Genetic Algorithms

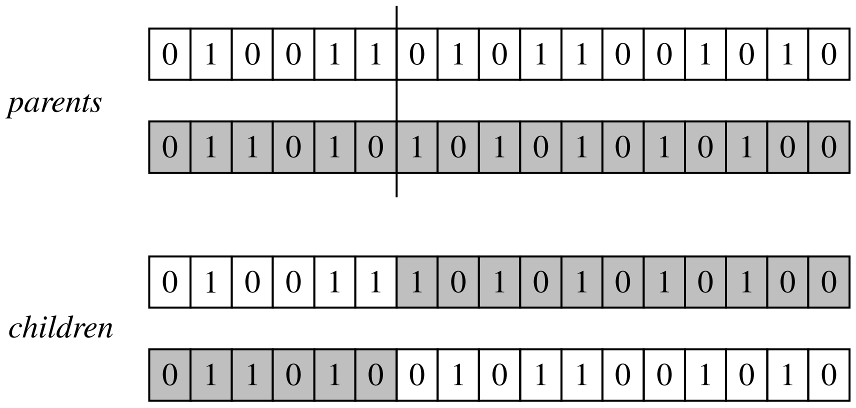
Introduction

- The *genetic algorithm* (GA), developed by John Holland and his collaborators in the 1960s and 1970s, is a model or abstraction of biological evolution based on Charles Darwin's theory of natural selection.
- There are many advantages of GAs over conventional optimization algorithms. Two of the most notable are:
  - The ability to deal with complex problems.
  - Parallelism.
- GAs can deal with various types of optimization, whether the objective function is linear or nonlinear, continuous or discontinuous, or with random noise.

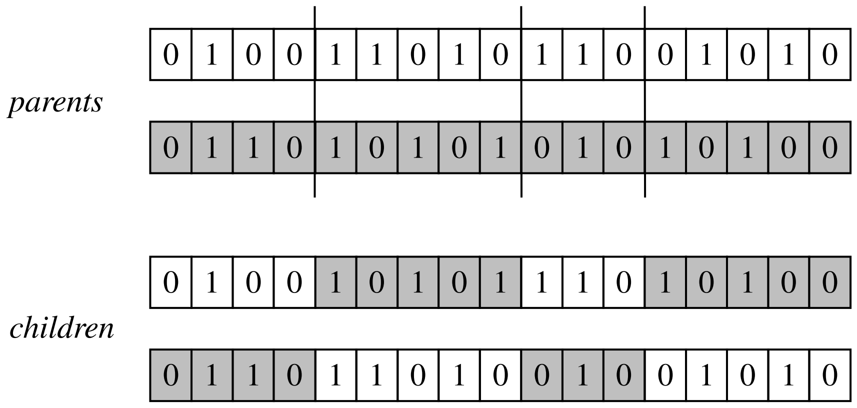
- GAs have three main genetic operators: crossover, mutation, and selection.
- Pseudo code of GAs:

```
BEGIN
Initialize the rates of survival ($p_s$) and mutation ($p_m$)
Generate the initial population;
Compute the fitness of each individual;
While (t<Max number of generations)
    Select the fittest individuals;
    Generate the new solution by crossover and mutation;
    Compute the fitness of offspring;
    Update $t=t+1$
end while
```

- *Representation*:
  - For simplicity, we consider binary strings for encoding and decoding.
  - Chromosome (string): 01001101011001010
    genes: 0、1、0、0、1、1….
- *Crossove*:

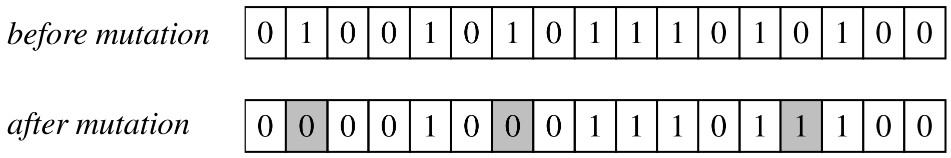


  - Step 1. Choose a random point on the two parents.
    Step 2. Split parents at this crossover point.
    Step 3. Create children by exchanging tails.
  - The main role is to provide mixing of the solutions and convergence in a subspace.
  - N-point crossover:



- Drawbacks:
  - ✧ For example, for two strings S_1=[1100] and S_2=[1011], whatever the crossover actions will be, their offspring will always be in the form [1...].
  - ✧ Two identical solutions will result in two identical offspring, no matter how the crossover has been applied.

