COST Action CA22137 2023-2027

ROAR-NET

Problem Representation

First ROAR-NET Training School

Luca Di Gaspero

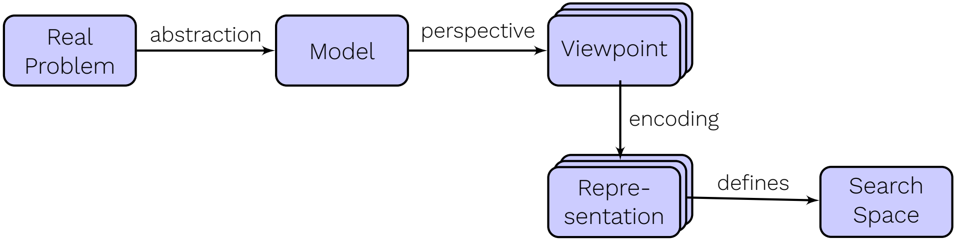
EUROPEAN COOPERATION IN SCIENCE & TECHNOLOGY
Funded by the European Union

Outline

1. Core Concepts and Examples
2. ROAR-NET API Representation
3. Direct vs. Indirect Representations
4. Represenation in Algorithmic Frameworks
5. Practical Guidelines
6. Conclusion

Core Concepts

- **Problem**: The real-world situation to be solved
- **Model**: A formal (e.g., mathematical) abstraction of the problem
- **Viewpoint**: A particular way of looking at the model
- **Representation**: How solutions are encoded within a viewpoint
- **Search Space**: All possible encoded solutions



Core Concepts: exemplified

- **Problem**: The real-world situation or question to be solved.
  - *Example*: Find the shortest route visiting all cities.
- **Model**: An abstraction of the problem, often mathematical.
  - *Example*: Graph $G = (V, E)$ with edge weights, find Hamiltonian cycle of minimum total weight.
- **Representation (Encoding)**: How a candidate solution to the model is stored and manipulated by the algorithm.
  - *Example*: The sequence of visited cities (i.e., a permutation of city indices for TSP).
- **Search Space (Solution Space)**: The set of all possible candidate solutions that the algorithm can explore, defined by the chosen representation.
  - *Example*: All permutations of n cities, i.e., $n!$ possible solutions.

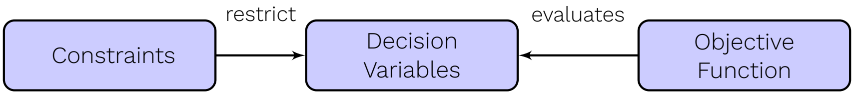
Representation defines the Search Space

- The representation dictates the structure and size of the search space.
- Different representations of the same problem can lead to radically different search spaces.
- This impacts:
  - The number of candidate solutions (size).
  - The “distance” between solutions (neighborhood).
  - The presence and structure of local optima (search landscape).
  - The effectiveness of search operators (hardness).

Building Blocks

Key components of a representation:

- **Decision Variables**: The controllable inputs that define the solution space.
- **Constraints**: Conditions that solutions must satisfy to be valid.
- **Objective Function**: Evaluates the quality of candidate solutions.



Decision Variables

Decision Variables

- Represent the **choices** or **decisions** that can be made.
- Their values determine a **candidate solution**.
- The optimization algorithm **searches** for the best values for these variables.
- They define the **dimensions** of the search space.

Example: In a production planning problem, decision variables might be:

- x_i = quantity of product i to produce;
- y_j = whether to open facility j (binary);
- z_{ijk} = amount shipped from facility i to customer j using transport k.

Types of Decision Variables (1/2)

Binary Variables

- Domain: $\{0, 1\}$
- Represent yes/no decisions
- *Examples*: Select item, open facility, assign task

Integer Variables

- Domain: $\mathbb{Z}$ or $\{a, a+1, \ldots, b\}$
- Discrete quantities or choices
- *Examples*: Number of units, worker assignments

Categorical Variables

- Domain: Finite set of categories
- Unordered discrete choices
- *Examples*: Color, method, route type

Permutation Variables

- Domain: Permutations of $\{1, 2, \ldots, n\}$
- Ordering/sequencing decisions
- *Examples*: Task sequence, tour order

Types of Decision Variables (2/2)

Continuous Variables

- Domain: $\mathbb{R}$ or $[a, b] \subset \mathbb{R}$
- Real-valued quantities
- *Examples*: Flow rates, coordinates, weights

Set Variables

- Domain: Subsets of a given set
- Selection of multiple items
- *Examples*: Feature selection, coalition formation

The type of decision variables heavily influences the choice of operators

Decision Variables: Examples by Problem Type

Problem	Decision Variables	Variable Type
Knapsack	x_i = 1 if item i selected	Binary
TSP	π = permutation of cities	Permutation
Portfolio Optimization	w_i = weight of asset i	Continuous
Job Scheduling	s_j = start time of job j	Continuous/Integer
Facility Location	y_i = 1 if facility i opened x_{ij} = flow from i to j	Binary Continuous
Vehicle Routing	Route assignment for each vehicle	Permutation/Set
Neural Network Design	w_{ij} = weight of connection Architecture choices	Continuous Categorical/Integer
Course Timetabling	(t, r) = time and room for course	Categorical

Most real problems involve **mixed** variable types, requiring hybrid representations.

