

Genetic Algorithms

Theory and Practice

A Comprehensive Guide to Evolutionary Optimization

Course Materials Collection

November 24, 2025

Contents

1	Introduction to Optimization and Evolutionary Computation	1
1.1	Overview	1
1.2	What is Optimization?	1
1.3	Types of Optimization Problems	1
1.3.1	Based on Variable Types	1
1.3.2	Based on Problem Characteristics	2
1.4	Traditional Optimization Methods	2
1.4.1	Limitations of Traditional Methods	2
1.5	Introduction to Evolutionary Computation	2
1.5.1	Advantages of Evolutionary Approaches	3
1.6	Types of Evolutionary Algorithms	3
1.7	Applications of Evolutionary Computation	3
1.8	Simple Examples of Genetic Algorithm Applications	3
1.8.1	Maximizing a Quadratic Function	4
1.8.2	Travelling Salesman Problem	4
1.9	Genetic Algorithms for Real-World Problems	4
1.9.1	Scheduling	4
1.9.2	Telecommunication Network Design	5
1.9.3	VLSI Design (Very Large Scale Integration)	5
1.10	Genetic Algorithm Flow	5
1.11	Genetic Algorithm Variations	6
1.11.1	Hybrid GA	6
1.11.2	Adaptive GA	6
1.11.3	Parallel GA	6
1.12	Chapter Summary	6
1.13	Key Concepts	6
1.14	Further Reading	7
2	What is a Genetic Algorithm?	9
2.1	Introduction	9
2.2	Biological Inspiration	9
2.3	Basic Terminology	9
2.3.1	Genetic Algorithm Terms	9
2.4	Basic Structure of a Genetic Algorithm	10
2.5	Key Components of GA	10
2.5.1	Representation	10
2.5.2	Fitness Function	10
2.5.3	Selection	11

2.5.4	Crossover (Recombination)	11
2.5.5	Mutation	11
2.6	Example: Maximizing a Simple Function	11
2.6.1	Step 1: Representation	11
2.6.2	Step 2: Initial Population	11
2.6.3	Step 3: Selection	11
2.6.4	Step 4: Crossover	12
2.6.5	Step 5: Mutation	12
2.7	Advantages of Genetic Algorithms	12
2.8	Disadvantages of Genetic Algorithms	12
2.9	When to Use Genetic Algorithms	12
2.10	Variations of Genetic Algorithms	13
2.11	Chapter Summary	13
2.12	Key Concepts	13
3	GA Cycle and Holland Schema Theory	15
3.1	The Genetic Algorithm Cycle	15
3.1.1	Detailed GA Cycle	16
3.1.2	Phase 1: Initialization	16
3.1.3	Phase 2: Evaluation	16
3.1.4	Phase 3: Termination Check	16
3.1.5	Phase 4: Selection	17
3.1.6	Phase 5: Crossover	17
3.1.7	Phase 6: Mutation	17
3.1.8	Phase 7: Replacement	17
3.2	Holland Schema Theory	17
3.2.1	What is a Schema?	18
3.2.2	Schema Properties	18
3.2.3	Schema Theorem (Fundamental Theorem)	18
3.2.4	Building Block Hypothesis	20
3.2.5	Role of Schema Order in Genetic Algorithms	20
3.2.6	Relationship Between Holland Schema and Genome	21
3.2.7	Schema Functions in Genetic Algorithms	21
3.3	Implicit Parallelism	21
3.4	Deception and Schema Theory	22
3.4.1	Deceptive Problems	22
3.4.2	Overcoming Deception	22
3.5	Practical Implications	22
3.5.1	Encoding Design	22
3.5.2	Parameter Settings	22
3.6	Limitations of Schema Theory	22
3.7	Modern Extensions	23
3.7.1	Walsh Analysis	23
3.7.2	Fitness Landscapes	23
3.7.3	No Free Lunch Theorem	23
3.8	Chapter Summary	23
3.9	Key Concepts	23

4 Genetic Algorithm Encoding	25
4.1 Introduction to Encoding	25
4.2 Requirements for Good Encoding	25
4.2.1 Completeness	25
4.2.2 Soundness	25
4.2.3 Non-redundancy	25
4.2.4 Locality	25
4.3 Binary Encoding	25
4.3.1 Basic Binary Encoding	26
4.3.2 Decoding Binary Strings	26
4.3.3 Multi-variable Binary Encoding	26
4.3.4 Advantages of Binary Encoding	26
4.3.5 Disadvantages of Binary Encoding	26
4.4 Overview of Encoding Types	27
4.4.1 Problem Identification and Formulation	27
4.4.2 What is Encoding	27
4.4.3 Encoding Type Categories	27
4.5 Gray Code Encoding	28
4.5.1 Binary to Gray Code Conversion	28
4.5.2 Gray Code to Binary Conversion	29
4.5.3 Example: 4-bit Gray Code	30
4.6 Real-valued Encoding	30
4.6.1 Representation	30
4.6.2 Advantages	30
4.6.3 Disadvantages	30
4.6.4 Real-valued Crossover Operators	30
4.6.5 Real-valued Mutation Operators	31
4.7 Integer Encoding	31
4.7.1 Representation	31
4.7.2 Integer Crossover	31
4.7.3 Integer Mutation	32
4.8 Permutation Encoding	32
4.8.1 Representation	32
4.8.2 Permutation Crossover Operators	32
4.8.3 Permutation Mutation Operators	32
4.9 Tree Encoding	33
4.9.1 Applications	33
4.9.2 Representation	33
4.9.3 Tree Crossover	33
4.9.4 Tree Mutation	33
4.10 Problem-specific Encodings	34
4.10.1 Graph Coloring	34
4.10.2 Job Scheduling	34
4.10.3 Neural Network Weights	34
4.11 Choosing the Right Encoding	34
4.11.1 Factors to Consider	34
4.11.2 Guidelines	34
4.12 Chapter Summary	34

4.13 Key Concepts	35
5 Selection Methods in Genetic Algorithms	37
5.1 Introduction to Selection	37
5.1.1 Definition and Function of Selection Operator	37
5.1.2 Selection Pressure	37
5.1.3 Types of Selection Operators	38
5.2 Selection Pressure	38
5.3 Fitness Proportionate Selection (FPS)	38
5.3.1 Roulette Wheel Selection	38
5.3.2 Stochastic Universal Sampling (SUS)	40
5.4 Rank-based Selection	41
5.4.1 Overview	42
5.4.2 Linear Ranking	42
5.4.3 Exponential Ranking	42
5.4.4 Advantages of Rank Selection	43
5.4.5 Disadvantages	43
5.5 Tournament Selection	43
5.5.1 Overview	43
5.5.2 Tournament Selection Mechanism	44
5.5.3 Binary Tournament	44
5.5.4 k-Tournament Selection	44
5.5.5 Tournament Size Effects	44
5.5.6 Selection Probability	45
5.5.7 Advantages	45
5.5.8 Disadvantages	45
5.6 Truncation Selection	45
5.6.1 Algorithm	45
5.6.2 Selection Ratio	45
5.6.3 Advantages	46
5.6.4 Disadvantages	46
5.7 Boltzmann Selection	46
5.7.1 Formula	46
5.7.2 Temperature Schedule	46
5.7.3 Advantages	46
5.7.4 Disadvantages	46
5.8 Elitist Selection	47
5.8.1 Pure Elitism	47
5.8.2 Elitist Replacement	47
5.8.3 Benefits	47
5.8.4 Drawbacks	47
5.9 Diversity-Preserving Selection	47
5.9.1 Fitness Sharing	47
5.9.2 Crowding	47
5.9.3 Speciation	47
5.10 Multi-objective Selection	48
5.10.1 Pareto Dominance	48
5.10.2 Non-dominated Sorting	48

5.10.3	NSGA-II Selection	48
5.11	Selection Comparison	48
5.12	Selection Guidelines	48
5.12.1	Problem Characteristics	48
5.12.2	Population Size	48
5.12.3	Generation Number	49
5.13	Hybrid Selection Strategies	49
5.13.1	Adaptive Selection	49
5.13.2	Multi-level Selection	49
5.13.3	Combined Methods	49
5.14	Chapter Summary	49
5.15	Key Concepts	49
6	Crossover (Recombination) in Genetic Algorithms	51
6.1	Introduction to Crossover	51
6.2	Biological Inspiration	51
6.3	Crossover Principles	51
6.3.1	Exploration vs. Exploitation	51
6.3.2	Crossover Probability	51
6.4	Binary Crossover Operators	52
6.4.1	Definition and Function of Crossover Operator	52
6.4.2	One-Point Crossover	52
6.4.3	One-Point Crossover	52
6.4.4	Two-Point Crossover	53
6.4.5	Uniform Crossover	54
6.4.6	Multi-Point Crossover	56
6.5	Integer Chromosome Crossover	56
6.5.1	Single-Point Crossover for Integer	56
6.5.2	Multi-point Crossover for Integer	56
6.5.3	Uniform Crossover for Integer	56
6.6	Real-Valued Crossover Operators	57
6.6.1	Arithmetic Crossover	57
6.6.2	BLX- α Crossover (Blend Crossover)	60
6.6.3	SBX (Simulated Binary Crossover)	61
6.7	Permutation Crossover Operators	61
6.7.1	Order Crossover (OX)	61
6.7.2	Partially Mapped Crossover (PMX)	62
6.7.3	Cycle Crossover (CX)	62
6.7.4	Edge Recombination Crossover	62
6.8	Crossover Analysis	63
6.8.1	Schema Disruption	63
6.8.2	Building Block Preservation	63
6.9	Advanced Crossover Techniques	63
6.9.1	Adaptive Crossover	63
6.9.2	Multiple Parent Crossover	64
6.9.3	Problem-Specific Crossover	64
6.10	Crossover Guidelines	64
6.10.1	Choosing Crossover Type	64

6.10.2	Parameter Setting	64
6.10.3	Empirical Testing	64
6.11	Crossover vs. Mutation	65
6.12	Chapter Summary	65
6.13	Key Concepts	65
7	Real-World Applications and Visual Examples	67
7.1	Game AI and Entertainment	67
7.1.1	Super Mario Bros Level Learning	67
7.1.2	Tower Defense Game Balancing	67
7.2	Pathfinding and Navigation	68
7.2.1	Maze Navigation	68
7.2.2	Robot Navigation	68
7.3	Evolution Simulation	68
7.3.1	Simulated Evolution of Creatures	68
7.4	Human Analogy Examples	69
7.4.1	Evolution of Movement	69
7.4.2	Work Journey Optimization	69
7.5	Academic Context	69
7.5.1	GA in Computational Intelligence	69
7.6	Historical Perspective	70
7.6.1	Natural Selection Theories	70
7.7	Summary	70
8	Mutation and Generation Update	71
8.1	Introduction to Mutation	71
8.1.1	What is Mutation?	71
8.1.2	Mutation in Evolutionary Algorithms vs. Biological Evolution	72
8.2	Mutation for Different Representations	72
8.2.1	Mutation for Binary Representation	72
8.2.2	Mutation for Integer Representation	73
8.2.3	Mutation for Real-Valued Representation	73
8.2.4	Mutation for Permutation Representation	74
8.3	Generation Update Mechanisms	75
8.3.1	Holland's Original Model (Generational Replacement)	75
8.3.2	Generational Model with Elitism	76
8.3.3	Steady-State Update	76
8.3.4	Continuous Update	77
8.4	GA Parameters	77
8.4.1	Crossover Probability (P_c)	77
8.4.2	Mutation Probability (P_m)	78
8.4.3	Population Size (N)	78
8.4.4	Number of Generations (G)	79
8.4.5	General Parameter Setting Guidelines	79
8.5	Parameter Observation Study	79
8.5.1	Test Problem	80
8.5.2	Experimental Setup	80
8.5.3	Sample Results	80
8.6	Summary and Conclusions	81

