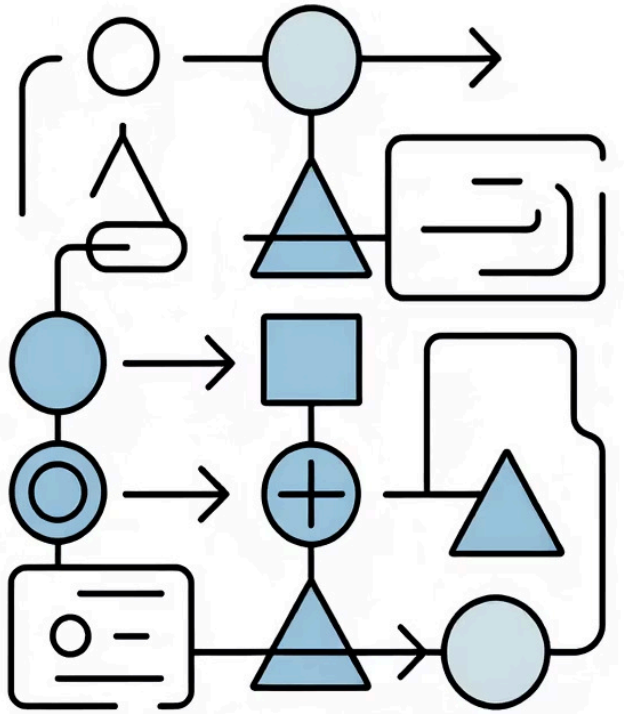
# 8.4.4 Exemple of algorithm use in MODAPTO:

# Proposed Dynamic Grouping Algorithm (2)

## Optimizing with Genetic Algorithm

# Genetic Algorithm



# Introduction to Genetic Algorithms for Maintenance Grouping

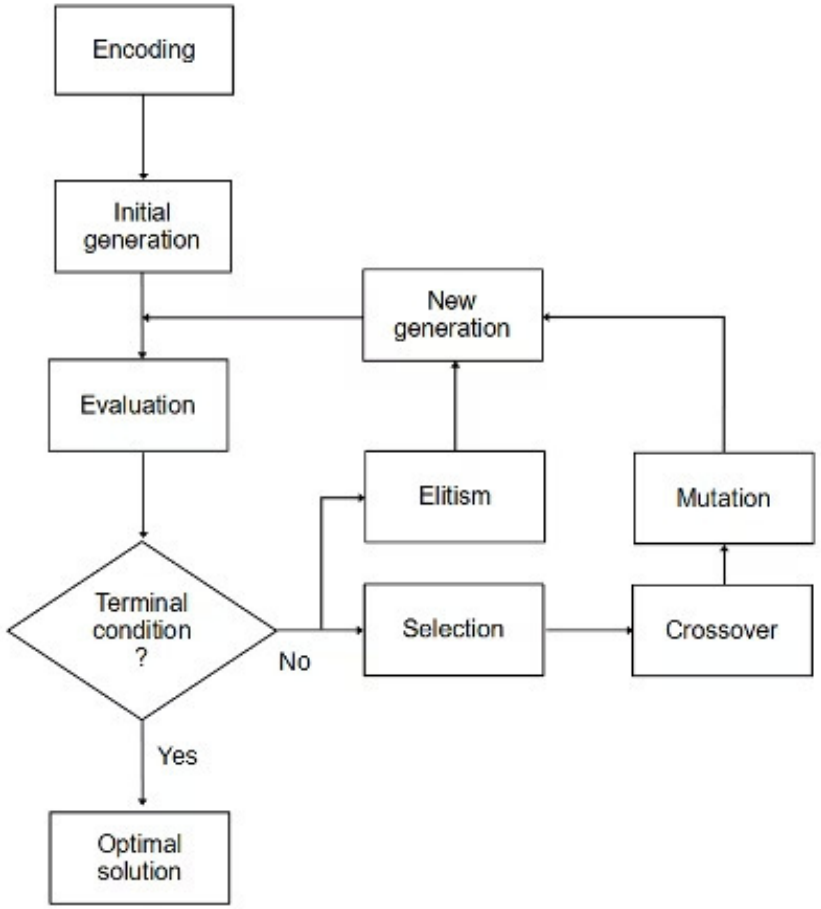
Genetic Algorithms (GAs) represent a powerful class of evolutionary computation techniques inspired by natural selection principles. In the context of maintenance optimization, GAs provide a robust framework for solving complex grouping problems where traditional methods often fail due to computational complexity and non-linearity.

