Evolutionary algorithms

Sáfár Orsolya

Constrain handling

Types of optimization problems

Let us suppose that for the representation of the solutions has the variables $x_1, x_2, \ldots, x_n$ whose values are in the domains $D_1, D_2, \ldots, D_n$. We call the set $D_1 \times D_2 \times \cdots \times D_n$ free searching space. If we are looking for the solution on a subset of this set then we have constrains to satisfy.

The types of the problems are

- We have no constrains, our goal is to maximize an object function. FOP: free optimization problem
- We have constrains, we are searching one feasible solution. CSP: constrain satisfaction problem
- We have constrains and an objective function. We are looking for the optimal solution, which satisfies the constrains. COP: constrained optimization problem

Examples for COP

Backpacking

The number of variables are the number of items. The values of the variables can be either 0 or zero. The free searching space is $\{0,1\}^n$. However we have a constrain: the sum of the weights can't surpass the capacity. We are looking for a packing with maximum value, hence it's a COP.

Traveling agent

Let us denote the number of cities by n. We we represent our solutions as an list of integers, whose values are between 1 and n, that we also have a constrain: no repetitions are allowed. This makes our problem a COP (for this particular representation).

Examples for CSP

Graph coloring with k colors

A graph is given (for example with its adjacency matrix), and we want to color it with k colors. It's a CSP, since the constrain is, that two neighboring vertex can't have the same color. But we don't optimize: any of the right coloring will do.

8 queens An 8×8 chessboard is given, we want to put 8 queens on it so that none of them hits any other queen. Our task is to find a good position for each. Since we don't have an objective function it's also a CSP.

Using genetic operators for constrained problems

The direct approach starts with feasible solutions in the starting population, and makes sure every offspring is feasible.

One way to do that is: the genetic operators create feasible offspring from feasible parents. In this case the constrains are satisfied for every candidate solution.

Example For the Traveling Salesman problem a permutation is a feasible solution. If our operators create permutations (like INS mutation, or CX crossover), than the constrain 'visit every city once' is satisfied.

Decoding

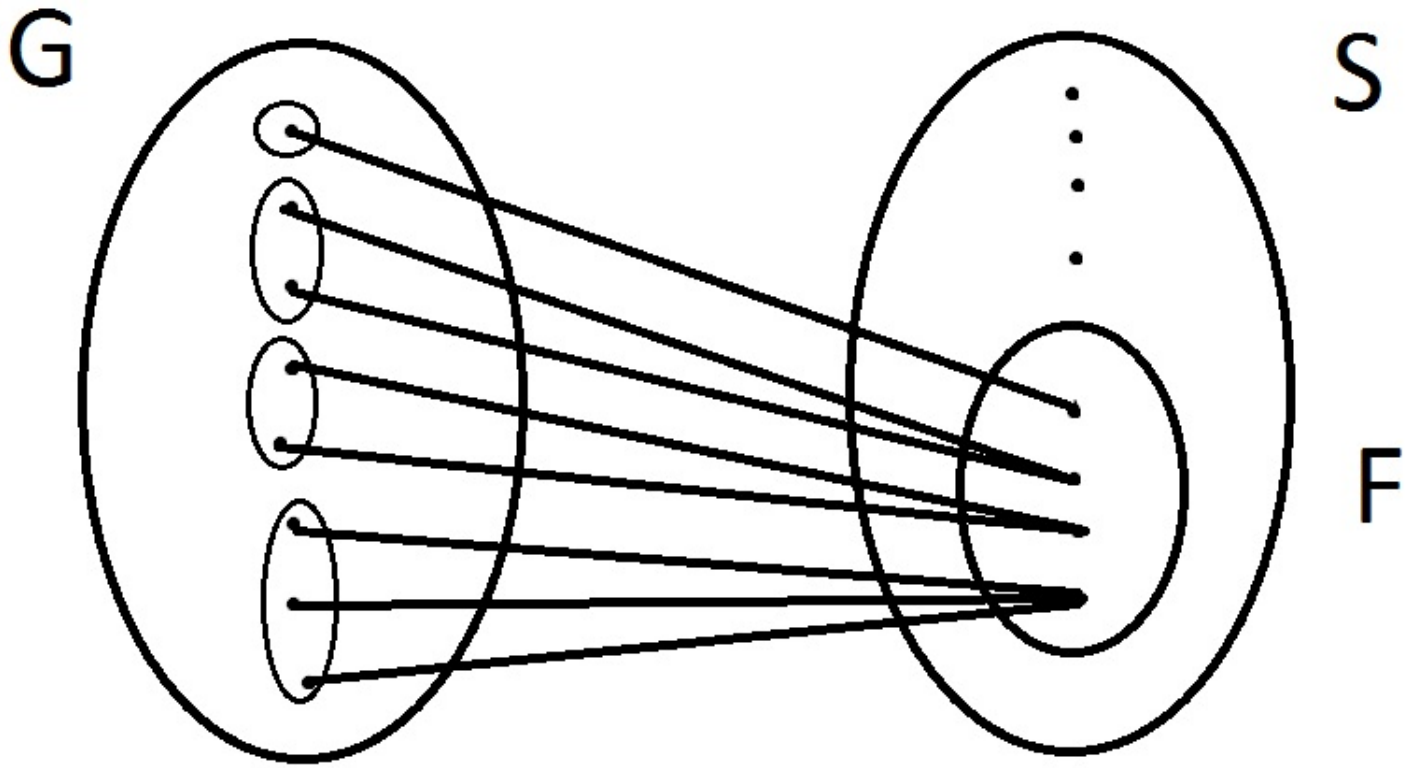
An other approach is modifying the solutions violating the constrains to feasible solutions with a decoding transformation.

