Advanced Genetic Algorithm Design for Combinatorial Optimization: A Study of Assignment, Location, and Set Problems

December 5, 2025

1 Foundational Concepts in Evolutionary Computation and Genetic Algorithms

The successful application of Genetic Algorithms (GAs) to solve complex optimization challenges necessitates a rigorous definition of the underlying evolutionary components, specifically focusing on representation, fitness evaluation, and operator design.

1.1 Chromosomal Representation, Genes, and Alleles

In the framework of Evolutionary Algorithms (EAs), a potential solution to an optimization problem is codified as a *chromosome*, often referred to as the genotype. The variables that constitute this solution are termed *genes*, and their possible values are known as *alleles*. The specific position of a gene within the chromosome sequence is designated the *locus*.

Concrete Example - Knapsack Problem:

Consider a knapsack problem with 5 items to choose from. A chromosome might be represented as:

$$\text{Chromosome: } [1, 0, 1, 1, 0]$$

- **Gene 1** (locus 1): Value = 1, meaning Item 1 is included in the knapsack
- **Gene 2** (locus 2): Value = 0, meaning Item 2 is excluded
- **Gene 3** (locus 3): Value = 1, meaning Item 3 is included
- **Gene 4** (locus 4): Value = 1, meaning Item 4 is included
- **Gene 5** (locus 5): Value = 0, meaning Item 5 is excluded
- **Alleles**: In this binary representation, alleles are $\{0, 1\}$
- **Genotype**: The chromosome [1, 0, 1, 1, 0]
- **Phenotype**: The actual solution (Items 1, 3, and 4 in the knapsack)

For a permutation problem (like TSP), a chromosome might be $[3, 1, 4, 2, 5]$, where each gene represents a city and alleles are city indices, with the constraint that each city appears exactly once.

The evaluation of a candidate solution's quality is handled by the objective function, which, in EA terminology, is commonly called the *fitness function*. The evolutionary process is driven by *selection pressure*, which biases the choice of parents toward individuals exhibiting higher fitness. Selection strategies include Proportional Fitness Assignment, which utilizes the absolute fitness value, and Rank-Based Fitness Assignment, which uses the relative rank of an individual within the population. Common selection mechanisms designed to implement this pressure include:

Roulette Wheel Selection: This strategy assigns a probability of selection p_i to each individual proportional to its relative fitness: $p_i = f_i/(\sum_{j=1}^{n} f_j)$. However, exceptionally fit individuals may introduce bias early in the search, potentially leading to premature convergence.

Example: Given a population of 4 individuals with fitness values:

- Individual A: fitness = 10
- Individual B: fitness = 20
- Individual C: fitness = 30
- Individual D: fitness = 40

Total fitness = 100. Selection probabilities:

- $p_A = 10/100 = 0.10$ (10% chance)
- $p_B = 20/100 = 0.20$ (20% chance)
- $p_C = 30/100 = 0.30$ (30% chance)
- $p_D = 40/100 = 0.40$ (40% chance)

Individual D has 4 times the selection probability of Individual A, potentially causing premature convergence if D is selected repeatedly.

Stochastic Universal Sampling (SUS): Developed to mitigate the bias of the Roulette Wheel, SUS uses equally spaced pointers on the fitness wheel, allowing for the simultaneous selection of μ individuals in a single spin, which helps maintain population diversity.

Example: To select 4 individuals from the same population, SUS creates 4 equally-spaced pointers at intervals of 25 (100/4). Starting from a random position (say 5), pointers are at positions 5, 30, 55, 80. This ensures:

- Pointer at 5: selects A
- Pointer at 30: selects C (cumulative ranges: A=0-10, B=10-30, C=30-60)
- Pointer at 55: selects C
- Pointer at 80: selects D (cumulative: D=60-100)

This provides fairer selection compared to spinning the wheel 4 separate times.

Tournament Selection: This robust strategy involves randomly selecting k individuals (the tournament size) and choosing the best among them as the parent. The procedure is repeated μ times to select the required number of parents.

Example: With tournament size $k = 3$:

- Randomly pick individuals B, C, D
- Compare fitness: B=20, C=30, D=40
- Select D (highest fitness) as parent
- Repeat for next parent selection

Tournament selection is widely used because it's simple, doesn't require fitness scaling, and the selection pressure can be easily controlled by adjusting k (larger k = higher pressure).

1.2 The Critical Role of Operator Constraints: Validity, Heritability, and Locality

The effectiveness of reproduction operators—mutation (unary) and crossover (binary)—is contingent upon their ability to maintain crucial characteristics of the search space.

Validity: Operators must strive to produce valid (feasible) solutions. This is particularly challenging for constrained optimization problems, often necessitating specialized operators or external repair mechanisms.

Example - TSP Validity: Consider a 5-city TSP where a valid tour must visit each city exactly once:

- Valid chromosome: $[1, 3, 2, 5, 4]$ (each city appears once)
- Invalid chromosome: $[1, 3, 2, 3, 4]$ (city 3 appears twice, city 5 is missing)

If standard 1-point crossover is applied to two valid TSP tours:

- Parent 1: $[1, 3, 2, 5, 4]$, crossover point after position 2
- Parent 2: $[4, 2, 5, 1, 3]$, same crossover point
- Offspring: $[1, 3|5, 1, 3]$ - INVALID (duplicate 1s and 3s, missing 2 and 4)

This demonstrates why permutation problems require specialized operators that maintain validity.

Heritability: The crossover operator must successfully transmit genetic material from both parents. An operator is deemed *respectful* if common decisions shared by both parents are preserved in the offspring. It is *assorting* if the distance d between the parent and the offspring is less than or equal to the distance between the parents themselves, satisfying $d(p_1, o) \leq d(p_1, p_2)$.

Example - Respectfulness: Consider two binary chromosomes:

- Parent 1: $[1, 0, 1, 1, 0]$
- Parent 2: $[1, 1, 1, 0, 0]$

- Common genes: Positions 1 and 5 have the same values (1 and 0 respectively)

A respectful crossover operator must ensure offspring also have gene 1 = 1 and gene 5 = 0. If an operator produces offspring $[0, 1, 1, 0, 1]$, it violates respectfulness because it changed common genes.

Example - Assorting Property: Using Hamming distance:

- Parent 1: $[1, 0, 1, 1, 0]$
- Parent 2: $[0, 1, 0, 0, 1]$
- Distance $d(P1, P2) = 5$ (all bits differ)
- Offspring: $[1, 0, 0, 0, 1]$
- Distance $d(P1, O) = 2$ (positions 3 and 4 differ)
- Since $2 \leq 5$, the assorting property is satisfied

Locality: Locality ensures that minimal changes in the genotype (the encoded solution) result in minimal changes in the phenotype (the solution quality). Poor adherence to this principle, known as weak locality, results in highly disruptive mutations or crossovers that generate low-quality solutions from high-quality parents, potentially making the search inefficient.

Example - Good Locality: In a binary knapsack problem:

- Parent: $[1, 0, 1, 1, 0]$, fitness = 85
- Mutate bit 2: $[1, 1, 1, 1, 0]$, fitness = 82 (small change in genotype $\rightarrow$ small change in fitness)
- This exhibits good locality

Example - Weak Locality: In a poorly encoded TSP:

- Parent: $[1, 2, 3, 4, 5]$, tour length = 100
- Swap positions 2 and 3: $[1, 3, 2, 4, 5]$, tour length = 250 (small genotype change $\rightarrow$ large fitness change)
- This exhibits weak locality, making the search landscape rugged and difficult to navigate

Good locality is crucial for efficient optimization because it allows the GA to make incremental improvements rather than random jumps in solution quality.

2 Comprehensive Analysis of Genetic Crossover Operators and Child Extraction

Crossover operators are differentiated by the data representation they manipulate. While standard methods suffice for binary or discrete representations, permutation-based problems require specialized operators to ensure solution validity.

2.1 Standard Crossover Mechanisms (Binary and Discrete Representations)

Standard crossovers, such as n-point and uniform crossover, are typically applied to chromosomes represented by binary strings or discrete value vectors, where the position of an allele does not necessarily impose strict ordering constraints (e.g., in facility selection vectors).

2.1.1 1-Point, 2-Point, and N-Point Crossover

These methods define crossover sites that dictate where the exchange of genetic material occurs. For 1-point crossover, a single site is randomly selected.

Child Extraction Example (1-Point Crossover):

If Parent 1 (P1) is 100111001001 and Parent 2 (P2) is 011100100111, and the crossover site is after the 9th bit:

- P1 Head: 100111001 — P1 Tail: 001
- P2 Head: 011100100 — P2 Tail: 111
- Offspring 1 (O1): P1 Head + P2 Tail $\rightarrow$ 100111001111
- Offspring 2 (O2): P2 Head + P1 Tail $\rightarrow$ 011100100001

2.1.2 Uniform Crossover (U-X)

Uniform crossover achieves maximum mixing by selecting the source parent for each element (gene) independently and randomly, often guided by a binary mask.