Decision Variable Dependencies and Constraints

Decision variables often have complex relationships:

- **Independent Variables**: Can be set without affecting others
  - Example: Investment amounts in different sectors (if no budget constraint)
- **Dependent Variables**: Value depends on other variables
  - Example: Total cost depends on individual quantities produced
- **Conditional Dependencies**: Constraints only apply under certain conditions
  - Example: If facility is opened, then capacity constraints apply
- **Precedence Dependencies**: Some decisions must precede others
  - Example: Cannot schedule task B before deciding when A has been completed

Impact on Representation

Dependencies affect how we can encode and modify solutions

Example: Production Planning Problem

Production Planning Problem:

- Produce n products using m resources
- Binary decisions: Which products to produce
- Integer decisions: Batch sizes
- Continuous decisions: Resource allocation levels

Decision Variables:

- $y_i \in \{0, 1\}$: 1 if product i is produced
- $b_i \in \mathbb{Z}^+$: Number of batches of product i
- $r_{ij} \in \mathbb{R}^+$: Amount of resource j for product i

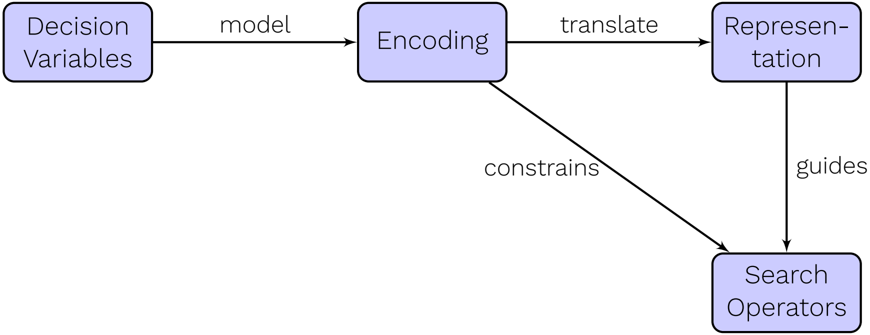
Representation Options:

1. **Direct**: $[y_1, \ldots, y_n, b_1, \ldots, b_n, r_{11}, \ldots, r_{nm}]$
2. **Hierarchical**: First decide y_i, then b_i, finally optimize r_{ij}
3. **Hybrid**: Binary for y_i, integer for b_i, real for r_{ij}

Dependencies: $b_i = 0$ and $r_{ij} = 0$ if $y_i = 0$ (conditional constraints)

From Problem Variables to Search Representation

Bridging the Problem Definition and Search Process:



Decision Variables Representation Questions

- What is the number, type, and domain of problem variables?
- Are variables continuous, discrete, categorical, or mixed?
- Which variables are interdependent or constrained?
- How should variables be grouped, transformed, or decomposed?
- How do variable structures affect search effectiveness?

Design Principle

Effective representations preserve meaningful structure and enable efficient search operations.

Decision Variables handling (1/2)

Different strategies for handling decision variables:

Direct Encoding

- One element per decision variable
- Natural mapping
- Example: $[x_1, x_2, x_3, x_4]$

Implicit Encoding

- Some variables computed from others
- Reduce search space
- Example: Priorities → assignments

Decomposed Encoding

- Separate different variable types
- Specialized operators
- Example: Binary + continuous parts

Decision Variables handling (2/2)

Different strategies for handling decision variables:

Grouped Encoding

- Group related variables by structure/dependency
- Exploit natural problem decomposition
- *Example*: Vehicle routing with 3 vehicles, each group represents a vehicle's route
- Encoding: $[[r_1^1, r_2^1, r_3^1], [r_1^2, r_2^2], [r_1^3, r_2^3, r_3^3, r_4^3]]$
- Operators can work within/across groups

Hierarchical Encoding

- High-level decisions first
- Conditional sub-decisions
- Example: Select facility, then assign customers

Choice depends on:

Variable relationships, constraint structure, algorithm requirements.

Constraints (1/2)

Constraints are conditions that solutions must satisfy to be considered valid.

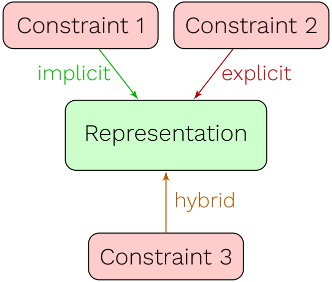
- They can be hard (must be satisfied) or soft (preferable but not mandatory).
- They define the boundaries of the search space.
- They can be linear, nonlinear, discrete, continuous, etc.

Define feasibility - they separate valid from invalid solutions

Constraints (2/2)

Different representations handle constraints differently:

- **Implicit constraint handling**: Built into the representation
- **Explicit constraint handling**: Checked separately (penalties, repairs)
- **Hybrid approaches**: Some constraints implicit, others explicit

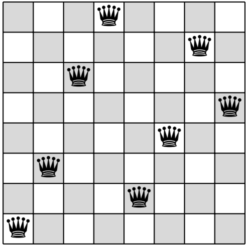


Design Principle

The more constraints you can build into the representation, the more efficient your search becomes.