8.7 Exercises	82
A Algorithm Implementations	85
A.1 Basic Genetic Algorithm Implementation	85
A.1.1 Python Implementation	85
A.2 Real-Valued Genetic Algorithm	89
A.3 Traveling Salesman Problem GA	92
A.4 NSGA-II for Multi-Objective Optimization	97
B Practical Examples and Case Studies	105
B.1 Function Optimization Problems	105
B.1.1 OneMax Problem	105
B.1.2 Sphere Function	106
B.1.3 Rastrigin Function	106
B.1.4 Rosenbrock Function	107
B.2 Combinatorial Optimization Problems	107
B.2.1 Traveling Salesman Problem (TSP)	107
B.2.2 Knapsack Problem	108
B.3 Real-World Applications	109
B.3.1 Neural Network Training	109
B.3.2 Feature Selection	109
B.3.3 Job Shop Scheduling	110
B.4 Parameter Tuning Guidelines	110
B.4.1 Population Size	110
B.4.2 Crossover and Mutation Rates	110
B.4.3 Selection Pressure	111
B.5 Performance Analysis	111
B.5.1 Convergence Metrics	111
B.5.2 Statistical Testing	111
B.5.3 Comparison with Other Methods	111
B.6 Common Pitfalls and Solutions	111
B.6.1 Premature Convergence	111
B.6.2 Slow Convergence	112
B.6.3 Constraint Handling Issues	112
B.7 Advanced Techniques	113
B.7.1 Hybrid Genetic Algorithms	113
B.7.2 Adaptive Parameter Control	113
B.7.3 Parallel Genetic Algorithms	113
B.8 Implementation Best Practices	113
B.8.1 Code Organization	113
B.8.2 Testing and Validation	113
B.8.3 Documentation	114
B.9 Chapter Summary	114
B.10 Key Takeaways	114

List of Figures

1.1	Illustration of GA cycle and variations	4
3.1	Genetic Algorithm Cycle	16
3.2	Schema characteristics: Order and Defining Length examples	19
4.1	Binary encoding examples and characteristics	28
4.2	Various encoding types: Value, Permutation, and Tree encoding	29
5.1	Basic selection process in Genetic Algorithms	37
5.2	Roulette-wheel selection process with sample trials	39
5.3	Stochastic universal sampling with equally spaced pointers	41
5.4	How the situation changes after converting fitness to order number (rank) .	42
5.5	Tournament selection mechanism	43
6.1	Single-Point Crossover for binary chromosomes	52
6.2	Multi-point Crossover for binary chromosomes	54
6.3	Uniform Crossover for binary chromosomes	55
6.4	Single-Point Crossover for integer chromosomes	57
6.5	Multi-point Crossover for integer chromosomes	58
6.6	Uniform Crossover for integer chromosomes	59
6.7	Single Arithmetic Crossover for real chromosomes	59
6.8	Simple Arithmetic Crossover for real chromosomes	59
6.9	Whole Arithmetic Crossover for real chromosomes	60
7.1	Genetic Algorithm Learning to Play Super Mario Bros at 4x Speed	67
7.2	Cat Navigating Circular Maze to Reach Cheese Using Genetic Algorithm .	68
7.3	Simulated Evolution Using Genetic Algorithm - Creatures Adapting Over Generations	68
7.4	Human Movement Evolution: From Sitting to Athletic Performance	69
7.5	Optimizing Daily Commute Route Using GA Principles	69
7.6	Position of Genetic Algorithms in Machine Learning and Soft Computing Landscape	69
7.7	Comparison of Lamarck vs Darwin-Wallace Evolution Theories Using Giraffe Example	70

List of Tables

2.1	Initial Population Example	9
2.2	Initial Population Example	11
4.1	Binary vs Gray Code Representation	30
5.1	Selection probability and fitness value (from Buku Ajar)	39
5.2	Roulette Wheel Selection Example	40
5.3	Comparison of Selection Methods	48
6.1	Crossover vs. Mutation Comparison [42, 38]	65
8.1	GA Parameter Observation Results	80
B.1	OneMax GA Configuration	105
B.2	Population Size Guidelines	110
B.3	Crossover and Mutation Rate Guidelines	110
B.4	Algorithm Comparison	111

Chapter 1

Introduction to Optimization and Evolutionary Computation

1.1 Overview

This chapter provides a foundational understanding of optimization problems and introduces the concept of evolutionary computation as a powerful approach to solving complex optimization challenges.

1.2 What is Optimization?

Optimization is the process of finding the best solution from a set of available alternatives. In mathematical terms, an optimization problem can be formulated as:

$$\begin{aligned} & \text{minimize (or maximize)} && f(x) \\ & \text{subject to} && g_i(x) \leq 0, \quad i = 1, 2, \dots, m \\ & && h_j(x) = 0, \quad j = 1, 2, \dots, p \\ & && x \in X \end{aligned} \tag{1.1}$$

where:

- $f(x)$ is the objective function to be optimized
- $g_i(x)$ are inequality constraints
- $h_j(x)$ are equality constraints
- X is the feasible region

1.3 Types of Optimization Problems

1.3.1 Based on Variable Types

- **Continuous Optimization:** Variables can take any real value
- **Discrete Optimization:** Variables can only take discrete values
- **Mixed-Integer Optimization:** Combination of continuous and discrete variables

1.3.2 Based on Problem Characteristics

- **Linear Programming:** Objective function and constraints are linear
- **Nonlinear Programming:** At least one function is nonlinear
- **Convex Optimization:** Objective function is convex
- **Multi-objective Optimization:** Multiple conflicting objectives

1.4 Traditional Optimization Methods

Traditional optimization methods include:

- Gradient-based methods (Newton's method, quasi-Newton methods)
- Simplex method for linear programming
- Branch and bound for integer programming
- Dynamic programming

1.4.1 Limitations of Traditional Methods

- Require differentiability of objective function
- Can get trapped in local optima
- Computationally expensive for large-scale problems
- Difficulty handling discrete variables
- Problems with discontinuous or noisy functions

1.5 Introduction to Evolutionary Computation

Evolutionary computation is a family of algorithms inspired by biological evolution [26, 14]. These algorithms use mechanisms such as:

- **Selection:** Survival of the fittest
- **Reproduction:** Creating offspring
- **Mutation:** Random changes
- **Crossover:** Combining genetic material

1.5.1 Advantages of Evolutionary Approaches

- No requirement for gradient information
- Can handle discontinuous, noisy, and multi-modal functions
- Suitable for both continuous and discrete optimization
- Population-based search provides robustness
- Can find global optima

1.6 Types of Evolutionary Algorithms

- **Genetic Algorithms (GA)**: Inspired by natural selection [26, 22]
- **Evolution Strategies (ES)**: Focus on real-valued optimization [7]
- **Evolutionary Programming (EP)**: Emphasis on behavioral evolution [18, 19]
- **Genetic Programming (GP)**: Evolution of computer programs [28]

1.7 Applications of Evolutionary Computation

Evolutionary algorithms have been successfully applied to [21, 40, 14]:

- Engineering design optimization [49]
- Machine learning and neural network training [33, 34]
- Scheduling and timetabling [24, 17]
- Financial modeling
- Bioinformatics
- Game playing and strategy

1.8 Simple Examples of Genetic Algorithm Applications

Genetic Algorithms possess an extraordinary capability in finding optimal solutions for problems with vast search spaces.

1.8.1 Maximizing a Quadratic Function

A classic example often used to illustrate how Genetic Algorithms work is the problem of maximizing a simple quadratic function, such as $f(x) = x^2$ in the range $0 \leq x \leq 31$. In this case, the optimal solution is clear: $x = 31$, yielding $f(x) = 961$. However, this problem is used to demonstrate how Genetic Algorithms, despite starting with a random population (for example, four 5-bit binary chromosomes), can gradually combine features (building blocks) from existing solutions to achieve better results.

Through cycles of selection, crossover, and mutation, Genetic Algorithms show how both the maximum and average performance of the population increases from one generation to the next. Within a few generations, individuals representing $x = 31$ (binary 11111) will dominate the population.

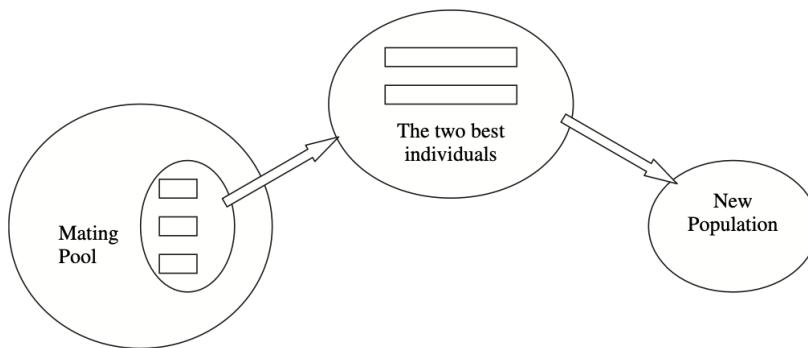


Figure 1.1: Illustration of GA cycle and variations

1.8.2 Travelling Salesman Problem

Another example of genetic algorithm application is the Travelling Salesman Problem (TSP), where Genetic Algorithms are used to find the shortest route visiting a number of cities—a highly complex combinatorial optimization problem.

1.9 Genetic Algorithms for Real-World Problems

Genetic Algorithms are becoming increasingly popular in industrial engineering, management science, and operations research due to their capability in handling complex optimization problems. Many real-world engineering problems are too complex, non-linear, or contain intricate constraints that are difficult to solve with conventional methods. Genetic Algorithms offer a robust and widely applicable method for stochastic search and optimization.

Rather than searching for a single solution point, Genetic Algorithms work on a population of solutions and only require the objective function (fitness) value without requiring derivatives or deep prior knowledge about the search space [32, 22]. This makes Genetic Algorithms highly suitable for problems such as:

1.9.1 Scheduling

Scheduling problems, such as course scheduling [17], factory production scheduling (Job Shop Scheduling) [24, 36], or flight scheduling, are combinatorial optimization problems

characterized by a large number of tasks, limited resources, and strict time constraints. In this context, GA chromosomes represent one task sequence or allocation (schedule). The fitness function is designed to minimize criteria such as total completion time (makespan), amount of delays, or operational costs. Genetic Algorithms can navigate discrete and large solution spaces very efficiently, searching for the best task sequence that meets all complex constraints.

1.9.2 Telecommunication Network Design

In network design, Genetic Algorithms are used to optimize topology, capacity, and node or router placement to achieve the best performance. The goal is often to minimize network construction costs while maximizing service quality, such as minimizing delay or maximizing throughput. Chromosomes can encode connection configurations between nodes or technology choices. Because each configuration change can significantly impact overall network performance, Genetic Algorithms are very useful for exploring various possible network architectures that are mathematically difficult to solve with traditional methods.

1.9.3 VLSI Design (Very Large Scale Integration)

VLSI is the process of designing semiconductor chips with millions of transistors. Genetic Algorithms are highly relevant in two main sub-problems: placement and routing. In the placement problem, Genetic Algorithms optimize the position of circuit blocks on the chip to minimize total wire length and avoid overlaps. In routing, Genetic Algorithms search for the best path to connect all circuit pins without violating physical constraints. The high dimensionality and geometric constraints (such as area, power, and timing) make the VLSI search space explosive, making Genetic Algorithms a feasible search technique for generating near-optimal layouts.

1.10 Genetic Algorithm Flow

Genetic algorithms maintain a population of individuals, denoted as $P(t)$ for generation t , where each individual represents a potential solution to the problem at hand. This cycle runs iteratively through steps that mimic the evolution process.

