Chapter 5. Metaheuristics

Algorithmics and Programming III

FIB

Albert Oliveras Enric Rodríguez

Q1 2019–2020

Version November 18, 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 that guide the search to efficiently 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 that guide the search to efficiently 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 function $f : D_1 \times D_2 \times \cdots D_n \rightarrow \mathbb{R}^+$,

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

Our goal is to find a (globally) optimal solution $s^* \in \underset{x \in S}{\text{argmin}}\ f(x)$.

EXAMPLES:

- 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 function $f : D_1 \times D_2 \times \cdots D_n \rightarrow \mathbb{R}^+$,

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

Our goal is to find a (globally) optimal solution $s^* \in \underset{x \in S}{\text{argmin}}\ f(x)$.

EXAMPLES:

- 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 about the quality of the solutions found by 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 about the quality of the solutions found by 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

What determines the areas of the search space to explore is the concept of neighborhood:

- 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 wrt 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

What determines the areas of the search space to explore is the concept of neighborhood:

- 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 wrt 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, we could define $\mathcal{N}(s)$ as the set of solutions that:

- can be obtained from s by replacing one item by another one, and
- whose sum does not exceed W

For example:

- $\{7, 4\}$ belongs to $\mathcal{N}(\{7, 5\})$
- $\{7, 4\}$ 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, we could define $\mathcal{N}(s)$ as the set of solutions that:

- can be obtained from s by replacing one item by another one, and
- whose sum does not exceed W

For example:

- $\{7, 4\}$ belongs to $\mathcal{N}(\{7, 5\})$
- $\{7, 4\}$ 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, we could define $\mathcal{N}(s)$ as the set of solutions that:

- can be obtained from s by replacing one item by another one, and
- whose sum does not exceed W

For example:

- $\{7, 4\}$ belongs to $\mathcal{N}(\{7, 5\})$
- $\{7, 4\}$ 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.
We will present them in their general setting,
without biasing them towards one concrete COP.

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.

We will present them in their general setting,
without biasing them towards one concrete COP.

Chapter 5. Metaheuristics

Basic Local Search: Iterative Improvement

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

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

In Pseudo-code:

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

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

Basic Local Search: Iterative Improvement

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 previous Knapsack example)
  - 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*$(\mathcal{N}(s))$?
  - **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)

Iterative improvement disallows to temporarily worsen the objective function, and hence it cannot escape from bad local optima.

Basic Local Search: Iterative Improvement

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 previous Knapsack example)
  - 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*$(\mathcal{N}(s))$?
  - **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)

Iterative improvement disallows to temporarily worsen the objective function, and hence it cannot escape from bad local optima.

Basic Local Search: Iterative Improvement

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 previous Knapsack example)
  - 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*$(\mathcal{N}(s))$?
  - **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)

Iterative improvement disallows to temporarily worsen the objective function, and hence it cannot escape from bad local optima.

Chapter 5. Metaheuristics

Simulated Annealing

- Fundamental idea: allow moves resulting in solutions of worse quality than the current solution in order to escape from local minima.
- 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 is the probability of accepting a worsening move defined?
  - 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 closer $f(s')$ is to $f(s)$, the higher the probability of accepting the move
  - Given fixed s, s', since $f(s') > f(s)$, the higher the temperature T the higher the probability of accepting a worsening move
- How do we update the temperature?
  - The temperature at iteration k is defined as a function $Q(T_k, k)$ of the current temperature T_k and the iteration number
  - 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 is the probability of accepting a worsening move defined?
  - 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 closer $f(s')$ is to $f(s)$, the higher the probability of accepting the move
  - Given fixed s, s', since $f(s') > f(s)$, the higher the temperature T the higher the probability of accepting a worsening move
- How do we update the temperature?
  - The temperature at iteration k is defined as a function $Q(T_k, k)$ of the current temperature T_k and the iteration number
  - 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 minima 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.
- About the tabu-list length:
  - It controls the memory consumed by the process
  - With small lengths the search concentrates on small areas of the search space
  - With large lengths the search explores larger regions
  - The length can be varied dynamically during the search

