CS616 - Optimization Algorithms

Chapter 2 - Main Common Concepts for Metaheuristics

Dr. Mahdi Khemakhem

Department of Computer Science
College of Computer Engineering and Science
Prince Sattam bin Abdulaziz University

AY - 2025/2026

Outline

1. General Schema of Metaheuristics
1.1 Macroscopic View
1.2 Initial Solution(s)
1.3 Examples of Initial Solution(s)
1.4 Key Takeaways

2. Solution Encoding (Representation)
2.1 Definition
2.2 Characteristics
2.3 Encoding Example
2.4 Key Takeaways

3. Objective Function
3.1 Definition
3.2 Types of Objective Function
3.3 Key Takeaways

4. Constraint Handling
4.1 Introduction
4.2 Strategies
4.3 Key Takeaways

5. Parameter Tuning

Outline (Contd.)

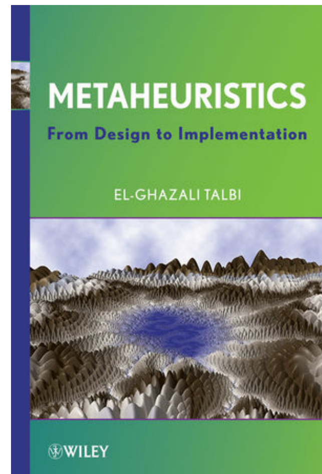
5.1 Definition and Importance
5.2 Tuning Strategies
5.3 Key Takeaways

6. Performance Analysis
6.1 Introduction
6.2 Three-Step Methodology
6.3 Common Pitfalls
6.4 Key Takeaways

7. Common Concepts for Metaheuristics
7.1 General Methodology Overview
7.2 Single-Solution Based Metaheuristics
7.3 Population-Based Metaheuristics
7.4 Key Takeaways

Materials

- **Textbook:** "Metaheuristics: From Design to Implementation" by *El-Ghazali Talbi*
- $\Rightarrow$ **Read Chapter 1 (Sections 1.4, 1.5, 1.6, and 1.7), Chapter 2 (Section 2.1), and Chapter 3 (Section 3.1) for this chapter's material**

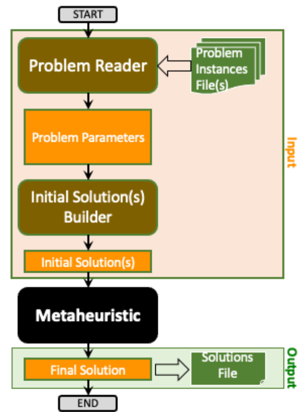


Outline

1. General Schema of Metaheuristics
   - 1.1 Macroscopic View
   - 1.2 Initial Solution(s)
   - 1.3 Examples of Initial Solution(s)
   - 1.4 Key Takeaways
2. Solution Encoding (Representation)
3. Objective Function
4. Constraint Handling
5. Parameter Tuning
6. Performance Analysis

General Schema of Metaheuristics

- For now, we consider that a Metaheuristic is a black box!
- Since a *Metaheuristic* is an algorithm so it should have **Input** and **Output**.
- Two **modules**[a] are responsible to build the **Input**:
  - **Problem reader:** a module that reads the **problem instance** (example) then provides the **problem parameters** stored in some **data structures**.
  - **Initial solution(s) builder:** a module that, from the **problem parameters**, build an **initial solution(s)** (feasible solution(s)) considered as an initial starting point to the *Metaheuristic*.
- Starting from an **initial solution(s)**, the *Metaheuristic* provides a **final solution** (not necessarily optimal) considered as an **Output** to be stored in a **solution file**.



[a] functions, methods, procedures, etc.

Initial Solution(s)

- The majority of **Metaheuristics** start their searching process from a **single** or **multiple** **initial solution**.
- **Initial solution(s)** should be built from the problem parameters and **respecting all problem constraints**.
- Generally, the **quality/qualities** (**objective function value(s)**) of **initial solution(s)** are **relatively ignored**.
- A good Metaheuristic should be **insensitive to the quality of the initial solution** used as starting point.
- Many strategies used in the literature to build **initial solution(s)** for Metaheuristics.
  - The well known named **"Greedy Algorithms"**

Greedy Algorithms (1/2)

In *greedy* or *constructive algorithms*[1], **we start from scratch** (**empty solution**) and construct a solution by assigning values to one decision variable at a time, until a **complete solution** is generated.

- Given an optimization problem, where:
  - a **solution** can be defined by the **presence/absence** of a **finite set of elements** $E = \{e_1, e_2, \ldots, e_n\}$,
  - the **objective function** may be defined as $f : 2^E \rightarrow \mathbb{R}$,
  - the **search space** is defined as $F \subset 2^E$.
- A *partial solution* s may be seen as a subset $\{e_1, e_2, \ldots, e_k\}$ of elements e_i from the set of all elements E. Initially s is empty.
- At each step, a *local heuristic* is used to select the new element to be included in the set s.
- Once an element e_i is selected to be part of the solution s, it is never replaced by another element.
- *There is no backtracking of the already taken decisions.*
- Typically, *greedy heuristics* are **deterministic algorithms**.

[1] Also referred to as successive augmentation algorithms.

Greedy Algorithms (2/2)

Algorithm 1 shows the template of a greedy algorithm.

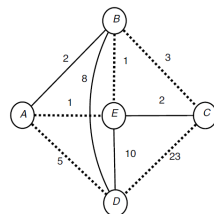
Algorithm 1: Template of a greedy algorithm

Input: Problem Parameters, $s = \{\}$; /* Initial solution (null) */
Output: Initial solution s ; /* Feasible solution (full) */

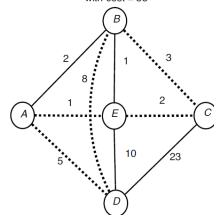
1 **repeat**
2 $e_i = \text{Local-Heuristic}(E - \{e/e \in s\})$; /* next element selected from E minus already selected elements */
3 **if** $s \cup e_i \in F$ **then**
4 $s = s \cup e_i$;
5 **end**
6 **until** Complete solution found;

Greedy Algorithm for the Traveling Salesman Problem

- In the **Traveling Salesman Problem (TSP)**, the set E is defined by the **set of edges**.
- The set F of **feasible solutions** is defined by the subsets of 2^E that forms all possible **Hamiltonian cycles** in the graph.
- A **solution** can be considered as a set of edges that form a **Hamiltonian cycle**.
- A *heuristic* that can be used to select the next edge may be based on the distance. **One possible greedy heuristic** is to **select the nearest neighbor**.
- Figure at the right illustrates the application of the *nearest-neighbor* **greedy heuristic** on the graph beginning from the node A.
- Figure at the bottom illustrates the **optimal solution** that can be provided after applying a *Metaheuristic* on the **initial solution** on the top.