- *Mutation*:



  - Alter each gene independently with a probability p_m.
  - Mutation increases the diversity of the population and provides a mechanism for escaping from a local optimum.
  - The mutation probability p_m is usually small, typically is the range of 0.01~0.3.
    - ✧ If the mutation probability is too high, the solutions could still "jump around", even if the optimal solution is approaching.

- *Selection*:
  - The selection of an individual in a population is carried out by the evaluation of its fitness.
  - Main idea: better individuals get higher chance to survive (elitism).
  - Implementation:
    - ✧ Roulette wheel selection
    - ✧ Fitness-proportional selection
    - ✧ Tournament selection

- Survival rate p_s is typically in the range of 0.3~0.6.
  - If p_s is too small (very strong elitism), it may lead to premature convergence. If p_s is too high, the crossover will occur sparsely, which is not efficient for the evolution.

- The choice of the right population size is also very important.
  - Small population size:
    - There is a danger of premature convergence.
  - Large population size:
    - More evaluations of the objective function are needed, which will require extensive computing time.
  - Studies and empirical observations suggest that population size $n = 20$ to 200 works for most problems.
- For more complex problems, various GA variants can be used to suit specific tasks.

Applying GA to the Knapsack Problem

- The 0/1 knapsack problem is a well-known combinatorial optimization problem that represents a certain class of real-world problems.
- 0/1 Knapsack Problem:

  Given a set of m items and a knapsack with

  P_i = Profit of item i

  W_i = Weight of item i

  C = Capacity of knapsack.

  Find a vector $\boldsymbol{x} = (x_1, x_2, ..., x_m) \in \{0,1\}^m$, such that $\sum_{i=1}^{m} W_i \cdot x_i \le C$ and for which

  $f(x) = \sum_{i=1}^{m} P_i \cdot x_i$ is maximum.

  - Greedy repair method:

    With the constraint of knapsack capacity, all infeasible binary solutions should be repaired before being evaluated.

    - ✧ Firstly, all items are sorted by their profit/weight ratio.
    - ✧ Then, items are removed step by step according to profit/weight ratio in ascending order until the capacity constraint is fulfilled.
    - ✧ Finally, items are added step by step according to profit/weight ratio in descending order as long as all capacity constraints are fulfilled.

Applying GA to the Traveling Salesman Problem

- The *traveling salesman problem* (TSP) is another well-known combinatorial optimization problem.
- Representation
  - *Permutation Representation*
    - ✧ A tour of a 9-city TSP
      3−2−5−4−7−1−6−9−8
      is simply represented as follows:

      $$[3 \quad 2 \quad 5 \quad 4 \quad 7 \quad 1 \quad 6 \quad 9 \quad 8]$$

    - ✧ This representation is also called a *path representation* or *order representation.*
    - ✧ This representation may lead to illegal tours if the tradidtional one-point crossover operator is used.
      - Some cities may be missed while some cities may be duplicated in the offspring.
  - *Random Keys Representation*
    - ✧ This representation encodes a solution with *random number* from (0,1).
    - ✧ A chromosome to a 9-city problem may be

      $$[0.23 \quad 0.82 \quad 0.45 \quad 0.74 \quad 0.87 \quad 0.11 \quad 0.56 \quad 0.69 \quad 0.78]$$

      where position i represents city i. Sort the random number in ascending order to get the following tour:
      6−1−3−7−8−4−9−2−5
- Crossover Operators
  - *Partial-Mapped Crossover (PMX)*

    Step 1. Select two positions along the string uniformly at random. The substings defined by the two positions are called the mapping sections.

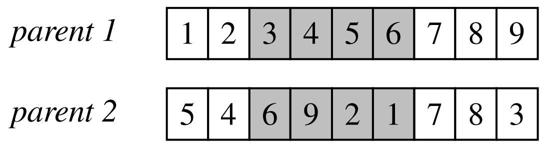
    Step 2. Exchange two substrings between parents to produce proto-children.

    Step 3. Determine the *mapping relationship* between two mapping sections.

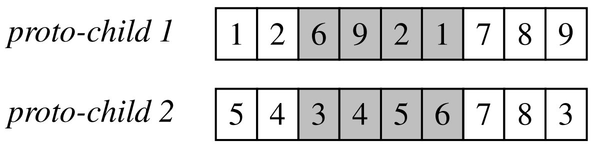
    Step 4. Legalize offspring with the mapping relationship.