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 minima 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.
- About the tabu-list length:
  - It controls the memory consumed by the process
  - With small lengths the search concentrates on small areas of the search space
  - With large lengths the search explores larger regions
  - The length can be varied dynamically during the search

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 minima 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.
- About the tabu-list length:
  - It controls the memory consumed by the process
  - With small lengths the search concentrates on small areas of the search space
  - With large lengths the search explores larger regions
  - The length can be varied dynamically during the search

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 assigns the tabu status to probably more than one solution. It is hence possible that unvisited solutions of good quality are excluded from the allowed set.
  - To overcome this problem, aspiration criteria are defined. They allow to move to a solution even if forbidden by the tabu conditions.

    Example: always allow moving to 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 some aspiration condition}
    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:
  - Phase 1: solution construction
  - Phase 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 set of items).
    The solution is constructed by adding one new element at a time
  - Next element is chosen uniformly at random from a candidate list as follows
  - Each element is assigned a score that estimates the benefit if 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:
  - Phase 1: solution construction
  - Phase 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 set of items).
    The solution is constructed by adding one new element at a time
  - Next element is chosen uniformly at random from a candidate list as follows
  - Each element is assigned a score that estimates the benefit if 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

- 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
    (Iterative Improvement, 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
    (Iterative Improvement, 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
    (Iterative Improvement, 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 minima
- One drawback of GRASP is that it does not use the history of the search
- However, due to is simplicity, it is generally very fast and produces quite good solutions in a very short amount of time

Chapter 5. Metaheuristics

Variable Neighborhood Search

- Key idea of VNS: dynamically change neighborhood structures
- 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.
- Starting with an initial solution s, the algorithm repeats three 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

- Key idea of VNS: dynamically change neighborhood structures
- 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.
- Starting with an initial solution s, the algorithm repeats three 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
- Ideally, s' should belong to the basin of attraction of another local minimum
- But it should not be too different from s. Otherwise, we are doing a simple random multi-start
- Intuitively, choosing s' in some neighborhood of s maintains some good features of s
- Changing neighborhood corresponds to a diversification of the search
- A solution that is locally optimal wrt one neighborhood is probably not locally optimal in another
- Different neighborhood structures have different landscape topologies, and hence search behave differently on them.
  This property is exploited by Variable Neighborhood Descent (VND)

Variable Neighborhood Search

- Shaking perturbs s to provide a good starting point for local search
- Ideally, s' should belong to the basin of attraction of another local minimum
- But it should not be too different from s. Otherwise, we are doing a simple random multi-start
- Intuitively, choosing s' in some neighborhood of s maintains some good features of s
- Changing neighborhood corresponds to a diversification of the search
- A solution that is locally optimal wrt one neighborhood is probably not locally optimal in another
- Different neighborhood structures have different landscape topologies, and hence search behave differently on them.
  This property is exploited by Variable Neighborhood Descent (VND)

Variable Neighborhood Search

Variable Neighborhood Descent:

```
s := generateInitialSolution()
while termination conditions not met do
    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
endwhile
```

- The choice of the neighborhoods is the critical point of VNS and VND.
- They should exploit different properties of the search space, i.e, should provide different abstractions of it.

Chapter 5. Metaheuristics

Guided Local Search

- In order to diversify the search and escape from local minima, GLS dynamically changes the objective function so as to make them less desirable.
- Let us fix a set of m solution features, which can be any kind of properties or characteristics that discriminate between solutions (e.g. in knapsack, a solution feature can be if a fixed object is chosen).
- We will use m indicator functions $I_i(s)$ that return value 1 iff the feature i is present in solution s.
- Each solution feature has a fixed associated cost c_i (e.g. in knapsack $-$value, the negated of the value).
- After every solution found, the new objective function f' is redefined by

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

where

  - λ is called the regularization parameter, and
  - p_i are the penalty parameters (which change during the execution)

Guided Local Search

- In Pseudo-code:

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

- Every time a local optimum is found, GLS tries to penalize the features of this solution with highest cost
- Given 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 highest cost c_i, but the presence of $1 + p_i$ dividing prevents a feature from being penalized again and again.

