fitness means larger values, of course. It can be adapted to the other case by using the inverse of the fitness values, for example.

[17]For each individual, we normalize its fitness by dividing its value by the total sum of the population's fitness. This way, each normalized fitness is between 0 and 1, and their sum across the whole population is 1.

[18]Some flowering plants also adopt sexual reproduction through pollination.

[19]The process is conceptually similar to early phases of **meiosis**, the mechanism used by cells for sexual reproduction.

[20]Mutations also happen spontaneously, with on-null probability, during **mitosis**, the mechanism used for asexual reproduction of all cells (*gametes* and *somatic cells*). These mutations can be key to the evolution of the organisms and species, when they happen in gametes, the cells involved in sexual reproduction.

[21]An *unfair* coin, where the probability of applying a mutation would be far smaller than .

[22]See http://mng.bz/nMMd.

Manning's focus is on computing titles at professional levels. We care about the quality of our books. We work with our authors to coax out of them the best writing they can produce. We consult with technical experts on book proposals and manuscripts, and we may use as many as two dozen reviewers in various stages of preparing a manuscript. The abilities of each author are nurtured to encourage him or her to write a first-rate book.

July 15, 2021

Articles, Development

advanced-algorithms-and-data-structures, algorithms, data, data structures

Previous post

Next post

COMMENTS ARE CLOSED.

SEARCH FORM

Search form

LATEST POSTS

» Svelte REPL, Part 2

» Learn How to Model Language as Tensors

» How Fluentd fits into the Modern Software Landscape

» Node Web Applications and how to Use NPM

» Ask Dr. Chong: become a leader in data science part 1

FEATURED CONTENT

Animated GIFs

Podcasts

Free eBooks

UP ↑