Example: The N-Queens Problem

- **Problem**: Place N chess queens on an $N \times N$ chessboard so that no two queens attack each other.
- Queens attack horizontally, vertically, and diagonally.
- A classic constraint satisfaction problem often solved with optimization techniques (i.e., minimize the number of attacking pairs).



An 8×8 chessboard

Key Questions

How do we represent a potential solution (a placement of N queens)?
How does this representation affect the search for valid solutions?

N-Queens: Representation Viewpoint 1 (Coordinates)

- **Idea**: Represent each queen's position by its $(row, column)$ coordinates.
- A solution is a list of N coordinate pairs: $[(r_1, c_1), (r_2, c_2), \ldots, (r_N, c_N)]$.
- For an $N \times N$ board:
  - Each r_i can be from 1 to N.
  - Each c_i can be from 1 to N.
- **Search Space Size**: $C(N^2, N)$ possible solutions (choosing N distinct coordinates among N^2 possibilities).
- **Challenges**:
  - Extremely large search space.
  - Most solutions are invalid (multiple queens in same row/col/diag).
  - Example for $N = 4$: $C(16, 4) = 1,280$ solutions.

N-Queens: Representation Viewpoint 2 (Fixed Column)

- **Observation**: A valid N-Queens solution must have exactly one queen per column.
- **Idea**: Fix the columns (e.g., $1, 2, \ldots, N$) and just decide the row for each queen.
- A solution is a list of N row indices: $[r_1, r_2, \ldots, r_N]$, where r_i is the row of the queen in column i.
- Each r_i can be from 1 to N.
- **Search Space Size**: N^N possible solutions.
- **Challenges**:
  - Still many invalid solutions (queens attack horizontally or diagonally).
  - Significantly smaller than Viewpoint 1.
  - Example for $N = 4$: $4^4 = 256$ solutions.

N-Queens: Representation Viewpoint 3 (Permutation)

- **Observation**: A valid N-Queens solution must have exactly one queen per row AND one queen per column.
- **Idea**: A queen in column i is placed in row p_i, where $(p_1, p_2, \dots, p_N)$ is a permutation of $(1, 2, \dots, N)$.
- This implicitly handles the ”one queen per row” and ”one queen per column” constraints.
- **Search Space Size**: $N!$ possible solutions.
- **Challenges**:
  - Only diagonal attacks need to be checked.
  - Even smaller search space, but solutions still might be invalid.
  - Example for $N = 4$: $4! = 24$ solutions.

N-Queens: Comparing Search Spaces

Representation	Encoding	Search Space Size	Constraints Checks
Coordinates	$[(r_1, c_1), \ldots, (r_N, c_N)]$	$C(N^2, N)$	Row, Col, Diagonal
Fixed Column	$[r_1, r_2, \ldots, r_N]$	N^N	Row, Diagonal
Permutation	$[p_1, p_2, \ldots, p_N]$	$N!$	Diagonal Only

- The choice of representation dramatically changes the size and characteristics of the search space.
- A "smarter" representation incorporates more problem constraints implicitly.

Example: Nurse Scheduling Problem

- **Problem**: Four (head) nurses assigned to eight-hour shifts over a week
- **Shifts**: Shift 1 (day), Shifts 2 and 3 (night)
- **Constraints**:
  1. Every shift is assigned exactly one nurse
  2. Each nurse works at most one shift per day
  3. Each nurse works at least five days a week (the others should be days off)
  4. No shift can be staffed by more than two different nurses in a week
  5. A nurse working night shifts (2 or 3) must do so at least two consecutive days

Viewpoint 1: Assignment Worker

- **Idea**: Directly represent which nurse is assigned to which shift-day combination
- **Representation**: X_{sd} where $X_{sd} = n$ if nurse n works shift s on day d
- **Search Space**: 3×7 variables, $3 \times 7 \times 4 = 84$ possible assignments
- **Constraint Handling**:
  - Constraint 1: one nurse per shift (implicit)
  - Constraint 2-5: (explicit, to check)
- **Challenges**: Most random assignments violate constraints

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Shift1	A	B	A	A	A	A	A
Shift2	C	C	C	B	B	B	B
Shift3	D	D	D	D	C	C	D

Viewpoint 2: Shift Sequence to Nurse

- **Idea**: Represent each assignment to a nurse for the week as sequence of shifts
- **Representation**: Three sequences, one per nurse: S_A, S_B, S_C, S_D
- Each sequence: $[s_1, s_2, \ldots, s_7]$ where s_i is the shift assigned for day i
- **Search Space**: $4^7 = 16,384$ possible sequences (each day can be one of 3 shifts + day off $\perp$)
- **Constraint Handling**:
    - Constraint 1: (explicit, to check)
    - Constraint 2: one shift per day (implicit)
    - Constraint 3-5: (easier to check)

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Worker A	1	$\perp$	1	1	1	1	1
Worker B	$\perp$	1	$\perp$	2	2	2	2
Worker C	2	2	2	$\perp$	3	3	$\perp$
Worker D	3	3	3	3	$\perp$	$\perp$	3

Comparing the Two Viewpoints

Aspect	Assignment-Based	Sequence-Based
Search Space Size	84	$16,384$
Constraint 1 (One nurse per shift)	Explicit	Implicit
Constraint 2 (One shift per day)	Implicit	Explicit
Constraint 3-5	Complex	Simple

Synchronization (channeling): The two viewpoints should be synchronized. Operators in the assignment-based representation should modify the sequences in the sequence-based representation and vice versa.