Chapter 5. Metaheuristics

Evolutionary Computation (Genetic Algorithms)

- At every iteration of the algorithm, we will deal with a set of solutions, to which we will refer as individuals.
- This population of individuals evolves at each iteration by the following rules:
  - New individuals are created by applying recombination, that combines two or more individuals (so-called parents) to produce one or more new individuals (children or offspring).
  - Mutation allows the appearance of new traits in the offspring to promote diversity.
  - Selection of which individuals will be maintained into the next generation is done by evaluating their fitness, i.e. how good they are.
- As a termination condition, a predefined number of generations may be used

Evolutionary Computation (Genetic Algorithms)

- In Pseudo-code:

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

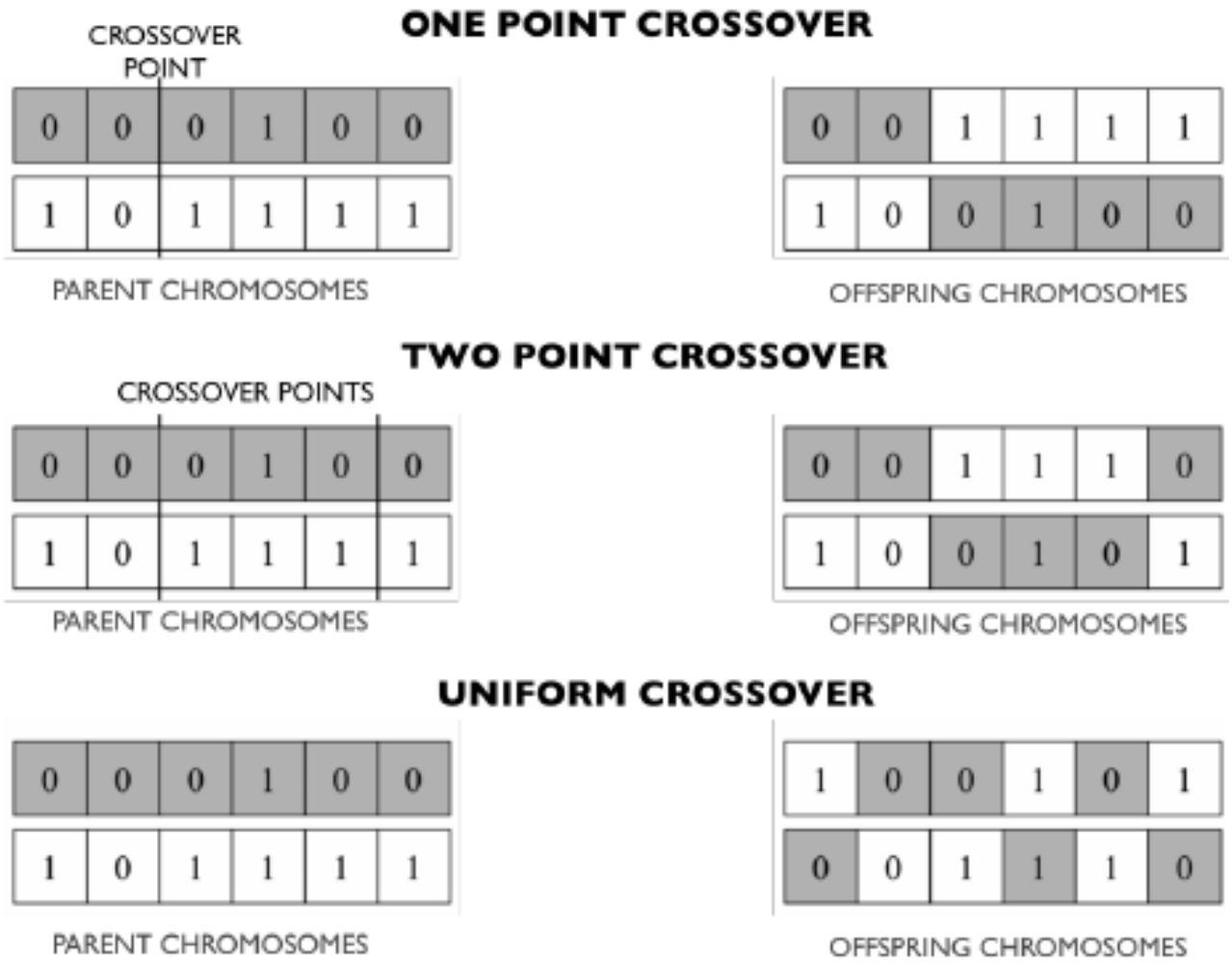
- A concrete instance is the well-known case of Genetic Algorithm