    - ✧ Cities 1, 2, and 9 are duplicated in *proto-child* 1, while cities 3, 4, and 5 are missing from it. According to the mapping relationship determined in Step 3, the excessive cities 1, 2, and 9 should be replaced by the missing cities 3, 5, and 4, repsecitvely, while keeping the swapped substring unchanged.

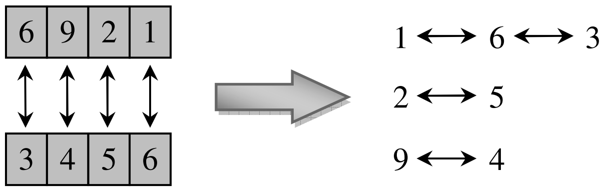
1. *select the substring at random*



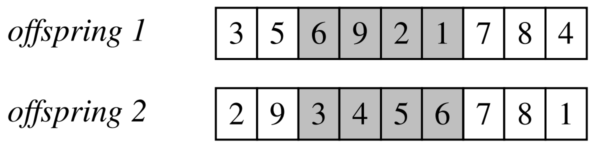
2. *exchange substrings between parents*



3. *determine mapping relationship*



4. *legalize offspring with mapping relationship*



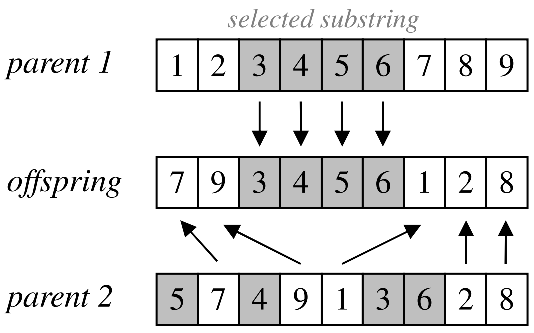
— *Order Crossover (OX)*

Step 1. Select a substring from one parent at random.

Step 2. Produce a proto-child by copying the substring into the correspoiding of it.

Step 3. Delete the cities which are already in the substring from the second parent. The resulted sequence of cities contains the cities that the proto-child needs.

Step 4. Place the cities into the unfixed positions of the proto-child from left to right according to the order of the sequence to produce an offspring.



✧ With the same steps, we can produce the second offspring as

$[2 \quad 5 \quad 4 \quad 9 \quad 1 \quad 3 \quad 6 \quad 7 \quad 8]$

from the other parent.

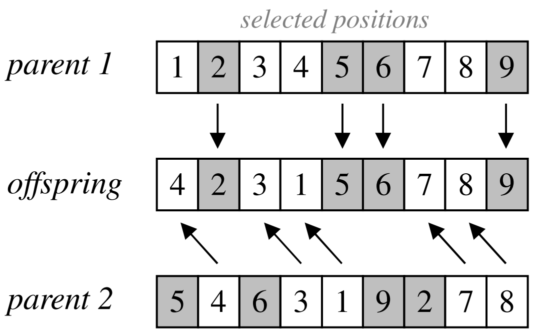
— *Position-Based Crossover*

Step 1. Select a set of positions from one parent at random.

Step 2. Produce a proto-child by copying the cites on these positions into the corresponding positions of the proto-child.

Step 3. Delete the cities which are already selected from the second parent. The resulting sequence of cities contains the cities the proto-child needs.

Step 4. Place the cities into the unfixed positions of the proto-child from left to right according to the order of the sequence to produce one offspring.



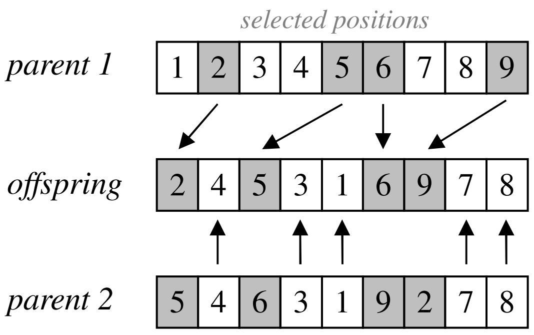
✧ With the same steps, we can produce the second offspring as

$[2 \quad 4 \quad 3 \quad 5 \quad 1 \quad 9 \quad 6 \quad 7 \quad 8]$

from the other parent.

