Evolutionary algorithms

Orsolya Sáfár

Farkas Miklós seminar, 2019.03.07.

Motivation

Let us say we have some optimization problem, that is, we have to choose the best from the possible candidates in a reasonable time. For example:

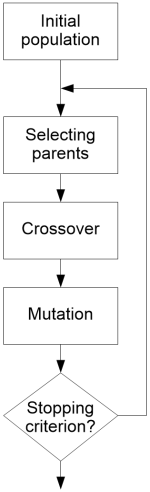
1. Find an optimal path for an walking tour
2. Find an optimal path for a robotic arm
3. Make an optimal timetable (for our university)
4. Find a good strategy in the game 'noughts and crosses'

Biological background

These methods are called evolutionary or genetic algorithms since their basic concepts mimic the evolution of species. They select some 'parent solution' from a pool of possible candidates and by using two operators on them (called crossover and mutation) they make offspring solutions. The offspring are similar to their parents, but they are not necessary identical. The selection of the parents is random, but the better (more 'fit') a solution is, the most likely that it will be chosen as a parent.

Environment	Problem
Individual	Candidate solution
Population	Multiset of candidate solutions
Fitness	Quality of the solution
Genome	Representation of the solution

Scheme


Initial population
Selecting parents
Crossover
Mutation
Stopping criterion?

The backpack problem

Given a set of different items, each one with an associated value and weight, determine which items you should pick in order to maximize the value of the items without surpassing the capacity of your backpack. We illustrate the design of a genetic algorithm on this problem.

We have n items, each of them has a weight $\mathbf{s} = (s_1, s_2, \ldots, s_n) \in \mathbb{R}_+^n$ and a value $\mathbf{v} = (v_1, v_2, \ldots, v_n) \in \mathbb{R}_+^n$. Let us denote the capacity of the backpack by $C \in \mathbb{R}^+$.

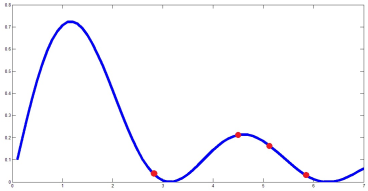
There are 2^n possible solutions, since there are two possible states for each item (either we put it in the backpack or not). A brute force solution would check each of them, hence its running time is exponential. Our goal is to find a better solution.

Solving the backpack problem with a genetic algorithm

- Representation (coding): a 0–1 sequence of length n. A 1 in position i means that we put the ith item in the backpack, 0 means that we don't.
- Fitness: the sum of the values of the items we have chosen, if the sum of the weights is less than (or equal to) the capacity, 0 otherwise. Our goal is to maximize the fitness.
- Individual: a possible choice of items
- Population: a multiset of choices (there can be identical items)
- Genotype: the individual's 0-1 series
- Phenotype: the set of items we have chosen

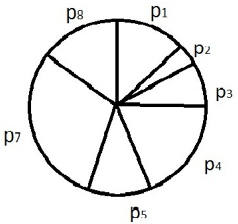
Choosing the parents

From the population we wish to choose the solutions with high fitness as parents, hoping that their offsprings will be good or even better solutions as their parents were.
However, it's important, that the low fitness solutions still have a chance to reproduce. This decreases the chance of getting stuck in a local optimum (which is the greatest danger using any genetic algorithm). That is, we have to maintain the diversity of the population, at least in the beginning.



Roulette wheel method

Let us suppose that for a given population the fitness of the individuals are $f_1, f_2, \dots f_k \geq 0$. We make a probability distribution by dividing each f_i by $\sum_{i=1}^{k} f_i$-s to obtain the probabilities $p_1, p_2, \dots p_k$ (normalizing).



Now we choose k parents independently in k rounds. In each round we choose the ith individual with probability p_i.

Issues with the roulette wheel method

- In the beginning, the offspring of one outstanding solution can dominate the population. If we lose diversity too quickly, we are going to stuck in a local optimum.
- When we are near the global optimum, the solution and hence their fitness are very similar. The result is a slow convergence.
- The relative frequency of the individuals can deviate strongly from the probabilities, especially if the population size is small. But we might not have the time or memory available to work with a large population.
- Seemingly slight changes of the fitness (like adding a constant to every f_i) function can bring radical changes in the efficiency of the algorithm. This can be useful, if we use this property with caution.

