Chapter 5. Metaheuristics

Algorithmics and Programming III

FIB

Albert Oliveras Enric Rodríguez

Q1 2019–2020

Version December 10, 2019

Chapter 5. Metaheuristics

Chapter 5. Metaheuristics

Computationally difficult problems

In this course we have mostly considered computationally intractable problems and ways to tackle them:

- For small inputs we can simply apply brute force
- For some problems,
  if enough memory is available we can apply dynamic programming
- For some problems,
  polynomial-time greedy algorithms can be used in certain subproblems
- Now we present metaheuristics:
  general strategies to search and quickly find (near-)optimal solutions

Computationally difficult problems

In this course we have mostly considered computationally intractable problems and ways to tackle them:

- For small inputs we can simply apply brute force
- For some problems,
  if enough memory is available we can apply dynamic programming
- For some problems,
  polynomial-time greedy algorithms can be used in certain subproblems
- Now we present metaheuristics:
  general strategies to search and quickly find (near-)optimal solutions

Computationally difficult problems

We are interested in solving Combinatorial Optimization Problems (COP)

Formally, given

- a set of variables $X = (x_1, x_2, \ldots, x_n)$ with domains $D_1, D_2, \ldots, D_n$,
- a set of constraints $C_1, C_2, \ldots, C_m$ among variables (each constraint can be seen as a subset $C_i \subseteq D_1 \times D_2 \times \cdots D_n$),
- an objective or cost function $f : D_1 \times D_2 \times \cdots D_n \to \mathbb{R}$,

we define the set of solutions $S := \cap_{i=1}^{m} C_i$, and

our goal is to find a (globally) minimal solution $s^* \in \underset{x \in S}{\operatorname{argmin}}\ f(x)$.

For example:

- Knapsack
- Min Graph Coloring
- Traveling Salesman Problem
- ...

Computationally difficult problems

We are interested in solving Combinatorial Optimization Problems (COP)

Formally, given

- a set of variables $X = (x_1, x_2, \ldots, x_n)$ with domains $D_1, D_2, \ldots, D_n$,
- a set of constraints $C_1, C_2, \ldots, C_m$ among variables
  (each constraint can be seen as a subset $C_i \subseteq D_1 \times D_2 \times \cdots D_n$),
- an objective or cost function $f : D_1 \times D_2 \times \cdots D_n \to \mathbb{R}$,

we define the set of solutions $S := \cap_{i=1}^{m} C_i$, and

our goal is to find a (globally) minimal solution $s^* \in \underset{x \in S}{\operatorname{argmin}} \ f(x)$.

For example:

- Knapsack
- Min Graph Coloring
- Traveling Salesman Problem
- ...

Chapter 5. Metaheuristics

Metaheuristics

- A metaheuristic is an algorithm for solving approximately a wide range of hard optimization problems without much adaptation to each problem
- Metaheuristics are:
  - efficient
  - approximate
  - non problem-specific
  - usually non-deterministic
- **DISCLAIMER:** in general,
  no guarantee on the quality of the solutions of metaheuristics is given

Metaheuristics

- A metaheuristic is an algorithm for solving approximately a wide range of hard optimization problems without much adaptation to each problem
- Metaheuristics are:
  - efficient
  - approximate
  - non problem-specific
  - usually non-deterministic
- **DISCLAIMER:** in general,
  no guarantee on the quality of the solutions of metaheuristics is given

Metaheuristics

A key aspect of metaheuristics is to provide a balance between:

- Intensification:
  intensively explore areas of the search space with good solutions
- Diversification:
  move to unexplored areas of the search space when necessary

The areas of the search space to explore are determined by neighborhoods:

- A neighborhood structure is a function $\mathcal{N} : S \to 2^S$.
  Given $s \in S$, we call $\mathcal{N}(s)$ the neighborhood of s.
- A local minimum with respect to a neighborhood structure $\mathcal{N}$ is
  a solution $\hat{s}$ such that $\forall s \in \mathcal{N}(\hat{s}) : f(\hat{s}) \leq f(s)$

Metaheuristics

A key aspect of metaheuristics is to provide a balance between:

- Intensification:
  intensively explore areas of the search space with good solutions
- Diversification:
  move to unexplored areas of the search space when necessary

The areas of the search space to explore are determined by neighborhoods:

- A neighborhood structure is a function $\mathcal{N} : S \to 2^S$.
  Given $s \in S$, we call $\mathcal{N}(s)$ the neighborhood of s.
- A local minimum with respect to a neighborhood structure $\mathcal{N}$ is
  a solution $\hat{s}$ such that $\forall s \in \mathcal{N}(\hat{s}) : f(\hat{s}) \leq f(s)$

Let us consider the Knapsack Problem:

given a set of items, each with a certain value and weight, the goal is to find a subset not exceeding a certain weight W and with maximum value

Item Id	1	2	3	4	5	6	7
Value	7	2	1	4	3	4	8
Weight	10	7	2	8	4	6	15

$W = 23$

Given a solution s, its neighborhood $\mathcal{N}(s)$ can be the set of solutions that can be obtained from s by replacing one item by another one

For example:

- $\{7, 5\}$ belongs to $\mathcal{N}(\{7, 4\})$
- $\{7, 4\}$ is a local optimum
- However, it is not global. A global optimum is $\{1, 3, 5, 6\}$.

Let us consider the Knapsack Problem:

given a set of items, each with a certain value and weight, the goal is to find a subset not exceeding a certain weight W and with maximum value

Item Id	1	2	3	4	5	6	7
Value	7	2	1	4	3	4	8
Weight	10	7	2	8	4	6	15

$W = 23$

Given a solution s, its neighborhood $\mathcal{N}(s)$ can be the set of solutions that can be obtained from s by replacing one item by another one

For example:

- $\{7, 5\}$ belongs to $\mathcal{N}(\{7, 4\})$
- $\{7, 4\}$ is a local optimum
- However, it is not global. A global optimum is $\{1, 3, 5, 6\}$.

Metaheuristics

Let us consider the Knapsack Problem:

given a set of items, each with a certain value and weight, the goal is to find a subset not exceeding a certain weight W and with maximum value

Item Id	1	2	3	4	5	6	7
Value	7	2	1	4	3	4	8
Weight	10	7	2	8	4	6	15

$W = 23$

Given a solution s, its neighborhood $\mathcal{N}(s)$ can be the set of solutions that can be obtained from s by replacing one item by another one

For example:

- $\{7, 5\}$ belongs to $\mathcal{N}(\{7, 4\})$
- $\{7, 4\}$ is a local optimum
- However, it is not global. A global optimum is $\{1, 3, 5, 6\}$.