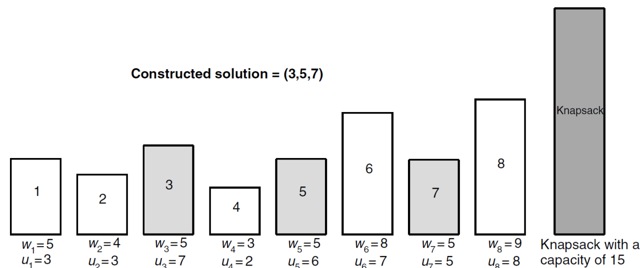
Greedy final solution : A – E – B – C – D – A
with cost = 33



Better solution : A – E – C – B – D – A
with cost = 19

Greedy Algorithm for the Knapsack Problem

- In the **Knapsack Problem (KP)**, the set E is defined by the **set of objects to be packed**.
- The set F represents all subsets of E that are **feasible solutions** (*i.e., the total weight of the selected objects does not exceed the capacity of the knapsack*).
- A **greedy algorithm** that can be used to solve the KP consists in choosing the object minimizing the ratio $\frac{w_i}{u_i}$ where w_i (resp. u_i) represents the weight (resp. profit) of the object i.
- The figure below illustrates this **greedy heuristic** for a KP instance.



Key Takeaways

Key Takeaways

- A **Metaheuristic** can be viewed as a **black box** with **inputs** (problem parameters and initial solutions) and **outputs** (final solutions).
- Two essential **modules** prepare the input: the **Problem Reader** (loads problem instance and parameters) and the **Initial Solution Builder** (constructs feasible starting solutions).
- **Initial solutions** should be **feasible** and respect all problem constraints, though their **quality is relatively ignored** - good metaheuristics are **insensitive** to initial solution quality.
- **Greedy (constructive) algorithms** are commonly used to build initial solutions by incrementally adding elements without backtracking until a complete solution is formed.
- In **Greedy algorithms**: solutions grow from scratch using **local heuristics** (e.g., nearest neighbor for TSP, weight/profit ratio for Knapsack), decisions are **permanent** (no backtracking), and they are typically **deterministic**.

Outline

1. General Schema of Metaheuristics

2. Solution Encoding (Representation)
   - 2.1 Definition
   - 2.2 Characteristics
   - 2.3 Encoding Example
   - 2.4 Key Takeaways

3. Objective Function

4. Constraint Handling

5. Parameter Tuning

6. Performance Analysis

Solution Encoding (Representation)

Definition - Solution Encoding

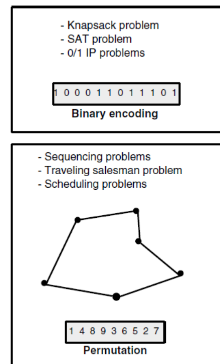
A **solution encoding (or representation)** defines **how a solution** of the optimization problem is **represented as a data structure** that can be manipulated by the algorithm.

- *Solution encoding* is a **fundamental design decision** when developing any **iterative Metaheuristic** with **critical impact** on both **efficiency** (computational speed) and **effectiveness** (solution quality).
- The **choice of encoding** directly determines:
  - How solutions are **represented** and **stored** in memory
  - How **search operators** transform and explore solutions
  - The **searchability** of the solution space
  - The overall **performance** of the Metaheuristic
- A **good encoding** should balance:
  - **Expressiveness:** ability to represent all valid solutions
  - **Simplicity:** ease of manipulation and implementation
  - **Efficiency:** low computational overhead

Solution Encoding - Characteristics (1/2)

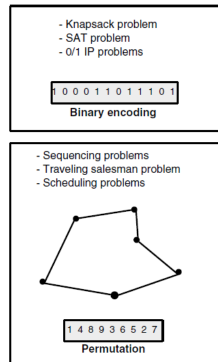
Multiple alternative encodings may exist for a given problem. A **valid representation** must satisfy the following characteristics:

- **Required Characteristics (Mandatory):**
  - **Completeness:** all feasible solutions of the problem can be represented. Every valid solution to the optimization problem has a corresponding representation.
  - **Connexity:** a **search path must exist between any two solutions of the search space**. Any solution, especially the global optimum, can be reached through successive transformations.
  - **Efficiency: encoding must be easy to manipulate by the search operators**. The time and space complexities of operators dealing with the representation must be low.



Solution Encoding - Characteristics (2/2)

- **Desirable Properties (Recommended):**
  - **Suitability:** naturally aligned with problem structure
  - **Relevance:** captures essential solution characteristics
  - **Compatibility:** works seamlessly with chosen search operators
- **Design Considerations:**
  - **Evaluation efficiency:** enables fast objective function computation
  - **Operator effectiveness:** facilitates meaningful transformations
  - **Constraint handling:** properly manages problem constraints



Solution Encoding Example - Permutation Encoding for TSP

- In the **Traveling Salesman Problem (TSP)**, a common **solution encoding** is the *permutation encoding*.
- A **solution** is represented as a **permutation** of the cities to be visited.
- For example, the permutation $(3, 1, 4, 2)$ represents a tour visiting cities in the order: city 3 $\rightarrow$ city 1 $\rightarrow$ city 4 $\rightarrow$ city 2 $\rightarrow$ back to city 3.
- This encoding is **complete** (all tours can be represented), **connected** (any tour can be reached from any other through swaps), and **efficient** (easy to manipulate).

Key Takeaways

Key Takeaways

- **Solution encoding (representation)** defines how solutions are represented as data structures for manipulation by Metaheuristics.
- The choice of encoding is a **fundamental design decision** that critically impacts both the **efficiency** (computational speed) and **effectiveness** (solution quality) of the Metaheuristic.
- A **valid representation** must satisfy **completeness** (all feasible solutions can be represented), **connexity** (any solution can be reached from any other), and **efficiency** (easy manipulation by search operators).
- **Desirable properties** include **suitability** (aligned with problem structure), **relevance** (captures essential characteristics), and **compatibility** (works with chosen operators).
- The **permutation encoding** for TSP is a common example, representing tours as permutations of cities, satisfying the required characteristics for effective search.