Genetic Algorithms:

- The basic GA is very generic. It will vary depending on: representation of individuals, selection strategy, recombination procedure and mutator operators.
- Representation of individuals: individuals are usually represented using bit-strings of fixed length, although permutations of integer numbers are also frequent
- Selection strategy for parents: after evaluating the fitness of each individual f_i, several selection strategies exist:
  - Roulette-wheel selection: the probability to choose one individual is $f_i / \sum_{j=1}^{n} f_j$
  - Ranking selection: sort all n individuals wrt their fitness. The i-th individual in order gets rank i. The probability to select an individual x is $2 \cdot rank(x)/(n \cdot (n-1))$.
  - Tournament selection: choose k elements at random and pick the one with the highest fitness. Repeat until selecting enough individuals to mate.

Evolutionary Computation (Genetic Algorithms)

- Recombination: it is called crossover in this context.
- When individuals are encoded as bit-strings, we can use, among others:



- Recombination: it is called crossover in this context.
- When individuals are encoded permutations of integers, we can use, among others:
  - Partially-mapped crossover:

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

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

Evolutionary Computation (Genetic Algorithms)

- Recombination: it is called crossover in this context.
- When individuals are encoded permutations of integers, we can use, among others:
  - 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

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

We can fill the leftmost and the rightmost parts O_1 as P_1.
If there is a conflict, use mapping σ.

Similarly with O_2 using mapping τ.

Evolutionary Computation (Genetic Algorithms)

- Recombination: it is called crossover in this context.
- When individuals are encoded permutations of integers, we can use, among others:
  - 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
    $$\sigma(1) = 4, \tau(4) = 1$$
    $$\sigma(8) = 5, \tau(5) = 8$$
    $$\sigma(7) = 6, \tau(6) = 7$$

    We can fill the leftmost and the rightmost parts O_1 as P_1.
    If there is a conflict, use mapping σ.

    Similarly with O_2 using mapping τ.

- Order crossover:

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

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

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

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

- Order crossover:

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

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

To generate O_2, take its order 9 3 4 5 2 1 8 7 6 and remove $\{4, 5, 6, 7\}$. We obtain the order 9 3 2 1 8, that we use to fill it

- Order crossover:

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

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

To generate O_2, take its order 9 3 4 5 2 1 8 7 6 and remove $\{4, 5, 6, 7\}$. We obtain the order 9 3 2 1 8, that we use to fill it

- Order crossover:

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

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

To generate O_1, take its order 8 9 1 2 3 4 5 6 7 and remove $\{1, 8, 7, 6\}$.
We obtain the order 9 2 3 4 5, that we use to fill it

- Order crossover:

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

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

To generate O_1, take its order 8 9 1 2 3 4 5 6 7 and remove $\{1, 8, 7, 6\}$.
We obtain the order 9 2 3 4 5, that we use to fill it

- 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 identify a collection of cycles: those of $P_2 \circ P_1^{-1}$

- The first cycle starts with the first element from P_1, which is 1.
- The position of 1 in P_2 is occupied by 3, which is added to the cycle.
- Elem 3 is in the 7th position of P_1. Add the 7th element of P_2: 10
- Elem 10 is in the last position of P_1. Add the last element of P_2: 8
- Elem 8 is in the 6th position of P_1. Add the 6th element of P_2: 1. This closes the cycle: $(1, 3, 10, 8)$
- We now consider the first elem of P_1 not in the previous cycle: 6.
- The position of 6 in P_2 is occupied by 2, which is added to the cycle.
- Elem 2 is in the 9th position of P_1. Add the 9th element of P_2: 4
- Elem 4 is in the 5th position of P_1. Add the 5th element of P_2: 6. This closes the cycle: $(6, 2, 4)$
- Similarly, the last cycle is $(5, 7, 9)$.