Example: Portfolio Optimization

Problem: Allocate resources across multiple assets maximizing return while minimizing risk.

- **Objective**: Maximize expected return while minimizing risk
- **Variables**: Asset weights w_i (proportions of total capital)
- **Objective Function** (risk minimization): $\sum_{i=1}^{n} \sum_{j=1}^{n} \sigma_{ij} w_i w_j$, where σ_{ij} is the covariance between assets i and j (variance when $i = j$).
- **Constraints**:
  1. Return constraint, e.g., $\sum_{i=1}^{n} r_i w_i \geq R$, where r_i is the expected return of asset i.
  2. Total investment equals available capital, i.e., $\sum_{i=1}^{n} w_i = 1$
  3. Individual asset weights within bounds, e.g., $\epsilon_i \leq w_i \leq \delta_i$ (or $w_i = 0$ for no investment).
  4. Number of assets must be between minimum and maximum limits, e.g., $n_{min} \leq |\{w_i | w_i > 0, i = 1, \ldots, n\}| \leq n_{max}$.

Portfolio Optimization: Decomposition

Strategy:

- **Subproblem 1**: Asset selection (which assets to include)
- **Subproblem 2**: Weight allocation (how much to invest in each selected asset)

Representations:

- For asset selection: Binary vector $[s_1, s_2, ..., s_n]$ where $s_i = 1$ if asset i is selected.
- For weight allocation: Real-valued vector $[w_{i_1}, w_{i_2}, ..., w_{i_k}]$ for weights of selected k assets.

Selecting assets through the binary vector eases the satisfaction of the cardinality constraints (i.e., $n_{min} \leq k \leq n_{max}$).

Once the asssets are selected the weights can be allocated to the selected assets only, considerably reducing the search space of the (quadratic programming) weight allocation problem.

Constraint Handling Strategies

Penalties

- Add cost for constraint violations
- Allows exploration of infeasible regions
- Risk: Algorithm might converge to infeasible solutions

Repair

- Fix violated constraints after operator application
- Ensures all solutions are feasible
- Risk: May be computationally expensive

Specialized Operators

- Design operators that preserve feasibility
- Guarantees feasible offspring
- Risk: Limited exploration capability

Decoder Functions

- Indirect representation with constraint-aware decoder
- Can handle complex constraints elegantly
- Risk: Many-to-one mapping inefficiency

Objective Function (1/2)

Objective Function evaluates the quality or fitness of candidate solutions

- Provides a **mechanism for comparing** different solutions
- Guides the search process toward better solutions
- May incorporate constraint violations through penalties or other mechanisms
- Defines what constitutes “improvement” in the search space

Types of Objective Functions:

- **Single-objective**: $f(x) \to \mathbb{R}$ (minimize cost, maximize profit)
- **Multi-objective**: $\mathbf{f}(x) \to \mathbb{R}^k$ (minimize cost AND time)
- **Multi-criteria**: Hierarchical or lexicographic preferences
- **Constraint-augmented**: $f(x) + \sum$ penalty(violations)

Objective Function (2/2)

Handling Incomplete Solutions:

- **Partial evaluation**: Assess quality of incomplete solutions during construction
- **Lower bound estimation**: Predict best possible completion of partial solution
- **Heuristic completion**: Use fast heuristics to complete partial solutions for evaluation

> The objective function shapes the fitness landscape that the algorithm explores

Symmetry and Redundancy in Representations

Symmetry: Multiple different encodings represent the same solution
Redundancy: Search space contains equivalent or duplicate solutions

Examples of Symmetry:

- **TSP**: Tours $[1, 2, 3, 4]$ and $[4, 3, 2, 1]$ represent the same cycle
- **Graph Coloring**: Node permutations with same color pattern
- **Set Partitioning**: Different orderings of the same partition
- **Vehicle Routing**: Different vehicle-route assignments for identical solutions

Impact on Search:

- Wasted computational effort
- Slower convergence
- Difficulty comparing solutions
- Larger effective search space
- Population diversity issues

Strategies to Handle Symmetry and Redundancy

Representation-Level Solutions:

- **Canonical Forms**:
  - Fix ordering conventions
  - TSP: Always start from city 1
  - Graph: Use lexicographic ordering
- **Reduced Representations**:
  - Eliminate redundant variables
  - Relative vs. absolute encodings
  - Random keys instead of permutations
- **Invariant Encodings**:
  - Representations insensitive to symmetries
  - Edge-based instead of node-based