Outline

1. General Schema of Metaheuristics

2. Solution Encoding (Representation)

3. Objective Function
3.1 Definition
3.2 Types of Objective Function
3.3 Key Takeaways

4. Constraint Handling

5. Parameter Tuning

6. Performance Analysis

7. Common Concepts for Metaheuristics

Objective Function - Definition

Definition - Objective Function

The **objective function**[a] $f : S \to \mathbb{R}$ formulates the **optimization goal** to achieve. It **associates** with each **solution** $s \in S$ of the **search space** S a **numerical value** that quantifies the **quality** or **fitness** of that solution.

[a] *Also called cost function, evaluation function, fitness function, or utility function.*

- **Role and properties:**
  - Provides an **absolute measure** of solution quality
  - Enables **complete ordering** of all solutions in the search space
  - **Guides the search** toward better (**optimal** or **near-optimal**) solutions
- **Critical importance:**
  - The *objective function* is a **fundamental component** in designing any **Metaheuristic**
  - It **directly influences** the search direction and convergence behavior
  - **Warning:** An **improperly defined objective function** can lead to **unacceptable solutions**, **premature convergence**, or **stagnation** in **local optima**[2]

[2] *We will discuss this more in the next slides.*

Objective Function - Design Considerations

Key design considerations when defining an objective function

- Must **accurately reflect** the problem's optimization goals and objectives
- Should be **computationally efficient** to evaluate (will be called many times)
- Must handle the **scale and range** of solution values appropriately
- May need to **normalize or weight** different components

- **Common challenges:**
  - **Multiple objectives:** balancing conflicting goals (Pareto optimization)
  - **Dynamic objectives:** goals that change over time or with context
  - **Noisy evaluations:** uncertainty in fitness measurements
  - **Expensive evaluations:** computationally costly to calculate
  - **Constraint incorporation:** integrating feasibility requirements
- **Impact on search performance:**
  - A **well-designed objective function** guides the search efficiently toward optimal solutions
  - A **poorly-designed objective function** can mislead the search, causing premature convergence or stagnation

Types of Objective Function (1/2)

Objective functions can be classified according to several criteria:

1. **By Number of Objectives:**
   - **Single-objective:** One criterion to optimize, $f : S \to \mathbb{R}$
     - Examples: minimize total distance, maximize profit
   - **Multi-objective:** Multiple (often conflicting) criteria, $f : S \to \mathbb{R}^k$
     - Examples: minimize cost AND maximize quality, minimize time AND energy
     - Requires *Pareto optimization*[3] or **scalarization approaches** to handle trade-offs
2. **By Problem Domain:**
   - **Continuous:** Solutions in continuous space (e.g., $\mathbb{R}^n$)
   - **Discrete:** Solutions in discrete/combinatorial space (e.g., permutations, binary vectors)
   - **Mixed:** Combination of continuous and discrete variables
3. **By Mathematical Properties:**
   - **Linear vs. Non-linear:** form of the mathematical expression
   - **Separable vs. Non-separable:** independence of variables

[3] *Not covered in this course.*

Types of Objective Function (2/2)

4. **By Nature of Evaluation:**
   - **Deterministic:** Same solution always yields same fitness value
     - $f(s) = f(s)$ for all evaluations
   - **Stochastic/Noisy:** Fitness evaluation includes randomness or uncertainty
     - $f(s) = g(s) + \epsilon$ where ϵ is noise
     - Requires multiple evaluations or robust methods
   - **Dynamic:** Objective function changes over time
     - $f_t(s)$ depends on time t
     - Requires adaptive search strategies
5. **By Computational Cost:**
   - **Cheap to evaluate:** Fast computation (analytical formulas)
   - **Expensive to evaluate:** Requires simulations, experiments, or complex calculations
     - May require surrogate models or reduced evaluation budgets
6. **Special Cases:**
   - **Black-box:** Internal structure unknown, only input-output behavior observable
   - **Constrained:** Includes feasibility constraints (see next section)
   - **Hierarchical:** Objectives organized in priority levels

Key Takeaways

Key Takeaways

- The **objective function** defines the optimization goal by associating each solution with a numerical value quantifying its quality or fitness.
- It plays a **critical role** in guiding the search process, influencing both the **direction** and **convergence** behavior of Metaheuristics.
- Key design considerations include accurately reflecting **optimization goals**, **computational efficiency**, **appropriate scaling**, and **handling multiple objectives or constraints**.
- Objective functions can be classified by **number of objectives** (single vs. multi-objective), **problem domain** (continuous, discrete, mixed), **mathematical properties** (linear, non-linear, separable), **nature of evaluation** (deterministic, stochastic, dynamic), and **computational cost** (cheap vs. expensive).
- A **well-designed objective function** is essential for effective optimization, while a **poorly designed one** can lead to suboptimal solutions or stagnation.

Outline

1. General Schema of Metaheuristics

2. Solution Encoding (Representation)

3. Objective Function

4. Constraint Handling
4.1 Introduction
4.2 Strategies
4.3 Key Takeaways

5. Parameter Tuning

6. Performance Analysis

7. Common Concepts for Metaheuristics

Constraint Handling - Introduction

- **Why is constraint handling important?**
  - Many **continuous** and **discrete** optimization problems include **constraints**
  - Constraints define the **feasible region** of the search space
  - **Dealing with constraints** is a critical challenge in designing efficient **Metaheuristics**
  - **It is not trivial** to handle constraints effectively
- **Types of constraints:**
  - Can be **Linear** or **Nonlinear** constraints under the form of **Equality** or **Inequality**.
  - May involve **hard constraints** (must be satisfied) or **soft constraints** (preferred but not mandatory)

Key concepts

- **Feasible solution:** satisfies all constraints
- **Infeasible solution:** violates at least one constraint
- **Constraint violation degree:** measures how much constraints are violated
- **Global optimum** is often located on the **boundary** between feasible and infeasible regions

Constraint Handling Strategies - Overview

- **Constraint handling strategies mainly act on:**
  - The **representation of solutions**
  - The **objective function**
  - The **search operators**

Four main categories of constraint handling strategies:

1. *Reject Strategies*
   - Discard infeasible solutions
   - Simple but limited
2. *Penalizing Strategies*
   - Add penalty to objective function
   - Most popular approach
3. *Repairing Strategies*
   - Transform infeasible to feasible
   - Uses problem knowledge
4. *Preserving Strategies*
   - Prevent infeasibility
   - Special operators/encoding
   - Search never produces infeasible solutions

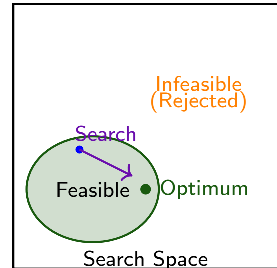
Strategy	Encounter Infeasible?	Complexity	Effectiveness
Reject	Yes (but discarded)	Low	Low
Penalizing	Yes (considered)	Medium	High
Repairing	Yes (transformed)	Medium	Medium-High
Preserving	No	High	High

Reject Strategies

Reject Strategies (Death Penalty)

A simple approach where **only feasible solutions are kept** during the search and **infeasible solutions can be encountered but are immediately discarded**.