- 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 identify a collection of cycles: those of $P_2 \circ P_1^{-1}$

- The first cycle starts with the first element from P_1, which is 1.
- The position of 1 in P_2 is occupied by 3, which is added to the cycle.
- Elem 3 is in the 7th position of P_1. Add the 7th element of P_2: 10
- Elem 10 is in the last position of P_1. Add the last element of P_2: 8
- Elem 8 is in the 6th position of P_1. Add the 6th element of P_2: 1. This closes the cycle: $(1, 3, 10, 8)$
- We now consider the first elem of P_1 not in the previous cycle: 6.
- The position of 6 in P_2 is occupied by 2, which is added to the cycle.
- Elem 2 is in the 9th position of P_1. Add the 9th element of P_2: 4
- Elem 4 is in the 5th position of P_1. Add the 5th element of P_2: 6. This closes the cycle: $(6, 2, 4)$
- Similarly, the last cycle is $(5, 7, 9)$.

- 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 identify a collection of cycles: those of $P_2 \circ P_1^{-1}$

- The first cycle starts with the first element from P_1, which is 1.
- The position of 1 in P_2 is occupied by 3, which is added to the cycle.
- Elem 3 is in the 7th position of P_1. Add the 7th element of P_2: 10
- Elem 10 is in the last position of P_1. Add the last element of P_2: 8
- Elem 8 is in the 6th position of P_1. Add the 6th element of P_2: 1. This closes the cycle: $(1, 3, 10, 8)$
- We now consider the first elem of P_1 not in the previous cycle: 6.
- The position of 6 in P_2 is occupied by 2, which is added to the cycle.
- Elem 2 is in the 9th position of P_1. Add the 9th element of P_2: 4
- Elem 4 is in the 5th position of P_1. Add the 5th element of P_2: 6. This closes the cycle: $(6, 2, 4)$
- Similarly, the last cycle is $(5, 7, 9)$.

- 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 identify a collection of cycles: those of $P_2 \circ P_1^{-1}$

- The first cycle starts with the first element from P_1, which is 1.
- The position of 1 in P_2 is occupied by 3, which is added to the cycle.
- Elem 3 is in the 7th position of P_1. Add the 7th element of P_2: 10
- Elem 10 is in the last position of P_1. Add the last element of P_2: 8
- Elem 8 is in the 6th position of P_1. Add the 6th element of P_2: 1. This closes the cycle: $(1, 3, 10, 8)$
- We now consider the first elem of P_1 not in the previous cycle: 6.
- The position of 6 in P_2 is occupied by 2, which is added to the cycle.
- Elem 2 is in the 9th position of P_1. Add the 9th element of P_2: 4
- Elem 4 is in the 5th position of P_1. Add the 5th element of P_2: 6. This closes the cycle: $(6, 2, 4)$
- Similarly, the last cycle is $(5, 7, 9)$.

- 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 identify a collection of cycles: those of $P_2 \circ P_1^{-1}$

- The first cycle starts with the first element from P_1, which is 1.
- The position of 1 in P_2 is occupied by 3, which is added to the cycle.
- Elem 3 is in the 7th position of P_1. Add the 7th element of P_2: 10
- Elem 10 is in the last position of P_1. Add the last element of P_2: 8
- Elem 8 is in the 6th position of P_1. Add the 6th element of P_2: 1. This closes the cycle: $(1, 3, 10, 8)$
- We now consider the first elem of P_1 not in the previous cycle: 6.
- The position of 6 in P_2 is occupied by 2, which is added to the cycle.
- Elem 2 is in the 9th position of P_1. Add the 9th element of P_2: 4
- Elem 4 is in the 5th position of P_1. Add the 5th element of P_2: 6. This closes the cycle: $(6, 2, 4)$
- Similarly, the last cycle is $(5, 7, 9)$.