Algorithm-Level Solutions:

- **Symmetry Breaking**:
  - Add constraints to eliminate symmetries
  - Preprocessing to identify symmetric elements
- **Normalization**:
  - Convert solutions to canonical form
  - During evaluation or comparison
- **Duplicate Detection**:
  - Hash tables for quick lookup
  - Distance-based equivalence checking

Example: Handling Tour Symmetries in TSP

TSP Example: Handling Tour Symmetries

Problem	Standard Permutation	Canonical Form
Multiple equivalent representations	$[1, 2, 3, 4]$, $[2, 3, 4, 1]$, $[4, 3, 2, 1]$, $[3, 2, 1, 4]$	Always start with smallest: $[1, 2, 3, 4]$
Search space reduction	$n!$ possibilities	$\frac{(n-1)!}{2}$ possibilities (fix start + direction)

Design Principle

Good representations minimize symmetry while preserving solution quality and operator effectiveness

Complete vs. Incomplete Solution Representations

Complete Solutions

- All decision variables assigned
- Ready for immediate evaluation
- Often combined with selective heuristics

Examples:

- TSP: Full tour $[1, 3, 2, 4, 1]$
- Knapsack: $[1, 0, 1, 1, 0]$ for all items
- Graph coloring: Color for every node $[{\color{blue}\blacksquare}, {\color{green}\blacksquare}, {\color{red}\blacksquare}, {\color{blue}\blacksquare}]$

Incomplete Solutions

- Partial variable assignments
- Might allow lower bound estimation
- Often combined with constructive heuristics
- Requires to represent unassigned variables

Examples:

- TSP: Partial tour $[1, 3, ?, ?, 1]$
- Knapsack: Set of selected items $\{0, 1\}$
- Graph Coloring: Partial colors $[{\color{blue}\blacksquare}, ?, {\color{red}\blacksquare}, ?]$

ROAR-NET API Representation

- The ROAR-NET API is **representation agnostic** by design.
- Core types:
  - `Problem`: Encapsulates problem-specific data.
  - `Solution`: Encodes a candidate solution in chosen representation.
- Key operations (`random_solution`, `heuristic_solution`, `empty_solution`, `copy_solution`, `objective_value`) are **representation-aware**:
  - Support direct, indirect, hybrid, or problem-specific encodings.
  - Generate, copy, and evaluate solutions using underlying representation logic.
- Algorithmic components interact only with abstract `Problem`/`Solution` interfaces.
- Enables experimentation with different representations using the same framework.

ROAR-NET API Representation

- **Implementation Note**: `Solution` can include **auxiliary data** beyond minimal encoding:
  - *Examples*: Cached objective values, incremental cost updates, auxiliary structures (adjacency lists, partial schedules).
  - Useful for algorithms requiring frequent evaluation or incremental updates.
  - API allows transparent updates when solution is modified.

Design Principle

Practical representations combine compact encoding with auxiliary fields for efficient search and evaluation.

Direct Representation

- **Definition**: The decision variables of the problem are directly encoded as part of the solution structure.
- Each element in the representation directly corresponds to a decision variable.
- Often intuitive and straightforward.

Examples:

- **Binary Encoding**: For problems with binary decisions (e.g., Knapsack Problem, where each item is either included or not).
  - Solution: $[1, 0, 1, 1, 0]$ (items 1, 3, 4 are selected)
- **Permutation Encoding**: For ordering problems (e.g., TSP, Scheduling).
  - Solution: $[3, 1, 4, 2]$ (order of tasks/cities)
- **Real-Valued Encoding**: For continuous optimization problems.
  - Solution: $[x_1, x_2, x_3]$ (values for continuous variables)

Direct Encoding: Pros and Cons

Advantages:

- **Simplicity**: Easy to implement and interpret.
- **Clarity**: Direct correspondence between variables and representation.
- **Efficiency**: Enables fast, generic search operators.
- **Validity**: Fewer infeasible solutions.

Disadvantages:

- **Operator Design Overhead**: Requires problem-specific variation operators (e.g., permutation-preserving crossover).
- **Constraint Handling**: Difficult to express complex constraints without repairs or penalty functions.
- **Limited Structural Insight**: Encodings may ignore latent structure or relationships among variables (general also to indirect).

Indirect Representation

- **Definition**: The representation does not directly encode the decision variables. Instead, it encodes parameters or rules that, when decoded, generate a candidate solution.
- Requires a **decoder function** or **construction heuristic**.
- Often used when direct encoding is difficult or leads to many invalid solutions.

Examples:

- **Priority-based Encoding**: For scheduling problems.
  - Solution: $[0.7, 0.2, 0.9, 0.5]$ (priorities for tasks).
  - Decoder sorts tasks by priority to generate a schedule.
- **Rule-based Encoding**: For designing complex systems.
  - Solution: A set of production rules (e.g., for L-systems in evolutionary art).
  - Decoder interprets rules to generate a structure.
- **Neural Network Weights**: For evolving neural networks.
  - Solution: A vector of weights and biases.
  - Decoder constructs the network and evaluates its performance.