- **Principle:**
  - Assign **infinite penalty** (or very bad fitness) to infeasible solutions
  - Only feasible solutions survive
- **When to use:**
  - **Feasible region is large** relative to search space
  - Easy to generate feasible solutions
  - **Constraints are simple to check**
- **Advantages:** **Simple** to implement - No parameter tuning needed - Guarantees feasibility
- **Disadvantages:** **Does not exploit** infeasible solutions - May miss **optimal solutions** on boundaries - Inefficient when feasible region is small - Cannot traverse infeasible regions to reach better feasible areas

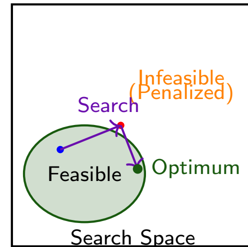


Penalizing Strategies

Penalizing Strategies

Infeasible solutions are considered during search. The **objective function is extended** by a **penalty function** that penalizes constraint violations.

- **Principle:**
  - Modified objective: $f'(s) = f(s) + \lambda \cdot c(s)$
  - $c(s)$: **constraint violation cost**
  - λ: **penalty weight** (tuning parameter)
- **Types of penalties: Static:** λ fixed throughout search - **Dynamic:** λ changes over time - **Adaptive:** λ adjusts based on search progress - **Annealing:** λ increases gradually
- **Advantages: Most popular** and widely used - **Exploits infeasible solutions** for guidance - Allows search paths: **feasible** $\rightarrow$ **infeasible** $\rightarrow$ **better feasible** - Can reach easily optimal solutions on boundaries
- **Disadvantages: Requires tuning** of penalty weight λ - Difficult to balance feasibility and optimality - Too high λ: behaves like reject strategy - Too low λ: search may stay in infeasible region

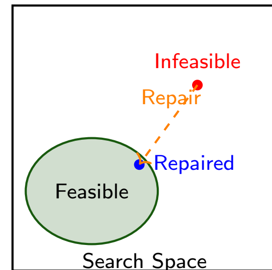


Repairing Strategies

Repairing Strategies

Heuristic algorithms that **transform infeasible solutions into feasible ones** through repair procedures (heuristic operators).

- **Principle:**
  - When search operator generates **infeasible solution**
  - Apply **repair procedure**: $s_{infeasible} \longrightarrow s_{feasible}$
  - Continue search with repaired solution
- **When to use:**
  - Search operators **frequently generate** infeasible solutions
  - **Problem-specific knowledge** available for repair
  - Repair procedure is **computationally cheap**
- **Examples: Knapsack:** Remove items until capacity satisfied - **TSP:** Fix tour violations (missing/duplicate cities) - **Graph coloring:** Reassign colors to eliminate conflicts
- **Advantages: Guarantees feasibility** after repair - **Uses problem knowledge** effectively - Can improve solution quality during repair
- **Disadvantages: Problem-specific:** requires custom design - May be **computationally expensive** - Repair may **bias search** toward certain regions

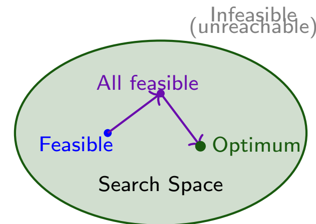


Preserving Strategies

Preserving Strategies

Use **special representation and operators** that **ensure generation of only feasible solutions**, thus **preserving feasibility** throughout the search.

- **Principle:**
  - Design **encoding** that can only represent feasible solutions
  - Design **operators** that preserve feasibility
  - **Incorporate problem knowledge** into representation and operators
- **Techniques:**
  - **Specialized encoding:** only feasible solutions representable
  - **Custom operators:** crossover/mutation maintain feasibility
  - **Decoder functions:** map any representation to feasible solution
- **Example: TSP:** permutation encoding ensures valid tours
- **Advantages: Guarantees feasibility** by design - **No wasted evaluations** on infeasible solutions - No penalty tuning required - Often **more efficient** than other strategies
- **Disadvantages: Requires significant problem knowledge** - **Design effort:** custom encoding and operators - May be **difficult or impossible** for complex constraints - **Less general:** not easily transferable to other problems



Constraint Handling - Comparison and Guidelines

- **Strategy selection depends on:**
  - **Problem characteristics:** size of feasible region, constraint complexity
  - **Available knowledge:** understanding of problem structure
  - **Computational budget:** time available for evaluation and repair
  - **Constraint type:** hard vs. soft, linear vs. nonlinear

Aspect	Reject	Penalizing	Repairing	Preserving
Feasibility guarantee	Yes	No	Yes (after repair)	Yes (by design)
Uses infeasible info	No	Yes	Limited	No
Implementation	Very easy	Easy	Medium	Hard
Problem knowledge	None	None	Medium	High
Parameter tuning	None	Yes (λ)	Possibly	None
Generality	High	High	Medium	Low
Efficiency	Low-Medium	High	Medium	High
Best for	Large feasible region	General problems	Known repair	Well-structured

- **Recommendations:**
  - **Start with penalizing** if no problem-specific knowledge available
  - **Use preserving** when possible (most efficient)
  - **Consider hybrid approaches:** combine multiple strategies
  - **Experiment and compare** different strategies for your specific problem

Key Takeaways

Key Takeaways

- Constraints define the **feasible region** of the search space and are critical in many optimization problems.
- Effective constraint handling is essential for Metaheuristics to find high-quality feasible solutions.
- Four main strategies exist: **Rejecting**, **Penalizing**, **Repairing**, and **Preserving**.
- Each strategy has its own **advantages and disadvantages**, and the choice depends on problem characteristics, available knowledge, and computational budget.
- **Hybrid approaches** combining multiple strategies can often yield better results.
- Experimentation and comparison are key to selecting the most effective constraint handling method for a given problem.
- A well-chosen constraint handling strategy can significantly enhance the performance of Metaheuristic algorithms.

Outline

1. General Schema of Metaheuristics
2. Solution Encoding (Representation)
3. Objective Function
4. Constraint Handling
5. Parameter Tuning
5.1 Definition and Importance
5.2 Tuning Strategies
5.3 Key Takeaways
6. Performance Analysis
7. Common Concepts for Metaheuristics

Parameter Tuning - Definition and Importance

Parameter Tuning

Parameter tuning is the process of **selecting and adjusting the values of algorithm parameters** to **optimize the performance** of Metaheuristic algorithms on specific optimization problems.