- 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 identify a collection of cycles: those of $P_2 \circ P_1^{-1}$

- The first cycle starts with the first element from P_1, which is 1.
- The position of 1 in P_2 is occupied by 3, which is added to the cycle.
- Elem 3 is in the 7th position of P_1. Add the 7th element of P_2: 10
- Elem 10 is in the last position of P_1. Add the last element of P_2: 8
- Elem 8 is in the 6th position of P_1. Add the 6th element of P_2: 1. This closes the cycle: $(1, 3, 10, 8)$
- We now consider the first elem of P_1 not in the previous cycle: 6.
- The position of 6 in P_2 is occupied by 2, which is added to the cycle.
- Elem 2 is in the 9th position of P_1. Add the 9th element of P_2: 4
- Elem 4 is in the 5th position of P_1. Add the 5th element of P_2: 6. This closes the cycle: $(6, 2, 4)$
- Similarly, the last cycle is $(5, 7, 9)$.

- 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 identify a collection of cycles: those of $P_2 \circ P_1^{-1}$

- The first cycle starts with the first element from P_1, which is 1.
- The position of 1 in P_2 is occupied by 3, which is added to the cycle.
- Elem 3 is in the 7th position of P_1. Add the 7th element of P_2: 10
- Elem 10 is in the last position of P_1. Add the last element of P_2: 8
- Elem 8 is in the 6th position of P_1. Add the 6th element of P_2: 1. This closes the cycle: $(1, 3, 10, 8)$
- We now consider the first elem of P_1 not in the previous cycle: 6.
- The position of 6 in P_2 is occupied by 2, which is added to the cycle.
- Elem 2 is in the 9th position of P_1. Add the 9th element of P_2: 4
- Elem 4 is in the 5th position of P_1. Add the 5th element of P_2: 6. This closes the cycle: $(6, 2, 4)$
- Similarly, the last cycle is $(5, 7, 9)$.

- 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 identify a collection of cycles: those of $P_2 \circ P_1^{-1}$

- The first cycle starts with the first element from P_1, which is 1.
- The position of 1 in P_2 is occupied by 3, which is added to the cycle.
- Elem 3 is in the 7th position of P_1. Add the 7th element of P_2: 10
- Elem 10 is in the last position of P_1. Add the last element of P_2: 8
- Elem 8 is in the 6th position of P_1. Add the 6th element of P_2: 1. This closes the cycle: $(1, 3, 10, 8)$
- We now consider the first elem of P_1 not in the previous cycle: 6.
- The position of 6 in P_2 is occupied by 2, which is added to the cycle.
- Elem 2 is in the 9th position of P_1. Add the 9th element of P_2: 4
- Elem 4 is in the 5th position of P_1. Add the 5th element of P_2: 6. This closes the cycle: $(6, 2, 4)$
- Similarly, the last cycle is $(5, 7, 9)$.

- 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 identify a collection of cycles: those of $P_2 \circ P_1^{-1}$

- The first cycle starts with the first element from P_1, which is 1.
- The position of 1 in P_2 is occupied by 3, which is added to the cycle.
- Elem 3 is in the 7th position of P_1. Add the 7th element of P_2: 10
- Elem 10 is in the last position of P_1. Add the last element of P_2: 8
- Elem 8 is in the 6th position of P_1. Add the 6th element of P_2: 1. This closes the cycle: $(1, 3, 10, 8)$

- We now consider the first elem of P_1 not in the previous cycle: 6.
- The position of 6 in P_2 is occupied by 2, which is added to the cycle.
- Elem 2 is in the 9th position of P_1. Add the 9th element of P_2: 4
- Elem 4 is in the 5th position of P_1. Add the 5th element of P_2: 6. This closes the cycle: $(6, 2, 4)$

- Similarly, the last cycle is $(5, 7, 9)$.