Maintenance grouping seeks to identify optimal combinations of preventive maintenance activities that should be performed simultaneously to reduce setup costs, minimize system downtime, and maximize economic benefits. The combinatorial nature of this problem makes it particularly suitable for genetic algorithm applications.

Key advantages of applying GAs to maintenance grouping include:

- Ability to handle large-scale systems with numerous components
- Flexibility in incorporating various constraints and objectives
- Capacity to find near-optimal solutions in reasonable computational time
- Adaptability to dynamic environments where maintenance requirements change over time

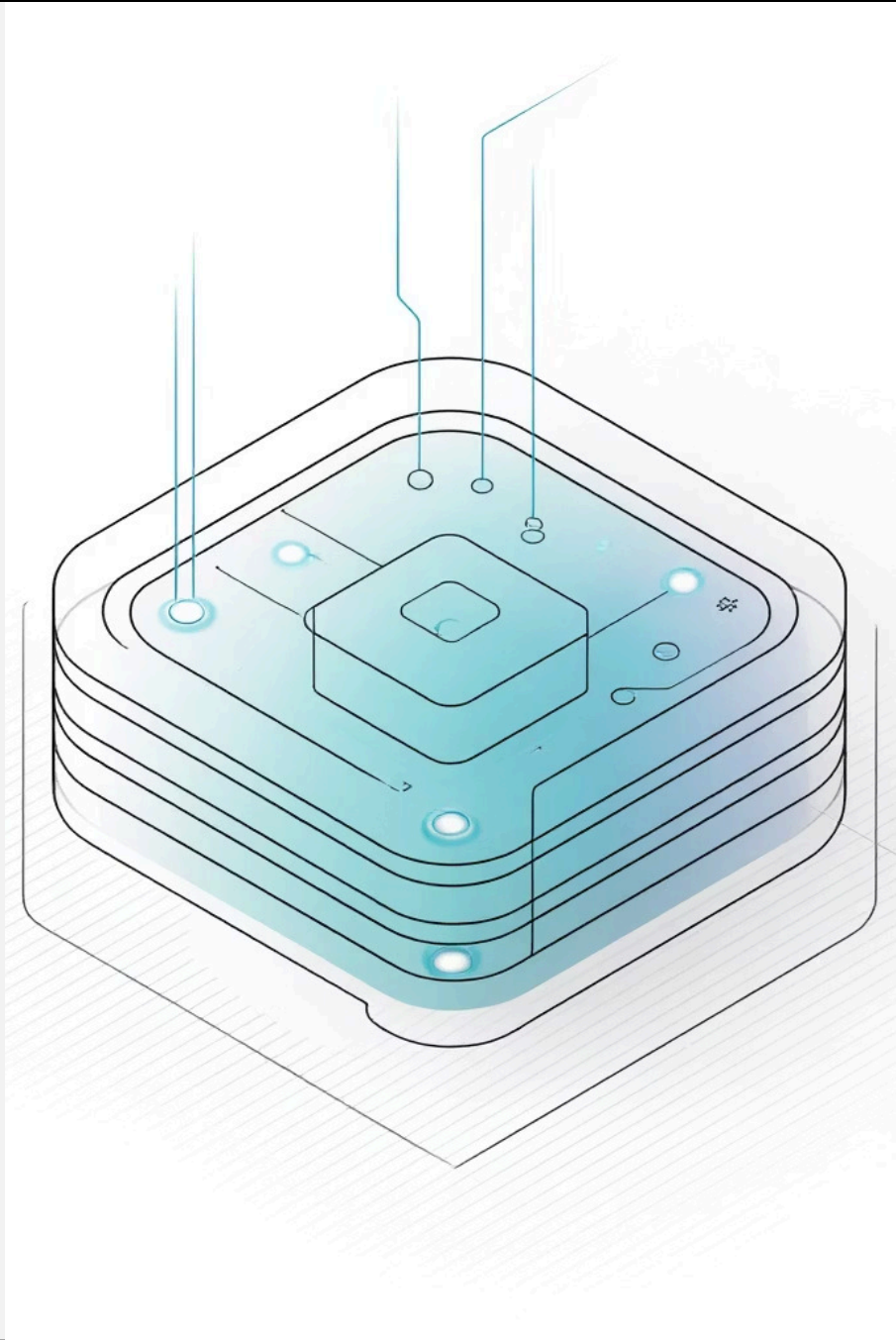
# GA Flowchart




## Algorithm Process Flow

1. **Initialize Population:** Generate random initial solutions with diverse grouping configurations
2. **Evaluate Fitness:** Calculate the maintenance cost for each solution using the fitness function
3. **Check Stopping Criteria:** Determine if termination conditions are met
4. **Apply Elitism:** Identify and preserve the two best solutions from the current generation
5. **Select Parents:** Use linear ranking selection to choose parent solutions for reproduction
6. **Apply Crossover:** Combine parent solutions to create offspring with new grouping arrangements
7. **Apply Mutation:** Introduce random variations to maintain genetic diversity
8. **New Generation:** Form the new generation from elite solutions and offspring
9. **Return Best Solution:** Output the optimal maintenance grouping when algorithm terminates