Pros and Cons of Indirect Representation

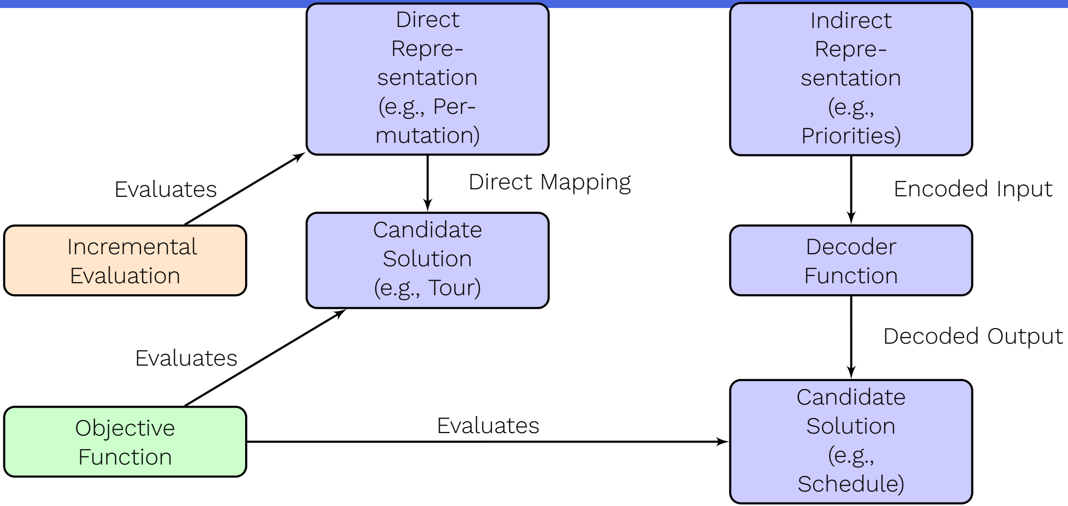
Advantages:

- **Generality**: Standard operators (e.g., bit flip, uniform crossover) can often be used on the indirect encoding.
- **Compactness**: May allow for more compact representations of complex solutions.
- **Exploration of Building Blocks**: Can focus search on "good" parameters or rules.
- **Decoder Flexibility**: The decoder can handle complex business logic and rules, simplifying the search.

Disadvantages:

- **Decoder Complexity**: Requires a sophisticated and often computationally expensive decoder function.
- **Epistasis**: Complex relationships between encoded variables and actual solution components, making search difficult.
- **Redundancy**: Different indirect encodings might decode to the same solution (neutrality).
- **Loss of Locality**: Small changes in the indirect encoding might lead to large changes in the decoded solution.

Direct vs. Indirect Representation



Direct and Indirect Representations

Example: Random Key Encoding for TSP

Problem: Visit all cities exactly once with minimum travel distance.

Direct (Permutation)

- Encoding: $[1, 3, 2, 4]$
- Meaning: Visit cities in order 1→3→2→4→1
- Constraints: All permutations valid
- Operators: Need specialized crossover

Indirect (Random Keys)

- Encoding: $[0.7, 0.2, 0.9, 0.1]$
- Decoder: Sort by values
- Result: $[4, 2, 1, 3]$ (tour order)
- Operators: Standard real-valued

Biased Random Key Genetic Algorithms (BRKGA)

Key Concept: Use random keys to encode solutions for complex combinatorial problems.

Representation: Vector of real numbers in $[0, 1]$: $[k_1, k_2, \ldots, k_n]$.

Advantages:

- **Universal representation**: Same encoding for many problem types
- **Standard operators**: Use real-valued crossover and mutation
- **Always feasible**: Decoder ensures constraint satisfaction
- **Problem-independent search**: GA doesn't need problem knowledge

Core Principle

Separate the *search mechanism* (GA) from *problem knowledge* (decoder)

BRKGA: General Decoding Framework

- **Priority-Based Decoding**:
  - Keys represent priorities of elements
  - Sort elements by key values
  - Process in priority order using greedy heuristic
- **Threshold-Based Decoding**:
  - Keys represent selection probabilities
  - Elements with $k_i > \theta$ are selected
  - Threshold θ can be fixed or adaptive
- **Parameter-Based Decoding**:
  - Keys represent heuristic parameters
  - Use constructive algorithm with key-derived parameters
  - Different keys $\rightarrow$ different algorithmic behavior

Design Principle

The decoder should be **fast, deterministic** and provide **good** solutions.

BRKGA Example 1: Job Scheduling

Problem: Schedule n jobs on m machines to minimize makespan.

Representation: n random keys $[k_1, k_2, \ldots, k_n]$.
Decoder Algorithm:

1. Sort jobs by key values (descending): $k_{j_1} \geq k_{j_2} \geq \cdots \geq k_{j_n}$
2. **For each** job j_i in sorted order:
   - Assign j_i to machine with earliest completion time (second decision)
   - Update machine completion time

Example:

Job	Processing Time	Random Key	Priority Rank	Assigned Machine
1	5	0.3	3	M2
2	8	0.9	1	M1
3	3	0.7	2	M2
4	6	0.1	4	M1

Schedule: M1: [Job2, Job4], M2: [Job3, Job1]

BRKGA Example 2: Set Cover Problem

Problem: Select minimum cost subset of sets that covers all elements.