- 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 identify a collection of cycles: those of $P_2 \circ P_1^{-1}$

- The first cycle starts with the first element from P_1, which is 1.
- The position of 1 in P_2 is occupied by 3, which is added to the cycle.
- Elem 3 is in the 7th position of P_1. Add the 7th element of P_2: 10
- Elem 10 is in the last position of P_1. Add the last element of P_2: 8
- Elem 8 is in the 6th position of P_1. Add the 6th element of P_2: 1. This closes the cycle: $(1, 3, 10, 8)$
- We now consider the first elem of P_1 not in the previous cycle: 6.
- The position of 6 in P_2 is occupied by 2, which is added to the cycle.
- Elem 2 is in the 9th position of P_1. Add the 9th element of P_2: 4
- Elem 4 is in the 5th position of P_1. Add the 5th element of P_2: 6. This closes the cycle: $(6, 2, 4)$
- Similarly, the last cycle is $(5, 7, 9)$.

- Cycle crossover:

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

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

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

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

- Let us remember our 3 cycles: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.
- O_1 will place the elements of $(1, 3, 10, 8)$ in the positions they have in P_1. O_2 will place them in the positions they have in P_2

- Cycle crossover:

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

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

- Let us remember our 3 cycles: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.
- O_1 will place the elements of $(1, 3, 10, 8)$ in the positions they have in P_1. O_2 will place them in the positions they have in P_2

- Cycle crossover:

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

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

- Let us remember our 3 cycles: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.
- O_1 will place the elements of $(1, 3, 10, 8)$ in the positions they have in P_1.
  O_2 will place them in the positions they have in P_2
- O_1 will place the elements of $(6, 2, 4)$ in the positions they appear in P_2.
  O_2 will place them in the positions they have in P_1

- Cycle crossover:

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

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

- Let us remember our 3 cycles: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.
- O_1 will place the elements of $(1, 3, 10, 8)$ in the positions they have in P_1.
  O_2 will place them in the positions they have in P_2
- O_1 will place the elements of $(6, 2, 4)$ in the positions they appear in P_2.
  O_2 will place them in the positions they have in P_1

- Cycle crossover:

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

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

- Let us remember our 3 cycles: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.
- O_1 will place the elements of $(1, 3, 10, 8)$ in the positions they have in P_1.
  O_2 will place them in the positions they have in P_2
- O_1 will place the elements of $(6, 2, 4)$ in the positions they appear in P_2.
  O_2 will place them in the positions they have in P_1
- O_1 will place the elements of $(5, 7, 9)$ in the positions they appear in P_1.
  O_2 will place them in the positions they have in P_2.

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

- Cycle crossover:

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

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

- Let us remember our 3 cycles: $(1, 3, 10, 8)$, $(6, 2, 4)$, $(5, 7, 9)$.
- O_1 will place the elements of $(1, 3, 10, 8)$ in the positions they have in P_1.
  O_2 will place them in the positions they have in P_2
- O_1 will place the elements of $(6, 2, 4)$ in the positions they appear in P_2.
  O_2 will place them in the positions they have in P_1
- O_1 will place the elements of $(5, 7, 9)$ in the positions they appear in P_1.
  O_2 will place them in the positions they have in P_2.

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

Evolutionary Computation (Genetic Algorithms)

- Mutator operators:
  - Introduces some randomness to prevent the optimization from getting trapped in 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 (or a sequence of them) and insert in somewhere else, etc.
  - Selection of new generation individuals:
    - We use again the fitness function to evaluate the quality of individuals
    - Usually, one keeps the best individuals from the previous generation
    - Among the offsprings, a subset of them is selected according to their fitness
    - One can decide to keep a fixed-size population or vary its size

- Mutator operators:
  - Introduces some randomness to prevent the optimization from getting trapped in 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 (or a sequence of them) and insert in somewhere else, etc.
- Selection of new generation individuals:
  - We use again the fitness function to evaluate the quality of individuals
  - Usually, one keeps the best individuals from the previous generation
  - Among the offsprings, a subset of them is selected according to their fitness
  - One can decide to keep a fixed-size population or vary its size