Scaling

The first two problems can be remedied by scaling of the fitness function.

We can increase the differences by applying x^n or $\exp(x)$ type scaling. It helps to accelerate convergence in the final phase of the algorithm.

We can maintain the diversity by slowing the convergence using $\ln(x)$ or $\sqrt{x}$ type scaling.

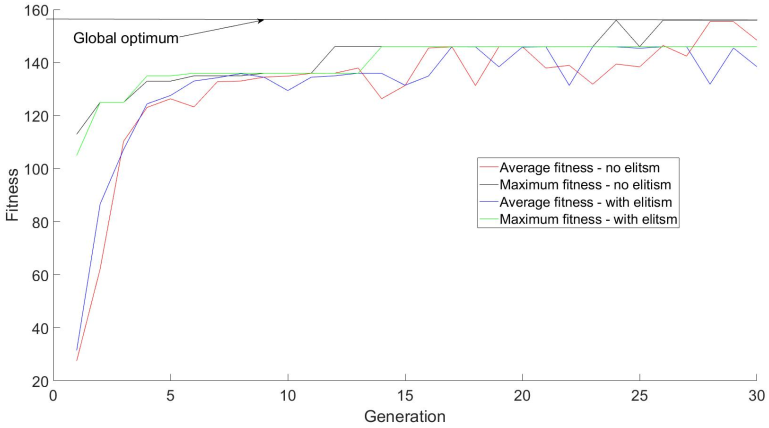
The main problem of choosing the right scaling is that you need to act while the algorithm is running, since what helps in the first phase makes things worse in the end and vice versa.

Elitism

To make sure that the maximal fitness doesn't decrease in the next generation, we can use elitism, that is, we copy the individual with maximum fitness (elite solution) from the parent population to the offspring population.

The elite solution can replace a random offspring or the offspring with minimal fitness. Elitism guarantees that the maximal fitness doesn't decrease in the next generation. However, it has a major drawback: we might get stuck in a local optimum.

Elitism for the backpacking problem



Tournament selection

We can select the parents by the roulette wheel method. Another possibility is tournament selection. In one round, we select a fixed number of individuals from the population (let's say k) completely randomly, and from this k individuals, the one with the maximal fitness will be chosen as parent. We repeat this selection as many times as many parents we want to choose.

If k is relatively big, then the individuals with lower fitness have lower chance to be selected as parent, hence k is an input of a genetic algorithm. We can control the selection pressure (that is, how rapidly we want to discard low fitness solutions) by applying a suitable k. For example, if the convergence is slow, we can increase k.

Genetic operators

First, we use a crossover operator, which generates two offspring from two parents. Our goal is that the offspring inherit features from their parents. A suitable crossover can be:

- onepoint-crossover: we choose a random position, and break both parents in that position. We stick the first half of the first parent to the second half of the second parent, and for the second offspring, stick the first half of the second parent to the second half of the first parent. For example, the offsprings of 11100 and 00011 if we break in the second position: 11011 and 00100.
- multipoint-crossover similar to the one point-crossover, but we choose more positions to break the parents
- uniform crossover in each position with probability $\frac{1}{2}$ we choose a gene from the first or the second parent, and we can have the second offspring by reversing our selection.

Mutation

The goal of the mutation is the exploration of the searching space, in order to prevent our algorithm winding up in a local optimum. On the other hand a too strong perturbation can slow down the convergence, and we might even end up not converging at all. Finding the optimal parameter is crucial for the success of the algorithm.

As a rule of thumb: choose the expected value of the number of mutations between one gene per generation and one individual per generation.

For the backpacking problem, a good mutation operator can be that we flip one gene in a random position with some fixed probability for each individual.

If we use elitism, then a bigger probability of mutation is recommended.

The Traveling Salesman problem

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

This problem is in fact finding the minimum weight Hamiltonian cycle in a complete weighted graph (or digraph).

Let us denote the number of cities by n. A naive algorithm, which checks every possible permutation of the vertices computes $(n-1)!$ sums, each with n terms. Using dynamical programming (Karp-Held algorithm) we can solve the problem in $\mathcal{O}(n^2 2^n)$ time.