Let us denote the searching space by S, the set of the feasible solutions by F, where $F \subset S$, and the set of genotypes by G.

A $G \to F$ transformation is called decoding if

- For all $g \in G$ has a unique image in F
- For all $s \in F$ has at least one inverse image in G

Decoding



Example

Let us consider the backpacking problem. Now S is the set of every possible packing, while F is the set of the packings where the sum of weights doesn't surpass the capacity. G is the set of every $0-1$ sequence.

Let us define the decoding the following way: by going from left to right a 0 means that we don't pick that item, 1 means that we pick that item, provided we aren't over the capacity yet.

This way every $0-1$ sequence has a feasible image. Also every feasible solution has at least one inverse image, since the decoding doesn't change the feasible solutions.

Example continued

In a typical case a solution in F has multiple inverse image, and the number of inverse images aren't the same for different solutions.

Example Let us have 8 items, for the first 4 the weights are 10 and they worth 100, for the second 4 item the weights and the values is 1. Suppose, that the capacity is 12.

Let us denote the following packing by f_1: we pick the second, fifth and sixth items (it's a feasible solution). Let use denote by f_2 the packing where we've chosen the third, fifth and sixth item.

The number of inverse images for f_1 is 16, because every solution fitting the 01##11## maps to it, while the number of inverse images for f_2 8, since they are the packings fitting the 001#11## scheme.

Example: graph coloring

Let us suppose we have a graph with n vertices. We are looking for a coloring with 3 colors.

A possible representation for a coloring is a sequence of length n, which items are $1, 2, 3$. The ith element of the sequence is the color of the ith vertex. This is a CSP, the constrain is that vertices connected with an edge can't have the same color.

Graph coloring continued

In order to handle the constrains we will represent the possible solutions with a permutation instead of a sequence. The permutation is the order of the vertices. We decode the coloring by going trough the vertices in the order of the permutation, trying to color the next vertex 1. If it already has a neighbor colored 1, the we try coloring it 2. In case it already has a neighbor colored 2, we try 3. If we also have a 3 colored neighbor we leave that vertex colorless.

Every feasible solution has at least one permutation coding it: if we order the vertices according to their color.