After the initial population $P(0)$ is initialized and its fitness evaluated, the selection process begins, where fitter individuals are probabilistically selected to enter the mating pool. Selected individuals then undergo stochastic transformation through genetic operators. The crossover operator is responsible for exploiting the best information from parents, while mutation is tasked with exploring the search space by introducing new genetic material.

Newly generated individuals, called offspring $C(t)$, are then evaluated for their fitness. A new population $P(t + 1)$ is formed by selecting fitter individuals from both the parent and offspring populations through a replacement scheme. This process is repeated for several generations until the algorithm converges, pointing to the best individual, which is expected to represent an optimal or suboptimal solution to the problem.

1.11 Genetic Algorithm Variations

Although the Simple Generational Genetic Algorithm serves as the basic framework, various modifications have been developed to improve performance in dealing with problem complexity. Some of these modifications include:

1.11.1 Hybrid GA

Hybrid GA (HGA) combines Genetic Algorithms with conventional local search techniques [30, 34]. In a hybrid approach, Genetic Algorithms perform global exploration across the population, while local search is used for refinement around promising solutions. This approach often outperforms single methods because it leverages the complementary advantages of both search techniques.

1.11.2 Adaptive GA

Adaptive GA (AGA) is a Genetic Algorithm where strategic parameters (such as crossover or mutation probabilities) are dynamically adjusted during the evolution process, often using feedback from population performance to balance exploitation and exploration effectively [39, 43].

1.11.3 Parallel GA

Parallel Genetic Algorithms divide the population into different sub-populations across various processors, allowing periodic information exchange (migration) to increase diversity and convergence speed.

1.12 Chapter Summary

This chapter introduced the fundamental concepts of optimization and evolutionary computation. We explored the limitations of traditional optimization methods and highlighted the advantages of evolutionary approaches. We also examined specific examples of GA applications including function optimization, TSP, scheduling, network design, and VLSI design. The GA flow and various modifications were discussed to provide a comprehensive understanding of genetic algorithms.

1.13 Key Concepts

- Optimization problem formulation
- Objective functions and constraints
- Local vs. global optima
- Evolutionary computation principles
- Population-based search

1.14 Further Reading

- Deb, K. (2001). Multi-objective optimization using evolutionary algorithms.
- Eiben, A. E., & Smith, J. E. (2015). Introduction to evolutionary computing.
- Goldberg, D. E. (1989). Genetic algorithms in search, optimization, and machine learning.

Chapter 2

What is a Genetic Algorithm?

2.1 Introduction

Genetic Algorithms (GAs) are search

Individual	Binary	Decimal	Fitness
1	01101	13	169
2	11000	24	576
3	01000	8	64
4	10011	19	361

Table 2.1: Initial Population Example

t mimic the process of natural selection [26, 22, 32]. They belong to the larger class of evolutionary algorithms and are particularly useful for optimization and search problems.

2.2 Biological Inspiration

GAs are inspired by Charles Darwin's theory of natural evolution. In nature:

- Individuals with better fitness have higher chances of survival
- Successful traits are passed to offspring through reproduction
- Genetic diversity is maintained through mutation
- Population evolves over generations toward better adaptation

2.3 Basic Terminology

2.3.1 Genetic Algorithm Terms

- **Individual/Chromosome:** A candidate solution
- **Gene:** A single element of an individual

- **Allele:** The value of a gene
- **Population:** Collection of individuals
- **Generation:** One iteration of the algorithm
- **Fitness:** Quality measure of an individual
- **Genotype:** Encoded representation of a solution
- **Phenotype:** Decoded representation of a solution

2.4 Basic Structure of a Genetic Algorithm

Algorithm 1 Basic Genetic Algorithm

```

Initialize population randomly
while termination condition not met do
    Evaluate fitness of each individual
    Select parents for reproduction
    Apply crossover to create offspring
    Apply mutation to offspring
    Select survivors for next generation
    Increment generation counter
end while
Return best individual found

```

2.5 Key Components of GA

2.5.1 Representation

The representation defines how solutions are encoded:

- **Binary representation:** Solutions encoded as binary strings
- **Real-valued representation:** Solutions as real numbers
- **Permutation representation:** Solutions as ordered sequences
- **Tree representation:** Solutions as tree structures

2.5.2 Fitness Function

The fitness function evaluates the quality of each individual:

$$\text{fitness}(x) = f(x) \quad (2.1)$$

For maximization problems, higher fitness values are better. For minimization problems, fitness is often defined as:

$$\text{fitness}(x) = \frac{1}{1 + f(x)} \quad (2.2)$$

2.5.3 Selection

Selection determines which individuals become parents:

- **Roulette Wheel Selection:** Probability proportional to fitness
- **Tournament Selection:** Best individual from random subset
- **Rank Selection:** Selection based on fitness ranking

2.5.4 Crossover (Recombination)

Crossover combines genetic material from two parents:

- **One-point crossover:** Single crossover point
- **Two-point crossover:** Two crossover points
- **Uniform crossover:** Random selection from parents

2.5.5 Mutation

Mutation introduces random changes to maintain diversity:

- **Bit-flip mutation:** For binary representation
- **Gaussian mutation:** For real-valued representation
- **Swap mutation:** For permutation representation

2.6 Example: Maximizing a Simple Function

Consider maximizing $f(x) = x^2$ where $x \in [0, 31]$.

2.6.1 Step 1: Representation

Use 5-bit binary strings: $x = 10110_2 = 22_{10}$

2.6.2 Step 2: Initial Population

Individual	Binary	Decimal	Fitness
1	01101	13	169
2	11000	24	576
3	01000	8	64
4	10011	19	361

Table 2.2: Initial Population Example

2.6.3 Step 3: Selection

Select individuals 2 and 4 (highest fitness) as parents.

2.6.4 Step 4: Crossover

Parents: 11000 and 10011 Crossover point: position 3 Offspring: 11011 (27) and 10000 (16)

2.6.5 Step 5: Mutation

Apply bit-flip mutation with low probability.

2.7 Advantages of Genetic Algorithms

- **Global search:** Can escape local optima [22]
- **Parallelizable:** Population-based approach [14]
- **Flexible:** Applicable to various problem types [40]
- **No gradient required:** Works with discontinuous functions [32]
- **Robust:** Handles noisy fitness functions [25]

2.8 Disadvantages of Genetic Algorithms

- **Computational cost:** May require many function evaluations
- **Parameter tuning:** Many parameters to set
- **No guarantee:** May not find global optimum
- **Premature convergence:** Population may lose diversity

2.9 When to Use Genetic Algorithms

GAs are particularly suitable when:

- Search space is large and complex
- Little is known about the problem structure
- Traditional methods fail or are inappropriate
- Multiple objectives need to be optimized
- Robustness is more important than efficiency

2.10 Variations of Genetic Algorithms

- **Steady-state GA:** Replace one individual per generation
- **Parallel GA:** Multiple populations evolve simultaneously
- **Hybrid GA:** Combine with local search methods
- **Multi-objective GA:** Handle multiple objectives

2.11 Chapter Summary

This chapter introduced genetic algorithms as optimization tools inspired by natural evolution. We covered the basic components, terminology, and a simple example. The key insight is that GAs use population-based search with selection, crossover, and mutation to evolve solutions toward optimality.

2.12 Key Concepts

- Biological inspiration and evolution metaphor
- Basic GA structure and components
- Representation, fitness, selection, crossover, mutation
- Advantages and limitations of GAs
- When to apply genetic algorithms

Chapter 3

GA Cycle and Holland Schema Theory

3.1 The Genetic Algorithm Cycle

The genetic algorithm follows a cyclic process that mimics natural evolution. Understanding this cycle is crucial for implementing and analyzing GA performance.

3.1.1 Detailed GA Cycle

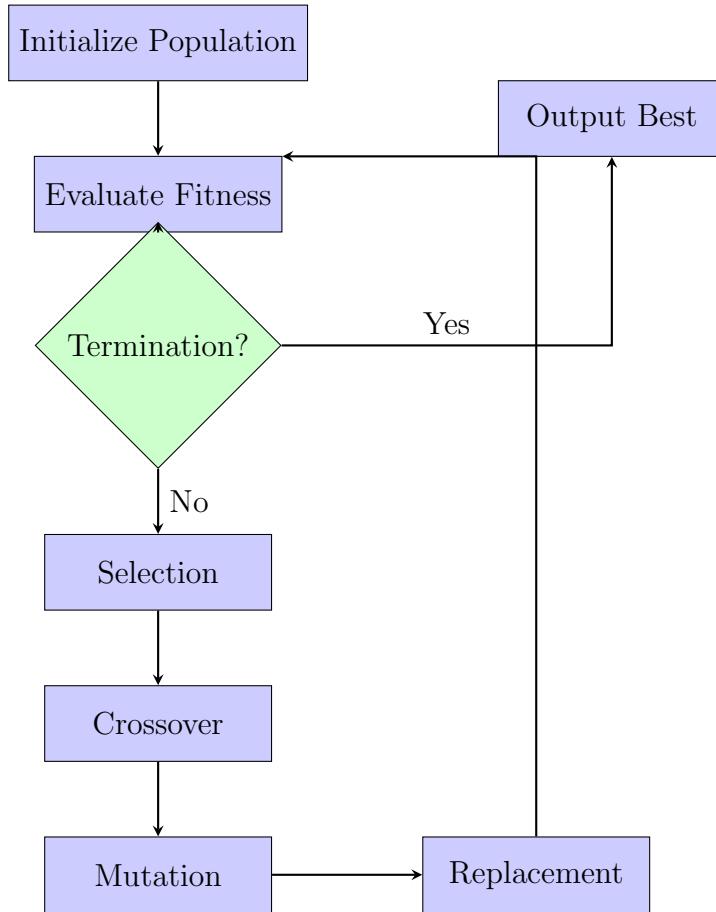


Figure 3.1: Genetic Algorithm Cycle

3.1.2 Phase 1: Initialization

- Create initial population of size N
- Generate individuals randomly or using heuristics
- Ensure population diversity
- Set generation counter $t = 0$

3.1.3 Phase 2: Evaluation

- Calculate fitness for each individual
- Identify best and worst individuals
- Compute population statistics (average, variance)

3.1.4 Phase 3: Termination Check

Common termination criteria:

- Maximum number of generations reached
- Fitness threshold achieved
- Population convergence (low diversity)
- No improvement for specified generations
- Maximum function evaluations reached

3.1.5 Phase 4: Selection

- Choose parents for reproduction
- Bias selection toward fitter individuals
- Maintain population diversity

3.1.6 Phase 5: Crossover

- Combine genetic material from selected parents
- Create offspring with traits from both parents
- Apply with probability p_c (typically 0.6-0.9)

3.1.7 Phase 6: Mutation

- Introduce random changes to offspring
- Maintain genetic diversity
- Apply with probability p_m (typically 0.001-0.1)

3.1.8 Phase 7: Replacement

- Form new population from parents and offspring
- Increment generation counter $t = t + 1$
- Return to evaluation phase

3.2 Holland Schema Theory

Schema theory, developed by John Holland [26], provides a theoretical foundation for understanding why genetic algorithms work effectively.