Representation: Random key k_i for each set S_i.
Decoder Algorithm:

1. Sort sets by $\frac{k_i \cdot |S_i \cap U|}{cost_i}$ (greedy ratio with randomization)
2. Initialize uncovered elements U
3. **While** $U \neq \emptyset$:
   - Select highest-ratio set that covers elements in U
   - Add set to solution
   - Remove covered elements from U

Key Insight

Random keys bias the greedy selection, leading to different solutions.

Keys: $[0.8, 0.3, 0.9, 0.2]$, Ratios: $[2.4, 0.9, 1.8, 0.4]$ (key × coverage/cost), Selection order: S_1, S_3, S_2, S_4

Binary Chromosomes in Genetic Algorithms

Binary Representation: Solutions encoded as strings of 0s and 1s.

Natural Applications:

- **Selection Problems**: Each bit indicates inclusion/exclusion
- **Feature Selection**: Bit i = 1 if feature i is selected
- **Knapsack Problems**: Bit i = 1 if item i is taken
- **Network Design**: Bit i = 1 if edge i is included

Advantages:

- Simple, well-understood operators (one-point, uniform crossover)
- Efficient bit manipulation
- Schema theorem analysis applies
- Many theoretical results available

Chromosome: $[1, 0, 1, 1, 0, 1, 0, 1]$, Meaning: Select items $\{1, 3, 4, 6, 8\}$, Fitness: evaluate selected subset

Binary Decoding Strategies

Direct Binary Decoding:

- Bit i directly corresponds to decision variable x_i
- $x_i = 1$ if bit $i = 1$, else $x_i = 0$
- Simple but may violate constraints

Gray Code Decoding:

- For numerical optimization with binary strings
- Adjacent integers differ by exactly one bit
- Reduces the impact of crossover disruption

Constraint-Based Decoding:

- **Repair**: Fix violations after direct decoding
- **Greedy Construction**: Use bits to guide feasible construction
- **Priority-Based**: Bits represent priorities, then apply decoding algorithm

Example of Constraint-Based Decoding

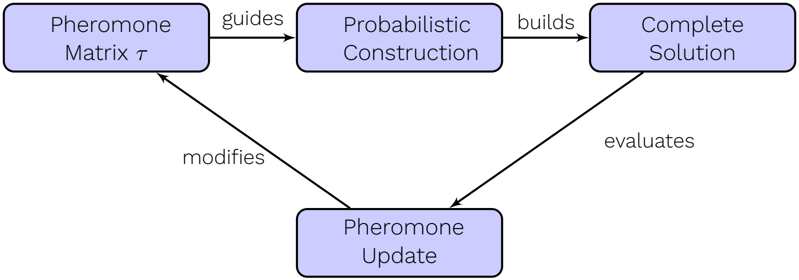
Knapsack with repair

1. Direct decode: $[1, 1, 1, 1] \to$ select all items
2. Check capacity: Total weight = 100, capacity = 80 (infeasible)
3. Repair: Remove lowest value/weight ratio items until feasible

ACO: Indirect Solution Construction

ACO Representation Concept:

- **No explicit chromosomes** - solutions are constructed incrementally
- **Pheromone trails** are attached to the "representation" (solution elements)
- Each ant builds a solution by following probabilistic rules
- Pheromone concentrations encode collective knowledge



ACO: Pheromone Trail Representation

Pheromone Trail Structure:

- **Components**: Problem-specific building blocks (cities, jobs, edges)
- **Connections**: Pheromone τ_{ij} on transitions between components
- **Meaning**: τ_{ij} = learned desirability of choosing j after i

Construction Rule (general form): $p_{ij} = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{k \in \mathcal{N}_i} [\tau_{ik}]^\alpha \cdot [\eta_{ik}]^\beta}$, where:

- τ_{ij} = pheromone trail (learned)
- η_{ij} = heuristic information (problem-specific)
- α, β = parameters balancing exploration vs. exploitation
- $\mathcal{N}_i$ = feasible components from state i

Pheromone Update: $\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} + \sum_k \Delta\tau_{ij}^k$, where ρ = evaporation rate, $\Delta\tau_{ij}^k$ = pheromone deposited by ant k on selection sequence (i, j) (according to solution quality).

ACO Example 1: Traveling Salesman Problem

Problem: Find shortest tour visiting all cities exactly once.

Components: Cities $\{1, 2, \dots, n\}$
Pheromone Trails: τ_{ij} = desirability of traveling from city i to city j
Heuristic Information: $\eta_{ij} = \frac{1}{d_{ij}}$ (inverse of distance)

Construction Process:

1. Start at random city
2. **While** unvisited cities remain:
    - Calculate probabilities for unvisited cities
    - Select next city probabilistically
    - Move to selected city
3. Return to starting city

ACO Example 2: Job Shop Scheduling

Problem: Schedule jobs on machines to minimize makespan.