- **Every Metaheuristic** has **multiple parameters** that need to be configured (e.g., iterations, population size, mutation rate, cooling schedule, tabu tenure)
- *Parameter tuning* provides: **Flexibility:** adapt algorithm to different problem types - **Robustness:** improve performance across instances - **Efficiency:** optimize computational resources
- **Challenge:** Finding the right parameter values is **non-trivial** and **problem-dependent**
- **Parameters have significant impact on: Solution quality** (objective value) - **Convergence speed** (computation time) - **Exploration vs exploitation** balance - **Robustness** across different instances
- **Key Issues:**
  - **No universal optimal values** exist
  - **Optimal values depend on:** Problem type and characteristics - Specific instance features - Available computational budget - Solution quality requirements

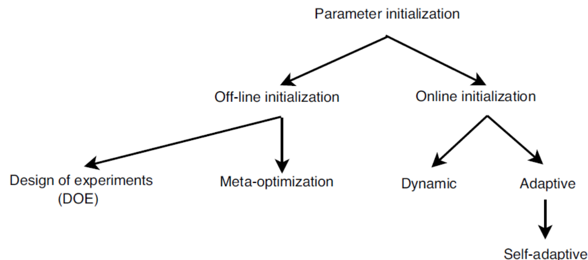
Parameter Tuning: Two Main Strategies

Off-line Tuning

- Parameters **fixed before execution**
- Also called *parameter initialization*
- Based on **preliminary experiments**
- **Pros:** Simple, predictable
- **Cons:** Static, may not adapt

On-line Tuning

- Parameters **updated during execution**
- Also called *adaptive control*
- Based on **search progress**
- **Pros:** Adaptive, flexible
- **Cons:** Complex, overhead



Off-line Parameter Tuning

Off-line Parameter Tuning

Parameters are **set to fixed values before** the algorithm execution based on **preliminary experiments** and **empirical analysis**.

- **Common approaches:**
  - **Trial-and-error:** manual experimentation
  - *Grid Search:* systematic exploration of parameter combinations
  - **Design of Experiments (DOE):** systematic exploration (e.g., factorial designs, Taguchi methods)
  - **Meta-optimization:** using another optimization algorithm to tune parameters (e.g., Random Search, Bayesian Optimization, Genetic Algorithms)
- **Best for:**
  - Well-understood problems with **stable characteristics**
  - When **computational budget allows** extensive preliminary testing
  - Production environments requiring **predictable behavior**

On-line Parameter Tuning

On-line Parameter Tuning

Parameters are **dynamically adjusted during** the algorithm execution based on **real-time feedback** from the search process.

- **Types of adaptation:**
  - **Deterministic:** predefined schedule (e.g., linear/exponential cooling)
  - **Adaptive:** based on search progress indicators
    - Success rate, diversity measures
    - Improvement rate, convergence detection
  - **Self-adaptive:** parameters encoded in solutions (common used in Evolutionary Algorithms)
- **Common approaches:**
  - **Feedback control:** adjust based on performance metrics
  - *Reinforcement learning:* learn optimal parameter policies
  - *Hyper-heuristics:* high-level strategies to select parameter settings
- **Best for:** **Diverse problem instances, long runs, when exploration/exploitation balance needs adjustment**

Comparison: Off-line vs On-line Tuning

Aspect	Off-line Tuning	On-line Tuning
Timing	Before execution	During execution
Adaptability	Static, fixed	Dynamic, adaptive
Complexity	Simple to implement	More complex, requires monitoring
Computational overhead	Low (only during setup)	Higher (due to adjustments)
Robustness	May not generalize well	More robust across instances
Best for	Stable problems, predictable behavior	Diverse problems, long runs
Examples	Grid Search, DOE, Meta-optimization	Feedback control, Reinforcement learning, Hyper-heuristics

- **Recommendation: Can combine both approaches (hybrid tuning)**

Key Takeaways

Key Takeaways

- Parameter tuning is essential for **optimizing the performance** of Metaheuristic algorithms.
- Two main strategies exist: **Off-line tuning** (fixed before execution) and **On-line tuning** (dynamic during execution).
- **Off-line** tuning is **simpler** but **static**, while **On-line** tuning is **adaptive** but **more complex**.
- The choice of tuning strategy depends on **problem characteristics**, **computational budget**, and **desired adaptability**.
- **Hybrid approaches** combining both strategies can leverage their strengths.
- Effective parameter tuning can significantly enhance **solution quality**, **convergence speed**, and **robustness**.

Outline

1. General Schema of Metaheuristics

2. Solution Encoding (Representation)

3. Objective Function

4. Constraint Handling

5. Parameter Tuning

6. Performance Analysis

6.1 Introduction

6.2 Three-Step Methodology

6.3 Common Pitfalls

6.4 Key Takeaways

Why Performance Analysis Matters

Performance Analysis

Performance analysis involves the **systematic evaluation and measurement** of Metaheuristic algorithms to **assess their effectiveness, efficiency, and robustness** on optimization problems.

- **Performance analysis** is **crucial** to:
  - **Validate** the effectiveness of metaheuristics
  - **Compare** different algorithms fairly
  - **Identify** strengths and weaknesses
  - **Guide** parameter tuning and improvements
  - **Justify** algorithmic choices in research/practice
- **Key Principles:**
  - Must be done on a **fair and rigorous basis**
  - **Theoretical analysis alone is insufficient** (NP-hard problems)
  - Requires **empirical evaluation** with proper methodology
  - Should follow **scientific standards** for reproducibility

Performance Analysis: Three Essential Steps

The performance analysis of Metaheuristic algorithms can be systematically conducted through a **three-step methodology** encompassing **experimental design, measurement, and reporting**.