Metaheuristics

Metaheuristics can be classified according to various criteria:

- Population-based vs single point search
- Dynamic vs static objective function
- One vs various neighborhood structures
- Memory usage vs memory-less methods
- Nature-inspired vs non-nature inspired

In the following we will introduce some of the most successful metaheuristics

Metaheuristics

Metaheuristics can be classified according to various criteria:

- Population-based vs single point search
- Dynamic vs static objective function
- One vs various neighborhood structures
- Memory usage vs memory-less methods
- Nature-inspired vs non-nature inspired

In the following we will introduce some of the most successful metaheuristics

Chapter 5. Metaheuristics

Basic Local Search

Basic Local Search, a.k.a Iterative Improvement or Hill Climbing:

1. Start with an arbitrary solution $s \in S$.
2. While possible, replace s by an element of $\mathcal{N}(s)$ with better cost

In pseudo-code:

```
s := generateInitialSolution()
repeat
  s := improve(s, 𝒩(s))
until no improvement is possible
```

The algorithm terminates at a local minimum,
but may not be a global minimum

Basic Local Search

This scheme indeed describes a large family of concrete algorithms:

- How do we choose the neighborhood structure $\mathcal{N}$?
  - It should be rich enough so that we do not tend to get stuck in bad local optima (see the previous example of knapsack)
  - It should not be too large, since we want to be able to efficiently search the neighborhood for possible local moves
- How do we implement $improve(s, \mathcal{N}(s))$?
  - **First improvement:** choose the first s' we find s.t. $f(s') < f(s)$
  - **Best improvement:** choose $s' \in \operatorname{argmin}_{\hat{s} \in \mathcal{N}(s)} f(\hat{s})$ (i.e. explore the whole $\mathcal{N}(s)$ and pick the best element)

Basic local search disallows to temporarily worsen the objective function, and hence it cannot escape from bad local optima!

Basic Local Search

This scheme indeed describes a large family of concrete algorithms:

- How do we choose the neighborhood structure $\mathcal{N}$?
  - It should be rich enough so that we do not tend to get stuck in bad local optima (see the previous example of knapsack)
  - It should not be too large, since we want to be able to efficiently search the neighborhood for possible local moves
- How do we implement *improve*$(s, \mathcal{N}(s))$?
  - **First improvement:** choose the first s' we find s.t. $f(s') < f(s)$
  - **Best improvement:** choose $s' \in \operatorname{argmin}_{\hat{s} \in \mathcal{N}(s)} f(\hat{s})$ (i.e. explore the whole $\mathcal{N}(s)$ and pick the best element)

Basic local search disallows to temporarily worsen the objective function, and hence it cannot escape from bad local optima!

Basic Local Search

This scheme indeed describes a large family of concrete algorithms:

- How do we choose the neighborhood structure $\mathcal{N}$?
  - It should be rich enough so that we do not tend to get stuck in bad local optima (see the previous example of knapsack)
  - It should not be too large, since we want to be able to efficiently search the neighborhood for possible local moves
- How do we implement *improve*$(s, \mathcal{N}(s))$?
  - **First improvement:** choose the first s' we find s.t. $f(s') < f(s)$
  - **Best improvement:** choose $s' \in \operatorname{argmin}_{\hat{s} \in \mathcal{N}(s)} f(\hat{s})$ (i.e. explore the whole $\mathcal{N}(s)$ and pick the best element)

Basic local search disallows to temporarily worsen the objective function, and hence it cannot escape from bad local optima!

Chapter 5. Metaheuristics

Simulated Annealing

- Key idea: allow moves resulting in solutions of worse quality than the current solution in order to escape from local optima
- The probability of doing such a move is decreased during the search

```
s := generateInitialSolution()
T := T0
while termination conditions not met do
    s' := pickAtRandom(N(s))
    if f(s') < f(s) then s' := s
    else with probability p(T,s',s), s' := s
    endif
    update(T)
endwhile
```

Simulated Annealing

- How do we define the probability of accepting a worsening move?
  - The probability is usually computed with the Boltzmann distribution

$$p(T, s', s) = exp(-\frac{f(s') - f(s)}{T})$$

  - Given a fixed temperature T, the farther $f(s')$ is from $f(s)$, the lower the probability of accepting the move
  - Given fixed s, s', since $f(s') > f(s)$, the lower the temperature T the lower the probability of accepting a worsening move
- How do we update the temperature?
  - The temperature at iteration k is defined as a function of the current temperature T_k and the iteration counter k
  - There are functions that guarantee the convergence to a global optimum, but they are too slow to be feasible in practice
  - One of the most useful functions follows a geometric law $T_{k+1} = \alpha \cdot T_k$, with $\alpha \in [0, 1]$

Simulated Annealing

- How do we define the probability of accepting a worsening move?
  - The probability is usually computed with the Boltzmann distribution

$$p(T, s', s) = exp(-\frac{f(s') - f(s)}{T})$$

  - Given a fixed temperature T, the farther $f(s')$ is from $f(s)$, the lower the probability of accepting the move
  - Given fixed s, s', since $f(s') > f(s)$, the lower the temperature T the lower the probability of accepting a worsening move

- How do we update the temperature?
  - The temperature at iteration k is defined as a function of the current temperature T_k and the iteration counter k
  - There are functions that guarantee the convergence to a global optimum, but they are too slow to be feasible in practice
  - One of the most useful functions follows a geometric law $T_{k+1} = \alpha \cdot T_k$, with $\alpha \in [0, 1]$

Chapter 5. Metaheuristics

Tabu Search

- The simplest form of Tabu Search applies a best-improvement local search as basic ingredient and uses a memory (tabu list) to escape from local optima and avoid cycles
- The tabu list keeps track of the most recently visited solutions. The solutions in the tabu list are excluded from the neighborhood of the current solution
- At each iteration, the best solution in neighborhood is chosen and added to the tabu list and the oldest of the solutions in the list is removed in a FIFO order.

Tabu Search

- How do we store the solutions in the tabu list?
  - Storing complete solutions is impractical,
    and so only attributes are stored (e.g. components of solutions).
    One tabu list for each attribute is kept.
  - Forbidding an attribute discards probably more than one solution.
    So it is possible that good unvisited solutions are disallowed
  - To overcome this problem, aspiration criteria are defined.
    They allow a solution even if forbidden by the tabu conditions.

    Example: allow solutions that are better than the current best one.