3.2.1 What is a Schema?

A schema is a template describing a subset of strings with similarities at certain positions [26, 22, 1]. It uses three symbols:

- **0**: Fixed bit value 0
- **1**: Fixed bit value 1
- *****: Don't care symbol (wild card)

Example: Schema $H = 1 * 0 * 1$ represents all 5-bit strings with:

- First bit = 1
- Third bit = 0
- Fifth bit = 1
- Second and fourth bits can be anything

Strings matching this schema: 10001, 10011, 11001, 11011

3.2.2 Schema Properties

Order of a Schema

The order $o(H)$ is the number of fixed positions (non-* symbols):

$$o(H) = \text{number of defined bits in } H \quad (3.1)$$

For $H = 1 * 0 * 1$: $o(H) = 3$

Defining Length

The defining length $\delta(H)$ is the distance between the first and last fixed positions:

$$\delta(H) = \text{last fixed position} - \text{first fixed position} \quad (3.2)$$

For $H = 1 * 0 * 1$: $\delta(H) = 5 - 1 = 4$

Examples from Buku Ajar:

- $S_1 = (* * * 001 * 110)$: $\delta(S_1) = 10 - 4 = 6$
- $S_2 = (* * * * 00 * * 0 *)$: $\delta(S_2) = 9 - 5 = 4$
- $S_3 = (11101 * * 001)$: $\delta(S_3) = 10 - 1 = 9$

3.2.3 Schema Theorem (Fundamental Theorem)

The schema theorem describes how the expected number of strings matching a schema changes from generation to generation.

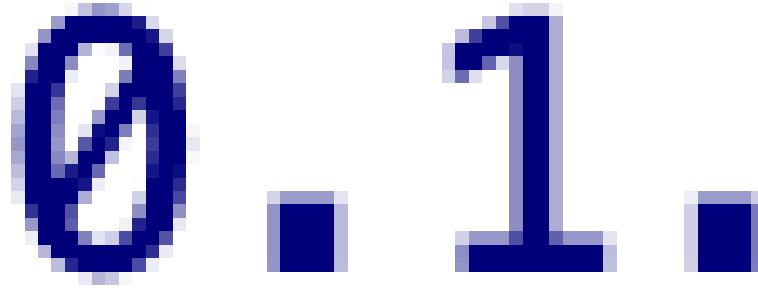


Figure 3.2: Schema characteristics: Order and Defining Length examples

Selection Effect

If $m(H, t)$ is the number of strings matching schema H at generation t , and $f(H)$ is the average fitness of strings matching H , then:

$$E[m(H, t + 1)] \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \quad (3.3)$$

where \bar{f} is the average fitness of the population.

This means schemas with above-average fitness will increase in representation.

Crossover Effect

Crossover can disrupt a schema if the crossover point falls between the defining positions. The probability of schema survival is:

$$P_s = 1 - p_c \cdot \frac{\delta(H)}{l - 1} \quad (3.4)$$

where:

- p_c is the crossover probability
- l is the string length

Mutation Effect

The probability that a schema survives mutation is:

$$P_m = (1 - p_m)^{o(H)} \quad (3.5)$$

where p_m is the mutation probability per bit.

Combined Schema Theorem

Combining all effects:

$$E[m(H, t + 1)] \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \cdot \left(1 - p_c \cdot \frac{\delta(H)}{l - 1}\right) \cdot (1 - p_m)^{o(H)} \quad (3.6)$$

3.2.4 Building Block Hypothesis

The building block hypothesis states that:

- Short, low-order, above-average schemas (building blocks) increase exponentially
- GA combines these building blocks to form optimal solutions
- Good solutions contain good building blocks

Characteristics of Good Building Blocks

1. **Short defining length:** $\delta(H)$ is small
2. **Low order:** $o(H)$ is small
3. **Above-average fitness:** $f(H) > \bar{f}$

3.2.5 Role of Schema Order in Genetic Algorithms

Pattern Specificity Level

The order of a schema indicates the level of specificity of the pattern represented in a chromosome. The higher the order, the more specific the pattern described. For example, a low-order schema like `1 * * * *` still represents many possible chromosomes because only one position is fixed, while a high-order schema like `101011` is very specific and only matches one particular chromosome. Thus, order plays a role in determining how broad a schema's representation is within the population.

Survival Probability in Evolution

The order of a schema greatly influences the schema's chance of surviving through the evolution process. In genetic algorithms, two main operators that often cause changes are crossover and mutation.

Low-order schemas are relatively safer against these changes because they have few fixed positions. For example, the schema `1 * * * *` only locks one bit at the beginning. If mutation occurs at another position or crossover cuts through the middle, the chance of schema disruption is very small. In other words, the fewer fixed bits that must be maintained, the greater the likelihood that the schema will survive and be inherited to the next generation.

Conversely, high-order schemas like `101011` have many fixed bits that must be exactly the same to remain valid. In such conditions, even one small mutation at one of the fixed positions can destroy the entire schema. Similarly, in the crossover process, the probability of being cut between fixed positions becomes larger. As a result, high-order schemas often quickly disappear from the population because they struggle to survive the combination of genetic variations that occur.

Relationship with Selection

Although they appear simple, low-order schemas actually play a very important role in genetic algorithms. The simplicity of this structure allows schemas to be more resistant to damage from crossover and mutation, so these patterns survive more often and are inherited to the next generation. If a simple schema contains patterns relevant to the optimal solution—for example, certain bit combinations that increase fitness value—then that schema will appear repeatedly on various chromosomes in the population.

This phenomenon aligns with the building block hypothesis put forward by Holland. Genetic algorithms do not directly search for complex solutions as a whole, but work by maintaining and arranging simple pattern blocks that have a positive contribution to fitness. These blocks are then combined through selection, crossover, and mutation, thus forming more complex genetic structures that approach the optimal solution.

The selection process plays a central role here. Individuals who have schemas with high contribution to fitness will be selected more often to reproduce. Thus, beneficial simple schemas can spread widely in the population. Over time, combinations of building blocks from various simple schemas produce solutions that are not only more complex, but also more efficient in solving problems.

3.2.6 Relationship Between Holland Schema and Genome

The number of genome sequences that can be represented by a schema depends on the number of don't care symbols (*). Each don't care symbol can have a value of 0 or 1, so a schema with k don't care symbols will produce 2^k possible genome sequences.

Examples:

- S_4 has 1 don't care, so there are $2^1 = 2$ possible genome sequences: 110010, 110110
- S_5 has 2 don't care, so there are $2^2 = 4$ possible genome sequences: 1110000, 1110100, 0110000, 0110100
- S_6 has 3 don't care, so there are $2^3 = 8$ possible genome sequences: 101100111, 101100110, 101100010, 101100011, 100100111, 100100110, 100100011, 100100111

3.2.7 Schema Functions in Genetic Algorithms

- Provides a powerful and compact way to represent well-defined similarities among strings
- Serves as the theoretical basis for explaining how good (fit) patterns in the population can be maintained and multiplied through crossover and mutation
- Helps understand that Genetic Algorithms work not only on individual strings, but also on classes of strings that share patterns (schemas)

3.3 Implicit Parallelism

GAs process many schemas simultaneously. For a population of size n with string length l :

- Number of possible schemas: 3^l
- Useful schemas processed: $O(n^3)$

This massive implicit parallelism is a key strength of GAs.

3.4 Deception and Schema Theory

3.4.1 Deceptive Problems

Problems where low-order building blocks mislead the search away from the global optimum.

Example: Trap function where building blocks point toward local optima.

3.4.2 Overcoming Deception

- Increase population size
- Use diversity-preserving techniques
- Apply linkage learning
- Use multi-objective approaches

3.5 Practical Implications

3.5.1 Encoding Design

- Minimize epistasis (gene interactions)
- Keep related variables close together
- Use appropriate representation for building blocks

3.5.2 Parameter Settings

- Low mutation rate to preserve building blocks
- Moderate crossover rate for effective recombination
- Sufficient population size for schema sampling

3.6 Limitations of Schema Theory

- Assumes infinite population size
- Ignores finite population effects
- Doesn't account for epistasis
- Limited to binary representations
- Overlooks linkage effects

3.7 Modern Extensions

3.7.1 Walsh Analysis

Mathematical framework extending schema theory using Walsh functions.

3.7.2 Fitness Landscapes

Analysis of problem difficulty using landscape topology.

3.7.3 No Free Lunch Theorem

States that no algorithm is superior across all possible problems.

3.8 Chapter Summary

This chapter covered the genetic algorithm cycle and Holland's schema theory. The GA cycle provides a systematic approach to evolutionary search, while schema theory explains why GAs work by processing building blocks in parallel. Understanding these concepts is essential for effective GA design and application.

3.9 Key Concepts

- GA cycle phases and their purposes
- Schema representation and properties
- Schema theorem and its implications
- Building block hypothesis
- Implicit parallelism in GAs
- Deception and its challenges

Chapter 4

Genetic Algorithm Encoding

4.1 Introduction to Encoding

Encoding (also called representation) is the process of transforming problem solutions into a form that genetic algorithms can manipulate [22, 31]. The choice of encoding significantly impacts GA performance and is one of the most critical design decisions.

4.2 Requirements for Good Encoding

4.2.1 Completeness

Every possible solution to the problem should have at least one corresponding representation in the genetic encoding.

4.2.2 Soundness

Every encoded string should correspond to a valid solution of the problem.

4.2.3 Non-redundancy

Each solution should have a unique representation (one-to-one mapping preferred).

4.2.4 Locality

Small changes in genotype should correspond to small changes in phenotype, ensuring smooth search.

4.3 Binary Encoding

Binary encoding is the most traditional and widely studied representation in genetic algorithms [26, 22].

4.3.1 Basic Binary Encoding

Solutions are represented as fixed-length binary strings.

Example: Optimizing $f(x) = x^2$ where $x \in [0, 31]$

- Use 5-bit representation
- $x = 13 \rightarrow 01101_2$
- $x = 27 \rightarrow 11011_2$

4.3.2 Decoding Binary Strings

For integer values in range $[x_{min}, x_{max}]$ using l bits:

$$x = x_{min} + \frac{x_{max} - x_{min}}{2^l - 1} \times \text{binary_value} \quad (4.1)$$

Example: 5-bit string $10110_2 = 22_{10}$ for range $[0, 31]$:

$$x = 0 + \frac{31 - 0}{2^5 - 1} \times 22 = 0 + \frac{31}{31} \times 22 = 22 \quad (4.2)$$

4.3.3 Multi-variable Binary Encoding

For multiple variables, concatenate individual encodings:

Example: Two variables $x_1 \in [0, 15]$, $x_2 \in [0, 7]$

- x_1 : 4 bits
- x_2 : 3 bits
- Combined: 7-bit string x_1x_2
- String 1011001: $x_1 = 1011_2 = 11$, $x_2 = 001_2 = 1$

4.3.4 Advantages of Binary Encoding

- Simple and easy to implement
- Well-studied theoretical foundation
- Standard crossover and mutation operators available
- Schema theory directly applicable

4.3.5 Disadvantages of Binary Encoding

- Hamming cliff problem (adjacent values may have large Hamming distance)
- Fixed precision may be inadequate
- Long strings for high precision
- Epistasis between variables
- Not natural for many problems with real-valued features
- Often requires correction after genetic operations

4.4 Overview of Encoding Types

4.4.1 Problem Identification and Formulation

One thing to consider when using genetic algorithms is how we represent the data we are studying. Therefore, we must determine which encoding is most suitable for the problem we face. Like other search or learning methods, how encoding is done on candidate solutions is one of, if not the most important factor in the success of the genetic algorithm itself. Most genetic algorithm applications use bit strings with predetermined length and order. However, recently many experiments have been conducted with other types of encoding.

4.4.2 What is Encoding

Encoding is the process of representing individual genes. This process can be done with bits, numbers, trees, arrays, lists, or any other object. The encoding is primarily dependent on how the related object is solved. For example, encoding can be done with integers or real numbers.

4.4.3 Encoding Type Categories

1. Binary Encoding

Binary encoding, i.e., bit strings, is the simplest and most widely used type of encoding. Many renowned researchers use binary encoding in their problems, for example Holland, as in the Holland schema. Each chromosome is encoded as a binary string. Each bit in it represents a feature or characteristic of that solution. Binary encoding can store a lot of information with a minimal number of alleles (e.g., 0 or 1, compared to decimal which has many alleles).