Child Extraction Example (Uniform Crossover):

Given:

- Parent 1 (P1): $[1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1]$
- Parent 2 (P2): $[0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0]$
- Random Mask: $[1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0]$ (1 = take from P1, 0 = take from P2)

Step-by-step construction of Offspring:

- Position 1: Mask=1 $\rightarrow$ take from P1 $\rightarrow$ 1
- Position 2: Mask=1 $\rightarrow$ take from P1 $\rightarrow$ 0
- Position 3: Mask=0 $\rightarrow$ take from P2 $\rightarrow$ 1
- Position 4: Mask=0 $\rightarrow$ take from P2 $\rightarrow$ 0
- Position 5: Mask=1 $\rightarrow$ take from P1 $\rightarrow$ 1
- Position 6: Mask=0 $\rightarrow$ take from P2 $\rightarrow$ 0
- Position 7: Mask=1 $\rightarrow$ take from P1 $\rightarrow$ 0
- Position 8: Mask=0 $\rightarrow$ take from P2 $\rightarrow$ 1

- Position 9: Mask=1 $\rightarrow$ take from P1 $\rightarrow$ 0
- Position 10: Mask=1 $\rightarrow$ take from P1 $\rightarrow$ 1
- Position 11: Mask=0 $\rightarrow$ take from P2 $\rightarrow$ 0
- Position 12: Mask=0 $\rightarrow$ take from P2 $\rightarrow$ 0

Resulting Offspring: $[1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0]$

Uniform crossover maximizes disruption (high exploration) but may break beneficial building blocks that span multiple adjacent genes. It's useful when there's no strong positional linkage between genes, but potentially combining distant beneficial features from both parents.

2.2 Specialized Crossover Operators for Permutation Problems

When the chromosome represents a permutation (e.g., the assignment order in QAP), standard crossover fails because it produces illegal solutions containing duplicates and missing elements. Specialized permutation operators are mandatory to maintain feasibility.

2.2.1 Order Crossover (OX)

Order Crossover (OX) preserves a contiguous section of the first parent and maintains the relative ordering of the remaining elements from the second parent. This is crucial for permutation problems where each element must appear exactly once.

Child Extraction Example (Order Crossover):
Given:

- P1: $[A, B, C, D, E, F, G, H, I]$
- P2: $[C, D, E, F, b, g, h, a, i]$ (using lowercase for P2 for clarity)
- Two crossover points randomly selected: positions 3-6 (delimiting segment C, D, E, F)

Step 1: Copy the selected segment from P1 to offspring:

$$\text{Offspring: } [\_, \_, C, D, E, F, \_, \_, \_]$$

Step 2: Identify elements from P2 not in the copied segment:

- P2 elements: $[C, D, E, F, b, g, h, a, i]$
- Already in offspring: C, D, E, F
- Remaining from P2 (in order): $[b, g, h, a, i]$

Step 3: Fill empty positions starting after the segment and wrapping around:

- Position 7: b
- Position 8: g
- Position 9: h

- Position 1 (wrap): a
- Position 2 (wrap): i

Final Offspring: $[a, i, C, D, E, F, b, g, h]$
Key Properties:

- Preserves the subsequence $CDEF$ from P1 in its original relative positions
- Maintains the relative order of remaining elements from P2 (b comes before g, which comes before h, etc.)
- Guarantees a valid permutation (no duplicates, no missing elements)
- Useful when the absolute position of a contiguous segment is important

2.2.2 Partially Matched Crossover (PMX)

PMX uses a matching section to establish a positional mapping that resolves conflicts arising from copying genes from both parents, ensuring the resulting chromosome remains a valid permutation. It preserves both absolute positions and relationships from both parents.

Child Extraction Example (PMX):
Given:

- P1: $[9, 8, 4, |5, 6, 7|1, 3, 2, 10]$
- P2: $[8, 7, 1, |2, 3, 10|9, 5, 4, 6]$
- Matching section (crossover points): positions 4-6

Step 1: Copy matching sections directly:

- Offspring 1: $[\_, \_, \_, |5, 6, 7|\_, \_, \_, \_]$ (from P1)
- Offspring 2: $[\_, \_, \_, |2, 3, 10|\_, \_, \_, \_]$ (from P2)

Step 2: Establish positional mapping from matching sections:

- Position 4: $5 \leftrightarrow 2$
- Position 5: $6 \leftrightarrow 3$
- Position 6: $7 \leftrightarrow 10$

Step 3: Fill Offspring 1 using elements from P2's external positions:
Try to copy from P2's position 1 (value = 8):

- 8 is not in O1's matching section $\rightarrow$ place at position 1: $[8, \_, \_, 5, 6, 7, \_, \_, \_, \_]$

Try P2's position 2 (value = 7):

- 7 is already in O1 (matching section) $\rightarrow$ use mapping: $7 \leftrightarrow 10$
- Try to place 10 at position 2, but 10 maps to $10 \leftrightarrow 7$ (already in O1)

- Follow chain: $7 \leftrightarrow 10$, and 10 has no further mapping $\rightarrow$ Cannot place directly
- Use P1's value at position 2 instead: $[8, 8, \_, 5, 6, 7, \_, \_, \_, \_]$ - conflict!
- Actually, place 10 at position 2: $[8, 10, \_, 5, 6, 7, \_, \_, \_, \_]$ (10 maps from 7)

Try P2's position 3 (value = 1):

- 1 is not in matching section $\rightarrow$ place at position 3: $[8, 10, 1, 5, 6, 7, \_, \_, \_, \_]$

Continue for remaining positions 7-10 using P2 values: 9, 5, 4, 6

- Position 7: Try 9 $\rightarrow$ not in matching $\rightarrow$ place 9: $[8, 10, 1, 5, 6, 7, 9, \_, \_, \_]$
- Position 8: Try 5 $\rightarrow$ already in matching $\rightarrow$ use mapping $5 \leftrightarrow 2 \rightarrow$ place 2: $[8, 10, 1, 5, 6, 7, 9, 2, \_, \_]$
- Position 9: Try 4 $\rightarrow$ not in matching $\rightarrow$ place 4: $[8, 10, 1, 5, 6, 7, 9, 2, 4, \_]$
- Position 10: Try 6 $\rightarrow$ already in matching $\rightarrow$ use mapping $6 \leftrightarrow 3 \rightarrow$ place 3: $[8, 10, 1, 5, 6, 7, 9, 2, 4, 3]$

Resulting Offspring 1: $[8, 10, 1, 5, 6, 7, 9, 2, 4, 3]$ - but this needs verification.
Correct Simplified Result: $[9, 8, 4, 2, 3, 10, 1, 6, 5, 7]$
Key Properties:

- Preserves absolute positions from both parents when possible
- The mapping ensures no duplicates through conflict resolution
- More complex than OX but can better preserve positional information
- Useful when absolute position of elements matters (like in QAP)

It is important to note that while these specialized operators guarantee validity for permutation problems, the intricate nature of the mapping and re-insertion steps (as seen in PMX and OX) can be highly disruptive, meaning that a small change in parental input can lead to a large rearrangement in the offspring. This structural observation suggests that even structurally correct permutation operators can exhibit weak locality, necessitating the combination of GAs with local search procedures to refine the highly diversified, yet feasible, solutions they produce.

2.3 Mutation Operators and Their Role

Mutation is a unary operator that introduces small random changes to maintain genetic diversity and prevent premature convergence. The mutation rate p_m typically ranges from 0.001 to 0.01.

2.3.1 Mutation for Binary Representations

Bit Flip Mutation:
Example: Given chromosome $[1, 0, 1, 1, 0, 1, 0, 0]$ and $p_m = 0.125$ (mutate 1 in 8 genes on average):

- Gene 1: Random(0,1) = $0.05 < 0.125 \rightarrow$ Flip: $1 \rightarrow 0$
- Gene 2: Random(0,1) = $0.82 > 0.125 \rightarrow$ Keep: 0
- Gene 3: Random(0,1) = $0.95 > 0.125 \rightarrow$ Keep: 1
- Genes 4-8: No mutations (random values > threshold)
- Mutated: $[0, 0, 1, 1, 0, 1, 0, 0]$

2.3.2 Mutation for Permutations

Swap Mutation: Exchange two randomly selected positions.
Example: $[A, B, C, D, E, F]$ with positions 2 and 5 selected:

- Original: $[A, B, C, D, E, F]$
- Swap positions 2 and 5: $[A, E, C, D, B, F]$

Inversion Mutation: Reverse a subsequence between two random points.
Example: $[A, B, C, D, E, F]$ with points 2-5:

- Original: $[A, |B, C, D, E|F]$
- Reverse segment: $[A, E, D, C, B, F]$

Insertion Mutation: Remove an element and insert it at a different position.
Example: $[A, B, C, D, E, F]$ remove position 2, insert before position 5:

- Remove B: $[A, C, D, E, F]$
- Insert B before position 5: $[A, C, D, E, B, F]$

2.3.3 Mutation for Real-Valued Representations

Gaussian Mutation: Add random noise from normal distribution.
Example: $x = [2.5, 3.7, 1.2]$ with $\sigma = 0.1$:

- $x'_1 = 2.5 + N(0, 0.1) = 2.5 + 0.08 = 2.58$
- $x'_2 = 3.7 + N(0, 0.1) = 3.7 - 0.05 = 3.65$
- $x'_3 = 1.2 + N(0, 0.1) = 1.2 + 0.12 = 1.32$
- Mutated: $[2.58, 3.65, 1.32]$

2.4 Complete GA Workflow Example

To consolidate understanding, here's a complete mini-example of one GA generation:

Problem: Maximize $f(x) = x^2$ for $x \in [0, 31]$ (5-bit encoding)

Generation t:

Individual	Binary	x value	Fitness
A	01101	13	169
B	11000	24	576
C	01000	8	64
D	10011	19	361

Step 1 - Selection (Tournament, k=2):

- Tournament 1: Compare A(169) vs C(64) $\rightarrow$ Select A
- Tournament 2: Compare B(576) vs D(361) $\rightarrow$ Select B

Parents: A and B

Step 2 - Crossover (1-point at position 3, $P_c = 0.8$):

- Random(0,1) = 0.65 $<$ 0.8 $\rightarrow$ Apply crossover
- P1 (A): 011|01
- P2 (B): 110|00
- Offspring 1: 011|00 = 01100 (x=12, f=144)
- Offspring 2: 110|01 = 11001 (x=25, f=625)

Step 3 - Mutation ($p_m = 0.05$ per bit):

- O1: $[0, 1, 1, 0, 0]$ - bit 3 mutates $\rightarrow$ $[0, 1, 0, 0, 0] = 01000$ (x=8, f=64)
- O2: $[1, 1, 0, 0, 1]$ - no mutations $\rightarrow$ $[1, 1, 0, 0, 1] = 11001$ (x=25, f=625)

Step 4 - Replacement (Elitism, keep 4 best):

- Combine: A(169), B(576), C(64), D(361), O1(64), O2(625)
- Sort by fitness: O2(625) $>$ B(576) $>$ D(361) $>$ A(169) $>$ C(64) = O1(64)
- New population: O2, B, D, A

Result: Best fitness improved from 576 to 625 in one generation!

Table 1: Comparison of Crossover and Mutation Operators

Operator	**Purpose**	**Typical Rate**	**Best Used When**
1-Point Crossover	Exploit building blocks	$P_c = 0.6 - 0.9$	Schema have low defining length
Uniform Crossover	Maximum exploration	$P_c = 0.5 - 0.7$	No positional linkage between genes
OX Crossover	Preserve order	$P_c = 0.7 - 0.9$	Relative order matters (TSP, scheduling)
PMX Crossover	Preserve position	$P_c = 0.7 - 0.9$	Absolute position matters (QAP)
Bit Flip Mutation	Maintain diversity	$p_m = 0.001 - 0.01$	Binary/discrete representations
Swap Mutation	Small permutation change	$p_m = 0.01 - 0.1$	Permutation problems
Gaussian Mutation	Local refinement	$\sigma = 0.05 - 0.2$	Real-valued continuous optimization

3 GA Application to Assignment and Location Problems

3.1 The Quadratic Assignment Problem (QAP)

3.1.1 Problem Explanation and Common Applications

The Quadratic Assignment Problem (QAP) is a foundational, NP-hard combinatorial optimization problem categorized under facility location problems. It involves assigning n facilities to n unique locations in a one-to-one mapping. The cost function is defined by two input matrices: a distance matrix D between locations and a flow (or weight) matrix W representing the interaction between facilities. The goal is to find an assignment (a permutation π) that minimizes the total sum of the product of flow and distance for all pairs of facilities.

Mathematical Formulation: Given:

- n facilities and n locations
- Distance matrix D: where d_{ij} is the distance between location i and location j
- Flow matrix W: where w_{ab} is the flow/interaction between facility a and facility b
- Permutation π: where $\pi(a)$ indicates the location assigned to facility a

The cost function is inherently quadratic because it relies on the product of two binary assignment decisions, $x_{ij}x_{kl}$, defining the cost structure.

Concrete Example - Hospital Department Layout:

Consider a hospital with 4 departments (Surgery, Emergency, Radiology, Pharmacy) that must be assigned to 4 available locations (Wings A, B, C, D). The flow matrix represents patient/staff movement between departments per day:

$$W = \begin{bmatrix} 0 & 50 & 30 & 20 \\ 50 & 0 & 40 & 25 \\ 30 & 40 & 0 & 15 \\ 20 & 25 & 15 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 0 & 10 & 20 & 30 \\ 10 & 0 & 15 & 25 \\ 20 & 15 & 0 & 10 \\ 30 & 25 & 10 & 0 \end{bmatrix}$$

If Surgery is assigned to Wing A and Emergency to Wing B, the cost contribution from their interaction is $w_{Surgery,Emergency} \times d_{A,B} = 50 \times 10 = 500$ travel-distance units. The QAP seeks the assignment that minimizes the total travel-distance across all department pairs.

Common applications include:

- *Manufacturing*: Optimizing factory floor layouts to minimize material handling costs between workstations
- *Campus Planning*: Positioning university departments to minimize student walking distances
- *Electronic Design*: Placing circuit components on a chip to minimize wire length
- *Keyboard Layout*: Arranging keys to minimize finger travel distance for common letter pairs

3.1.2 Solution Representation, Objective Function, Crossover, and Mutation

Solution Representation: QAP is naturally represented using permutation encoding. A chromosome of length n represents the assignment, where the i-th gene indicates the facility assigned to location i.

Objective Function (Fitness): The fitness function is the direct minimization of the total assignment cost:

$$\text{Minimize } C(\pi) = \sum_{a,b} w(a,b) \cdot d(\pi(a), \pi(b)) \tag{1}$$

where $w(a,b)$ is the flow between facilities a and b, and $d(\pi(a), \pi(b))$ is the distance between their assigned locations.

Crossover and Mutation: Since the solution must be a valid permutation, standard crossover operators are inadequate. The GA must employ permutation-specific operators such as Order Crossover (OX) or Partially Matched Crossover (PMX), as these mechanisms are designed to preserve validity. Mutation often involves permutation operators like swapping, inversion, or insertion (related to 2-opt local search).

Solving QAP with GA: Due to the difficulty and size of QAP instances (up to $n = 729$ in some studies), highly effective solutions are usually achieved through Hybrid Genetic Algorithms (HGAs). These HGAs couple the GA's global search capability with powerful local search procedures (e.g., Tabu Search) to efficiently balance diversification and intensification, enabling the discovery of (pseudo-)optimal solutions for small- and medium-sized instances.

Practical Implementation Tips for QAP:

1. *Initialization*: Use a mix of random and greedy solutions (e.g., 70% random, 30% greedy) to balance diversity and quality

2. *Local Search*: Apply 2-opt or 3-opt local search to each offspring before evaluation

3. *Distance Preservation*: Maintain population diversity by rejecting solutions too similar to existing ones (Hamming distance threshold)

4. *Adaptive Parameters*: Reduce mutation rate as search progresses (e.g., $p_m = 0.05$ initially, decrease to 0.01)

5. *Restart Strategy*: If no improvement for 100 generations, restart with new random population while keeping best solution

Common Pitfalls:

- *Premature Convergence*: All individuals become too similar. Solution: Increase population size or use fitness sharing
- *Invalid Solutions*: Standard crossovers create duplicates. Solution: Always use OX or PMX for permutations
- *Slow Convergence*: Pure GA explores too broadly. Solution: Hybridize with local search (2-opt, Tabu Search)
- *Poor Initial Solutions*: Starting with bad quality. Solution: Seed population with greedy heuristic solutions

3.2 The Facility Location Problem (FLP)

3.2.1 Problem Explanation and Common Applications

The Facility Location Problem (FLP) encompasses several models, such as the p-median problem, where the objective is to determine the optimal subset of p locations (facilities) to open from n candidate sites to serve a set of m demand nodes. Unlike QAP, FLP costs typically stem from the service distance between the selected facilities and external demand points (customers), rather than the interaction between facilities themselves. The Uncapacitated Facility Location Problem (UFLP) is a common variant where facilities have no capacity constraints.

Mathematical Formulation: Given:

- n candidate facility locations
- m demand points (customers) with demands d_i
- Fixed cost f_j for opening facility at location j
- Service cost c_{ij} for serving demand point i from facility j
- Decision variable $x_j \in \{0, 1\}$: 1 if facility j is opened, 0 otherwise
- Decision variable $y_{ij} \in \{0, 1\}$: 1 if customer i is served by facility j

Objective: Minimize total cost $= \sum_{j=1}^{n} f_j x_j + \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} y_{ij}$

Concrete Example - Emergency Response Centers:

A city needs to select 2 fire stations from 5 candidate locations to serve 8 neighborhoods. Each candidate location has:

- Opening cost (building/staffing): Location 1: \$500K, Location 2: \$450K, Location 3: \$600K, etc.
- Response times to each neighborhood (the service cost)

For instance:

- If Station 1 is opened, it can reach Neighborhood A in 3 minutes, Neighborhood B in 7 minutes
- If Station 2 is opened, it reaches Neighborhood A in 5 minutes, Neighborhood B in 2 minutes

The FLP determines which 2 locations to open and which station serves each neighborhood to minimize total cost (opening costs + weighted response times). Note that multiple neighborhoods may be served by the same station, and the goal is external service, not inter-facility interaction.

Applications are diverse:

- *Supply Chain*: Determining optimal warehouse locations to serve retail stores
- *Healthcare*: Locating vaccination centers to serve population centers during pandemics
- *Telecommunications*: Placing cell towers to provide coverage to users
- *Retail*: Selecting store locations to maximize market coverage while minimizing costs
- *Emergency Services*: Positioning ambulance stations to minimize response times

3.2.2 Solution Representation, Objective Function, Crossover, and Mutation

Solution Representation: FLP, especially the p-median and UFLP variants, is commonly represented by a binary string or direct value coding. A chromosome of length n (the number of potential locations) uses a binary allele $x_j \in \{0, 1\}$ to indicate whether a facility is opened at location j ($x_j = 1$).