Tabu Search

- How do we store the solutions in the tabu list?
  - Storing complete solutions is impractical,
    and so only attributes are stored (e.g. components of solutions).
    One tabu list for each attribute is kept.
  - Forbidding an attribute discards probably more than one solution.
    So it is possible that good unvisited solutions are disallowed
  - To overcome this problem, aspiration criteria are defined.
    They allow a solution even if forbidden by the tabu conditions.

    Example: allow solutions that are better than the current best one.

Tabu Search

- How do we store the solutions in the tabu list?
  - Storing complete solutions is impractical,
    and so only attributes are stored (e.g. components of solutions).
    One tabu list for each attribute is kept.
  - Forbidding an attribute discards probably more than one solution.
    So it is possible that good unvisited solutions are disallowed
  - To overcome this problem, aspiration criteria are defined.
    They allow a solution even if forbidden by the tabu conditions.

    Example: allow solutions that are better than the current best one.

Tabu Search

```
s := generateInitialSolution()
initializeTabuLists(TL_1, TL_2, ..., TL_r)
k := 0
while termination conditions not met do
    allowedSet = {s' ∈ N(s) | s' is not forbidden by tabu or
                              satisfies an aspiration cond.}
    s := chooseBestOf(allowedSet)
    updateTabuListAndAspirationConditions(s)
    k := k + 1
endwhile
```

Chapter 5. Metaheuristics

GRASP

- The Greedy Randomized Adaptive Search Procedure (GRASP) is a two-phase iterative procedure:
  1. Solution construction
  2. Solution improvement

- PHASE 1: Solution construction
  - Assume that a solution s consists of a subset of a set of elements (e.g. in knapsack it is a subset of items).
    The solution is constructed by adding one new element at a time
  - Next element is chosen randomly from a candidate list as follows
  - Each element is assigned a score that estimates the benefit if the element is inserted into the current partial solution
  - The Restricted Candidate List (RCL) is composed of the best α elements according to the scores
  - For $\alpha = 1$ the construction amounts to a greedy heuristic.
    For $\alpha = n$, the construction is completely random.

GRASP

- The Greedy Randomized Adaptive Search Procedure (GRASP) is a two-phase iterative procedure:
  1. Solution construction
  2. Solution improvement

- PHASE 1: Solution construction
  - Assume that a solution *s* consists of a subset of a set of elements (e.g. in knapsack it is a subset of items).
    The solution is constructed by adding one new element at a time
  - Next element is chosen randomly from a candidate list as follows
  - Each element is assigned a score that estimates the benefit if the element is inserted into the current partial solution
  - The Restricted Candidate List (RCL) is composed of the best α elements according to the scores
  - For $\alpha = 1$ the construction amounts to a greedy heuristic.
    For $\alpha = n$, the construction is completely random.

- PHASE 1 in pseudo-code:

```
s := ∅
α := determineCandidateListLength()
while solution not complete do
    RCL := generatedRestrictedCandidateList()
    x := selectElementAtRandom(RCL)
    s := s ∪ {x}
    updateGreedyFunction(s)
endwhile
```

- PHASE 2: Solution improvement

  It is a local search process
  (Basic Local Search, Simulated Annealing, Tabu Search, ...)

- The two phases are combined as follows:

```
while termination conditions not met do
    s := constructGreedyRandomizedSolution()
    s := applyLocalSearch(s)
    memorizeBestFoundSolution()
endwhile
```

GRASP

- PHASE 1 in pseudo-code:

```
s := ∅
α := determineCandidateListLength()
while solution not complete do
    RCL := generatedRestrictedCandidateList()
    x := selectElementAtRandom(RCL)
    s := s ∪ {x}
    updateGreedyFunction(s)
endwhile
```

- PHASE 2: Solution improvement

  It is a local search process
  (Basic Local Search, Simulated Annealing, Tabu Search, ...)

- The two phases are combined as follows:

```
while termination conditions not met do
    s := constructGreedyRandomizedSolution()
    s := applyLocalSearch(s)
    memorizeBestFoundSolution()
endwhile
```

GRASP

- PHASE 1 in pseudo-code:

```
s := ∅
α := determineCandidateListLength()
while solution not complete do
    RCL := generatedRestrictedCandidateList()
    x := selectElementAtRandom(RCL)
    s := s ∪ {x}
    updateGreedyFunction(s)
endwhile
```

- PHASE 2: Solution improvement

  It is a local search process
  (Basic Local Search, Simulated Annealing, Tabu Search, ...)

- The two phases are combined as follows:

```
while termination conditions not met do
    s := constructGreedyRandomizedSolution()
    s := applyLocalSearch(s)
    memorizeBestFoundSolution()
endwhile
```

GRASP

- GRASP is very effective if two conditions are satisfied:
  - Phase 1 samples the most promising regions of the search space
  - Phase 1 returns solutions belonging to basins of attraction of different local optima
- Thanks to is simplicity, GRASP is generally very fast and produces quite good solutions in a very short amount of time

Chapter 5. Metaheuristics

Variable Neighborhood Search

- Key idea: dynamically change neighborhood structures
- A local optimum with respect to one neighbourhood structure is not necessarily a local optimum for another one

Variable Neighborhood Search

- We assume a sequence of neighborhood structures $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{kmax}$
- Although they could be arbitrary, usually $|\mathcal{N}_1| < |\mathcal{N}_2| < \ldots < |\mathcal{N}_{kmax}|$.
  As we will see, $\mathcal{N}_1 \subset \mathcal{N}_2 \subset \ldots \subset \mathcal{N}_{kmax}$ is not an efficient choice because work would be repeated.
- A solution that is locally optimal wrt. $\mathcal{N}_k$ is probably not locally optimal wrt. $\mathcal{N}_{k+1}$
- This property is exploited by Variable Neighborhood Descent (VND)

Variable Neighborhood Search

- We assume a sequence of neighborhood structures $\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_{kmax}$
- Although they could be arbitrary, usually $|\mathcal{N}_1| < |\mathcal{N}_2| < \ldots < |\mathcal{N}_{kmax}|$.
  As we will see, $\mathcal{N}_1 \subset \mathcal{N}_2 \subset \ldots \subset \mathcal{N}_{kmax}$ is not an efficient choice because work would be repeated.
- A solution that is locally optimal wrt. $\mathcal{N}_k$ is probably not locally optimal wrt. $\mathcal{N}_{k+1}$
- This property is exploited by Variable Neighborhood Descent (VND)

Variable Neighborhood Search

Variable Neighborhood Descent:

```
s := generateInitialSolution()
k := 1
while k <= kmax do
    s' := chooseBestOf(N_k(s))
    if f(s') < f(s) then
        s := s'
        k := 1
    else
        k := k + 1
    endif
endwhile
```