Here are some examples of chromosomes with binary encoding:

- Chromosome 1: 1 1 0 1 0 0 0 1 1 0 1 0
- Chromosome 2: 0 1 1 1 1 1 1 1 1 1 0 0

On the other hand, this encoding is not natural and difficult to implement for most problems that generally have features in the form of real numbers, and corrections often have to be made after genetic operations are completed.

2. Value Encoding

Problems generally have complex values. Values like these are quite difficult to represent in binary form. This is where value encoding is used. Each chromosome is a string of values, where the value in each string itself can be anything related to the problem. This includes integers, letters, real numbers, and other values.

Here are some examples of chromosomes with value encoding:

- Chromosome A: 1.2324, 5.3242, 0.4556, 2.3293, 2.4545
- Chromosome B: ABDJEIFJDHDIERJFDLDFLEGT
- Chromosome C: (back), (back), (right), (forward), (left)

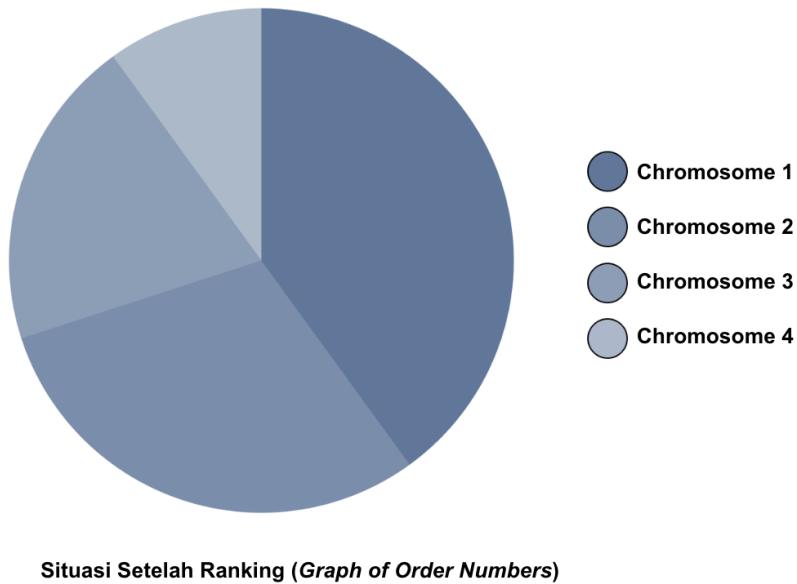


Figure 4.1: Binary encoding examples and characteristics

3. Permutation Encoding

There are some special problems that do not look at chromosomes from their individual gene values. They look at the position of occurrence of each gene, as in ordering problems. Each gene can be an integer or real, representing numbers in a sequence.

Here are some examples of chromosomes with permutation encoding:

- Chromosome A: 1 5 3 2 6 4 7 9 8
- Chromosome B: 8 5 6 7 2 3 1 4 9

4. Tree Encoding

Tree encoding is most often used in program expressions that constantly change in genetic programming. Each chromosome is a tree of an object such as a function or command from a programming language. Tree encoding has a number of advantages, one of which is enabling an open search space, because basically trees of any size can be formed through crossover and mutation.

However, this openness can be one of the weaknesses of this encoding method. The formed tree can become too large without being controlled, thus hindering the formation of more structured and hierarchical solution candidates. In addition, the resulting tree tends to be difficult to interpret and simplify due to its large size.

4.5 Gray Code Encoding

Gray code addresses the Hamming cliff problem by ensuring adjacent values differ by only one bit [25, 31].

4.5.1 Binary to Gray Code Conversion

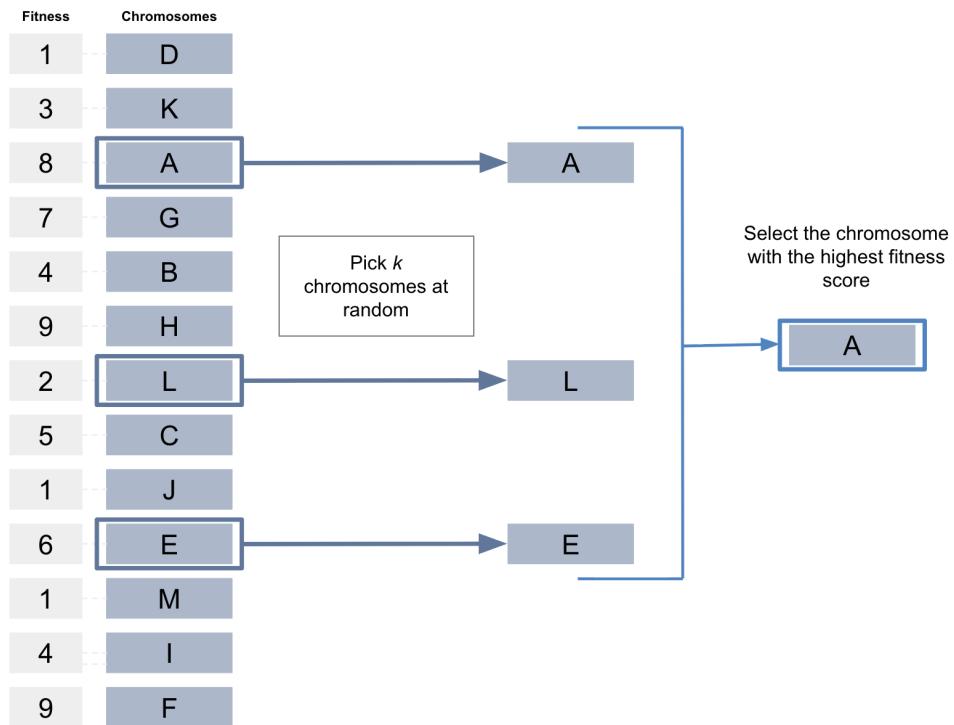


Figure 4.2: Various encoding types: Value, Permutation, and Tree encoding

Algorithm 2 Binary to Gray Code

```

 $g_0 = b_0$  (most significant bit)
for  $i = 1$  to  $n - 1$  do
     $g_i = b_{i-1} \oplus b_i$  (XOR operation)
end for
  
```

4.5.2 Gray Code to Binary Conversion

Algorithm 3 Gray Code to Binary

```

 $b_0 = g_0$ 
for  $i = 1$  to  $n - 1$  do
     $b_i = b_{i-1} \oplus g_i$ 
end for
  
```

4.5.3 Example: 4-bit Gray Code

Decimal	Binary	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100

Table 4.1: Binary vs Gray Code Representation

4.6 Real-valued Encoding

Real-valued encoding directly represents solutions as vectors of real numbers [7, 31].

4.6.1 Representation

Individual: $\mathbf{x} = (x_1, x_2, \dots, x_n)$ where $x_i \in \mathbb{R}$

Example: Function optimization $f(x_1, x_2) = x_1^2 + x_2^2$ Individual: $(2.5, -1.7)$

4.6.2 Advantages

- Natural representation for continuous problems
- No precision limitations
- More compact than binary encoding
- Smooth fitness landscapes

4.6.3 Disadvantages

- Requires specialized operators
- Loss of building block analysis
- Parameter tuning for operators

4.6.4 Real-valued Crossover Operators

Arithmetic Crossover

$$\begin{aligned}\text{Offspring}_1 &= \alpha \times \text{Parent}_1 + (1 - \alpha) \times \text{Parent}_2 \\ \text{Offspring}_2 &= (1 - \alpha) \times \text{Parent}_1 + \alpha \times \text{Parent}_2\end{aligned}\tag{4.3}$$

where $\alpha \in [0, 1]$ is a random number.

BLX- α Crossover

For parents x_1 and x_2 :

$$\text{Offspring} \sim U[x_{\min} - \alpha \cdot I, x_{\max} + \alpha \cdot I] \quad (4.4)$$

where:

- $x_{\min} = \min(x_1, x_2)$
- $x_{\max} = \max(x_1, x_2)$
- $I = x_{\max} - x_{\min}$
- $\alpha = 0.5$ typically

4.6.5 Real-valued Mutation Operators

Gaussian Mutation

$$x'_i = x_i + \mathcal{N}(0, \sigma^2) \quad (4.5)$$

where $\mathcal{N}(0, \sigma^2)$ is Gaussian noise with mean 0 and variance σ^2 .

Non-uniform Mutation

$$x'_i = \begin{cases} x_i + \Delta(t, x_{\max} - x_i) & \text{if random bit is 0} \\ x_i - \Delta(t, x_i - x_{\min}) & \text{if random bit is 1} \end{cases} \quad (4.6)$$

where:

$$\Delta(t, y) = y \times \left(1 - r^{(1-t/T)^b}\right) \quad (4.7)$$

4.7 Integer Encoding

For problems with integer variables.

4.7.1 Representation

Individual: $\mathbf{x} = (x_1, x_2, \dots, x_n)$ where $x_i \in \mathbb{Z}$

Example: Knapsack problem with item quantities Individual: $(3, 0, 2, 1, 4)$ represents taking 3 of item 1, 0 of item 2, etc.

4.7.2 Integer Crossover

- Discrete uniform crossover
- Arithmetic crossover with rounding
- Two-point crossover

4.7.3 Integer Mutation

- Random resetting: $x_i = \text{random integer in range}$
- Creep mutation: $x_i = x_i \pm \text{small integer}$
- Gaussian mutation with rounding

4.8 Permutation Encoding

For problems where solution is an ordering of elements.

4.8.1 Representation

Individual: Permutation of $\{1, 2, \dots, n\}$

Example: Traveling Salesman Problem Individual: $(3, 1, 4, 2, 5)$ represents visiting cities in order $3 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 3$

4.8.2 Permutation Crossover Operators

Order Crossover (OX)

1. Select random substring from parent 1
2. Copy substring to offspring in same positions
3. Fill remaining positions with elements from parent 2 in order they appear

Example:

$$\text{Parent 1: } (1, 2, 3, 4, 5, 6, 7, 8, 9) \quad (4.8)$$

$$\text{Parent 2: } (9, 3, 7, 8, 2, 6, 5, 1, 4) \quad (4.9)$$

$$\text{Selected: } (_, _, 3, 4, 5, 6, _, _, _) \quad (4.10)$$

$$\text{Offspring: } (7, 8, 3, 4, 5, 6, 2, 1, 9) \quad (4.11)$$

Partially Mapped Crossover (PMX)

1. Select two crossover points
2. Exchange segments between parents
3. Resolve conflicts using mapping relationship

Cycle Crossover (CX)

Preserves position information from both parents by identifying cycles.

4.8.3 Permutation Mutation Operators

Swap Mutation

Randomly select two positions and swap their values.

Insert Mutation

Remove element from one position and insert at another position.

Inversion Mutation

Reverse order of elements in randomly selected substring.

Scramble Mutation

Randomly shuffle elements in selected substring.

4.9 Tree Encoding

For problems with hierarchical or tree structures.

4.9.1 Applications

- Genetic programming
- Decision trees
- Parse trees
- Circuit design

4.9.2 Representation

Trees with:

- Internal nodes: operators/functions
- Leaf nodes: terminals/variables
- Variable tree size and shape

4.9.3 Tree Crossover

Exchange randomly selected subtrees between parents.