Concrete Representation Example for FLP:

Consider a facility location problem with 8 potential warehouse locations. A chromosome might look like:

$$\text{Chromosome: } [1, 0, 1, 0, 0, 1, 0, 1]$$

- **Gene 1**: Value = 1 $\rightarrow$ Open warehouse at Location 1
- **Gene 2**: Value = 0 $\rightarrow$ Location 2 not selected
- **Gene 3**: Value = 1 $\rightarrow$ Open warehouse at Location 3
- **Gene 6**: Value = 1 $\rightarrow$ Open warehouse at Location 6
- **Gene 8**: Value = 1 $\rightarrow$ Open warehouse at Location 8
- **Interpretation**: 4 warehouses opened (locations 1, 3, 6, 8)

- **Alleles**: $\{0, 1\}$ where 0 = closed, 1 = open

Objective Function (Fitness): The fitness function directly minimizes the total cost, which usually includes the fixed costs of opening the chosen facilities plus the variable costs of serving all demand from the nearest open facility.

Fitness Calculation Example:

Given the chromosome $[1, 0, 1, 0, 0, 1, 0, 1]$:

- Fixed costs: $f_1 = \$500K$, $f_3 = \$450K$, $f_6 = \$600K$, $f_8 = \$550K$
- Total fixed cost: $500 + 450 + 600 + 550 = \$2100K$
- Service costs: Each of 20 customers assigned to nearest open facility
- Customer 1 nearest to Location 3 (distance 5km, cost = $5 \times 10 = \$50K$)
- Customer 2 nearest to Location 1 (distance 3km, cost = $3 \times 10 = \$30K$)
- ... (sum all 20 customers)
- Total service cost: $\$800K$
- **Total Fitness** (minimize): $2100 + 800 = \$2900K$

Crossover for FLP - Detailed Examples:

Since the chromosome is a binary/discrete vector, standard operators are effective. Studies have shown that the two-point crossover operator, and variations that randomly alternate between one-point and two-point crossover, perform well for FLP representations.

1-Point Crossover Example for FLP:

Given two parent solutions:

- Parent 1: $[1, 0, 1, 0, |0, 1, 0, 1]$ (crossover point after position 4)
- Parent 2: $[0, 1, 0, 1, |1, 0, 1, 0]$ (same crossover point)

Step-by-step Crossover Process:

1. Copy P1 head to Offspring 1: $[1, 0, 1, 0, \_, \_, \_, \_]$
2. Copy P2 tail to Offspring 1: $[1, 0, 1, 0, 1, 0, 1, 0]$
3. Copy P2 head to Offspring 2: $[0, 1, 0, 1, \_, \_, \_, \_]$
4. Copy P1 tail to Offspring 2: $[0, 1, 0, 1, 0, 1, 0, 1]$

Results:

- Offspring 1: $[1, 0, 1, 0, 1, 0, 1, 0]$ - Opens locations 1, 3, 5, 7 (4 facilities)
- Offspring 2: $[0, 1, 0, 1, 0, 1, 0, 1]$ - Opens locations 2, 4, 6, 8 (4 facilities)

2-Point Crossover Example for FLP:

Given:

- Parent 1: $[1, 0, |1, 0, 0, 1|0, 1]$ (crossover points at positions 2 and 6)

- Parent 2: $[0, 1, |0, 1, 1, 0|1, 0]$

Step-by-step Process:

1. Copy P1 segments outside crossover region to O1: $[1, 0, \_, \_, \_, \_, 0, 1]$
2. Copy P2 middle segment to O1: $[1, 0, 0, 1, 1, 0, 0, 1]$
3. Copy P2 segments outside crossover region to O2: $[0, 1, \_, \_, \_, \_, 1, 0]$
4. Copy P1 middle segment to O2: $[0, 1, 1, 0, 0, 1, 1, 0]$

Results:

- Offspring 1: $[1, 0, 0, 1, 1, 0, 0, 1]$ - Opens locations 1, 4, 5, 8 (4 facilities)
- Offspring 2: $[0, 1, 1, 0, 0, 1, 1, 0]$ - Opens locations 2, 3, 6, 7 (4 facilities)

Uniform Crossover Example for FLP:
Given:

- Parent 1: $[1, 0, 1, 0, 0, 1, 0, 1]$
- Parent 2: $[0, 1, 0, 1, 1, 0, 1, 0]$
- Random Mask: $[1, 0, 1, 1, 0, 0, 1, 0]$ (1 = from P1, 0 = from P2)

Step-by-step Construction:

- Position 1: Mask=1 $\rightarrow$ take from P1 $\rightarrow$ 1
- Position 2: Mask=0 $\rightarrow$ take from P2 $\rightarrow$ 1
- Position 3: Mask=1 $\rightarrow$ take from P1 $\rightarrow$ 1
- Position 4: Mask=1 $\rightarrow$ take from P1 $\rightarrow$ 0
- Position 5: Mask=0 $\rightarrow$ take from P2 $\rightarrow$ 1
- Position 6: Mask=0 $\rightarrow$ take from P2 $\rightarrow$ 0
- Position 7: Mask=1 $\rightarrow$ take from P1 $\rightarrow$ 0
- Position 8: Mask=0 $\rightarrow$ take from P2 $\rightarrow$ 0

Result: Offspring = $[1, 1, 1, 0, 1, 0, 0, 0]$ - Opens locations 1, 2, 3, 5 (4 facilities)
Mutation for FLP - Bit Flip Mutation:
Mutation involves flipping a gene (changing 0 to 1 or vice versa), representing the opening or closing of a potential site.
Bit Flip Mutation Example:
Given chromosome $[1, 0, 1, 0, 0, 1, 0, 1]$ and mutation rate $p_m = 0.125$ (mutate 1 in 8 genes on average):
Step-by-step Mutation Process:

- Gene 1: Random(0,1) = $0.45 > 0.125 \rightarrow$ Keep: 1

- Gene 2: Random(0,1) = 0.89 > 0.125 → Keep: 0
- Gene 3: Random(0,1) = 0.08 < 0.125 → **Flip**: 1 → 0
- Gene 4: Random(0,1) = 0.72 > 0.125 → Keep: 0
- Gene 5: Random(0,1) = 0.93 > 0.125 → Keep: 0
- Gene 6: Random(0,1) = 0.11 < 0.125 → **Flip**: 1 → 0
- Gene 7: Random(0,1) = 0.55 > 0.125 → Keep: 0
- Gene 8: Random(0,1) = 0.24 > 0.125 → Keep: 1

Result:

- Original: $[1, 0, 1, 0, 0, 1, 0, 1]$ - 4 facilities (locations 1, 3, 6, 8)
- Mutated: $[1, 0, 0, 0, 0, 0, 0, 1]$ - 2 facilities (locations 1, 8)
- *Impact*: Closed locations 3 and 6; may reduce fixed costs but increase service costs

Constraint Handling: If the specific FLP variant includes constraints, such as the p-median constraint requiring exactly p facilities to be open, the GA must include mechanisms (penalties or repair procedures) to ensure $|X| = p$.

Constraint Repair Example for p-Median Problem:

Suppose we need exactly $p = 4$ facilities open, but after crossover/mutation we get:

$$\text{Chromosome: } [1, 0, 0, 0, 0, 0, 0, 1] \quad \text{(only 2 facilities)}$$

Repair Procedure:

1. Count open facilities: $|X| = 2 < 4$ (need to open 2 more)
2. Calculate service benefit of each closed location
3. Open the 2 closed locations with highest service benefit
4. Suppose Locations 3 and 5 have highest benefits
5. Repaired chromosome: $[1, 0, 1, 0, 1, 0, 0, 1]$ - Now exactly 4 facilities

Alternatively, use **penalty function**:

$$\text{Fitness} = \text{TotalCost} + \lambda \times |\text{NumOpen} - p|^2$$

For the infeasible solution with 2 facilities when $p = 4$:

$$\text{Penalty} = \lambda \times |2 - 4|^2 = \lambda \times 4$$

This makes infeasible solutions less attractive, guiding the search toward exactly p open facilities.

4 GA Application to Set Problems: Covering and Partitioning

The Set Problems (SCP and SPP) represent highly constrained resource allocation models that challenge GA feasibility maintenance, leading to the adoption of hybridized and indirect approaches.

4.1 The Set Covering Problem (SCP)

4.1.1 Problem Explanation and Common Applications

The Set Covering Problem (SCP) involves selecting a minimum-cost subset of columns (resources/sets) from a binary matrix such that every row (requirement/element) is covered by at least one selected column. Each column has an associated cost, and a column "covers" a row if the corresponding matrix entry is 1.

Mathematical Formulation: Given:

- Binary matrix $A = [a_{ij}]$ where $a_{ij} = 1$ if column j covers row i
- Cost vector $c = [c_1, c_2, \ldots, c_n]$ where c_j is the cost of selecting column j
- Decision variable $x_j \in \{0, 1\}$: 1 if column j is selected, 0 otherwise

Objective: Minimize $\sum_{j=1}^{n} c_j x_j$

Constraint: $\sum_{j=1}^{n} a_{ij} x_j \geq 1$ for all rows i (each requirement must be covered at least once)

Structure: SCP allows for redundancy; a row can be covered multiple times by different columns. This is key to differentiating it from SPP. SCP is strongly NP-hard.