- The algorithm terminates when it reaches a solution that is a local minimum with respect to all neighborhoods
- Finding the best of the neighborhood corresponds to intensification
- Changing neighborhood corresponds to diversification

Variable Neighborhood Search

- Variable Neighborhood Search (VNS) generalizes VND
- Starting with an initial solution s, the algorithm repeats 3 steps:
  1. Shaking: $s' \in \mathcal{N}_k(s)$ is randomly chosen
  2. Local search: obtain s'' from s' using any local search procedure
  3. Move:
     - if s'' better than s, replace it and set $k := 1$
     - otherwise $k := k + 1$

Variable Neighborhood Search

Variable Neighborhood Search:

```
s := generateInitialSolution()
while termination conditions not met do
    k := 1
    while k <= kmax do
        s' := pickAtRandom(N_k(s))
        s'' := localSearch(s')
        if f(s'') < f(s) then
            s := s''
            k := 1
        else
            k := k + 1
        endif
    endwhile
endwhile
```

Variable Neighborhood Search

- Shaking perturbs s to provide a good starting point for local search
  1. If s is a local minimum,
     s' should belong to the basin of attraction of another local minimum
  2. s' should not be too different from s so as to keep good features of s

- The choice of the neighborhoods is critical for VNS (and also for VND)
  - Neighborhoods should give different abstractions of search space

Chapter 5. Metaheuristics

Guided Local Search

- To escape from local optima,
  Guided Local Search (GLS) dynamically changes the objective function so as to make them less desirable.
- Let us fix a set of m solution features.
  A feature can be any property that discriminates between solutions
- We will use m indicator functions $I_i(x)$ that return value 1 iff the feature i is present in solution x.
- After a solution is found, the new objective function f' is redefined as

$$f'(x) = f(x) + \lambda \sum_{i=1}^{m} p_i \cdot I_i(x)$$

  where
  - $\lambda > 0$ is called the regularization parameter, and
  - $p_i \geq 0$ are the penalty parameters

Guided Local Search

- To escape from local optima,
  Guided Local Search (GLS) dynamically changes the objective function so as to make them less desirable.
- Let us fix a set of m solution features.
  A feature can be any property that discriminates between solutions
- We will use m indicator functions $I_i(x)$ that return value 1 iff the feature i is present in solution x.
- After a solution is found, the new objective function f' is redefined as

$$f'(x) = f(x) + \lambda \sum_{i=1}^{m} p_i \cdot I_i(x)$$

  where
  - $\lambda > 0$ is called the regularization parameter, and
  - $p_i \geq 0$ are the penalty parameters

Guided Local Search

- To escape from local optima,
  Guided Local Search (GLS) dynamically changes the objective function so as to make them less desirable.
- Let us fix a set of m solution features.
  A feature can be any property that discriminates between solutions
- We will use m indicator functions $I_i(x)$ that return value 1 iff the feature i is present in solution x.
- After a solution is found, the new objective function f' is redefined as

$$f'(x) = f(x) + \lambda \sum_{i=1}^{m} p_i \cdot I_i(x)$$

  where
  - $\lambda > 0$ is called the regularization parameter, and
  - $p_i \geq 0$ are the penalty parameters

Guided Local Search

- In pseudo-code:

```
s := generateInitialSolution()
s* := s
(p1,p2,...,pm) := (0,0,...,0)
while termination conditions not met do
    (s,sf) := localSearch(s,f',f)
     if f(sf) < f(s*) then
        s* := sf
    update(s,p1,p2,...,pm)
endwhile
```

- We explicitly keep two solutions
  - s, the current solution
  - s^*, the best solution found so far with respect to f
- The local search procedure provides two solutions:
  1. a solution s aimed at optimizing f'
  2. the best solution s_f with respect to f found in the local search

Guided Local Search

- The penalty parameters are updated as follows
- Each solution feature has a fixed cost c_i
- Every time local search produces a new solution,
  GLS tries to penalize the features of this solution with highest cost
- Given a new solution s, the utility of penalizing a feature i is

$$Util(s, i) = I_i(s) \cdot \frac{c_i}{1 + p_i}$$

- When updating, for all features i such that $Util(s, i)$ is maximum,
  we set $p_i := p_i + 1$.
- This scheme penalizes features with high cost c_i,
  but dividing by $1 + p_i$ prevents a feature from being repeatedly penalized.

Guided Local Search

- The penalty parameters are updated as follows
- Each solution feature has a fixed cost c_i
- Every time local search produces a new solution,
  GLS tries to penalize the features of this solution with highest cost
- Given a new solution s, the utility of penalizing a feature i is

$$Util(s, i) = I_i(s) \cdot \frac{c_i}{1 + p_i}$$

- When updating, for all features i such that $Util(s, i)$ is maximum,
  we set $p_i := p_i + 1$.
- This scheme penalizes features with high cost c_i,
  but dividing by $1 + p_i$ prevents a feature from being repeatedly penalized.

Guided Local Search

- The penalty parameters are updated as follows
- Each solution feature has a fixed cost c_i
- Every time local search produces a new solution,
  GLS tries to penalize the features of this solution with highest cost
- Given a new solution s, the utility of penalizing a feature i is

$$Util(s, i) = I_i(s) \cdot \frac{c_i}{1 + p_i}$$

- When updating, for all features i such that $Util(s, i)$ is maximum,
  we set $p_i := p_i + 1$.
- This scheme penalizes features with high cost c_i,
  but dividing by $1 + p_i$ prevents a feature from being repeatedly penalized.

Chapter 5. Metaheuristics

Genetic Algorithms

- At every iteration of the algorithm we deal with a set of solutions, which we refer to as individuals.
- This population of individuals evolves at each step by the following rules:
  - New individuals are created by applying recombination, which crosses over two or more individuals (parents) to produce one or more new individuals (children or offspring).
  - Mutation allows that new traits appear in the offspring
  - Selection determines which individuals will be maintained into the next generation by evaluating their fitness, i.e. how good they are.
- This process can be iterated for a predefined number of generations
- On termination we return the best individual ever found

Genetic Algorithms

- At every iteration of the algorithm we deal with a set of solutions, which we refer to as individuals.
- This population of individuals evolves at each step by the following rules:
  - New individuals are created by applying recombination, which crosses over two or more individuals (parents) to produce one or more new individuals (children or offspring).
  - Mutation allows that new traits appear in the offspring
  - Selection determines which individuals will be maintained into the next generation by evaluating their fitness, i.e. how good they are.
- This process can be iterated for a predefined number of generations
- On termination we return the best individual ever found