The genetic algorithm tries to minimazie the number of colorless vertices, so we transformed the problem to a COP.

Example: Traveling Salesman

Let us denote the number of cities by n. The representation of the possible solution are n-long sequences, for which the ith elements of the sequence is $\leq (n + 1 - i)$. There isn't any more condition, the sequences may have a certain number multiple times. In this case we can use the simple one-point and multi-point crossover operators.

Decoding Let us fix the order of the cities say A, B, C, D, E. A sequence will be transformed to a permutation of the cities the following way: going from left to right on the list the ith element means, that we visit the ith city on the list next. After we inserted the city to our path, we delete the city from the list.

Example: The list $\{4, 4, 1, 2, 1\}$ codes the order DEACB.

Soft constrains - fitness function

The other possible way of handling the constrains is to transform them to 'soft constrains'. The solutions which violate the constrains are punished by modifying their fitness's (decreasing, if we maximized in the original problem), and after that we try to find an optimum for this new fitness function.

In this case the optimization of the algorithm (hopefully) solves the problem of satisfying the constrains, this is called the indirect approach.

We can use the following penalties:

- static (constant in each generation)
- dynamic (the values depend on the number of generation)
- adaptive (the values depend on the quality of the solutions)

Statical penalty function

We try to create a function, which is easy to compute. Let us denote the number of constrains by m. A possible choice is:

$$P(x) = \sum_{i=1}^{m} w_i \cdot d_i(x),$$

where $d_i(x)$ is a metric, expressing the distance of the solution from F. A very simple choice for d_i, that it's values are 1 it it's a feasible solution, and 0 if it isn't. The w_i are the weights, finding the optimal ones is often a hard problem.

Dynamical penalty function

If the value penalty changes in each generation, then we have a dynamical penalty function. Their general form is:

$$P(x) = \sum_{i=1}^{m} w_i(t) \cdot d_i(x).$$

Since in the beginning we wish to explore the searching space, but after that we want to concentrate satisfying the constraints, we have to increase the penalties. A reasonable choice is:
$w_i(t) = (w_i \cdot t)^{\alpha}$, where $\alpha = 1$, or $\alpha = 2$.

Adaptive penalty function (SAW)

The penalty changes during running, and has the form

$$P(x) = \sum_{i=1}^{m} w_i \cdot d_i(x)$$

We are going to increase the w_i weights. Let us fix a T_p and δw quantities, which are the parameters of our algorithm.

After T_p fitness evaluation we change the $P(x)$ function. The w_i weight will be increased by δw but only for those constrains, which are violated by the elite solution (our the current best solution, if we don't use elitism).

The reason behind this, is that we want to press our algorithm to satisfy the hardest constrains (the ones even our best solution couldn't satisfy).

n queens problem

We have a $n \times n$ chessboard, and we put n queens on it. We wish to find a layout, where none of the queens hit any other queen.

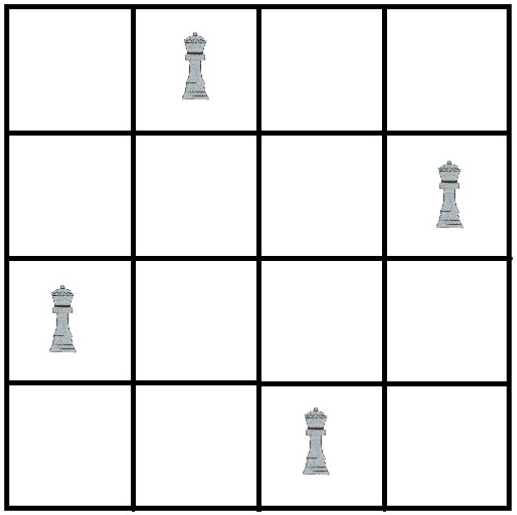


Figure: A feasible solution for $n = 4$.