Permutation representation

Since we don't have a polynomial algorithm for the Traveling Salesman problem, we will write a genetic algorithm. The natural representation for a graph with n edges is the a permutation. The representation is also appropriate for scheduling problems.

The main issue is to design the genetic operators. How could we mutate a permutation? If we change one position the result is no longer a permutation? Also the result of a one-point crossover is also probably not a permutation.

Problems coded with permutation representation

Two different types of problems have solutions whose natural representation are permutations.

First the scheduling problems, where we have to plan the order we manufacture the products. If the product A has to be manufactured before product B and C, all all three has to be manufactured before product D, then the fitness of (1234) and (1324) will be quite similar, while the fitness of (4123) will be low.

The other type is the adjacency problems, for example the traveling salesman problem. In this problem the fitness of (1234) will be identical to the fitness of (4123). The absolute position of an city is not important, rather its neighbors.

Mutation operators

Genetic algorithms with 0-1 sequence representations used bitwise mutation. In case of permutation representation, it isn't possible to change only one gene, we have to modify at least two. Here the probability of mutation will mean the probability of one individual mutates (instead of one gene).

The first three mutation operators are recommended for scheduling problems. They cause a relatively small change in the absolute position.

Mutation operators for scheduling

Swap mutation (SWAP): Exchange two items.
For example: (5 3 1 9 2 5 7 4 8) → (5 3 4 9 2 5 7 1 8).

Insert mutation (INS) We select two random positions, and move the item in the second position immediately after the first one.
For example: (5 3 1 9 2 5 7 4 8) → (5 3 1 4 9 2 5 7 8).

Scramble mutation (SCR) We select two random positions, and scramble everything between the two positions with a random permutation.
For example: (5 3 1 9 2 5 7 4 8) has length 4, that is, we have to scramble with a 4-long mutation i.e. with (3241). The result in this case is (5 3 2 9 5 1 7 4 8).

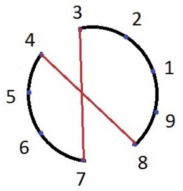
Mutation operator for adjacency problems

The following operator is recommended for adjacency problems, since it keeps most of the connections intact.

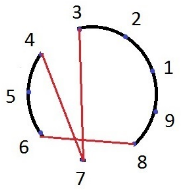
Inversion mutation (INV) We select two random positions, and then invert the order of the item between them.
For example: (6 3 1 9 2 5 7 4 8) $\rightarrow$ (6 3 5 2 9 1 7 4 8)

Comparing the mutation operators

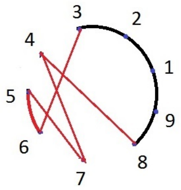
(123456789) INV (123765489)



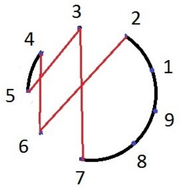
(123456789) INS (123745689)



(123456789) SCR (123657489)



(123456789) SWAP (126453789)



Crossover operators

The goal of the crossover operators is to preserve the parents' attributes. The exact type of attributes we want to preserve depends on the nature of our problem.

Two operators for each type will be presented.

All four operator create one offspring from two parents. In some cases even if we reverse the order of the parents, the same offspring will be created, in these cases we have to modify the parent selection mechanism.

Order crossover (OX)

This operator is recommended for scheduling, since it preserves the relative order of the items from the second parent (from now on the first parent is p1, the second is p2).

- We select two random positions, the items between them (the two positions included) is the matching segment.
- We copy to the offspring the matching segment from p1
- We copy the yet unused alleles from p2 to the offspring starting from the position immediately after endpoint of the matching segment.

Example: OX

Let p1=(123456789) and p2=(937826514) and the matching segment is positions 4–7.