Genetic Algorithms

- At every iteration of the algorithm we deal with a set of solutions, which we refer to as individuals.
- This population of individuals evolves at each step by the following rules:
  - New individuals are created by applying recombination, which crosses over two or more individuals (parents) to produce one or more new individuals (children or offspring).
  - Mutation allows that new traits appear in the offspring
  - Selection determines which individuals will be maintained into the next generation by evaluating their fitness, i.e. how good they are.
- This process can be iterated for a predefined number of generations
- On termination we return the best individual ever found

Genetic Algorithms

- In pseudo-code:

```
P := generateInitialPopulation()
while termination conditions not met do
    Par := selectParents(P)
    P′ := recombine(Par)
    P″ := mutate(P′)
    P := selectIndividuals(P″ ∪ P)
endwhile
```

- Genetic algorithms vary depending on:
  1. the representation of the individuals
  2. the selection strategy for parents
  3. the recombination (a.k.a. crossover) procedure
  4. the mutation operators
  5. the selection strategy of individuals for the next generation

Representation of individuals

- Individuals are often represented using bit strings of fixed length
- They are also sometimes viewed as permutations of integer numbers

Selection strategy for parents

- Let us assume that for each individual s_i we have evaluated its fitness f_i
- The fitness is usually the value of the objective function, i.e., $f_i = f(s_i)$
- Roulette-wheel selection:

  The probability to choose individual s_i is $f_i / \sum_{j=1}^{n} f_j$
- Ranking selection:

  Sort all n individuals in increasing order of fitness.
  The i-th individual in the sorted list gets rank i, for $i = 1, 2, \ldots$
  The probability to select an individual x is $2 \cdot rank(x)/(n \cdot (n+1))$.
- Tournament selection:

  Choose k (the tournament size) individuals at random
  Choose the best individual from the tournament with probability p
  Choose the second best individual with probability $p(1-p)$
  Choose the third best individual with probability $p((1-p)^2)$
  ...

Selection strategy for parents

- Let us assume that for each individual s_i we have evaluated its fitness f_i
- The fitness is usually the value of the objective function, i.e., $f_i = f(s_i)$
- Roulette-wheel selection:

  The probability to choose individual s_i is $f_i / \sum_{j=1}^{n} f_j$
- Ranking selection:

  Sort all n individuals in increasing order of fitness.
  The i-th individual in the sorted list gets rank i, for $i = 1, 2, \ldots$
  The probability to select an individual x is $2 \cdot rank(x)/(n \cdot (n+1))$.
- Tournament selection:

  Choose k (the tournament size) individuals at random
  Choose the best individual from the tournament with probability p
  Choose the second best individual with probability $p(1-p)$
  Choose the third best individual with probability $p((1-p)^2)$
  ...

Selection strategy for parents

- Let us assume that for each individual s_i we have evaluated its fitness f_i
- The fitness is usually the value of the objective function, i.e., $f_i = f(s_i)$
- Roulette-wheel selection:

  The probability to choose individual s_i is $f_i / \sum_{j=1}^{n} f_j$
- Ranking selection:

  Sort all n individuals in increasing order of fitness.
  The i-th individual in the sorted list gets rank i, for $i = 1, 2, \ldots$
  The probability to select an individual x is $2 \cdot rank(x)/(n \cdot (n+1))$.
- Tournament selection:

  Choose k (the tournament size) individuals at random
  Choose the best individual from the tournament with probability p
  Choose the second best individual with probability $p(1-p)$
  Choose the third best individual with probability $p((1-p)^2)$
  ...

Selection strategy for parents

- Let us assume that for each individual s_i we have evaluated its fitness f_i
- The fitness is usually the value of the objective function, i.e., $f_i = f(s_i)$
- Roulette-wheel selection:

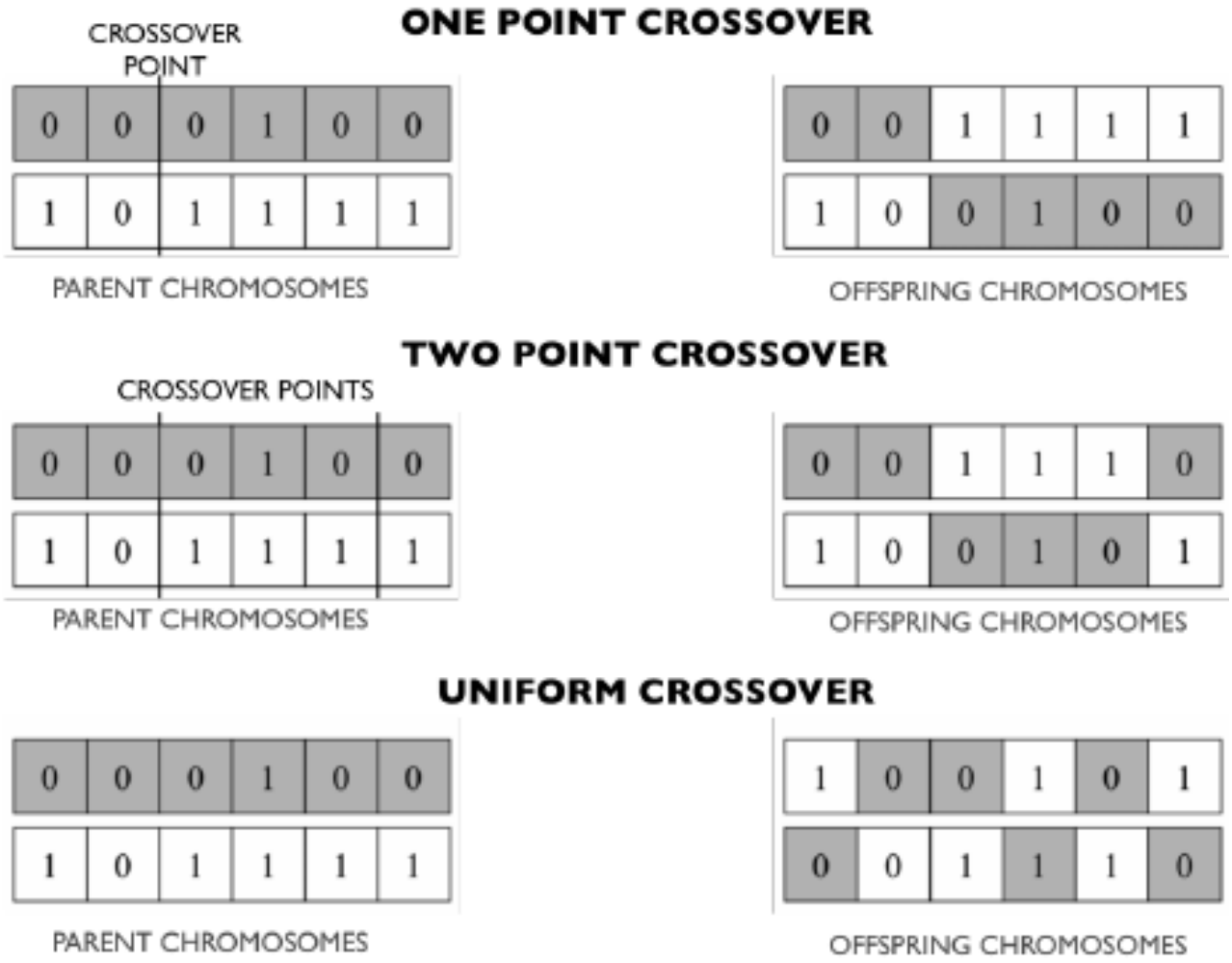
  The probability to choose individual s_i is $f_i / \sum_{j=1}^{n} f_j$
- Ranking selection:

  Sort all n individuals in increasing order of fitness.
  The i-th individual in the sorted list gets rank i, for $i = 1, 2, \ldots$
  The probability to select an individual x is $2 \cdot rank(x)/(n \cdot (n+1))$.
- Tournament selection:

  Choose k (the tournament size) individuals at random
  Choose the best individual from the tournament with probability p
  Choose the second best individual with probability $p(1-p)$
  Choose the third best individual with probability $p((1-p)^2)$
  ...

Crossover

- When individuals are encoded as bit strings:



Genetic Algorithms

- When individuals are encoded as permutations of integers:
- Order crossover:

$P_1 = (4\ 5\ 2 \mid 1\ 8\ 7\ 6 \mid 9\ 3)$ Select two random cut points
$P_2 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9)$

$O_1 = (\quad\quad \mid 1\ 8\ 7\ 6 \mid \quad\quad)$
$O_2 = (\quad\quad \mid 4\ 5\ 6\ 7 \mid \quad\quad)$

- When individuals are encoded as permutations of integers:
- Order crossover:

$P_1 = (4\ 5\ 2 \mid 1\ 8\ 7\ 6 \mid 9\ 3)$ Select two random cut points
$P_2 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9)$

$O_1 = (\quad\quad \mid 1\ 8\ 7\ 6 \mid \quad\quad)$
$O_2 = (\quad\quad \mid 4\ 5\ 6\ 7 \mid \quad\quad)$

To generate O_1, take P_2 and from $1, 2, 3, 4, 5, 6, 7, 8, 9$ remove $\{1, 8, 7, 6\}$. We obtain $2, 3, 4, 5, 9$, which we use to fill O_1.

Genetic Algorithms

- When individuals are encoded as permutations of integers:
- Order crossover:

$P_1 = (4\ 5\ 2 \mid 1\ 8\ 7\ 6 \mid 9\ 3)$ Select two random cut points
$P_2 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9)$

$O_1 = (2\ 3\ 4 \mid 1\ 8\ 7\ 6 \mid 5\ 9\ )$
$O_2 = (\quad\quad \mid 4\ 5\ 6\ 7 \mid \quad\ )$

To generate O_1, take P_2 and from $1, 2, 3, 4, 5, 6, 7, 8, 9$ remove $\{1, 8, 7, 6\}$. We obtain $2, 3, 4, 5, 9$, which we use to fill O_1.

- When individuals are encoded as permutations of integers:
- Order crossover:

$P_1 = (4\ 5\ 2 \mid 1\ 8\ 7\ 6 \mid 9\ 3)$ Select two random cut points
$P_2 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9)$

$O_1 = (2\ 3\ 4 \mid 1\ 8\ 7\ 6 \mid 5\ 9\ )$
$O_2 = (\quad\quad \mid 4\ 5\ 6\ 7 \mid \quad\ )$

To generate O_1, take P_2 and from $1, 2, 3, 4, 5, 6, 7, 8, 9$ remove $\{1, 8, 7, 6\}$. We obtain $2, 3, 4, 5, 9$, which we use to fill O_1.

To generate O_2, take P_1 and from $4, 5, 2, 1, 8, 7, 6, 9, 3$ remove $\{4, 5, 6, 7\}$. We obtain $2, 1, 8, 9, 3$, which we use to fill O_2.

- When individuals are encoded as permutations of integers:
- Order crossover:

$P_1 = (4\ 5\ 2 \mid 1\ 8\ 7\ 6 \mid 9\ 3)$ Select two random cut points
$P_2 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9)$

$O_1 = (2\ 3\ 4 \mid 1\ 8\ 7\ 6 \mid 5\ 9)$

$O_2 = (2\ 1\ 8 \mid 4\ 5\ 6\ 7 \mid 9\ 3\ )$

To generate O_1, take P_2 and from $1, 2, 3, 4, 5, 6, 7, 8, 9$ remove $\{1, 8, 7, 6\}$. We obtain $2, 3, 4, 5, 9$, which we use to fill O_1.

To generate O_2, take P_1 and from $4, 5, 2, 1, 8, 7, 6, 9, 3$ remove $\{4, 5, 6, 7\}$. We obtain $2, 1, 8, 9, 3$, which we use to fill O_2.

Genetic Algorithms

- Partially-mapped crossover:

  $P_1 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9)$ Select two random cut points

  $P_2 = (4\ 5\ 2 \mid 1\ 8\ 7\ 6 \mid 9\ 3)$

  $O_1 = (\quad\quad \mid 1\ 8\ 7\ 6 \mid \quad\quad)$

  $O_2 = (\quad\quad \mid 4\ 5\ 6\ 7 \mid \quad\quad)$

- Partially-mapped crossover:

  $P_1 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9)$ Select two random cut points
  $P_2 = (4\ 5\ 2 \mid 1\ 8\ 7\ 6 \mid 9\ 3)$

  $O_1 = (\quad\quad \mid 1\ 8\ 7\ 6 \mid \quad\quad)$
  $O_2 = (\quad\quad \mid 4\ 5\ 6\ 7 \mid \quad\quad)$

  This induces mappings

  $$\sigma(1) = 4, \quad \tau(4) = 1$$
  $$\sigma(8) = 5, \quad \tau(5) = 8$$
  $$\sigma(7) = 6, \quad \tau(6) = 7$$
  $$\sigma(6) = 7, \quad \tau(7) = 6$$

  We can try to fill the leftmost and the rightmost parts of O_1 as P_1.
  If i is already in the permutation, then put $\sigma(i)$ instead.