Solving the n queens problem

A first draft of a solution: check every layout. Since a queen hits her own row and column, it's enough to check permutation of the rows, that is $n!$ possible solutions.

For each permutation we have to check $\binom{n}{2}$ pairs whether they are on the same diagonal. That is

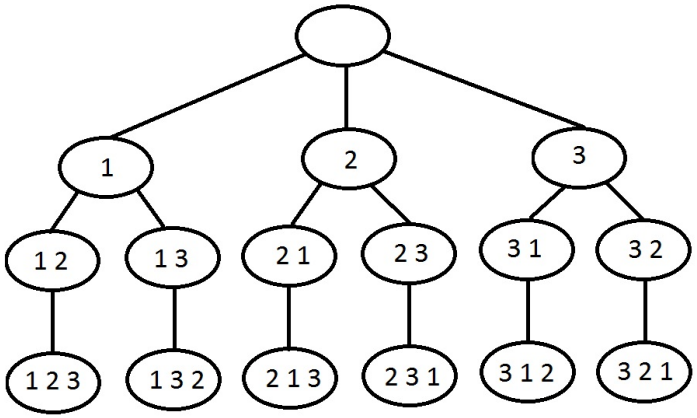
$$n! \frac{n(n-1)}{2} \sim \left(\frac{n}{\mathrm{e}}\right)^n \sqrt{2\pi n} \frac{n(n-1)}{2}$$

For example in the worst case for $n = 10$ it's about $1.7 \cdot 10^8$ checks, and for $n = 20$ it's $4.6 \cdot 10^{20}$.

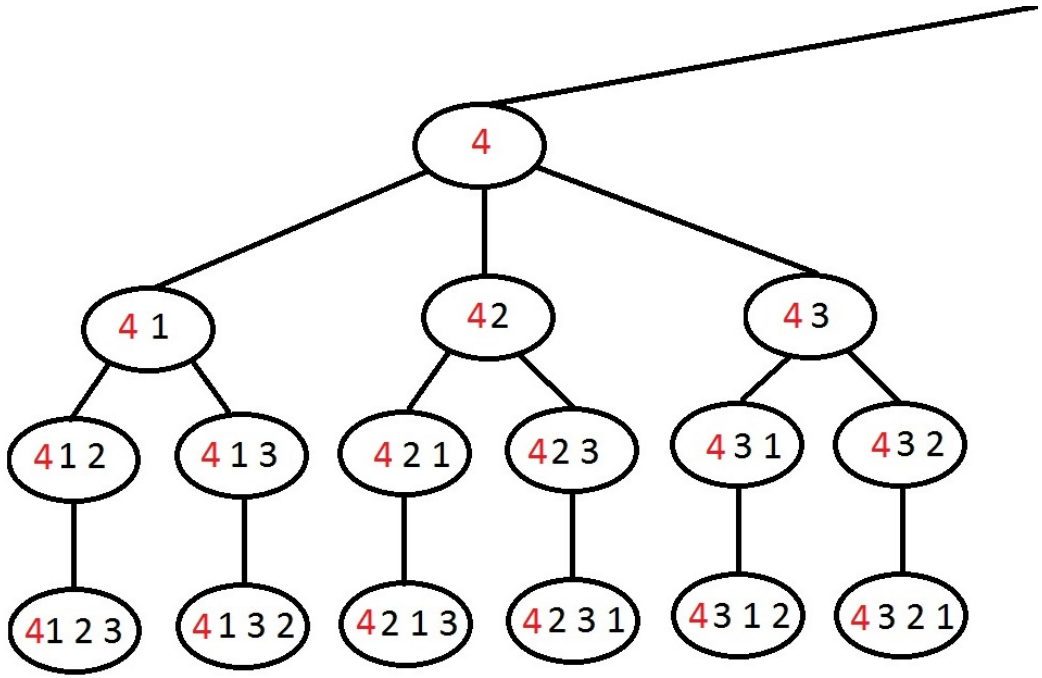
Solving the n queens (continued)

A better solution based on the following idea. If two queens hit each other in a partial layout, that none of the layouts are feasible, which has this part, hence we don't have to check them.