# Technical details



# GA Encoding

## Solution Representation

In our genetic algorithm implementation, each solution is encoded as an array **S** with **N_PM** elements, where N_PM represents the total number of preventive maintenance activities in the system.

The fundamental grouping rule is elegantly simple: if **S(i) = S(j)**, then activities *i* and *j* belong to the same maintenance group. This encoding scheme provides a compact and efficient representation of any possible grouping arrangement.

## Example Encoding

For a system with 7 maintenance activities (NPM=7), a potential solution might be encoded as:

```
S = [1, 2, 3, 4, 2, 3, 1]
```

This encoding produces 4 distinct groups:

- G1 = {1, 7} (activities 1 and 7 share group label "1")
- G2 = {2, 5} (activities 2 and 5 share group label "2")
- G3 = {3, 6} (activities 3 and 6 share group label "3")
- G4 = {4} (activity 4 has unique group label "4")

This representation allows for any possible grouping configuration while maintaining computational efficiency.

# Initial Population

## Generation Strategy

The genetic algorithm process begins with the generation of an initial population of candidate solutions. For each solution vector **S**, each element is randomly assigned a value between 1 and N_PM (the total number of maintenance activities).

This randomization strategy ensures maximum diversity in the initial population, allowing the algorithm to begin with a wide exploration of the solution space before converging toward optimal groupings.

## Population Size Considerations

The size of the initial population represents an important parameter that influences algorithm performance:

- **Larger populations:** Increase the probability of finding the global optimum by providing greater genetic diversity, but require more computational resources
- **Smaller populations:** Reduce computational overhead but may limit solution diversity and lead to premature convergence

Our implementation dynamically scales population size based on problem complexity, typically using population sizes between 50-200 individuals for maintenance systems of practical scale.

# Fitness Function

Once the groups (G1, ..., Gk) of maintenance activities are achieved, the economic benefit of each group is computed. The fitness of a grouping solution is the sum of the economic benefit of all groups.

## 1

### Objective Function

The primary objective of our genetic algorithm is to minimize the total maintenance cost, which is represented by the fitness function. A lower fitness value indicates a better solution.

$$\text{Fitness}(S) = C_{\text{setup}} \cdot |G(S)| + \sum_{i=1}^{N_c} C_{i,\text{marginal}} + C_{\text{penalty}}(S) + C_{\text{downtime}}(S)$$

## 2

### Cost Components

- **Setup cost:** Fixed cost per maintenance group (C_setup) multiplied by the number of groups |G(S)|
- **Marginal costs:** Variable costs specific to each maintenance activity
- **Penalty costs:** Costs associated with early or late maintenance execution relative to optimal timing
- **Downtime costs:** Production losses due to system unavailability during maintenance

## 3

### Constraint Handling

Constraints are incorporated into the fitness function using penalty terms that significantly increase the fitness value (cost) of infeasible solutions:

- **Hard constraints:** Applied as severe penalties that effectively eliminate infeasible solutions from consideration
- **Soft constraints:** Applied as proportional penalties that guide the algorithm toward more desirable regions of the solution space

# Elitism

Elitism is a crucial strategy in our genetic algorithm implementation that ensures the best solutions discovered are never lost during the evolutionary process. This mechanism guarantees monotonic improvement in solution quality across generations.

In our implementation, the **two best solutions** from each generation are identified based on their fitness values and copied directly into the next generation without undergoing any modification through crossover or mutation operations.

## Benefits of Elitism

**Preservation of high-quality solutions:**

Prevents the loss of excellent candidates that might otherwise be eliminated through random selection

**Acceleration of convergence:**

Ensures the best genetic material remains in the population, potentially speeding up optimization

**Performance guarantee:**

Provides a mathematical guarantee that solution quality will never decrease from one generation to the next

**Stability:**

Reduces algorithm sensitivity to randomization effects that might otherwise cause oscillation in solution quality

While elitism offers these advantages, we carefully limit it to only two solutions to maintain sufficient population diversity and avoid premature convergence to local optima.