  Similarly with O_2 using mapping τ.

- Partially-mapped crossover:

  $P_1 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9)$ Select two random cut points
  $P_2 = (4\ 5\ 2 \mid 1\ 8\ 7\ 6 \mid 9\ 3)$

  $O_1 = (4\ 2\ 3 \mid 1\ 8\ 7\ 6 \mid 5\ 9)$
  $O_2 = (1\ 8\ 2 \mid 4\ 5\ 6\ 7 \mid 9\ 3)$

  This induces mappings

  $$\sigma(1) = 4, \quad \tau(4) = 1$$
  $$\sigma(8) = 5, \quad \tau(5) = 8$$
  $$\sigma(7) = 6, \quad \tau(6) = 7$$
  $$\sigma(6) = 7, \quad \tau(7) = 6$$

  We can try to fill the leftmost and the rightmost parts of O_1 as P_1.
  If i is already in the permutation, then put $\sigma(i)$ instead.

  Similarly with O_2 using mapping τ.

- Cycle crossover:

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

We will first compute $P_2 \circ P_1^{-1}$

To that end, let us view permutations as functions from $\{1, 2, \ldots, 10\}$ to $\{1, 2, \ldots, 10\}$ by writing a top row.

$$P_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 6 & 5 & 9 & 4 & 8 & 3 & 7 & 2 & 10 \end{pmatrix}$$

$$P_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 2 & 7 & 5 & 6 & 1 & 10 & 9 & 4 & 8 \end{pmatrix}$$

Then

$$P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 9 & 7 & 5 & 3 & 2 & 8 & 6 & 4 & 10 \end{pmatrix}$$

$$P_2 \circ P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 4 & 10 & 6 & 7 & 2 & 9 & 1 & 5 & 8 \end{pmatrix}$$

- Cycle crossover:

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

We will first compute $P_2 \circ P_1^{-1}$

To that end, let us view permutations as functions from $\{1, 2, \ldots, 10\}$ to $\{1, 2, \ldots, 10\}$ by writing a top row.

$$P_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 6 & 5 & 9 & 4 & 8 & 3 & 7 & 2 & 10 \end{pmatrix}$$

$$P_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 2 & 7 & 5 & 6 & 1 & 10 & 9 & 4 & 8 \end{pmatrix}$$

Then

$$P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 9 & 7 & 5 & 3 & 2 & 8 & 6 & 4 & 10 \end{pmatrix}$$

$$P_2 \circ P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 4 & 10 & 6 & 7 & 2 & 9 & 1 & 5 & 8 \end{pmatrix}$$

- Cycle crossover:

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

We will first compute $P_2 \circ P_1^{-1}$

To that end, let us view permutations as functions from $\{1, 2, \ldots, 10\}$ to $\{1, 2, \ldots, 10\}$ by writing a top row.

$$P_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 6 & 5 & 9 & 4 & 8 & 3 & 7 & 2 & 10 \end{pmatrix}$$

$$P_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 2 & 7 & 5 & 6 & 1 & 10 & 9 & 4 & 8 \end{pmatrix}$$

Then

$$P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 9 & 7 & 5 & 3 & 2 & 8 & 6 & 4 & 10 \end{pmatrix}$$

$$P_2 \circ P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 4 & 10 & 6 & 7 & 2 & 9 & 1 & 5 & 8 \end{pmatrix}$$

- Cycle crossover:

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

We will first compute $P_2 \circ P_1^{-1}$

To that end, let us view permutations as functions from $\{1, 2, \ldots, 10\}$ to $\{1, 2, \ldots, 10\}$ by writing a top row.

$$P_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 6 & 5 & 9 & 4 & 8 & 3 & 7 & 2 & 10 \end{pmatrix}$$

$$P_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 2 & 7 & 5 & 6 & 1 & 10 & 9 & 4 & 8 \end{pmatrix}$$

Then

$$P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 9 & 7 & 5 & 3 & 2 & 8 & 6 & 4 & 10 \end{pmatrix}$$

$$P_2 \circ P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 4 & 10 & 6 & 7 & 2 & 9 & 1 & 5 & 8 \end{pmatrix}$$

- Cycle crossover:

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

We will first compute $P_2 \circ P_1^{-1}$

To that end, let us view permutations as functions from $\{1, 2, \ldots, 10\}$ to $\{1, 2, \ldots, 10\}$ by writing a top row.

$$P_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 6 & 5 & 9 & 4 & 8 & 3 & 7 & 2 & 10 \end{pmatrix}$$

$$P_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 2 & 7 & 5 & 6 & 1 & 10 & 9 & 4 & 8 \end{pmatrix}$$

Then

$$P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 9 & 7 & 5 & 3 & 2 & 8 & 6 & 4 & 10 \end{pmatrix}$$

$$P_2 \circ P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 4 & 10 & 6 & 7 & 2 & 9 & 1 & 5 & 8 \end{pmatrix}$$

$$P_2 \circ P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 4 & 10 & 6 & 7 & 2 & 9 & 1 & 5 & 8 \end{pmatrix}$$

Now let us find the cycles of $P_2 \circ P_1^{-1}$:

- Starting from 1 we move to 3, then to 10, then to 8, then back to 1: $(1, 3, 10, 8)$
- Starting from 6 we move to 2, then to 4, then back to 6: $(6, 2, 4)$
- Starting from 5 we move to 7, then to 9, then back to 5: $(5, 7, 9)$

$$P_2 \circ P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 4 & 10 & 6 & 7 & 2 & 9 & 1 & 5 & 8 \end{pmatrix}$$

Now let us find the cycles of $P_2 \circ P_1^{-1}$:

- Starting from 1 we move to 3, then to 10, then to 8, then back to 1: $(1, 3, 10, 8)$
- Starting from 6 we move to 2, then to 4, then back to 6: $(6, 2, 4)$
- Starting from 5 we move to 7, then to 9, then back to 5: $(5, 7, 9)$

$$P_2 \circ P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 4 & 10 & 6 & 7 & 2 & 9 & 1 & 5 & 8 \end{pmatrix}$$

Now let us find the cycles of $P_2 \circ P_1^{-1}$:

- Starting from 1 we move to 3, then to 10, then to 8, then back to 1: $(1, 3, 10, 8)$
- Starting from 6 we move to 2, then to 4, then back to 6: $(6, 2, 4)$
- Starting from 5 we move to 7, then to 9, then back to 5: $(5, 7, 9)$

$$P_2 \circ P_1^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 3 & 4 & 10 & 6 & 7 & 2 & 9 & 1 & 5 & 8 \end{pmatrix}$$