1. Design

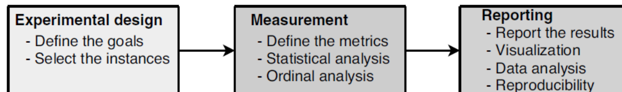
- Define goals
- Select instances
- Identify factors
- Plan experiments

2. Measurement

- Choose metrics
- Execute runs
- Collect data
- Apply statistics

3. Reporting

- Present results
- Visualize data
- Analyze findings
- Draw conclusions



Step 1: Experimental Design

- **Goal Definition: Clearly state what you want to evaluate**
  - **Quality-focused:** "Does algorithm A find better solutions than B?"
  - **Speed-focused:** "Which algorithm converges faster?"
  - **Robustness:** "Which algorithm is more consistent across instances?"
  - **Scalability:** "How does performance change with problem size?"
- **Instance Selection: Choose representative test problems**
  - **Benchmark instances:** standard datasets (TSP, VRP, SAT)
  - **Problem diversity:** vary size, structure, difficulty
  - **Real-world instances:** practical relevance
  - **Sufficient quantity:** statistical significance (30+ instances recommended)
- **Factors Identification: What variables affect performance?**
  - **Algorithm factors:** parameters, operators, strategies
  - **Problem factors:** size, constraints, structure
  - **Computational factors:** time limit, iterations, resources
- **Experimental Setup: Plan how to conduct experiments**
  - **Number of runs:** multiple independent runs (typically 30-50)
  - **Stopping criteria:** time limit, iteration count, or convergence
  - **Random seeds:** different seeds for stochastic algorithms
  - **Hardware specification:** document computing environment
  - **Parameter configuration:** document all settings

Step 2: Measurement - Performance Metrics

- **Solution Quality Metrics: Evaluate the effectiveness of the algorithm**
  - **Best solution found:** best objective value across all runs
  - **Average solution:** mean objective value over multiple runs
  - **Gap to optimum:** $\frac{|f(s)-f^*|}{f^*} \times 100\%$ (if optimum f^* known)
  - **Success rate:** % of runs reaching target quality
- **Computational Efficiency Metrics: Measure the resource usage of the algorithm**
  - **CPU time:** wall-clock time or processor time
  - **Iterations:** number of iterations to convergence
  - **Function evaluations:** total number of solution evaluations
  - **Time-to-target:** time to reach specific quality threshold
- **Robustness Metrics: Assess the consistency of the algorithm**
  - **Standard deviation:** spread of solution quality across runs
  - **Best/Average/Worst-case performance:** best, average, and worst solutions across runs
  - **Interquartile range (IQR):** middle 50% of results
- **Convergence Metrics: Analyze the search dynamics**
  - **Convergence curves:** quality over time/iterations
  - **Convergence rate:** speed of improvement
  - **Premature convergence:** early stagnation detection
  - **Diversity measures:** population/search diversity

Step 3: Reporting and Visualization

- **Tables: Summarize numerical results**
  - **Comparison table:** algorithms vs instances with best/avg/std
  - **Statistical significance:** mark significant differences
  - **Ranking:** rank algorithms per instance or overall, e.g., "Algorithm A: 95.2±2.1 (rank 1), Algorithm B: 93.4±3.5 (rank 2)"
- **Graphs: Visualize performance patterns**
  - **Convergence plots:** quality vs time/iterations
  - **Box plots:** distribution comparison across algorithms
  - **Performance profiles:** probability of solving within time thresholds
  - **Scatter plots:** instance-by-instance comparison
- **Analysis and Discussion: Interpret results meaningfully**
  - **Interpret results:** explain why certain algorithms perform better
  - **Identify patterns:** when/where algorithms excel or fail
  - **Discuss trade-offs:** quality vs speed, simplicity vs effectiveness
  - **Address limitations:** acknowledge experimental constraints
- **Best Practices for Reporting: Ensure clarity and reproducibility**
  - **Reproducibility:** provide all details (code, parameters, instances)
  - **Transparency:** report both successes and failures
  - **Fairness:** ensure equal computational budget for all algorithms
  - **Completeness:** document hardware, software, random seeds

Common Pitfalls in Performance Analysis

Pitfall	Impact	Best Practice
Single run evaluation	No statistical validity, unreliable conclusions	Run algorithm 30-50 times with different seeds
Cherry-picking instances	Biased results, overfitting to specific cases	Use diverse benchmark sets (TSPLIB, OR-Library)
Unfair comparison	Misleading conclusions about superiority	Equal CPU time/iterations, tune all algorithms
Ignoring stochasticity	Cannot claim significance	Apply Wilcoxon, Mann-Whitney, or ANOVA tests
Reporting only best	Hides algorithm variability	Report mean, std dev, median, worst case
Poor documentation	Results not reproducible	Document: code, parameters, instances, hardware
No convergence analysis	Misses premature convergence	Plot solution quality vs time/iterations
Parameter overfitting	Poor generalization to new instances	Use grid search and cross-validation
Ignoring problem structure	Algorithm may exploit specific features	Test on instances with varying characteristics
No baseline comparison	Cannot assess actual contribution	Compare against standard algorithms (greedy, LS)

Key Takeaways

Key Takeaways

- Performance analysis is essential for **validating and comparing** Metaheuristic algorithms.
- A systematic **three-step methodology** includes **experimental design**, **measurement**, and **reporting**.
- Proper **experimental design** involves clear goals, representative instances, factor identification, and a well-planned setup.
- Key **performance metrics** include solution quality, computational efficiency, robustness, and convergence behavior.
- Effective **reporting** requires clear tables, informative graphs, meaningful analysis, and adherence to best practices for reproducibility.
- Avoid common pitfalls such as single run evaluations, unfair comparisons, and poor documentation to ensure credible results.
- Rigorous performance analysis enhances the **credibility and impact** of optimization research.

Outline

1. General Schema of Metaheuristics

2. Solution Encoding (Representation)

3. Objective Function

4. Constraint Handling

5. Parameter Tuning

6. Performance Analysis

7. Common Concepts for Metaheuristics

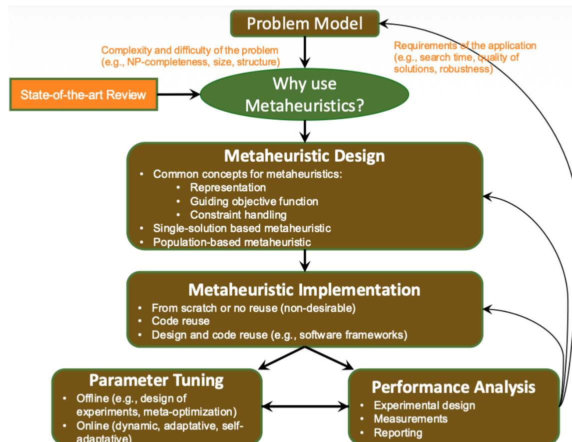
7.1 General Methodology Overview

7.2 Single-Solution Based Metaheuristics

7.3 Population-Based Metaheuristics

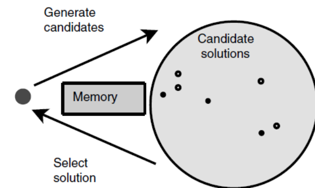
General Methodology Overview

- Solving optimization problems with metaheuristics follows a **six-step methodology**:
  - **Problem Modeling:** Define objectives, constraints, variables
  - **State-of-the-Art Review:** Analyze existing methods and solutions and argue the need for a new metaheuristic
  - **Metaheuristic Design:** Develop algorithmic strategies and components
  - **Metaheuristic Implementation:** Code and test the algorithm
  - **Parameter Tuning:** Optimize algorithm parameters for performance
  - **Performance Analysis:** Evaluate effectiveness and efficiency
- Each step is **critical** for developing **effective** and **efficient** metaheuristic solutions
- Form a **feedback loop** to iterate and refine each step based on insights gained