# Selection

## Linear Ranking Selection Process

Our algorithm employs a linear ranking selection mechanism that balances selection pressure with population diversity. The selection process begins by sorting the entire population according to their fitness values in ascending order (lower values are better in our minimization problem).

## Group Division and Probability Assignment

After sorting, the population is divided into groups, with each group assigned a different selection probability. Solutions with better fitness values (lower costs) are placed in groups with higher selection probabilities, creating a selection bias toward better solutions while still maintaining some chance for less optimal solutions to be selected.

## Parent Selection Mechanism

Parent solutions are selected based on these group probabilities, with better-performing groups having a higher chance of selection. This approach provides a more nuanced selection pressure than tournament selection or roulette wheel methods, allowing fine-tuning of the exploration-exploitation balance.

This linear ranking selection method creates an effective balance between **exploration** (maintaining genetic diversity by giving some chance to all solutions) and **exploitation** (favoring better solutions to drive convergence toward optimal groupings).

# Crossover

## Genetic Recombination

Crossover is the primary mechanism for generating new candidate solutions by combining genetic material from two parent solutions. This operation mimics biological reproduction and enables the exchange of beneficial grouping structures between different solutions.

In our implementation, we employ one-point crossover with the following procedure:

1. Select two parent solutions from the population using the linear ranking selection method
2. Randomly select a crossover point along the solution vector
3. Create two offspring by swapping the genetic material of the parents at the crossover point

## Example Crossover

```
Parent 1: [1, 2, 3, 4, 2, 3, 1]
Parent 2: [2, 2, 1, 3, 4, 1, 2]
Crossover point: k=3
Child 1: [1, 2, 3, 3, 4, 1, 2]
Child 2: [2, 2, 1, 4, 2, 3, 1]
```

## Group Preservation Challenges

A key challenge in crossover operations for grouping problems is that direct exchange of array elements can disrupt the semantic meaning of groups. For example, a group labeled "2" in Parent 1 may represent a completely different set of activities than a group labeled "2" in Parent 2.

To address this issue, more sophisticated crossover operators have been developed:

- **Group-Preserving Crossover:** Maintains the integrity of key groups during genetic exchange
- **Similarity-Based Crossover:** Identifies and preserves similar grouping patterns across parents
- **Group-Oriented Crossover:** Operates directly on the group structure rather than the array representation

The crossover probability (typically 0.7-0.9) controls how often crossover occurs versus simple copying of parents.

# Mutation

## Random Genetic Variation

Mutation introduces random changes to individual solutions, providing a mechanism to explore new regions of the solution space that might not be accessible through crossover alone. This operation is essential for maintaining genetic diversity and preventing premature convergence to local optima.

In our implementation, mutation operates by randomly selecting an element in the solution vector and changing its value according to the following rule:

$$S'(i) = \text{rand}(1, N_{PM})$$

Where S'(i) is the mutated value and rand(1, N_PM) generates a random integer between 1 and the total number of maintenance activities.

## Mutation Rate Considerations

The mutation rate—the probability of each gene being mutated—is a critical parameter that influences algorithm performance:

- **High mutation rates:** Increase exploration but may disrupt beneficial grouping structures and slow convergence
- **Low mutation rates:** Preserve good solutions but may lead to insufficient diversity and premature convergence

Our implementation uses an adaptive mutation rate that decreases as the algorithm progresses, starting with higher rates (around 0.1) to promote exploration and gradually reducing to lower rates (around 0.01) to refine solutions.

# Stopping Criteria

### Maximum Generations

The algorithm terminates when it reaches a predefined maximum number of generations. This criterion ensures the algorithm completes within a reasonable timeframe, even if convergence is slow.

For maintenance grouping problems, we typically set this limit between 100-500 generations, depending on problem complexity and computational resources available.

### Convergence Detection

The algorithm terminates when the best fitness value remains unchanged for k consecutive generations, indicating that further evolution is unlikely to yield significant improvements.

In our implementation, k is typically set to 20-50 generations, providing sufficient opportunity for the algorithm to escape local optima while still recognizing genuine convergence.

### Computation Time Limit

The algorithm terminates when a predefined maximum computation time is exceeded. This criterion is particularly important for time-sensitive applications where a good solution is needed within a specific timeframe.

For practical maintenance planning applications, we typically set this limit between 5-30 minutes, balancing solution quality with planning efficiency.

These stopping criteria work in parallel, with the algorithm terminating when any one criterion is met. This approach provides flexibility to accommodate various operational constraints and optimization requirements.