We build a tree (it's a graph) from the potential solutions (permutations). On it's leaves are the possible solutions (that is the layouts), on the inner vertices are the common part of the layouts.

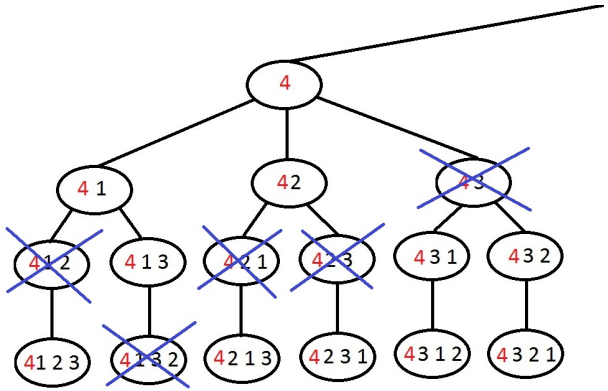


Parts of the tree



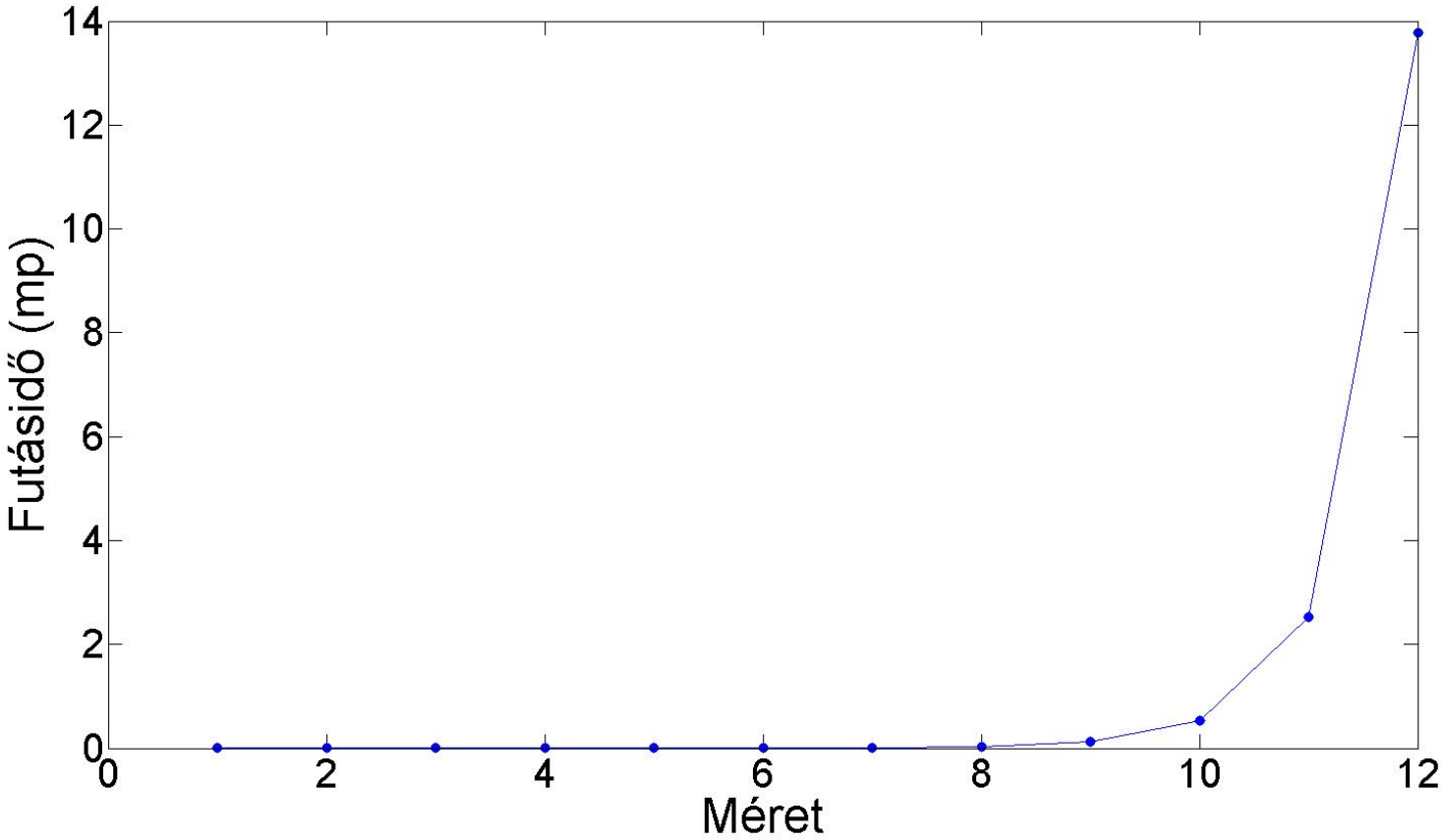
Solution (continued)