Components: Operations (j, i) where job j needs machine i
Pheromone Trails: $\tau_{(j,i),(k,\ell)}$ = desirability of scheduling operation (k, ℓ) after (j, i)
Heuristic Information: $\eta_{(j,i)} = \frac{1}{p_{j,i}}$ (inverse processing time)

Construction Process:

1. Initialize: All machines idle, all operations ready
2. **While** unscheduled operations exist:
    - Identify schedulable operations (precedence constraints satisfied)
    - Calculate selection probabilities using τ and η
    - Select operation probabilistically
    - Schedule operation and update machine/job states

Representation Insight: Pheromone encodes good operation sequencing patterns.

ACO: Construction Graph Design

Key Design Decisions for ACO Representations:

- **What are the components?**
  - Atomic decision elements
  - Example: Cities (TSP), tasks (scheduling), items (selection)
- **What are the connections?**
  - How components can be combined
  - Example: City-to-city transitions, operation sequences
- **How to handle constraints?**
  - Implicit: Only allow feasible components in $\mathcal{N}_i$
  - Explicit: Penalize constraint violations
- **What heuristic information to use?**
  - Problem-specific greedy guidance
  - Example: Distance, processing time, cost

ACO Representation Principle

The construction graph should naturally encode the problem structure and allow incremental solution building.

Representation Design Principles

Fundamental Principles

1. **Completeness**: All valid solutions should be representable
2. **Soundness**: All representations should decode to valid solutions
3. **Non-redundancy**: Minimize multiple encodings for same solution
4. **Locality**: Small changes in encoding → small changes in solution
5. **Constraint compatibility**: Representation should align with constraint structure

Practical Steps:

- Analyze constraint types and relationships
- Identify which constraints can be made implicit
- Consider problem decomposition possibilities
- Test multiple representations empirically
- Measure search space characteristics

Common Pitfalls and How to Avoid Them

- **Over-engineering**:
  - Problem: Too complex representations
  - Solution: Start simple, add complexity only when justified
- **Ignoring constraints**:
  - Problem: All constraints handled explicitly
  - Solution: Build constraints into representation structure
- **Poor scalability**:
  - Problem: Representation doesn't scale with problem size
  - Solution: Test on problems of different sizes early
- **Algorithm mismatch**:
  - Problem: Representation incompatible with chosen operators
  - Solution: Co-design representation and operators

Golden Rule

The best representation makes your problem easier to solve, not harder to encode

Key Takeaways

- **Representation is fundamental** - it often matters more than algorithm choice
- **Constraints are your guide** - use them to shape your representation design
- **Multiple viewpoints exist** for every problem - explore different perspectives
- **Search space size matters** - smaller, more focused spaces usually perform better
- **Implicit constraint handling** is almost always better than explicit
- **No silver bullet** - the best representation is problem-specific
- **Empirical validation** is essential - test your design choices

Take home message

A Good representation design can transform an intractable problem into a solvable one!

Thank You!

Questions?

Acknowledgments

This presentation is based upon work from COST Action Randomised Optimisation Algorithms Research Network (ROAR-NET), CA22137, supported by COST (European Cooperation in Science and Technology).

COST (European Cooperation in Science and Technology) is a funding agency for research and innovation networks. Our Actions help connect research initiatives across Europe and enable scientists to grow their ideas by sharing them with their peers. This boosts their research, career and innovation.

Funded by the European Union

References I

C. A. Coello Coello.
Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art.
Computer Methods in Applied Mechanics and Engineering, 191(11-12):1245–1287, 2002.

A. E. Eiben and J. E. Smith.
Introduction to Evolutionary Computing.
Springer-Verlag, Berlin, 2nd edition, 2015.

M. Gendreau and J.-Y. Potvin, editors.
Handbook of metaheuristics.
International Series in Operations Research & Management Science.
Springer International Publishing, Basel, Switzerland, 3 edition, Sept. 2018.

References II

J. F. Gonçalves and M. G. C. Resende.
Biased random-key genetic algorithms for combinatorial optimization.
Journal of Heuristics, 17(5):487–525, 2011.

R. Martí, P. Panos, and M. Resende, editors.
Handbook of heuristics.
Handbook of Heuristics. Springer International Publishing, Basel, Switzerland, 1 edition, Nov. 2016.

ROAR-NET API Contributors.
Roar-net api python implementation.
https://github.com/roar-net/roar-net-api-py, 2025.
Accessed: 2025-06-15.

References III

ROAR-NET API Contributors.
Roar-net api specification.
https://github.com/roar-net/roar-net-api-spec, 2025.
Accessed: 2025-06-15.

F. Rothlauf.
Representations for Genetic and Evolutionary Algorithms, volume 104 of *Studies in Fuzziness and Soft Computing*.
Springer-Verlag, Berlin, 2nd edition, 2006.

F. Rothlauf.
Design of modern heuristics.
Natural Computing Series. Springer, Berlin, Germany, 2011 edition, July 2011.