Concrete Example - Security Camera Placement:

A museum has 6 rooms that need surveillance coverage. There are 4 potential camera locations, each with different coverage and costs:

Room	Cam 1	Cam 2	Cam 3	Cam 4	Must Cover
Room A	1	0	1	0	≥ 1
Room B	1	1	0	0	≥ 1
Room C	0	1	1	0	≥ 1
Room D	0	1	0	1	≥ 1
Room E	0	0	1	1	≥ 1
Room F	1	0	0	1	≥ 1
Cost	\$300	\$250	\$200	\$280	

Camera 1 covers rooms A, B, and F (cost \$300). Camera 2 covers B, C, and D (cost \$250). The goal is to select the minimum-cost subset of cameras ensuring every room is monitored by at least one camera. Note that if both Camera 1 and Camera 2 are selected, Room B is covered twice—this redundancy is acceptable in SCP.

Applications:

- *Crew Scheduling*: Selecting shifts that cover all required time slots (redundancy acceptable)
- *Service Coverage*: Placing facilities to ensure all areas are within service range

- *Network Design*: Selecting nodes to monitor all network links
- *Sensor Placement*: Positioning sensors to detect all potential events
- *Boolean Satisfiability*: Clause satisfaction where multiple literals can satisfy a clause

4.1.2 Solution Representation, Objective Function, Crossover, and Mutation

Solution Representation: SCP is formally solved using a binary solution vector x_j, but applying standard GA operators directly to this binary vector often yields uncovered rows (infeasible solutions). Consequently, many high-performing GAs for SCP utilize an Indirect Genetic Algorithm approach. The chromosome in the indirect GA does not represent the solution itself but rather a permutation of solution variables or a parameter set that guides an external decoder.

Indirect Encoding Example for SCP:

Consider the security camera problem with 4 cameras. Instead of encoding the solution directly as $[x_1, x_2, x_3, x_4]$ where $x_j \in \{0, 1\}$, we use an indirect approach:

Direct Binary Encoding (problematic):

$$\text{Chromosome: } [1, 0, 1, 1] \quad \text{(select cameras 1, 3, 4)}$$

Problem: Standard crossover often creates infeasible solutions (e.g., some rooms uncovered)

Indirect Permutation Encoding (recommended):

$$\text{Chromosome: } [3, 1, 4, 2] \quad \text{(priority order for selection)}$$

Interpretation:

- This is not the solution itself, but a *construction order*
- The decoder will use this order to build a feasible solution
- Camera 3 has highest priority, Camera 1 second, Camera 4 third, Camera 2 lowest
- **Genotype**: $[3, 1, 4, 2]$ (the permutation)
- **Phenotype**: Actual binary solution $[1, 0, 1, 1]$ produced by decoder

Why Indirect Encoding Works:

- Permutation operators (OX, PMX, PUX) always produce valid permutations
- The decoder handles feasibility (ensuring all rooms covered)
- GA explores the space of construction priorities, not solutions directly
- Separates "how to search" (GA) from "how to repair" (decoder)

Objective Function (Fitness): The objective is cost minimization, Minimize $\sum c_j x_j$. Since the GA generates abstract genotypes, the fitness is evaluated after the genotype has been translated into a feasible binary solution (phenotype) by the decoder.

Fitness Evaluation Process:

1. GA generates genotype: $[3, 1, 4, 2]$
2. Decoder converts to phenotype: $[1, 0, 1, 1]$ (cameras 1, 3, 4 selected)
3. Calculate cost: $c_1 + c_3 + c_4 = 300 + 200 + 280 = \$780$
4. Verify feasibility: All 6 rooms covered? Yes
5. Return fitness: 780 (minimize)

Crossover for SCP - Permutation Uniform-like Crossover (PUX):
The operators act on the permutation encoding (or other indirect encoding). Robust permutation operators like the permutation uniform-like crossover (PUX) are commonly employed.
PUX Crossover Example:
Given two parent permutations (priority orders):

- Parent 1: $[3, 1, 4, 2]$
- Parent 2: $[2, 4, 1, 3]$
- Random Mask: $[1, 0, 1, 0]$ (1 = from P1, 0 = from P2)

PUX Process (maintains permutation validity):

1. Select elements from P1 where mask=1: positions 1 and 3 $\rightarrow$ elements [3, 4]
2. Place in offspring at same positions: $[3, \_, 4, \_]$
3. Get remaining elements from P2 in order: P2 = [2, 4, 1, 3], remove 3 and 4 $\rightarrow$ [2, 1]
4. Fill empty positions with [2, 1]: $[3, 2, 4, 1]$

Result: Offspring = $[3, 2, 4, 1]$ - a valid permutation
Alternative: Order Crossover (OX) for SCP:
Given:

- Parent 1: $[3, 1, |4, 2|]$ (crossover region: positions 3-4)
- Parent 2: $[2, 4, |1, 3|]$

OX Steps:

1. Copy segment from P1: $[\_, \_, 4, 2]$
2. Elements in P2 not in segment: [2, 4, 1, 3] - remove [4, 2] = [1, 3]
3. Fill from position 1 (after segment, wrapping): $[1, 3, 4, 2]$

Result: Offspring = $[1, 3, 4, 2]$ - represents new priority order
Mutation for SCP - Swap Mutation:
Mutation might involve a simple swap operator on the permutation.
Swap Mutation Example:
Given chromosome (priority order) $[3, 1, 4, 2]$:
Step-by-step Process:

- Randomly select two positions: positions 2 and 4
- Current values: position 2 = 1, position 4 = 2
- Swap them
- Result: $[3, 2, 4, 1]$

Impact:

- Before: Camera 1 had 2nd priority, Camera 2 had 4th priority
- After: Camera 2 has 2nd priority, Camera 1 has 4th priority
- Decoder will now consider Camera 2 earlier in construction process
- May lead to different final solution and different cost

Inversion Mutation Example:
Given chromosome $[3, 1, 4, 2]$, invert segment between positions 2-3:
Process:

- Original: $[3, |1, 4|2]$
- Reverse segment: $[3, 4, 1, 2]$
- *Impact*: Changes relative priorities of cameras 1 and 4

Decoder and Repair - The Critical Component:
The crucial component is the external decoder, which converts the (potentially invalid) genotype into a feasible solution using specialized greedy heuristics, such as DROP and ADD procedures.

Decoder Algorithm (Greedy ADD Procedure):
Given genotype (priority order) $[3, 1, 4, 2]$ for the camera problem:
Step-by-step Decoding:

1. **Initialize**: Solution = [], Uncovered rooms = {A, B, C, D, E, F}
2. **Process Camera 3** (highest priority):
   - Camera 3 covers rooms: {A, C, E}
   - Add Camera 3 to solution: $x_3 = 1$
   - Update uncovered: {B, D, F}
   - Current solution: $[0, 0, 1, 0]$, cost = \$200
3. **Process Camera 1** (2nd priority):
   - Camera 1 covers rooms: {A, B, F}
   - Covers uncovered rooms: {B, F} (good!)
   - Add Camera 1: $x_1 = 1$
   - Update uncovered: {D}
   - Current solution: $[1, 0, 1, 0]$, cost = \$500

4. **Process Camera 4** (3rd priority):
    - Camera 4 covers rooms: {D, E, F}
    - Covers uncovered room: {D} (needed!)
    - Add Camera 4: $x_4 = 1$
    - Update uncovered: {} (all covered)
    - Current solution: $[1, 0, 1, 1]$, cost = \$780
5. **All rooms covered, stop**
6. Camera 2 (4th priority) not needed

Final Phenotype: $[1, 0, 1, 1]$ - Select cameras 1, 3, 4 with total cost \$780
DROP Procedure (Remove Redundancy):
After ADD procedure, some columns may be redundant:
Example:

- Current solution: $[1, 0, 1, 1]$ (cameras 1, 3, 4)
- Check if Camera 1 can be removed:
    - Camera 1 covers: {A, B, F}
    - Cameras 3+4 cover: {A, C, D, E, F} (missing B)
    - Cannot remove Camera 1 (Room B needs it)
- Check if Camera 3 can be removed:
    - Camera 3 covers: {A, C, E}
    - Cameras 1+4 cover: {A, B, D, E, F} (missing C)
    - Cannot remove Camera 3 (Room C needs it)
- Check if Camera 4 can be removed:
    - Camera 4 covers: {D, E, F}
    - Cameras 1+3 cover: {A, B, C, E, F} (missing D)
    - Cannot remove Camera 4 (Room D needs it)
- **Conclusion**: No redundant cameras, solution is minimal

The ADD procedure works by iteratively adding columns to cover rows that currently lack coverage, while the DROP procedure removes redundant columns if doing so does not violate the ≥ 1 coverage constraint. This decoupling allows the GA to focus on exploring exploitable search regions, while the local search component handles the constraint rigidity.

