# Evolutionary algorithms

Sáfár Orsolya

## Permutation representation

# Decision problems

A decision problem is a yes-or-no question on an infinite set of inputs. It is traditional to define the decision problem as the set of possible inputs together with the set of inputs for which the answer is yes. These inputs can be natural numbers, but can also be values of some other kind, like binary strings or strings over some other alphabet.

The Boolean satisfiability problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. If this is the case, the formula is called satisfiable, and is in the SAT language. For example the $x_1 \wedge \neg x_1$ isn't in the SAT language, but $(x_1 \wedge x_2) \vee \neg x_1$ is.

# Complexity classes.

## Definition

*A problem is in P if we can determine that our input is in that language in polynomial time.*

Loosening this condition, we can define the NP class. Instead of the formal definition we give the characterizing theorem:

## Theorem

*A decision problem is in NP if the claim that an input is in the language can be verified in polynomial time.*

# Hamiltonian cycle

Let us take all graphs. A graph is in the $H$ language if it contains a Hamiltonian cycle.

If someone gives a permutation of vertices stating that this is a Hamiltonian cycle we can check if its true in polynomial time (for example by checking $n$ elements in the adjacency matrix).

## Definition

*A language is NP-hard, if any problem in NP can be reduced to it.*
*A problem is NP-complete if it's NP-hard, an is in NP.*

A famous conjecture is that $P \neq NP$.

# Famous NP-complete problems

- The language of graphs which can be colored with 3 colors
- To decide if a graph has a complete subgraph of $k$ vertices (the input is the graph and $k$)
- To decide if there is a Hamiltonian cycle in a graph
- To decide whether there is a Hamiltonian cycle with total weight at most $k$ in an edge-weighted graph
- The backpacking problem, where $s_i = v_i$ and the question is whether there is a packing which value is exactly $k$.
- We have the tasks $d_1, d_2, \ldots, d_k$ and a number $T$. Each task takes 1 day to complete. We have a partial ordering: $d_1 \prec d_2$ means that $d_1$ have to be completed before $d_2$. For each task we have a due date $h(d_i)$. We have to decide whether there is a scheduling for the tasks such that at most $T$ tasks are not ready by the corresponding due date.

# 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.

# Karp-Held algorithm

Using dynamical programming we can solve the problem in $\mathcal{O}(n^2 2^n)$ time. Let us denote the edges of the graph by $1, 2, \ldots, n$ and the weight of the vertex connecting $i$ and $j$ by $a_{ij}$.

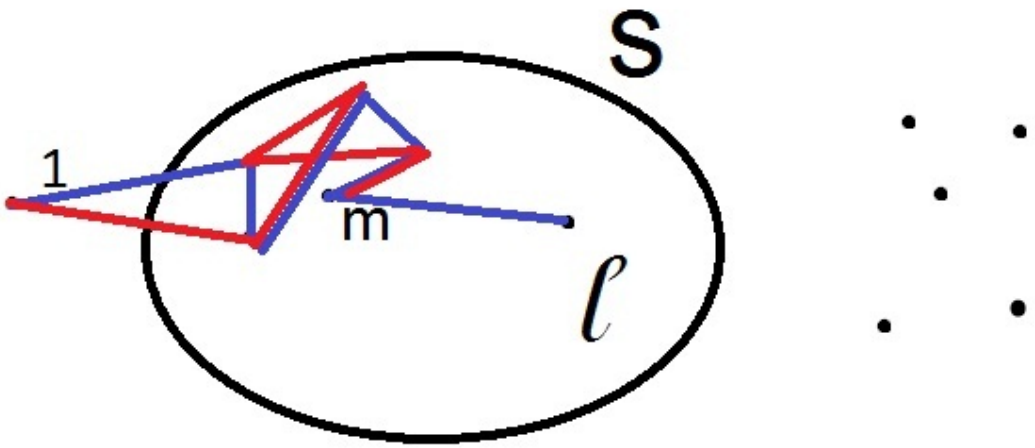
## Definition

*Let $S \subset \{2, 3, \ldots, n\}$, $\ell \in S$ and let us denote by $C(S, \ell)$ the shortest path starting from 1 that goes through every vertex of the graph and ends in $\ell$.*

We are going to compute $C(S, \ell)$ recursively, which not only determines the length of the shortest cycle but also the permutation of the vertices.