General Schema of S-Metaheuristics (1/3)

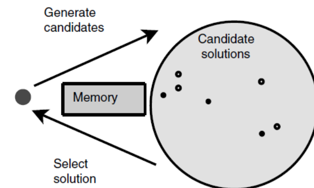
- **Single-Solution Based Metaheuristics** *S-Metaheuristics* **iteratively apply** the **generation** and **replacement** procedures **from the current single solution**.
- In the **generation phase**, a set $C(s)$ of **candidate solutions** are generated from the **current solution** s. This set $C(s)$ is generally obtained by local transformations of the solution.
- In the **replacement phase**[a], a **selection** is performed from the **candidate solution set** $C(s)$ to replace the **current solution**; that is, a solution $s' \in C(s)$ is selected to be the new solution.
- This process **iterates** until a given **stopping criteria**.



[a] Also named transition rule, pivoting rule, and selection strategy.

General Schema of S-Metaheuristics (2/3)

- The **generation** and the **replacement** phases may be:
  - **Memoryless:** the generation and replacement phases do not use any memory structure.
  - **Memory-based:** the generation and/or replacement phases use memory structures to store information from previous iterations to guide the search process.
- The **common search concepts** for all **S-Metaheuristics** are the **definition** of the **neighborhood structure**[a] and the **determination** of the **initial solution**.
- Popular examples of such **S-Metaheuristics** are *Local Search (LS)*, *Simulated Annealing (SA)*, *Tabu Search (TS)*, *Hill Climbing (HC)*, etc.



[a] *Will be discussed in next slides*

General Schema of S-Metaheuristics (3/3)

Algorithm 2 illustrates the **high-level template** (General Schema) of *S-Metaheuristics*.

Algorithm 2: High-level Template of S-Metaheuristics

Input: Initial solution s_0.
Output: Best Solution found

```
1 t = 0;
2 s_t = s_0;
3 repeat
4     Generate(C(s_t)) ; /* Generate candidate solutions (partial or complete
          neighborhood) from s_t                                          */
5     s_{t+1} = Select(C(s_t)); /* Select a solution from C(s_t) to replace the current
          solution s_t                                                     */
6     t = t + 1;
7 until Stopping Criteria Staisfied;
```

Neighborhood Structure (1/3)

Neighborhood Structure

Neighborhood structure defines the **set of solutions** that can be **reached from a given solution** by applying **small modifications** (moves).

- The **definition** of the *neighborhood structure* is a **required common step** for the design of any **S-Metaheuristic**.
- The **neighborhood structure** plays a **crucial role in the performance** of a S-Metaheuristic.
- ⇒ *If the neighborhood structure is not adequate to the problem, any S-Metaheuristic will fail to solve the problem.*

Neighborhood Structure (2/3)

A neighborhood function N is a mapping $N : S \rightarrow 2^S$ that assigns to each solution $s \in S$ a set of solutions $N(s) \subset S$, called the **neighborhood** of s.

- A solution s' in the **neighborhood** of s ($s' \in N(s)$) is called a *neighbor* of s.
- A **neighbor** is generated by the application of a *move operator* m that performs a **small perturbation to the solution** s.
- The **main property** that must characterize a **neighborhood** is *locality*.
- **Locality** is the **effect** on the solution when performing a **small change** in the **solution encoding**.
  - **Neighborhood** have a *strong locality* when **small changes** reveal **small effects** on the solution quality.
  - **Neighborhood** have a *weak locality* when **small changes** reveal **large effects** on the solution quality.
- The **structure of the neighborhood** depends on the **problem to be solved** and on the **solution representation**.

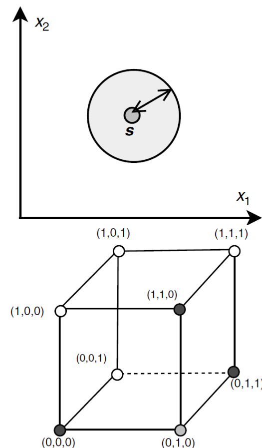
Neighborhood Structure (3/3)

Continuous Neighborhood

The neighborhood $N(s)$ of a solution s in a **continuous space** is the ball with center s and radius equal to ε with $\varepsilon > 0$.

Discrete Neighborhood

In a **discrete optimization** problem, the neighborhood $N(s)$ of a solution s is represented by the set $\{s'/d(s', s) \leq \varepsilon\}$, where d represents a given distance that is related to the move operator.



Example of TSP Neighborhood

- The complete neighborhood graph for the **Traveling Salesman Problem (TSP)** of size 4 is illustrated in the figure on the right.
- The **neighborhood structure** is defined over the set of **permutations** of the vertices.
- A **neighbor** is generated by the **swap** of each pair of adjacent vertices in the permutation.
- Thus, the **neighborhood** of a solution s of size n contains $n - 1$ neighbors.
- For instance, the neighbors of the solution **(1, 2, 3, 4)** are **(2, 1, 3, 4)**, **(1, 3, 2, 4)**, and **(1, 2, 4, 3)**.