Now take the matching segment from p1: (123456789)
Delete these alleles from p2: (93~~7~~82~~65~~1~~4~~)

Copy the matching segment from p1 to the offspring (␣␣␣4567␣␣) now copy the remaining alleles from p2 starting from the 8. position: (382456719)

Cycle crossover (CX)

This operator preserves the absolute position of the items from both parents as much as possible. We divide the parents into cycles. The goal is to find a set of positions where the same alleles are present in both parents (of course not necessarily in the same positions). We construct the cycles as follows:

- We start with the first unused allele of p1, that is, the first position in the first cycle
- Look at the allele in the same position in p2
- Go to the position with the same allele in p1, and add this allele to the cycle
- Repeat the previous two steps until we arrive at the first allele in p1

Example: CX

Let p1=(123456789) and p2=(937826514). The first cycle is:

(123456789) (123456789) (123456789) (123456789)
(937826514) (937826514) (937826514) (937826514)

The second one:

(123456789) (123456789) (123456789) (123456789)
(937826514) (937826514) (937826514) (937826514)

The third cycle contains only the 6 allele in position 5. Then the offspring is created from the $1., 3., 5., \dots$ cycle of p1 and the $2., 4., 6., \dots$ cycle of p2.
(137426589).

Partially mapped crossover (PMX)

This operator is recommended form adjacency type problems, since it tries to preserve most of the connections of the parents. It works as follows:

- We select two random positions, the items between them (the two positions included) is the matching segment.
- We copy to the offspring the matching segment from p1
- We search for a suitable place form the alleles in p2's matching segment in the offspring*
- Copy the items from p2 to the empty positions

Partially mapped crossover (PMX)

We search for a suitable place form the alleles in p2's matching segment in the offspring as follows:

- If the given allele is already in the matching segment of p1 then we don't have to do anything
- If it isn't in the matching segment, then let us denote the allele by p2(j) (that is, it's in position j)
- Look for the position of the p1(j) allele in p2, let us denote this position by k. If this position isn't in the matching segment then copy p2(j) here. If it is in the matching segment then we try again with p1(k) instead of p1(j) until we find a position outside of the matching segment, where we copy p2(j).

Example: PMX

Let p1=(123456789) and p2=(937826514) and the matching segment is positions 4-7.

Here p1=(123456789) and p2=(937826514). The offspring is: (␣␣␣4567␣␣)

Form the matching segment of p2: 8 isn't it p1's matching segment, so we search a suitable position. Since p1(4) is 4 and 4 is in the 9. position in p2. This is outside of the matching segment, so we found the right place for 8, which is the 9. position: (␣␣␣4567␣$\underline{8}$)

Example: PMX continued

The next item in p2's matching segment is 2. Since it isn't in p1's matching segment, we also have to find it a suitable position. Since p1(5)=5, and 5 is in the 7. position in p2 which is inside of the matching segment it isn't a suitable position for 2. Instead we search for p1(7) that is 7 in p2, which is in the 3. position and it's outside of the matching segment so we copy 2 here: $(\sqcup\sqcup\underline{2}4567\sqcup\underline{8})$.

The two remaining item from p2's matching segment is 6 and 5. They are both in p1's matching segment, so we don't have to do anything.

We copy to the remaining positions the alleles of p2 : $(93\underline{2}45671\underline{8})$.

As these example show, there are 6 edges of the offspring that is present in one of the parents. However the parents have a common edge {7-8} which isn't present in the offspring.

Edge crossover (EX)

This operator is recommended for adjacency problems. It tries to preserve the common edges, furthermore tries to copy as many connections as possible. For this operator the order of the parents is irrelevant.

We create an edge list for the parent pair, a list of neighboring vertices for each vertex, indicating the common edges.

Edge crossover (EX)

We build the offspring as follows:

- We select a random vertex as the current vertex, and copy the allele in the offspring
- We delete the current vertex from every other vertex's edge list
- If there is a common edge in the current vertex's edge list, then that vertex will be in the next current vertex (if there are 2 such vertices, we select randomly)
- If there is no common edge, the vertex with the shortest edge list will be the next current vertex. (if there is more than one such vertex, we select randomly)
- If we arrive to an empty edge list, we select randomly from the remaining vertices.

