Evolutionary algorithms

Orsolya Sáfár

2019 spring

Course requirements

- At least 70% attendance of the contact lessons is required
- Mark are based on homework assignments. A total of 120 points can be obtained from the problem sets issued each week. The homework assignments cover three main topics (simple algorithms, permutation representations, real representations). Students have to reach a total of 12 points in each topic to pass.

Marks based on the total score are as follows:

0-39:	1
40-49:	2
50-59:	3
60-69:	4
70+ :	5

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'

Running time

Let us denote the length of our input by $n \in \mathbb{N}$. Our algorithm's running time is $\mathbf{O}(f(n))$ if we can give an upper estimate for the number of necessary basic operations by $cf(n)$, where $c \in \mathbb{R}$ is a constant.

An algorithm has polynomial running time if its running time is $\mathbf{O}(n^k)$ for some $k \in \mathbb{N}$. An algorithm has exponential running time if its running time is $\mathbf{O}(a^n)$ for some $a > 1$.

Examples

We wish to sort an array of n entries.

Selection sort: entries to the left of the pointer are fixed and in ascending order, and no entry to the right of the pointer is smaller than any entry to the left of the pointer.
We move the pointer to the right, in each step we identify the minimum entry on right, and exchange it into position. Next round we have one less entry to deal with.
Since in round $n-i$ we use $i-1$ compares, and at most 1 exchange, the running time is

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}.$$

That is, the running time of the selection sort is $\mathbf{O}(n^2)$.

Examples cont'd

Merge sort: We divide the array into two halves (not necessarily same sized), recursively sort each half, then merge the two halves. $T(n)$ denotes the number of necessary steps for merge sorting an array of n entries.
In case we divided our array of n entries into two subarrays having n_1 and n_2 entries,

- the sorting of the two subarrays takes $T(n_1) + T(n_2)$ steps,
- the merging the two subarrays takes at most $n_1 + n_2$ compares.

Hence $T(1) = 0$ and $T(n) \leq n + 2T\left(\frac{n}{2}\right)$, we have

$$T(n) \leq n + 2 \cdot \frac{n}{2} + \cdots + 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) \leq n[\log_2 n]$$

Hence the running time of the merge sort is $\mathbf{O}(n \log_2 n)$.

Estimating the running time

In the previous two examples we could calculate the exact running time. But in some cases it's not possible, we can only estimate the number of steps from actual measured running time in seconds.

Let us suppose, that the running time is $\mathbf{O}(n^k)$ for some unknown $k \in \mathbb{N}$. This problem is overdetermined, since it has two unknowns (the exponent k and the constant factor c) but many equations (the number of measurements).

Let us denote the number of necessary steps by $T(n)$, where n is the length of the input. If $T(n) = cn^k$ then by the identities of the logarithm

$$\log(T(n)) = \log c + k \log(n).$$

That is, if we plot the logarithm of the running time on the y axis and the logarithm of n on the x axis, then we get a straight line. In this plot, called the log-log scale plot, the steepness of the line is k, which is the unknown exponent.

A brute force algorithms

A brute force algorithm solves an optimization problem by checking every candidate. Of course this isn't possible if you have infinitely many possible solutions (for example if the problem is continuous).

These algorithms are easy to write, but in many cases the running time isn't acceptable if n is large.

However, if n is small, the more sophisticated solutions might not be worth the effort.

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 $\mathbf{s} = (s_1, s_2, \dots, s_n) \in \mathbb{R}^n_+$ and a value $\mathbf{v} = (v_1, v_2, \dots, v_n) \in \mathbb{R}^n_+$. Let us denote the capacity of the backpack by $C \in \mathrm{R}^+$.

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.

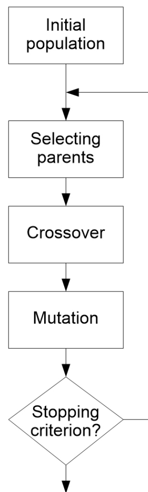
There exists a clever polynomial algorithm (using dynamic programming) if the weights are integers, but in our problem they can be arbitrary positive real numbers.

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 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

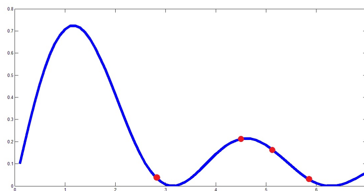


Solving the backpack problem with a genetic algorithm

- 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

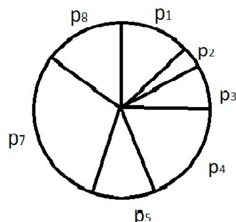
Choosing the parents

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 $f_1, f_2, \ldots f_k \geq 0$. We make a probability distribution by dividing each f_i by $\sum_{i=1}^{k} f_i$-s to obtain the probabilities $p_1, p_2, \ldots p_k$ (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.

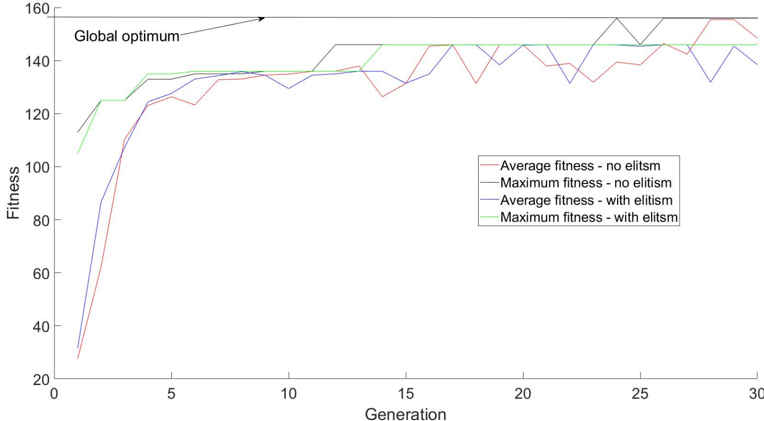
The main problem of choosing the right scaling is that you need to act while the algorithm is running, since what helps in the first phase makes things worse in the end and vice versa.

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.

At this point, we have chosen the parents and we are ready to generate the offspring.

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. To ensure this, we have to find a good representation and crossover operators. 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.

Stopping criteria

We iterate through generations, to find better and better solutions. The most primitive stopping criterion is that we run a fixed number of generation, and the output of our algorithm is the best individual up to that point.

This way we might iterate too many or too few times, depending on the rate of convergence. However, it has a great advantage: we know the running time exactly (which could otherwise be an issue for large inputs).

In the next lecture, we investigate the behavior of a genetic algorithm, and then we can propose better stopping criteria.