# The recursion

- $C(\{\ell\}, \ell) = a_{1\ell}$, $\forall \ell \in S$
- If $|S| > 1$ $C(S, \ell) = \min_{m \in S \setminus \{\ell\}} (C(S \setminus \{\ell\}, m) + a_{m\ell})$
- The length of the shortest cycle
  $\min_{\ell \in \{2,3,\ldots n\}} (C(\{2, 3, \ldots, n\}, \ell) + a_{\ell 1})$



# Running time

To prove the running time we distinguish the cases according to the size of $S$. Let $|S| = k$, then there are $\binom{n-1}{k}$ possible choices for $S$; we can choose $\ell$ in $k$ different ways, and $m$ in $k-1$ different ways. That is, the amount of necessary computations is:

$$\left(\sum_{k=1}^{n-1} k(k-1)\binom{n-1}{k}\right) + n - 1 =$$

$$\left(\sum_{k=2}^{n-3} \binom{n-3}{k-2}(n-1)(n-2)\right) + n + (n-1) =$$

$$(n-1)(n-2)2^{n-3} + (2n-1) = \mathcal{O}(n^2 2^n)$$

# Comparing the different running times

$\mathcal{O}(n^2 2^n)$ means that Karp-Held is still not a polynomial algorithm. Is it any better at all then the naive $\mathcal{O}(n!)$ algorithm?

Theorem (Stirling's formula)

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

This shows that the rate of growth for the brute force algorithm is much faster.

# 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.

# Denoting permutation

Given an order of the numbers $1, 2, \ldots n$ we put the sequence between parenthesis and denote it by $\pi$, for example $\pi = (3214)$

There is two possible coding for permutations. the first one is, where $\pi(i)$ is in the $i$th position. For example is the items are A,B,C,D, then (3124) means the order CABD.

For the second coding the $i$th item of the list means, that position of the $i$th, that is (3124) means the order BCAD. From now on we use the first coding.

# 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) $\rightarrow$ (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) $\rightarrow$ (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).

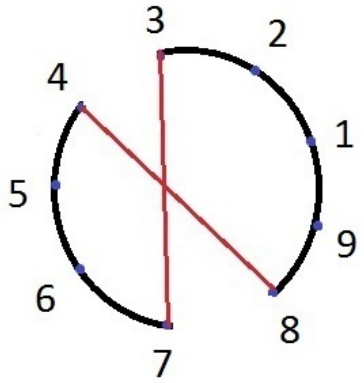
# Mutation operator for adjacency problems

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

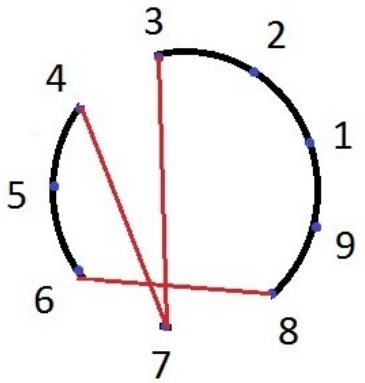
**Inversion mutation (INV)** We select two random positions, and then invert the order of the item between them.
For example: (6 3 1 9 2 5 7 4 8) $\rightarrow$ (6 3 5 2 9 1 7 4 8)

# Comparing the mutation operators

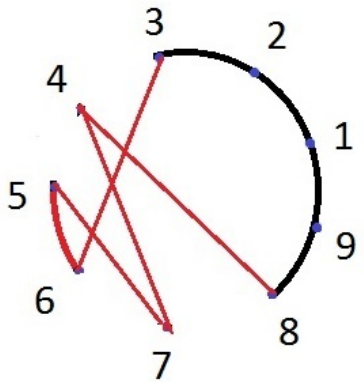
(123456789) INV (123765489)



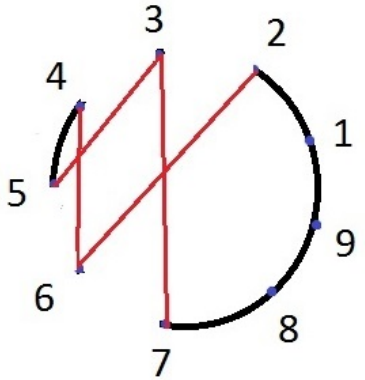

(123456789) INS (123745689)




(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: (9378265~~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 $1., 3., 5., \ldots$ cycle of p1 and the $2., 4., 6., \ldots$ 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␣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: (␣␣2456␣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 : (932456718).

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 | 2,4,5,9 | | | | |
| 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 | 4,6 | | |
| 6 | 2,5,7 | 6 | 2,5,7 | 6 | 2,7 | 6 | 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 | 3,8 | | | | | | |
| 3 | 4,7,9 | 3 | 4,7,9 | 3 | 4,9 | 3 | 4,9 |
| 4 | 3,9 | 4 | 3,9 | 4 | 3,9 | 4 | 9 |
| | | | | | | | |
| | | | | | | | |
| 7 | 3,8 | 7 | 3 | 7 | 3 | | |
| 8 | 7,9 | 8 | 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)