4.9.4 Tree Mutation

- Replace subtree with randomly generated subtree
- Change node value
- Grow or shrink tree

4.10 Problem-specific Encodings

4.10.1 Graph Coloring

Individual: (c_1, c_2, \dots, c_n) where c_i is color of vertex i

4.10.2 Job Scheduling

Individual: Priority list or schedule representation

4.10.3 Neural Network Weights

Individual: Vector of real-valued connection weights

4.11 Choosing the Right Encoding

4.11.1 Factors to Consider

- Problem domain and constraints
- Required precision
- Search space characteristics
- Available operators
- Computational efficiency

4.11.2 Guidelines

- Use natural representation when possible
- Ensure all solutions are reachable
- Minimize epistasis between variables
- Consider hybrid approaches
- Test multiple encodings empirically

4.12 Chapter Summary

This chapter covered various encoding schemes for genetic algorithms. The choice of encoding is crucial and should match the problem characteristics. Binary encoding provides theoretical foundation, real-valued encoding suits continuous optimization, permutation encoding handles ordering problems, and specialized encodings address domain-specific requirements.

4.13 Key Concepts

- Encoding requirements: completeness, soundness, non-redundancy
- Binary vs Gray code encoding
- Real-valued representation and operators
- Permutation encoding for ordering problems
- Tree encoding for hierarchical structures
- Problem-specific encoding considerations

Chapter 5

Selection Methods in Genetic Algorithms

5.1 Introduction to Selection

5.1.1 Definition and Function of Selection Operator

Selection is the process of choosing individuals from the population to become parents in the crossover process. This process determines which individuals will produce offspring in the next generation and how many offspring will be produced.

The main goal of selection is to give greater opportunities to individuals with higher fitness to be selected, with the hope that the offspring produced will also have higher fitness. Thus, selection functions as a mechanism that drives Genetic Algorithms toward optimal solutions through improving population quality from generation to generation.

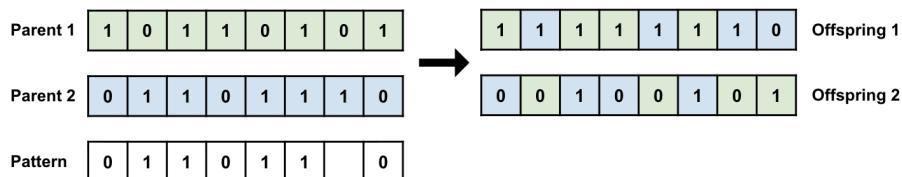


Figure 5.1: Basic selection process in Genetic Algorithms

5.1.2 Selection Pressure

Selection pressure is a measure of how much individuals with higher fitness are favored for selection compared to other individuals. The higher the selection pressure, the greater the chance that the best individuals are selected. This pressure plays an important role because it drives Genetic Algorithms to continuously improve the average fitness of the population from one generation to the next.

The magnitude of selection pressure greatly affects the convergence rate of Genetic Algorithms. If the pressure is high, Genetic Algorithms will reach solutions faster, but there is a risk of stopping too quickly at a wrong solution (premature convergence). Conversely, if the pressure is too low, the evolution process becomes slow because the best individuals are not spread enough.

Therefore, balance is needed. Selection must be strong enough to accelerate the search for solutions, but also maintain population diversity so that variation in the population is not lost. In this way, Genetic Algorithms can avoid premature convergence and still have the opportunity to find optimal or near-optimal solutions.

5.1.3 Types of Selection Operators

The main types of selection operators include:

1. Fitness Proportionate Selection (FPS)
 - Roulette Wheel Selection
 - Baker's SUS (Stochastic Universal Sampling)
2. Rank-Based Selection
3. Tournament Selection

5.2 Selection Pressure

Selection pressure is the degree to which better individuals are favored. It affects:

- **High pressure:** Fast convergence but risk of premature convergence
- **Low pressure:** Better exploration but slower convergence
- **Optimal pressure:** Balance between exploration and exploitation

5.3 Fitness Proportionate Selection (FPS)

The Genetic Algorithm developed by Holland uses Fitness Proportionate Selection (FPS) [26, 22], where the expected value of an individual (i.e., the expected number of times that individual will be selected for reproduction) is calculated as that individual's fitness divided by the population's average fitness.

In this method, each individual can be selected as a parent with a probability proportional to its fitness value. Therefore, individuals with higher fitness have greater opportunities to reproduce and spread their characteristics to the next generation. Thus, this method provides selection pressure on fitter individuals in the population, thus driving evolution toward better individuals over time.

5.3.1 Roulette Wheel Selection

Also known as fitness proportionate selection, individuals are selected with probability proportional to their fitness [22, 4, 5].

The simplest selection schema is roulette-wheel selection, also called stochastic sampling with replacement. This is a stochastic algorithm and involves the following technique:

Individuals are mapped to contiguous segments on a line, where the size of each segment is equal to that individual's fitness value. A random number is generated, and

the individual whose segment spans that random number is selected. This process is repeated until the desired number of individuals is reached, called the mating population. This technique is analogous to a roulette wheel, where each slice is proportional in size to the fitness value.

Number of Individual	Fitness Value	Selection Probability	Interval
1	2.0	0.18	[0.00, 0.18]
2	1.8	0.16	[0.18, 0.34]
3	1.6	0.15	[0.34, 0.49]
4	1.4	0.13	[0.49, 0.62]
5	1.2	0.11	[0.62, 0.73]
6	1.0	0.09	[0.73, 0.82]
7	0.8	0.07	[0.82, 0.89]
8	0.6	0.06	[0.89, 0.95]
9	0.4	0.03	[0.95, 0.98]
10	0.2	0.02	[0.98, 1.00]
11	0.0	0.0	—

Table 5.1: Selection probability and fitness value (from Buku Ajar)

Table 5.1 shows the selection probabilities for 11 individuals, with linear ranking with selective pressure of 2, along with their fitness values. Individual 1 is the individual with the highest fitness and occupies the largest interval, while individual 10 as the individual with the second lowest fitness has the smallest interval on the line. Individual 11, with the lowest fitness, has fitness value = 0 and gets no chance for reproduction.

To select the mating population, a number of uniformly distributed random numbers (uniformly distributed between 0.0 and 1.0) are generated independently.

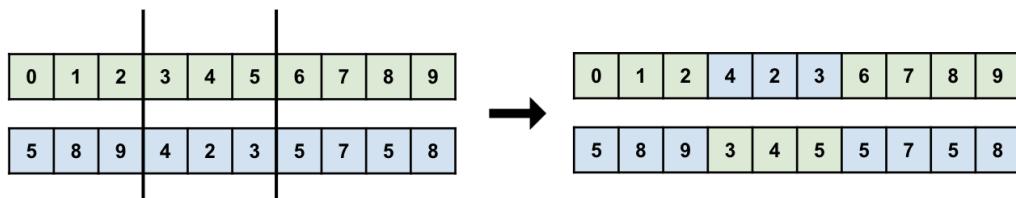


Figure 5.2: Roulette-wheel selection process with sample trials

The disadvantage of roulette-wheel selection is that although it provides zero bias, it does not guarantee minimum spread.

Algorithm

Selection Probability

The probability of selecting individual i is:

$$P_i = \frac{f_i}{\sum_{j=1}^N f_j} \quad (5.1)$$

Algorithm 4 Roulette Wheel Selection

Calculate total fitness: $F = \sum_{i=1}^N f_i$
 Generate random number: $r \sim U[0, F]$
 Set cumulative fitness: $sum = 0$
for $i = 1$ to N **do**
 $sum = sum + f_i$
 if $sum \geq r$ **then**
 Select individual i
 break
 end if
end for

Example

Individual	Fitness	Probability	Cumulative
1	10	0.25	0.25
2	20	0.50	0.75
3	5	0.125	0.875
4	5	0.125	1.0
Total	40	1.0	

Table 5.2: Roulette Wheel Selection Example

If random number $r = 0.6$, individual 2 is selected.

Advantages

- Simple to implement
- Fitness proportionate selection
- All individuals have chance of selection

Disadvantages

- Premature convergence with high fitness variance
- Poor selection pressure with similar fitness values
- Problems with negative fitness values
- Scaling issues

5.3.2 Stochastic Universal Sampling (SUS)

Improved version of roulette wheel selection that reduces variance [9].

Baker's SUS

Stochastic Universal Sampling (SUS) provides zero bias and minimum spread [9]. Individuals are mapped to contiguous segments on a line, where the size of each segment equals its fitness value, exactly as in roulette-wheel selection. In this method, equally spaced pointers are placed on the line equal to the number of individuals to be selected.

Let $N_{Pointer}$ be the number of individuals to be selected, then the distance between pointers is $1/N_{Pointer}$, and the position of the first pointer is determined by a random number generated in the range $[0, 1/N_{Pointer}]$.

For example, to select 6 individuals, the distance between pointers is $1/6 = 0.167$.

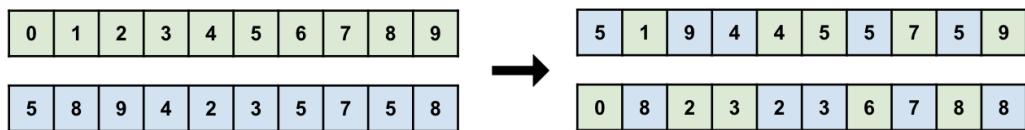


Figure 5.3: Stochastic universal sampling with equally spaced pointers

Stochastic universal sampling ensures offspring selection that is closer to the expected values compared to roulette-wheel selection.

Algorithm

Algorithm 5 Stochastic Universal Sampling

Calculate total fitness: $F = \sum_{i=1}^N f_i$

Calculate pointer distance: $distance = F/N$

Generate random start: $start \sim U[0, distance]$

Create pointers: $pointer_i = start + i \times distance$ for $i = 0, 1, \dots, N - 1$

for each pointer **do**

 Select individual using roulette wheel logic

end for

Advantages over Roulette Wheel

- Lower variance
- More uniform selection
- Guaranteed expected number of selections

5.4 Rank-based Selection

Rank-based selection assigns selection probabilities based on fitness rank rather than raw fitness values [23, 6].

5.4.1 Overview

Ranked-Based Selection introduces a different approach to selection in Genetic Algorithms. Instead of directly using fitness values to determine selection probability, individuals in the population are first sorted (ranked) based on their fitness values, then each individual is assigned a rank. The selection probability is then calculated based on that rank, not the actual fitness value.

This rank-based approach helps reduce problems associated with direct fitness-based selection, such as premature convergence and domination by a few very fit individuals in the early stages of the optimization process. By assigning ranks and using them for selection, Ranked-Based Selection provides more balanced and controlled selection pressure, allowing better exploration of the search space and maintaining diversity in the population.

Rankings are typically assigned linearly or exponentially, where the best individual receives the highest rank and the worst individual receives the lowest rank. Selection probability is then calculated based on that ranking using a predetermined formula or mapping function. This mapping function can be adjusted to control selection pressure, where higher pressure will favor individuals with the highest ranks, while lower pressure provides a more even distribution of selection probabilities.

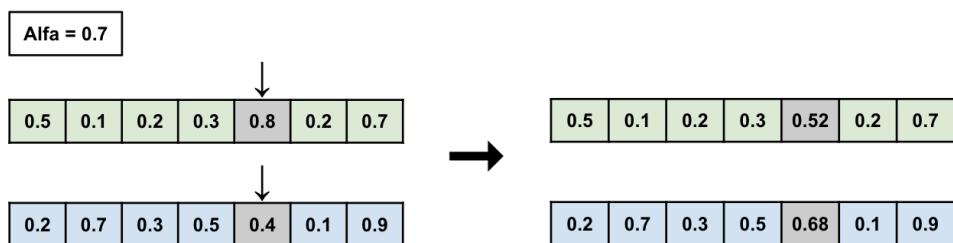


Figure 5.4: How the situation changes after converting fitness to order number (rank)

5.4.2 Linear Ranking

$$P_i = \frac{1}{N} \left[\eta^- + (\eta^+ - \eta^-) \frac{rank_i - 1}{N - 1} \right] \quad (5.2)$$

where:

- $rank_i$ is the rank of individual i ($1 = \text{worst}$, $N = \text{best}$)
- η^+ is the expected number of copies for best individual
- η^- is the expected number of copies for worst individual
- $\eta^+ + \eta^- = 2$ (to maintain population size)
- Typically: $\eta^+ = 2.0$, $\eta^- = 0.0$

5.4.3 Exponential Ranking

$$P_i = \frac{1 - e^{-rank_i}}{c} \quad (5.3)$$

where c is a normalization constant ensuring $\sum P_i = 1$.