— *Order-Based Crossover*

It is a slight variation of position-based crossover in which the order of cities in the selected position in one parent is imposed on the corresponding cities in the other parent.



✧ With the same steps, we can produce the second offspring as

$[4 \quad 2 \quad 3 \quad 1 \quad 5 \quad 6 \quad 7 \quad 9 \quad 8]$

from the other parent.

— *Cycle Crossover (CX)*

As with the position-based crossover, it takes some cities from one parent and selects the remaining cities from the other parent. The difference is that the cities from the first parent are not selected randomly and only those cities are selected which defined a cycle according to the cooresponding positions between parents.

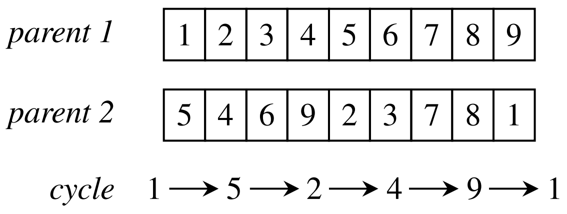
Step 1. Find the cycle which is defined by the corresponding positions of cities between parents.

Step 2. Copy the cities in the cycle to a child with the corresponding positions of one parent.

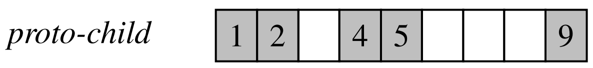
Step 3. Determine the remaining cities for the child by deleting those cities which are already in the cycle from the other parent.

Step 4. Fulfill the child with the remaining cities.

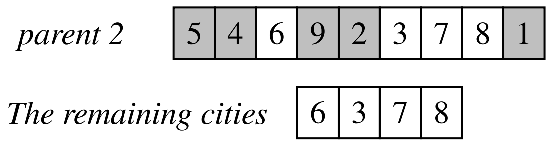
1. *Find the cycle defined by*



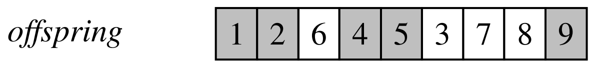
2. *Copy the cities in the cycle to a child*



3. *Determine the remaining cities for the child*



4. *Fulfill the child*



✧ With the same steps, we can produce the second offspring as

$$[5 \quad 4 \quad 3 \quad 9 \quad 2 \quad 6 \quad 7 \quad 8 \quad 1]$$

from the other parent.

— *Heuristic Crossover*

Step 1. For a pair of parents, pick a random city for the start.

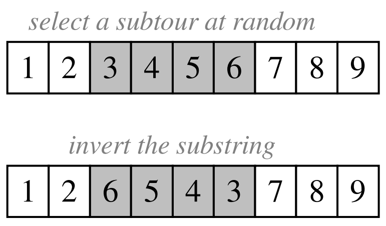
Step 2. Choose the shortest edge (that is presented in the parents) leading from the current city which does not lead to a cycle. If two edges lead to a cycle, chose a random city that continues the tour.

Step 3. If the tour is completed, stop; otherwise go to step 2.

- Mutation Operators

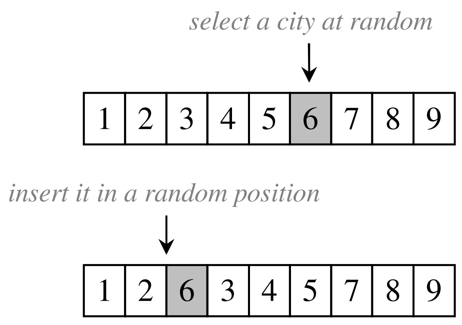
— *Inversion Mutation*

Inversion mutation selects two positions within a chromosome at random and then inverts the substring between these two positions.



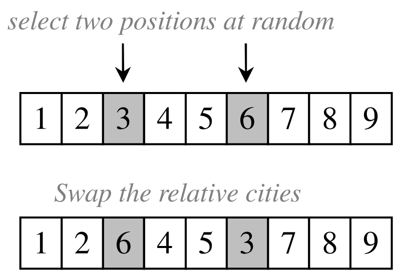
— *Insertion Mutation*

Insertion mutation selects a city at random and inserts it in a random position.



— *Reciprocal Exchange Mutation*

Reciprocal exchange mutation selects two positions at random and then swaps the cities on these positions.


select two positions at random
1 2 3 4 5 6 7 8 9
Swap the relative cities
1 2 6 4 5 3 7 8 9