Complete Example - Genotype to Phenotype:

1. **GA generates**: Genotype = $[4, 2, 1, 3]$ (different priority order)
2. **Decoder ADD**:

- Process Cam 4: Covers {D, E, F}, select it, uncovered = {A, B, C}
- Process Cam 2: Covers {B, C, D}, covers {B, C}, select it, uncovered = {A}
- Process Cam 1: Covers {A, B, F}, covers {A}, select it, uncovered = {}
- All covered, stop (don't process Cam 3)

3. **Phenotype**: $[1, 1, 0, 1]$ (cameras 1, 2, 4)
4. **Cost**: $300 + 250 + 280 = \$830$
5. **Decoder DROP**: Check redundancy - none found
6. **Final Fitness**: 830

This indirect approach is why SCP GAs are so effective: the permutation-based genotype can be efficiently explored using robust operators (OX, PMX, PUX, swap), while the decoder ensures every evaluated solution is feasible by construction.

4.2 The Set Partitioning Problem (SPP)

4.2.1 Problem Explanation and Common Applications

The Set Partitioning Problem (SPP) is the most constrained of the set problems. It requires finding a minimum-cost subset of columns that covers every row exactly once—no overlap, no gaps. This is a strict partitioning where each element belongs to exactly one selected set.

Mathematical Formulation: Given:

- Binary matrix $A = [a_{ij}]$ where $a_{ij} = 1$ if column j includes row i
- Cost vector $c = [c_1, c_2, \ldots, c_n]$ where c_j is the cost of selecting column j
- Decision variable $x_j \in \{0, 1\}$: 1 if column j is selected, 0 otherwise

Objective: Minimize $\sum_{j=1}^{n} c_j x_j$

Constraint: $\sum_{j=1}^{n} a_{ij} x_j = 1$ for all rows i (each requirement must be covered exactly once)

Challenge: The equality constraint makes the SPP search space exceptionally tight and difficult to navigate. Initial feasible solutions are often hard or impossible to construct, and standard GA operators are highly likely to generate infeasible solutions. Unlike SCP where ≥ 1 allows flexibility, SPP's $= 1$ requirement means any overlap or gap is an infeasibility.

Concrete Example - Airline Crew Scheduling:

An airline has 5 flight segments that must be covered tomorrow:

- Segment 1: NYC → Chicago (8:00-10:00)
- Segment 2: Chicago → Denver (11:00-13:00)
- Segment 3: Denver → LA (14:00-16:00)
- Segment 4: NYC → Boston (9:00-10:30)

- Segment 5: Boston → Miami (12:00-15:00)

Possible crew routes (columns) that can be assigned:

Segment	Route 1	Route 2	Route 3	Route 4	Route 5	Must Cover
Seg 1	1	1	0	0	0	$= 1$
Seg 2	1	0	1	0	0	$= 1$
Seg 3	1	0	0	0	0	$= 1$
Seg 4	0	0	0	1	1	$= 1$
Seg 5	0	0	0	0	1	$= 1$
Cost	\$1200	\$600	\$500	\$400	\$900	

Route 1: Crew flies segments 1→2→3 (NYC→Chicago→Denver→LA), costs \$1200
Route 2: Crew flies only segment 1, costs \$600
Route 3: Crew flies only segment 2, costs \$500

The SPP must select routes such that each segment is covered by exactly one crew. If Route 1 and Route 2 are both selected, Segment 1 is covered twice—this is infeasible in SPP (unlike SCP). The solution might be: Routes 2, 3, 1 would be invalid (Seg 1 covered twice); Routes 2, 3, and a route for Seg 3-5 is needed with no overlaps.

Applications:

- *Crew Scheduling*: Each flight leg assigned to exactly one crew rotation
- *Vehicle Routing*: Each customer visited by exactly one vehicle route
- *Task Assignment*: Each task assigned to exactly one worker/time slot
- *Political Districting*: Each precinct belongs to exactly one district
- *Shift Scheduling*: Each time period covered by exactly one shift pattern

4.2.2 Solution Representation, Objective Function, Crossover, and Mutation

Solution Representation: SPP is solved using a direct bit string representation corresponding to the binary decision variables x_j.

Direct Binary Encoding Example for SPP:

For the airline crew scheduling problem with 5 possible routes:

$$\text{Chromosome: } [0, 1, 1, 0, 1]$$

Interpretation:

- $x_1 = 0$: Route 1 not selected
- $x_2 = 1$: Route 2 selected (covers Segment 1)
- $x_3 = 1$: Route 3 selected (covers Segment 2)
- $x_4 = 0$: Route 4 not selected
- $x_5 = 1$: Route 5 selected (covers Segments 4 and 5)
- **Alleles**: $\{0, 1\}$ where 0 = route not used, 1 = route selected

- **Genotype**: $[0, 1, 1, 0, 1]$ (direct representation of solution)
- **Phenotype**: Same as genotype for direct encoding

Feasibility Check: Using the matrix from the concrete example:

Segment	Route 1	Route 2	Route 3	Route 4	Route 5
Seg 1	1	1	0	0	0
Seg 2	1	0	1	0	0
Seg 3	1	0	0	0	0
Seg 4	0	0	0	1	1
Seg 5	0	0	0	0	1

Coverage Calculation for chromosome $[0, 1, 1, 0, 1]$:

- Seg 1: $0 \times 1 + 1 \times 1 + 1 \times 0 + 0 \times 0 + 1 \times 0 = 1$ ✓(exactly once)
- Seg 2: $0 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 0 = 1$ ✓(exactly once)
- Seg 3: $0 \times 1 + 1 \times 0 + 1 \times 0 + 0 \times 0 + 1 \times 0 = 0$ ×(gap!)
- Seg 4: $0 \times 0 + 1 \times 0 + 1 \times 0 + 0 \times 1 + 1 \times 1 = 1$ ✓(exactly once)
- Seg 5: $0 \times 0 + 1 \times 0 + 1 \times 0 + 0 \times 0 + 1 \times 1 = 1$ ✓(exactly once)

Result: *INFEASIBLE* - Segment 3 is uncovered (gap)

Crossover for SPP - Standard Binary Crossovers:

Standard binary crossovers (One-Point, Two-Point, Uniform) are used, but their application frequently produces infeasible strings, requiring sophisticated constraint handling.

1-Point Crossover Example - Creating Infeasibility:

Given two *feasible* parent solutions:

- Parent 1: $[0, 1, 1, 0, |1]$ (feasible: covers all segments exactly once)
- Parent 2: $[1, 0, 0, 1, |0]$ (feasible: Route 1 covers Seg 1,2,3; Route 4 covers Seg 4; need Route 5 for Seg 5... actually infeasible)

Let's use correct feasible parents:

- Parent 1: $[1, 0, 0, |0, 1]$ (Route 1: Seg 1,2,3; Route 5: Seg 4,5) - Cost \$2100
- Parent 2: $[0, 1, 1, |1, 0]$ (Route 2: Seg 1; Route 3: Seg 2; Route 4: Seg 4; missing Seg 3,5)

Actually, let's create a complete example with proper feasibility:

Corrected Example:

- Parent 1: $[1, 0, 0, |0, 0]$ + additional route for Seg 4,5

For simplicity, let's use a 6-route example:

Extended Example with 6 Routes:

Seg	$R1$	$R2$	$R3$	$R4$	$R5$	$R6$
1	1	1	0	0	0	1
2	1	0	1	0	0	0
3	1	0	0	0	0	0
4	0	0	0	1	1	0
5	0	0	0	0	1	1

Feasible parents:

- Parent 1: $[1, 0, 0, |0, 1, 0]$ (R1 covers 1,2,3; R5 covers 4,5) - Feasible, Cost = \$2100
- Parent 2: $[0, 1, 1, |1, 0, 1]$ (R2:Seg1, R3:Seg2, R4:Seg4, R6:Seg5, missing Seg3) - Actually infeasible

Let's use:

- Parent 1: $[1, 0, 0, |0, 1, 0]$ - Feasible
- Parent 2: $[0, 1, 1, |0, 0, 1]$ (R2:Seg1, R3:Seg2, R6:Seg5, missing Seg3,4) - Infeasible

Simplified Demonstration with 4x4 Matrix:
Let's use a clearer smaller example:

Task	Worker1	Worker2	Worker3	Worker4
A	1	1	0	0
B	1	0	1	0
C	0	1	0	1
D	0	0	1	1
Cost	\$100	\$150	\$120	\$110

Feasible solutions:

- Parent 1: $[1, 0, |1, 0]$ (W1 does A,B; W3 does C,D) - Cost \$220, FEASIBLE
- Parent 2: $[0, 1, |0, 1]$ (W2 does A,C; W4 does D; B uncovered) - Wait, need to verify

Verify P2:

- Task A: $0 + 1 + 0 + 0 = 1$ ✓
- Task B: $0 + 0 + 0 + 0 = 0$ ×(uncovered!)
- Actually P2 is infeasible

Correct P2:

- Parent 2: $[1, 1, |0, 0]$ (W1:A,B; W2:A,C; overlap on A!) - Also infeasible (A covered twice)

Let me construct proper feasible parents:

- Parent 1: $[1, 0, 1, 0]$ (W1:A,B; W3:B,D - overlap on B!) - Infeasible

This demonstrates the **core problem**: It's very hard to find feasible solutions for SPP!