5.4.4 Advantages of Rank Selection

- Consistent selection pressure
- Handles negative fitness values
- Prevents premature convergence
- Scale-independent

5.4.5 Disadvantages

- Requires sorting population
- Loss of fitness magnitude information
- Computational overhead

5.5 Tournament Selection

Tournament selection randomly selects k individuals and chooses the best among them [22, 8].

5.5.1 Overview

Tournament selection is a strong and widely used selection mechanism in Genetic Algorithms because it can maintain a balance between diversity maintenance and selective pressure [8]. Unlike roulette-wheel selection, which directly depends on an individual's fitness relative to the total population fitness, tournament selection works by holding "tournaments" among subsets of individuals, and the winner of each tournament is selected for reproduction.

The main concept of tournament selection is quite simple: instead of considering the entire population at once, a subset of individuals is randomly selected to compete with each other. The individual with the highest fitness in that "tournament" is then selected. This process is repeated until the desired number of individuals for reproduction is reached.

This method has several advantages: tournament selection maintains diversity because individuals with low fitness still have the opportunity to participate in tournaments. Additionally, this method allows selective pressure to be adjusted by setting the tournament size.

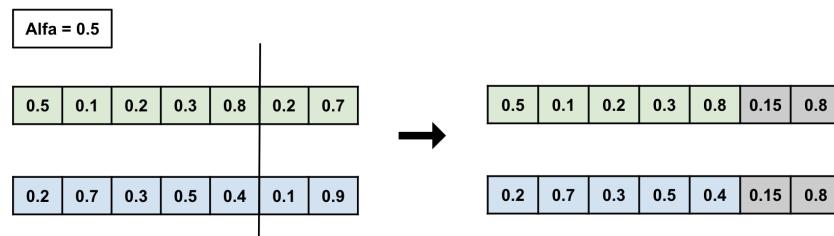


Figure 5.5: Tournament selection mechanism

5.5.2 Tournament Selection Mechanism

1. Determine tournament size (k), i.e., the number of individuals participating in each tournament.
2. Randomly select k individuals from the population.
3. Compare the fitness values of these individuals and select the individual with the highest fitness as the winner.
4. Add the winner to the mating pool.
5. Repeat steps 2–4 until the desired number of individuals is reached.

5.5.3 Binary Tournament

Most common form with $k = 2$.

Algorithm 6 Binary Tournament Selection

```

Randomly select individual  $i$ 
Randomly select individual  $j$  (where  $j \neq i$ )
if  $f_i > f_j$  then
    Select individual  $i$ 
else
    Select individual  $j$ 
end if
```

5.5.4 k-Tournament Selection

Algorithm 7 k-Tournament Selection

```

Create empty tournament set  $T$ 
for  $i = 1$  to  $k$  do
    Randomly select individual and add to  $T$ 
end for
Select best individual from  $T$ 
```

5.5.5 Tournament Size Effects

- $k = 1$: Random selection (no pressure)
- Small k : Low selection pressure
- Large k : High selection pressure
- $k = N$: Always selects best individual

5.5.6 Selection Probability

For individual with rank r out of N ($1 = \text{worst}$, $N = \text{best}$):

$$P_i = \frac{1}{N} \binom{N}{k} \sum_{j=0}^{r-1} \binom{j}{k-1} \binom{N-j-1}{0} \quad (5.4)$$

For binary tournament ($k = 2$):

$$P_i = \frac{2r-1}{N^2} \quad (5.5)$$

5.5.7 Advantages

- Simple implementation
- No global fitness information needed
- Adjustable selection pressure
- Parallelizable
- Handles negative fitness values

5.5.8 Disadvantages

- May select same individual multiple times
- Sensitive to tournament size parameter

5.6 Truncation Selection

Select the top μ individuals from population of size λ .

5.6.1 Algorithm

Algorithm 8 Truncation Selection

Sort population by fitness (descending)
 Select top μ individuals
 Create $\lambda - \mu$ offspring from selected parents

5.6.2 Selection Ratio

$$\text{Selection ratio} = \frac{\mu}{\lambda} \quad (5.6)$$

Common values: 0.5 (select top 50)

5.6.3 Advantages

- Simple and deterministic
- High selection pressure
- Efficient implementation

5.6.4 Disadvantages

- High risk of premature convergence
- Loss of diversity
- All-or-nothing selection

5.7 Boltzmann Selection

Selection probability based on Boltzmann distribution from statistical mechanics.

5.7.1 Formula

$$P_i = \frac{e^{f_i/T}}{\sum_{j=1}^N e^{f_j/T}} \quad (5.7)$$

where T is the temperature parameter.

5.7.2 Temperature Schedule

- High T : Nearly uniform selection (exploration)
- Low T : Strong selection pressure (exploitation)
- Common schedule: $T(t) = T_0 \cdot \alpha^t$ where $\alpha < 1$

5.7.3 Advantages

- Adaptive selection pressure
- Good balance of exploration/exploitation
- Prevents premature convergence

5.7.4 Disadvantages

- Requires temperature scheduling
- Computationally expensive (exponentials)
- Parameter tuning required

5.8 Elitist Selection

Ensures that the best individuals are preserved across generations.

5.8.1 Pure Elitism

Always copy the best individual(s) to the next generation.

5.8.2 Elitist Replacement

Replace worst individuals with best from previous generation if they're better.

5.8.3 Benefits

- Guarantees monotonic improvement
- Prevents loss of good solutions
- Faster convergence to local optima

5.8.4 Drawbacks

- May reduce diversity
- Risk of premature convergence
- Can slow exploration

5.9 Diversity-Preserving Selection

5.9.1 Fitness Sharing

Reduce fitness of similar individuals to maintain diversity.

$$f'_i = \frac{f_i}{\sum_{j=1}^N sh(d_{ij})} \quad (5.8)$$

where:

$$sh(d) = \begin{cases} 1 - \left(\frac{d}{\sigma_{share}}\right)^\alpha & \text{if } d < \sigma_{share} \\ 0 & \text{otherwise} \end{cases} \quad (5.9)$$

5.9.2 Crowding

Replace similar individuals in the population.

5.9.3 Speciation

Maintain multiple sub-populations (species) simultaneously.

5.10 Multi-objective Selection

For problems with multiple conflicting objectives.

5.10.1 Pareto Dominance

Individual \mathbf{x} dominates \mathbf{y} if:

- \mathbf{x} is at least as good as \mathbf{y} in all objectives
- \mathbf{x} is strictly better than \mathbf{y} in at least one objective

5.10.2 Non-dominated Sorting

1. Identify all non-dominated individuals (Rank 1)
2. Remove them and find next non-dominated set (Rank 2)
3. Continue until all individuals are ranked

5.10.3 NSGA-II Selection

Combines non-dominated sorting with crowding distance.

5.11 Selection Comparison

Method	Pressure	Diversity	Complexity	Scalability	Parameters
Roulette Wheel	Variable	Poor	$O(N)$	Poor	None
SUS	Variable	Good	$O(N)$	Poor	None
Rank Linear	Constant	Good	$O(N \log N)$	Good	η^+, η^-
Tournament	Adjustable	Good	$O(1)$	Excellent	k
Truncation	High	Poor	$O(N \log N)$	Good	μ/λ
Boltzmann	Adaptive	Excellent	$O(N)$	Good	$T(t)$

Table 5.3: Comparison of Selection Methods

5.12 Selection Guidelines

5.12.1 Problem Characteristics

- **Unimodal:** High selection pressure (truncation, large tournament)
- **Multimodal:** Moderate pressure (binary tournament, rank selection)
- **Deceptive:** Low pressure with diversity preservation

5.12.2 Population Size

- Small populations: Lower selection pressure
- Large populations: Higher pressure acceptable

5.12.3 Generation Number

- Early generations: Lower pressure for exploration
- Later generations: Higher pressure for exploitation

5.13 Hybrid Selection Strategies

5.13.1 Adaptive Selection

Change selection method or parameters during evolution.

5.13.2 Multi-level Selection

Apply different selection at different levels (e.g., parent selection vs. survival selection).

5.13.3 Combined Methods

Use multiple selection methods simultaneously.

5.14 Chapter Summary

This chapter covered various selection methods in genetic algorithms. Selection balances exploration and exploitation, with different methods offering different selection pressures and characteristics. Tournament selection is often preferred for its simplicity and effectiveness, while rank-based methods provide consistent pressure. The choice depends on problem characteristics, population size, and desired convergence behavior.

5.15 Key Concepts

- Selection pressure and its effects
- Proportional vs. rank-based selection
- Tournament selection and its variants
- Elitism and diversity preservation
- Multi-objective selection methods
- Guidelines for choosing selection methods

Chapter 6

Crossover (Recombination) in Genetic Algorithms

6.1 Introduction to Crossover

Crossover, also known as recombination, is the primary genetic operator in genetic algorithms [26, 22, 32]. It combines genetic material from two or more parent solutions to create offspring, potentially inheriting beneficial traits from multiple parents. Crossover exploits existing solutions to explore new regions of the search space.

6.2 Biological Inspiration

In nature, sexual reproduction combines genetic material from two parents:

- **Crossing over:** Exchange of genetic segments between homologous chromosomes
- **Independent assortment:** Random distribution of chromosomes
- **Genetic diversity:** Offspring differ from parents
- **Building blocks:** Beneficial gene combinations are preserved and mixed

6.3 Crossover Principles

6.3.1 Exploration vs. Exploitation

- **Exploitation:** Combines good building blocks from parents
- **Exploration:** Creates new combinations not present in parents
- **Heritability:** Offspring resemble parents but with variations

6.3.2 Crossover Probability

Crossover is typically applied with probability p_c (usually 0.6-0.9):

- High p_c : More exploration, faster convergence

- Low p_c : More exploitation of current solutions
 - $p_c = 1.0$: Always apply crossover
 - $p_c = 0.0$: No crossover (mutation-only evolution)

6.4 Binary Crossover Operators

6.4.1 Definition and Function of Crossover Operator

Crossover is a genetic operator used to vary the arrangement of chromosomes from one generation to the next [20, 44, 2]. The crossover method used depends on the encoding method applied.

Crossover occurs in three stages:

1. The reproduction operator randomly selects a pair of individual strings for the mating process.
 2. A cross site is randomly selected along the length of the string.
 3. The values after the cross site are exchanged between the two strings to form new offspring.

In binary chromosome representation, each individual in the population is represented as a sequence of bits (0 and 1) that express a potential solution to a problem.

6.4.2 One-Point Crossover

Single-point crossover is the simplest form of crossover. One crossover position is randomly selected, then the part of the two parents after that position is exchanged to form two new offspring. The strings resulting from this process have Positional Bias characteristics.