Example: EX

Let p1=(123456789) and p2=(937826514). First we construct the edge list:

1	2,4,5,9	1	2,4,$\underline{5}$,9				
2	1,3,6,8	2	3,6,8	2	3,6,8	2	3,8
3	2,4,7,9	3	2,4,7,9	3	2,4,7,9	3	2,4,7,9
4	1,3,5,9	4	3,5,9	4	3,9	4	3,9
5	1,4,6	5	4,6	5	4,$\underline{6}$		
6	2,5,7	6	2,5,7	6	2,7	6	$\underline{2}$,7
7	3,6,8	7	3,6,8	7	3,6,8	7	3,8
8	2,7,9	8	2,7,9	8	2,7,9	8	2,7,9
9	1,3,4,8	9	3,4,8	9	3,4,8	9	3,4,8
rc	(1)	sl	(15)	ce	(156)	rc	(1562)

Example continued

2	3,$\underline{8}$						
3	4,7,9	3	4,7,9	3	4,9	3	4,$\underline{9}$
4	3,9	4	3,9	4	3,9	4	9
7	3,8	7	3	7	$\underline{3}$		
8	7,9	8	$\underline{7}$,9				
9	3,4,8	9	3,4	9	3,4	9	4
sl	(15628)	ce	(156287)	sl	(1562873)	rc	(15628739)

The offspring is the permutation (156287394)

Continuous representation

For some optimization problem the natural selection for representation are floating point numbers. In this case the genotype is a $(x_1, \ldots, x_k) \in \mathbb{R}^k$ vector.

For these problems the gradient methods works fine, if the fitness function is differentiable and convex.

We can use simulated annealing for rough surfaces. We start with one point, generate several new ones by adding the logarithm of a uniform random number to each coordinate. If there is a better function value amongst the new points we select the one with the best value for the next round, otherwise we select one randomly according to a distribution, where better points (that is with higher function values) have better chances to be selected.

Rechenberg's algorithm

Another option for function without a smooth gradient is to design an evolutionary algorithm. Strictly speaking, this algorithm is not an evolutionary algorithm, but has many similarities to them. It differs from simulated annealing because

- It generates only one new point by adding to each coordinate of the parent a normally distributed random number (the expected value is always 0, the standard deviation is changing)
- If the new point is better, then it discards the old point, if not, then it discards the new one (elitism)
- The changes in standard deviation (step-size) is adapted to the function

Rechenberg's 1/5 rule

The bigger the standard deviation (σ), the bigger the expected step-size of the Rechenberg's algorithm is.

In the beginning a greater value of σ is recommended in order to explore the solution space. Later a smaller one is desirable to be able to have a precise estimate for the optima.

If there are many successful steps, then we are heading in the right direction, in this case we should increase the value of σ. In this way we get a quicker convergence.

If in most of the steps we discard the new point, then we are close to the optima (since in every direction the function increases), so we have to decrease σ.

Rechenberg's 1/5 rule

Let $0.817 < c < 1$ be fixed, and after 20 iterations we the ratio of successful steps is denoted by p. After each k iterations, we change the value of σ:

$$\sigma := \begin{cases} \frac{\sigma}{c} & p > \frac{1}{5} \\ \sigma \cdot c & p < \frac{1}{5} \\ \sigma & p = \frac{1}{5} \end{cases}$$

Continuous representation

An evolutionary algorithm for a continuations problem has many similarities with Rechenberg's algorithm. We use the generation mechanism of the Rechenberg algorithm as mutation.

The representation of the solution will be an $\mathbb{R}^n$ vector (if the function has n variables) supplemented with a parameter vector storing the parameters of the algorithm.

From now on an attribute of an individual is not only the function value itself, but also the expected step size (that is it tells us something about the fitness of its offsprings). The algorithm not only optimizes the function value, but also the step size. In this way our algorithm is self-adaptive.

Crossover operators

There are several options for contentious problems:

- discrete crossover, where the allele of the offspring is simply copied from one of the parents. It is recommended on the part of the vector where the coordinates are stored in order to maintain the diversity.
- arithmetic crossover: let $0 < \alpha < 1$ one of the offspring is $\alpha x + (1-\alpha)y$, the other is $(1-\alpha)x + \alpha y$. It is recommended on the part where the parameters are stored.
- Blend: instead of the previous α we choose a random number on $[-.5, 1.5]$ to avoid early convergence.

The crossover is global, it is possible to have different parents in each coordinates.

Selecting survivors

For continuous problems, we typically generate many more offspring in one generation (let's say λ), and then select the best μ (according to the function value). The selection pressure is the ratio of μ and λ.

It is recommended to select the new generation's individuals only from the offspring (the parents not includes), because it reduces the chance that a single individual dominates the population with a good function value but an inappropriate step-size.