1243 1342
1234 1324
2143 2134 3124 3142
2314 3214
2341 3241
2431 3421
4231 4321
2413 4213 4312 3412
4123 4132
1423 1432

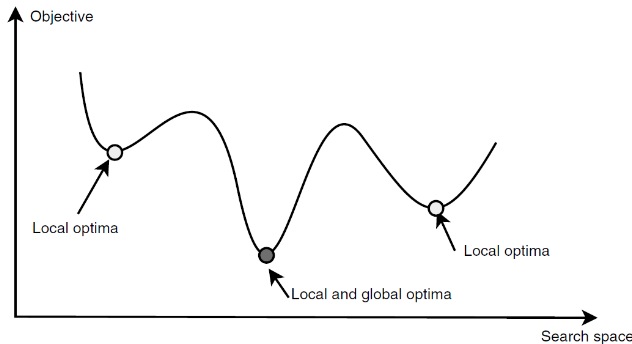
Local Optimum

Local Optimum

Relatively to a **given neighboring function** N, a solution $s \in S$ is a **local optimum** if it has a better quality than all its neighbors; that is, $f(s) \leq f(s') \forall s' \in N(s)$[a]

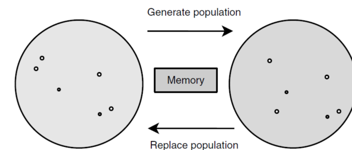
[a] *For a minimization problem.*

- For the **same optimization problem**, a **local optimum for a neighborhood** N_1 may not be a **local optimum for a different neighborhood** N_2.
- Thus, the **definition of the neighborhood structure** N **influences** the **number and the location of local optima** in the search space.



General Schema of P-Metaheuristics (1/3)

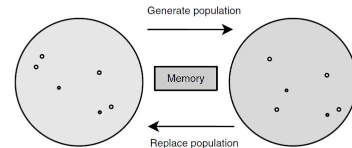
- **Population-based Metaheuristics** *P-Metaheuristics* **start** from an **initial population of solutions**[a].
- Then, they **iteratively apply** the **generation** of a **new population** and the **replacement** of the **current population**.
- In the **generation phase**, a **new population of solutions is created**.
- In the **replacement phase**, a **selection** is carried out **from the current and the new populations**.
- This process **iterates** until a given **stopping criteria**.



[a] *Some P-Metaheuristics such as Ant Colony Optimization start from partial or empty solutions.*

General Schema of P-Metaheuristics (2/3)

- The **generation** and the **replacement** phases may be:
  - **Memoryless:** the generation and replacement phases do not use any memory structure.
  - **Memory-based:** the generation and/or replacement phases use memory structures to store information from previous iterations to guide the search process.
- The **common search concepts** for all **P-Metaheuristics** are the **determination** of the **initial population** and the **diversity maintenance** in the population during the search process.
- **P-Metaheuristics** **differ** in the way they perform the **generation** and the **selection** procedures and the **search memory** they are **using during the search**.
- Most of the **P-Metaheuristics** are **nature-inspired algorithms**.
- Popular examples of such **P-Metaheuristics** are *Genetic Algorithms (GA)*, *Differential Evolution (DE)*, *Particle Swarm Optimization (PSO)*, *Ant Colony Optimization (ACO)*, etc.



General Schema of P-Metaheuristics (3/3)

Algorithm 3 illustrates the **high-level template** (General Schema) of *P-Metaheuristics*.

Algorithm 3: High-level template of P-Metaheuristics.

Input: Initial population P_0. ; /* Generation of the initial population */
Output: Best Solution found

```
1  t = 0;
2  P_t = P_0;
3  repeat
4  |   Generate(P'_t) ; /* Generation a new population */
5  |   P_{t+1} = Select-Population(P_t ∪ P'_t) ; /* Select new population */
6  |   t = t + 1
7  until Stopping Criteria Satisfied;
```

Initial Population

- Due to the **large diversity** of **initial populations**, **P-Metaheuristics** are naturally **more exploration** search algorithms whereas **S-Metaheuristics** are **more exploitation** search algorithms.
- The **determination of the initial population** is often disregarded in the design of a **P-Metaheuristic**. However, this step **plays a crucial role** in the *effectiveness* of the algorithm and its *efficiency*. Hence, it must be **carefully designed**.
- In the **generation of the initial population**, the **main criterion** to deal with is *diversification*.
- If the initial population is not well diversified, a **premature convergence can occur** for any **P-Metaheuristic**.
  $\Rightarrow$ *For instance, this may happen if the initial population is generated using a greedy heuristic or a S-Metaheuristic (e.g., Local Search, Tabu Search) for each solution of the population.*

Different Initialization Strategies

Strategies dealing with the **initialization of the population** may be **classified into four categories**:

- **Random Generation:** The initial population is **generated randomly** with respect of the problem constraints.
- **Sequential Diversification:** The initial population is **uniformly sampled in the decision space**. The **solutions are generated in sequence** in such a way that the diversity is optimized.
- **Parallel Diversification:** The **solutions of a population are generated in a parallel independent way**.
- **Heuristic Initialization:** Any **heuristic** (e.g., local search) **can be used to initialize the population**.
  - **Advantage:** The quality of the initial solutions is generally good.
  - **Disadvantage:** The diversity of the population is generally low.

Analysis of Initialization Strategies

- Strategies are evaluated on **three criteria: diversity**, **computational cost**, and **solution quality**

Strategy	Diversity	Computational Cost	Initial Quality
Pseudo-random	++ (Medium)	+++ (High)	+ (Low)
Quasi-random	+++ (High)	+++ (High)	+ (Low)
Sequential diversification	++++ (Very High)	++ (Medium)	+ (Low)
Parallel diversification	++++ (Very High)	+++ (High)	+ (Low)
Heuristic	+ (Low)	+ (Low)	+++ (High)

- **Trade-off:** Heuristic initialization provides **high-quality solutions** but **risks low diversity and premature convergence**

Stopping Criteria

Many *stopping criteria* **based on the evolution of a population** may be used. Some of them are similar to those designed for **S-metaheuristics**.

- **Static procedure: The end of the search may be known a priori**. For instance, one can use a **fixed number of iterations** (generations) or a **fixed amount of CPU time**.
- **Adaptive procedure: The end of the search cannot be known a priori**. Some examples are:
  - **No improvement** of the best solution found after a given number of generations.
  - **Population convergence:** All solutions in the population are identical or similar.
  - **Diversity threshold:** The diversity of the population falls below a predefined threshold.

Key Takeaways

Key Takeaways

- Metaheuristics can be broadly classified into **Single-Solution Based (S-Metaheuristics)** and **Population-Based (P-Metaheuristics)**.
- S-Metaheuristics iteratively improve a single solution through **generation** and **replacement** phases, relying on a well-defined **neighborhood structure**.
- The **neighborhood structure** is crucial for S-Metaheuristics, influencing the **search dynamics** and the **location of local optima**.
- P-Metaheuristics operate on a **population of solutions**, promoting **diversification** and **exploration** of the search space.
- The **initialization of the population** in P-Metaheuristics significantly impacts performance, with strategies balancing **diversity**, **computational cost**, and **solution quality**.
- Both S-Metaheuristics and P-Metaheuristics require carefully designed **stopping criteria** to ensure effective termination of the search process.

End of Chapter 2