Now let us find the cycles of $P_2 \circ P_1^{-1}$:

- Starting from 1 we move to 3, then to 10, then to 8, then back to 1: $(1, 3, 10, 8)$
- Starting from 6 we move to 2, then to 4, then back to 6: $(6, 2, 4)$
- Starting from 5 we move to 7, then to 9, then back to 5: $(5, 7, 9)$

Genetic Algorithms

- Let us mark in P_1 and P_2 the numbers of the cycles with different colors: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

- Let us mark in P_1 and P_2 the numbers of the cycles with different colors: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

- Let us mark in P_1 and P_2 the numbers of the cycles with different colors: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

- Now let us populate the offspring permutations O_1 and O_2

$$O_1 = (\qquad\qquad\qquad\qquad\qquad\qquad)$$
$$O_2 = (\qquad\qquad\qquad\qquad\qquad\qquad)$$

- Let us mark in P_1 and P_2 the numbers of the cycles with different colors: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

- Now let us populate the offspring permutations O_1 and O_2

$$O_1 = (\qquad\qquad\qquad\qquad)$$
$$O_2 = (\qquad\qquad\qquad\qquad)$$

- O_1 will place the elements of $(1, 3, 10, 8)$ as in P_1.
  O_2 will place them as in P_2

Genetic Algorithms

- Let us mark in P_1 and P_2 the numbers of the cycles with different colors: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

- Now let us populate the offspring permutations O_1 and O_2

$$O_1 = (\ 1\ \ \ \ \ \ \ \ \ \ \ \ \ \ 8\ \ 3\ \ \ \ \ \ \ \ \ 10\ )$$
$$O_2 = (\ 3\ \ \ \ \ \ \ \ \ \ \ \ \ \ 1\ \ 10\ \ \ \ \ \ \ \ \ 8\ )$$

- O_1 will place the elements of $(1, 3, 10, 8)$ as in P_1.
  O_2 will place them as in P_2

Genetic Algorithms

- Let us mark in P_1 and P_2 the numbers of the cycles with different colors: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

- Now let us populate the offspring permutations O_1 and O_2

$$O_1 = (\ 1\ \ \ \ \ \ \ \ \ \ \ \ \ \ 8\ \ 3\ \ \ \ \ \ \ \ 10\ )$$
$$O_2 = (\ 3\ \ \ \ \ \ \ \ \ \ \ \ \ \ 1\ \ 10\ \ \ \ \ \ \ \ 8\ )$$

- O_1 will place the elements of $(1, 3, 10, 8)$ as in P_1.
  O_2 will place them as in P_2
- O_1 will place the elements of $(6, 2, 4)$ as in P_2.
  O_2 will place them as in P_1

Genetic Algorithms

- Let us mark in P_1 and P_2 the numbers of the cycles with different colors: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

- Now let us populate the offspring permutations O_1 and O_2

$$O_1 = (\ 1\ \ 2\ \ \ \ \ \ 6\ \ 8\ \ 3\ \ \ \ \ 4\ \ 10\ )$$
$$O_2 = (\ 3\ \ 6\ \ \ \ \ \ 4\ \ 1\ \ 10\ \ \ \ \ 2\ \ 8\ )$$

- O_1 will place the elements of $(1, 3, 10, 8)$ as in P_1.
  O_2 will place them as in P_2
- O_1 will place the elements of $(6, 2, 4)$ as in P_2.
  O_2 will place them as in P_1

Genetic Algorithms

- Let us mark in P_1 and P_2 the numbers of the cycles with different colors: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

- Now let us populate the offspring permutations O_1 and O_2

$$O_1 = (\ 1\ \ 2\ \ \ \ \ \ 6\ \ 8\ \ 3\ \ \ \ \ 4\ \ 10\ )$$
$$O_2 = (\ 3\ \ 6\ \ \ \ \ \ 4\ \ 1\ \ 10\ \ \ \ \ 2\ \ 8\ )$$

- O_1 will place the elements of $(1, 3, 10, 8)$ as in P_1.
  O_2 will place them as in P_2
- O_1 will place the elements of $(6, 2, 4)$ as in P_2.
  O_2 will place them as in P_1
- O_1 will place the elements of $(5, 7, 9)$ as in P_1.
  O_2 will place them as in P_2.

  If there were more cycles, we would keep alternating the order.

Genetic Algorithms

- Let us mark in P_1 and P_2 the numbers of the cycles with different colors: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.

$$P_1 = (\ 1\ \ 6\ \ 5\ \ 9\ \ 4\ \ 8\ \ 3\ \ 7\ \ 2\ \ 10\ )$$
$$P_2 = (\ 3\ \ 2\ \ 7\ \ 5\ \ 6\ \ 1\ \ 10\ \ 9\ \ 4\ \ 8\ )$$

- Now let us populate the offspring permutations O_1 and O_2

$$O_1 = (\ 1\ \ 2\ \ 5\ \ 9\ \ 6\ \ 8\ \ 3\ \ 7\ \ 4\ \ 10\ )$$
$$O_2 = (\ 3\ \ 6\ \ 7\ \ 5\ \ 4\ \ 1\ \ 10\ \ 9\ \ 2\ \ 8\ )$$

- O_1 will place the elements of $(1, 3, 10, 8)$ as in P_1.
  O_2 will place them as in P_2
- O_1 will place the elements of $(6, 2, 4)$ as in P_2.
  O_2 will place them as in P_1
- O_1 will place the elements of $(5, 7, 9)$ as in P_1.
  O_2 will place them as in P_2.

  If there were more cycles, we would keep alternating the order.

Mutation operators

- Mutation introduces randomness to escape from local optima
- Typically, mutation is applied with less than 1% probability
- Examples:
  - For bit strings: randomly flip the value of one bit
  - For permutations of integers:
    - swap two numbers
    - remove one number and insert it somewhere else
    - the same, but with a sequence of numbers

Mutation operators

- Mutation introduces randomness to escape from local optima
- Typically, mutation is applied with less than 1% probability
- Examples:
  - For bit strings: randomly flip the value of one bit
  - For permutations of integers:
    - swap two numbers
    - remove one number and insert it somewhere else
    - the same, but with a sequence of numbers

Selection of individuals for the next generation

- We use a fitness function to evaluate the quality of individuals (usually, the objective function)
- The best individuals are kept
- We may distinguish whether the individual is new or comes from the previous generation
- The population may have a fixed-size or change along the execution