If to queens hit each other in a partial layout we discard the whole branch containing it.



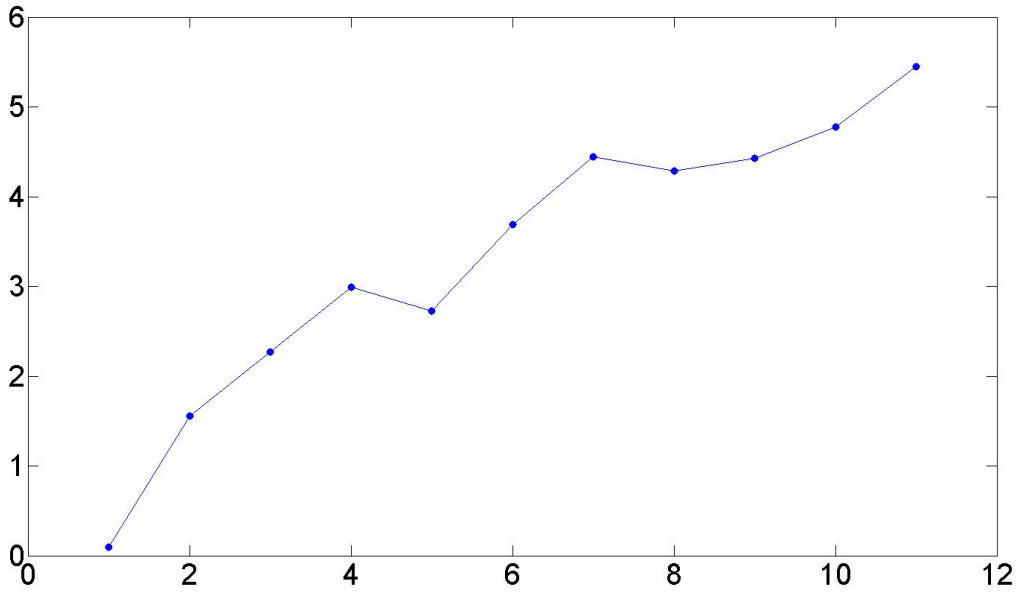
Running times

Figure: Running times



Running times

Figure: The rate of increasing



We see that the running time is still worse that a^n, that is for big chessboards we have to find a quicker solution. A genetic algorithm maybe?

Stopping criteria

It's advised to consider the following conditions as stopping criterion:

- We stop, if the fitness of the elite solution reaches a threshold (in this case we also have to have a cap in the number of generations). This is feasible for CSP problems, where 0 fitness means we satisfied all constrains.
- We stop, if the fitness of the elite solution doesn't (significantly) change for several generations. In this case we don't waste resource exploring the neighborhood of a (possibly) local optimum. This is a reasonable stopping criterion if we don't know the optimal fitness, that is for FOP and COP problems.