- Mutator operators:
  - Introduces some randomness to prevent the optimization from getting trapped in 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 (or a sequence of them) and insert in somewhere else, etc.
  - Selection of new generation individuals:
    - We use again the fitness function to evaluate the quality of individuals
    - Usually, one keeps the best individuals from the previous generation
    - Among the offsprings, a subset of them is selected according to their fitness
    - One can decide to keep a fixed-size population or vary its size

Chapter 5. Metaheuristics

Swarm Intelligence (Ant Colony Optimization)

- Swarm Intelligence is a paradigm for solving optimization problems inspired by the collective behavior of social insect colonies or animal societies.
- SI systems are made up of a population of simple agents interacting locally with one another and the environment
- These entities with very limited individual capability can jointly perform complex tasks necessary for their survival
- Although there is normally no centralized control structure, local interactions between agents often lead to global and self-organized behavior.
- Examples:
  - Ant Colony Optimization
  - Particle Swarm Intelligence
  - Bacterial Foraging Optimization
  - Bee Colony Optimization
  - ...

Swarm Intelligence (Ant Colony Optimization)

- We will focus on Ant Colony Optimization (ACO)
- ACO takes inspiration from the foraging behavior of real ants:
  - When searching for food, ants initially explore the area surrounding their nest with randomized walk
  - While walking, ants deposit a chemical pheromone trail on the ground to mark some favorable path that should guide other ants to the food source
  - After some time, the shortest path to the food source contains a higher concentration of pheromone and, therefore, attracts more ants

Swarm Intelligence (Ant Colony Optimization)

- ACO creates a set of m ants, which are in charge of incrementally and stochastically building solutions starting from the empty solution $s_p = \emptyset$. Intermediate solutions are referred to as solution states.
- At each step, each ant moves from a state x to state y, corresponding to adding a new element to the solution.
- How do we decide which transition is chosen?
  - The next state can only be chosen from the set of valid transitions $\mathcal{V}(x) = \{y \mid y \text{ if feasible and } x \to y \text{ is a transition}\}$.
  - Each valid transition has associated a dynamic value $\tau_{x,y}$ (pheromone) and fixed value $\eta_{i,j}$ (heuristic information about the problem)
  - The probability of choosing $x \to y$ is:

$$Prob(x \to y) = \frac{\tau_{x,y}^{\alpha} \cdot \eta_{x,y}^{\beta}}{\sum_{z \in \mathcal{V}(x)} \tau_{x,z}^{\alpha} \cdot \eta_{x,z}^{\beta}}$$

    with constants $\alpha, \beta \geq 0$

- How are the pheromone values $\tau_{x,y}$ updated to bias the ants towards good solutions?
  - We want to achieve two goals:
    - Move ants towards high-quality states. Pheromone values for good transitions should be increased (intensification)
    - Avoid ants from always going to the same state. Pheromone values should be evaporated (diversification)
  - Initially we can assume that all $\tau_{x,y}$ are equal
  - Once every ant has constructed a solution, we select of set S_{hq} of high-quality solutions. We update the pheromones $\tau_{x,y}$ as follows:

$$\tau_{x,y} := (1-\rho)\tau_{x,y} \quad + \sum_{s \in S_{hq} | y-x \subseteq s} f(s)$$

where $\rho \in (0,1]$ is the evaporation rate and $f(s)$ is the fitness function ($f : S \to \mathbb{R}^+$ such that $F(s) < F(s') \implies f(s) \geq f(s')$, where F is the function to minimize)

Swarm Intelligence (Ant Colony Optimization)

The overall ACO algorithm is:

```
Initialize pheromone values
while termination conditions not met do
    Construct Ants Solutions
    Update Pheromones
    Daemon Actions
endwhile
```

Daemon Actions refer to any centralized operation that a single ant cannot perform. The most used one is the application of local search to the constructed solutions.