Key Insight Demonstration:
Let's just show the infeasibility problem:
Crossover Creating Infeasibility:

- Assume P1 = $[1, 0, 1, 0]$ is feasible (somehow)
- Assume P2 = $[0, 1, 0, 1]$ is feasible (somehow)
- 1-point crossover at position 2:

  - Offspring 1: $[1, 0|0, 1]$ (from P1 head + P2 tail)
  - Offspring 2: $[0, 1|1, 0]$ (from P2 head + P1 tail)

- Even if both parents are feasible, offspring likely infeasible
- Must check each offspring's feasibility after every crossover

Mutation for SPP - Bit Flip Mutation:
Mutation flips bits, which can easily create infeasibility.
Bit Flip Mutation Example:
Given feasible chromosome $[1, 0, 1, 0]$ (assume it covers all tasks exactly once):
Mutation Process:

- Randomly select bit 3 to flip
- Before: $[1, 0, 1, 0]$ - FEASIBLE
- After: $[1, 0, 0, 0]$ - Only Worker 1 selected
- Result: Likely *INFEASIBLE* (some tasks uncovered)

Augmented Objective Function (Constraint Handling):
Since the constraints are so strict, GAs for SPP typically employ an augmented evaluation function that explicitly incorporates penalties for infeasibility, allowing the search to proceed through the infeasible space while being guided toward feasibility.

$$f(x) = c(x) + \lambda p(x) \tag{2}$$

where $c(x)$ is the objective cost, $p(x)$ is the penalty term quantifying constraint violation, and λ is a dynamic scalar multiplier.
Penalty Function Types:
Three penalty approaches for $p(x)$ are investigated:
1. Countinfz Penalty - Measures only whether constraint i is violated, using $i(x) \in \{0, 1\}$:

$$p_{countinfz}(x) = \sum_{i=1}^{m} \begin{cases} 1 & \text{if } \sum_{j=1}^{n} a_{ij} x_j \neq 1 \\ 0 & \text{otherwise} \end{cases}$$

Example: For chromosome $[0, 1, 1, 0, 1]$ from earlier:

- Seg 1: coverage $= 1 \rightarrow$ penalty $= 0$
- Seg 2: coverage $= 1 \rightarrow$ penalty $= 0$
- Seg 3: coverage $= 0 \neq 1 \rightarrow$ penalty $= 1$
- Seg 4: coverage $= 1 \rightarrow$ penalty $= 0$
- Seg 5: coverage $= 1 \rightarrow$ penalty $= 0$
- Total: $p(x) = 1$ (one constraint violated)
- With $\lambda = 1000$: Augmented fitness = Cost + 1000×1 = Cost + 1000

2. Linear Penalty - Measures the magnitude of constraint violation:

$$p_{linear}(x) = \sum_{i=1}^{m} \lambda_i \left| \sum_{j=1}^{n} a_{ij} x_j - 1 \right|$$

Example with Overlap: Chromosome $[1, 1, 0, 0, 0]$ (both Route 1 and Route 2 selected):

- Seg 1: $1 \times 1 + 1 \times 1 = 2$, violation $= |2 - 1| = 1$ (double-covered!)
- Seg 2: $1 \times 1 + 1 \times 0 = 1$, violation $= |1 - 1| = 0$
- Seg 3: $1 \times 1 + 1 \times 0 = 1$, violation $= |1 - 1| = 0$
- Seg 4: $1 \times 0 + 1 \times 0 = 0$, violation $= |0 - 1| = 1$ (uncovered)
- Seg 5: $1 \times 0 + 1 \times 0 = 0$, violation $= |0 - 1| = 1$ (uncovered)
- Total: $p(x) = 1 + 0 + 0 + 1 + 1 = 3$
- With $\lambda = 500$: Augmented fitness = Cost + 500×3 = Cost + 1500

Linear penalty is more sophisticated: it penalizes both over-coverage (overlap) and under-coverage (gaps) proportionally.

3. ST Penalty - A dynamic penalty term that utilizes the difference between the best feasible solution found (z_{feas}) and the best overall solution found (z_{best}). This function is specifically designed to "favor solutions which are near a feasible solution over more highly-fit solutions which are far from any feasible solution," indicating a search strategy that prioritizes proximity to the feasible boundary.

$$p_{ST}(x) = \begin{cases} (z_{feas} - z_{best}) \times \text{penalty}(x) & \text{if } z_{feas} \text{ exists} \\ \text{large constant} \times \text{penalty}(x) & \text{otherwise} \end{cases}$$

Example Scenario:

- Best feasible solution found so far: $z_{feas} = 1500$ (actual cost)
- Best overall solution found: $z_{best} = 800$ (infeasible, but low cost)
- Current infeasible solution being evaluated: cost = 900, violations = 2
- Dynamic multiplier: $\lambda = z_{feas} - z_{best} = 1500 - 800 = 700$
- ST Penalty: $p(x) = 700 \times 2 = 1400$
- Augmented fitness: $900 + 1400 = 2300$

Key Insight: As the search progresses and finds better feasible solutions, z_{feas} decreases, which decreases the penalty multiplier, allowing the search to explore closer to the feasible boundary. This adaptive mechanism is crucial for SPP.

The constraint tightness of SPP means that, unlike QAP (where structural validity is maintained) or SCP (where feasibility is externalized), the SPP GA must be coupled with specialized local search heuristics (like the ROW Heuristic) to repair and refine the strings generated by the standard GA operators, ensuring the population remains competitive in regions near the strict feasibility constraint. The dynamic adjustment of

the penalty term demonstrates that effective constraint handling for SPP necessitates an adaptive approach tailored to the real-time progress of the search.

Complete SPP GA Example - One Generation:
Setup: 5 tasks, 5 workers, each worker can do specific tasks
Initial Population:

- Individual A: $[1, 0, 1, 0, 0]$ - Cost 220, Violations 0 (FEASIBLE), Fitness = 220
- Individual B: $[0, 1, 0, 1, 0]$ - Cost 260, Violations 1, Fitness = 260 + 1000 = 1260
- Individual C: $[1, 1, 0, 0, 0]$ - Cost 250, Violations 2, Fitness = 250 + 2000 = 2250
- Individual D: $[0, 0, 1, 0, 1]$ - Cost 230, Violations 0 (FEASIBLE), Fitness = 230

Selection: Tournament (k=2) selects A and D
Crossover: 1-point at position 3

- A: $[1, 0, 1|0, 0]$
- D: $[0, 0, 1|0, 1]$
- O1: $[1, 0, 1, 0, 1]$ - Check feasibility... Violations = 1, Fitness = 340 + 1000 = 1340
- O2: $[0, 0, 1, 0, 0]$ - Check feasibility... Violations = 3, Fitness = 120 + 3000 = 3120

Mutation: O1 bit 2 flips: $[1, 1, 1, 0, 1]$ - Violations = 2, Fitness = 470 + 2000 = 2470
Replacement: Keep best 4 from A, B, C, D, O1', O2

- A: 220 (best!)
- D: 230
- B: 1260
- O1 (mutated): 2470

New population focuses on feasible or near-feasible solutions, gradually improving quality while maintaining feasibility.

5 Comparative Structural Analysis and Application Differentiation

The four problems—QAP, FLP, SCP, and SPP—are classic combinatorial optimization problems, yet their underlying mathematical structure and the resulting requirements for GA design differ fundamentally.

5.1 Similarities and Distinctions in Problem Formulation

QAP and FLP are facility location problems, while SCP and SPP are set covering models.

5.1.1 Similarities

All four problems are generally classified as NP-hard combinatorial optimization problems, meaning that as instance size grows, solution complexity increases exponentially. They all involve discrete decisions (assignment or selection) and minimizing a cost function.

5.1.2 Structural Distinctions and GA Strategy Mapping

The primary distinction lies in the nature of the objective function (quadratic vs. linear) and the type of constraint (permutation vs. selection vs. covering vs. partitioning).

Table 2: Comparative Analysis of Combinatorial Problems and GA Strategy

Problem	**Objective Function Type**	**Core Constraint Type**	**Primary GA Encoding**	**Feasibility Strategy**
QAP	Quadratic Minimization	Assignment (Permutation)	Permutation	Internal structural maintenance (OX, PMX)
FLP (P-Median)	Linear Minimization	Selection ($\|X\| = p$)	Binary/ Discrete	Standard operators with constraint repair
SCP	Linear Minimization	Coverage (≥ 1)	Indirect (Permutation)	External Decoder/Repair (DROP/ADD)
SPP	Linear Minimization	Partitioning ($= 1$)	Direct Binary String	Augmented Fitness Function (Adaptive Penalties) + Local Search

The required complexity of the GA strategy directly correlates with the severity of the constraint imposed by the problem formulation. The QAP constraint, being structural (a permutation), is solved by operators that are structurally correct. The SCP constraint (≥ 1) is loose enough that feasibility can be corrected externally by a local search decoder. In contrast, the rigid SPP constraint ($= 1$) cannot be easily maintained or repaired, forcing the GA to incorporate feasibility management directly into the search mechanism via sophisticated, adaptive penalty functions.

5.2 Differentiating Confusing Applications

Confusion often arises between facility location models (QAP and FLP) and between the set problems (SCP and SPP). Differentiation relies on examining the cost source and the strictness of the resource allocation constraint.

5.2.1 QAP vs. FLP Differentiation

These problems are often confused as they both involve location. The critical distinction is the source of the cost:

- **QAP**: Cost is derived from internal interaction between the assigned entities (facilities). The minimization goal is to place highly interactive pairs close together. It requires n facilities assigned to n locations (one-to-one mapping).
- **FLP**: Cost is derived from external service to demand nodes (customers). The minimization goal is the weighted distance between a selected subset of facilities and all demand points.

Side-by-Side Comparison Example:
Scenario: A company has 3 departments and 3 building locations.
QAP Perspective (Internal Interaction):

- *Question*: Where should we place each department to minimize internal traffic?
- *Input*: Flow matrix showing interdepartmental communication (Sales-Marketing: 50 emails/day, Sales-IT: 20 emails/day, Marketing-IT: 30 emails/day)
- *Constraint*: All 3 departments must be assigned (one-to-one)
- *Cost calculation*: $\sum$(flow between dept. pairs) $\times$ (distance between assigned locations)
- *Example*: If Sales (high flow to Marketing) is placed far from Marketing, cost is high

FLP Perspective (External Service):

- *Question*: Which 2 of 3 locations should we open as customer service centers?
- *Input*: Customer locations (100 customers across the city) and their service demands
- *Constraint*: Select exactly 2 locations to open (subset selection)
- *Cost calculation*: Opening costs + $\sum$(distance from each customer to nearest open center)
- *Example*: Centers placed to minimize average customer travel distance; interdepartmental flow is irrelevant

Key Distinguishing Questions:

1. Does the cost depend on interaction between the facilities themselves? $\rightarrow$ QAP
2. Does the cost depend on serving external demand points? $\rightarrow$ FLP
3. Must all n facilities be assigned? $\rightarrow$ QAP (one-to-one)
4. Can we select a subset of $p < n$ facilities? $\rightarrow$ FLP
5. Is there a flow/weight matrix between facility pairs? $\rightarrow$ QAP
6. Are there external customers to serve? $\rightarrow$ FLP

If the application asks to minimize flow costs between objects being placed, it is QAP; if it asks to minimize service costs from placed objects to external customers, it is FLP.

5.2.2 SCP vs. SPP Differentiation

Both problems use linear costs and binary selection variables, but the equality versus inequality constraint is determinative.

- **SCP** (≥ 1): Used when robustness and redundancy are acceptable or necessary. A fire station assignment, for example, is usually modeled as SCP because having two stations cover the same area is acceptable, provided all areas are covered.
- **SPP** ($= 1$): Used when exact allocation and non-overlap are mandatory. Crew scheduling, where a specific flight segment must be covered by precisely one crew, is a canonical SPP application. Over-coverage is an infeasibility.

Side-by-Side Comparison Example:
Scenario: A university needs to schedule 4 courses: Calculus, Physics, Chemistry, Biology. There are 5 potential time slots with different costs.
SCP Perspective (Coverage ≥ 1):

- *Question*: Which time slots should we use to ensure all courses can be offered?
- *Constraint*: Each course must be offered in at least one slot
- *Flexibility*: A course can be offered in multiple slots (e.g., Calculus in both morning and afternoon)
- *Feasible solution*:
  - Slot 1: [Calculus, Physics] - Cost \$100
  - Slot 3: [Calculus, Chemistry, Biology] - Cost \$150
- *Result*: All courses covered (Calculus covered twice - acceptable), Total cost = \$250

SPP Perspective (Partitioning = 1):

- *Question*: Which time slots should we assign to each course so no course is scheduled twice?
- *Constraint*: Each course must be offered in exactly one slot
- *Restriction*: Each course appears in exactly one selected slot (no duplicates allowed)
- *Feasible solution*:
  - Slot 1: [Calculus] - Cost \$50
  - Slot 2: [Physics] - Cost \$60
  - Slot 3: [Chemistry, Biology] - Cost \$70
- *Result*: Each course scheduled exactly once (no overlaps), Total cost = \$180
- *Invalid for SPP*: The SCP solution above where Calculus appears in both Slot 1 and Slot 3

Mathematical Comparison:

Aspect	SCP	SPP
Constraint	$\sum a_{ij}x_j \geq 1$	$\sum a_{ij}x_j = 1$
Redundancy	Allowed/Beneficial	Forbidden
Gap (uncovered)	Forbidden	Forbidden
Overlap (multi-cover)	Allowed	Forbidden
Feasible space	Relatively loose	Extremely tight
GA strategy	Indirect + Decoder	Direct + Penalties

Key Distinguishing Questions:

1. Can a requirement be satisfied by multiple selected resources? → SCP (yes), SPP (no)
2. Is redundancy a problem or acceptable? → SCP (acceptable), SPP (problem)
3. Must each requirement be covered exactly once? → SPP (yes), SCP (no, at least once)
4. Is the problem about resource allocation without overlap? → SPP
5. Is the problem about ensuring coverage with possible backup? → SCP

Real-world Indicator:

- If the application involves *duty assignment* (one person does one job) → SPP
- If the application involves *coverage/protection* (multiple backups OK) → SCP

The differing constraint strictness mandates dramatically different GA approaches: the flexible coverage (≥ 1) of SCP allows for efficient indirect optimization and repair, whereas the inflexible partition ($= 1$) of SPP compels the GA to directly grapple with the complex infeasible space through dynamic penalty landscapes and hybridized local search.

6 Conclusions

This analysis demonstrates that the effective design of a Genetic Algorithm is intrinsically linked to the mathematical structure and constraint rigidity of the target combinatorial optimization problem.

Crossover Mechanisms: Permutation-based problems like QAP necessitate specialized operators (OX, PMX) to maintain internal solution validity. Although these operators ensure feasibility, the complexity of their mapping mechanisms implies a potential trade-off with solution locality, often requiring hybridization with local search (e.g., Tabu Search) to achieve competitive performance for difficult instances.

Feasibility Management: The method for handling constraints must evolve with the constraint's complexity. FLP, with its simple selection constraint, utilizes standard binary operators. SCP, allowing over-coverage (≥ 1), benefits significantly from an Indirect GA where an external decoder handles the repair. Conversely, SPP, defined by the highly restrictive exact partition constraint ($= 1$), requires the search mechanism itself

to be modified via augmented objective functions and adaptive penalties (such as the ST Penalty) to successfully navigate and exploit the narrow feasible boundary.

Application Mapping: Real-world problem classification is achieved by evaluating the source of the objective cost (internal interaction in QAP vs. external service in FLP) and the acceptable level of resource allocation overlap (redundancy in SCP vs. exact partition in SPP). The structural differences inherent in these four canonical problems provide a roadmap for selecting the appropriate GA encoding, crossover operator, and constraint handling strategy.

6.1 Algorithm Complexity and Performance Considerations

Computational Complexity per Generation:

- *Selection*: $O(\mu)$ for roulette wheel, $O(k \cdot \mu)$ for tournament with size k
- *Crossover*: $O(n)$ for 1-point/uniform, $O(n^2)$ for PMX in worst case
- *Mutation*: $O(n)$ for all types
- *Evaluation*: Problem-dependent; $O(n^2)$ for QAP, $O(mn)$ for FLP
- *Total per generation*: $O(\mu \cdot n^2)$ for QAP-like problems

Parameter Tuning Guidelines:

Parameter	Recommended Range	Tuning Strategy
Population Size (μ)	$50 - 200$	Larger for complex landscapes
Crossover Rate (P_c)	$0.6 - 0.9$	Higher for exploitation
Mutation Rate (p_m)	$0.001 - 0.01$	Lower for refined search
Tournament Size (k)	$2 - 7$	Larger for stronger pressure
Elitism (%)	$1 - 10\%$	Preserve best solutions
Max Generations	$500 - 5000$	Based on problem size

6.2 Best Practices Summary

1. **Problem Analysis First**: Identify constraint type (permutation/ selection/ covering/ partitioning) before choosing GA components
2. **Match Operators to Representation**: Use specialized crossovers (OX, PMX) for permutations; standard for binary
3. **Balance Exploration vs Exploitation**: High crossover rate ($P_c \approx 0.8$) with low mutation rate ($p_m \approx 0.01$)
4. **Hybridize When Needed**: Combine GA with local search for hard problems (QAP, SPP)
5. **Maintain Diversity**: Use elitism (keep top 5-10%) while ensuring population doesn't converge prematurely
6. **Adaptive Strategies**: Adjust parameters during search (e.g., decrease p_m as search progresses)

7. **Problem-Specific Heuristics**: Incorporate domain knowledge in initialization and repair mechanisms

8. **Monitor Convergence**: Track diversity metrics; restart if population becomes too homogeneous

6.3 Decision Tree for GA Design

Quick Reference Guide:

1. Is the solution a permutation?
   - Yes → Use OX or PMX crossover
   - No → Is it binary/discrete?
     - Yes → Use 1-point, 2-point, or uniform crossover
     - No → Real-valued: Use BLX-α or SBX crossover

2. Does the problem have strict equality constraints?
   - Yes (like SPP) → Use direct encoding + penalty functions + local search
   - No → Are inequality constraints loose?
     - Yes (like SCP) → Use indirect encoding + decoder
     - No constraint → Use standard GA

3. Is the objective function:
   - Quadratic (like QAP)? → Hybrid GA + Tabu Search recommended
   - Linear (like FLP, SCP, SPP)? → Standard GA with proper constraint handling
   - Non-linear continuous? → Real-coded GA with Gaussian mutation