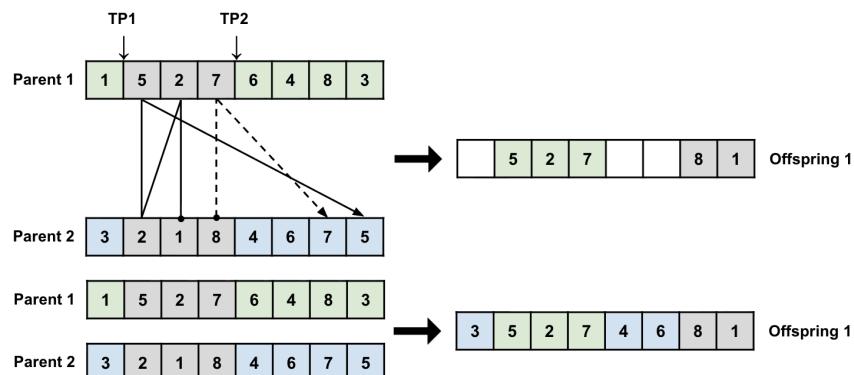


Figure 6.1: Single-Point Crossover for binary chromosomes

6.4.3 One-Point Crossover

Single crossover point divides chromosomes into two segments.

Algorithm

Algorithm 9 One-Point Crossover

Select random crossover point $k \in [1, l - 1]$

Create offspring:

$$\text{child}_1 = \text{parent}_1[1 : k] + \text{parent}_2[k + 1 : l]$$

$$\text{child}_2 = \text{parent}_2[1 : k] + \text{parent}_1[k + 1 : l]$$

Example

$$\text{Parent 1: } 1|1010011 \quad (6.1)$$

$$\text{Parent 2: } 0|0111100 \quad (6.2)$$

$$\text{Child 1: } 1|0111100 \quad (6.3)$$

$$\text{Child 2: } 0|1010011 \quad (6.4)$$

Crossover point at position 1.

Characteristics

- Simple and efficient
- Preserves long building blocks near chromosome ends
- May disrupt building blocks crossing the crossover point
- Positional bias (end positions less likely to be separated)

6.4.4 Two-Point Crossover

Two crossover points create three segments.

Multi-point crossover has a mechanism like single-point crossover; the difference is the number of randomly selected positions. In Multi-point crossover, a number N of positions are randomly selected along the length of the chromosome. These positions are exchanged to form two new offspring.

The objectives of multi-point crossover include increasing genetic diversity in the population, reducing positional bias, and increasing the chances of recombination from various schemas or different solution blocks. However, using too many cut points can become too disruptive, as it can damage already good gene combinations (coadapted alleles).

Algorithm

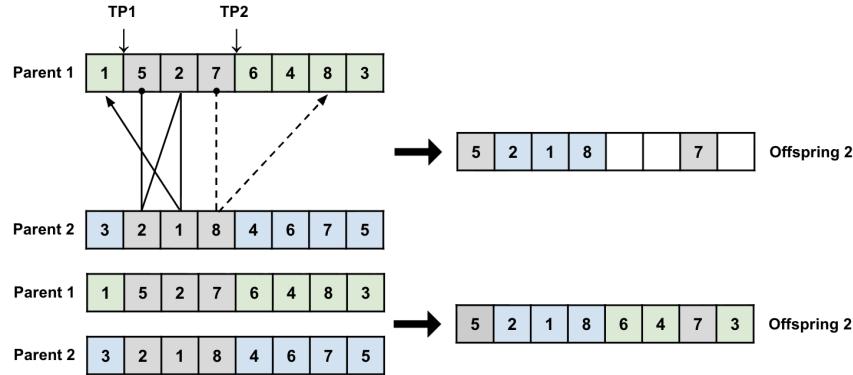


Figure 6.2: Multi-point Crossover for binary chromosomes

Algorithm 10 Two-Point Crossover

Select two random points k_1, k_2 where $1 \leq k_1 < k_2 \leq l - 1$

Create offspring:

$$\text{child}_1 = \text{parent}_1[1 : k_1] + \text{parent}_2[k_1 + 1 : k_2] + \text{parent}_1[k_2 + 1 : l]$$

$$\text{child}_2 = \text{parent}_2[1 : k_1] + \text{parent}_1[k_1 + 1 : k_2] + \text{parent}_2[k_2 + 1 : l]$$

Example

$$\text{Parent 1: } 11|010|011 \quad (6.5)$$

$$\text{Parent 2: } 00|111|100 \quad (6.6)$$

$$\text{Child 1: } 11|111|011 \quad (6.7)$$

$$\text{Child 2: } 00|010|100 \quad (6.8)$$

Crossover points at positions 2 and 5.

Advantages

- Reduces positional bias
- Can preserve building blocks at chromosome ends
- More disruptive than one-point crossover

6.4.5 Uniform Crossover

Each gene is independently chosen from either parent [42, 16].

In uniform crossover, each gene (bit) is randomly selected from one of the corresponding genes on the parent chromosome. This selection process is done independently for each gene position. This method can be analogized to tossing a coin. If the result is "head", the gene is taken from parent 1; if the result is "tail", the gene is taken from parent 2.

This method provides equal opportunity for each gene to be inherited from one of the parents, thereby increasing genetic diversity and eliminating positional bias that typically appears in single-point crossover or multi-point crossover.

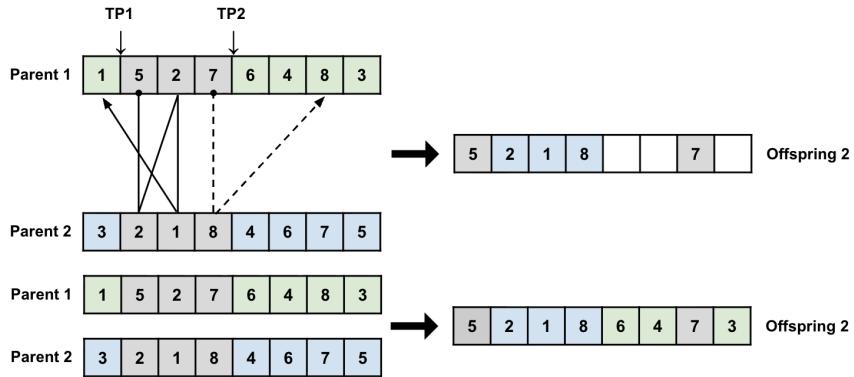


Figure 6.3: Uniform Crossover for binary chromosomes

Algorithm

Algorithm 11 Uniform Crossover

```

for each gene position  $i$  do
  Generate random number  $r \in [0, 1]$ 
  if  $r < 0.5$  then
     $child_1[i] = parent_1[i]$ ,  $child_2[i] = parent_2[i]$ 
  else
     $child_1[i] = parent_2[i]$ ,  $child_2[i] = parent_1[i]$ 
  end if
end for
  
```

Example with Mask

$$\text{Parent 1: } 11010011 \quad (6.9)$$

$$\text{Parent 2: } 00111100 \quad (6.10)$$

$$\text{Mask: } 10110100 \quad (6.11)$$

$$\text{Child 1: } 10111011 \quad (6.12)$$

$$\text{Child 2: } 01010100 \quad (6.13)$$

Mask bit 1: take from Parent 1, Mask bit 0: take from Parent 2.

Properties

- Maximum disruption potential
- No positional bias
- Good for problems where gene positions are independent
- May destroy long building blocks

6.4.6 Multi-Point Crossover

Generalization with k crossover points.

Characteristics

- $k = 0$: No crossover (copy parents)
- $k = 1$: One-point crossover
- $k = l - 1$: Uniform crossover (in expectation)
- As k increases, approaches uniform crossover

6.5 Integer Chromosome Crossover

Unlike binary chromosomes that use bits 0 and 1, integer representation is more suitable for problems involving discrete parameters or numerical values that have quantitative meaning, such as scheduling, sequencing, or combinatorial optimization.

6.5.1 Single-Point Crossover for Integer

Single-Point Crossover is the simplest form of crossover. One crossover position is randomly selected, then the part of the two parents after that position is exchanged to form two new offspring. The strings resulting from this process have Positional Bias characteristics.

6.5.2 Multi-point Crossover for Integer

Multi-point crossover has a mechanism like single-point crossover; the difference is the number of randomly selected positions. In Multi-point crossover, a number N of positions are randomly selected along the length of the chromosome. These positions are exchanged to form two new offspring.

The objectives of multi-point crossover include increasing genetic diversity in the population, reducing positional bias, and increasing the chances of recombination from various schemas or different solution blocks. However, using too many cut points can become too disruptive, as it can damage already good gene combinations (coadapted alleles).

6.5.3 Uniform Crossover for Integer

In uniform crossover, each gene is randomly selected from one of the corresponding genes on the parent chromosome. This selection process is done independently for each gene position. This method can be analogized to tossing a coin. If the result is "head", the gene is taken from parent 1; if the result is "tail", the gene is taken from parent 2.

This method provides equal opportunity for each gene to be inherited from one of the parents, thereby increasing genetic diversity and eliminating positional bias that typically appears in single-point crossover or multi-point crossover.

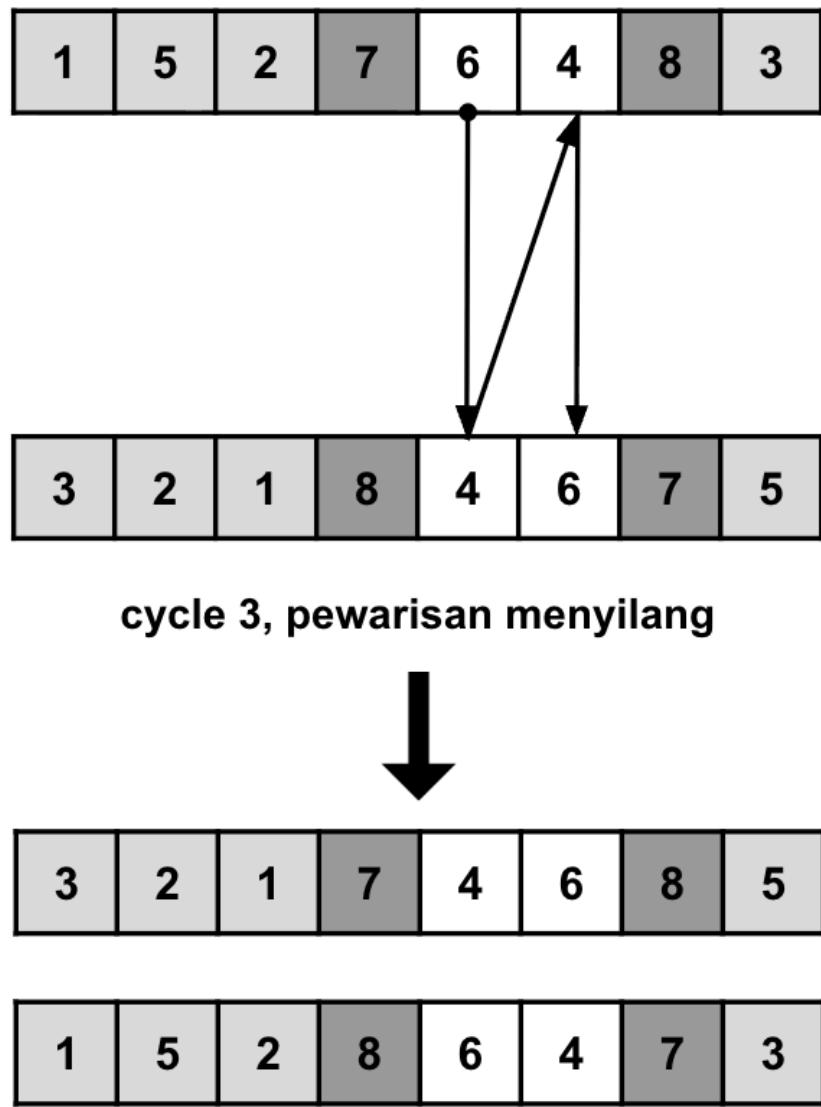


Figure 6.4: Single-Point Crossover for integer chromosomes

6.6 Real-Valued Crossover Operators

Crossover on real chromosomes is a genetic recombination process in Genetic Algorithms applied to chromosomes represented in real number form (floating-point representation). This representation is commonly used to solve continuous optimization problems, where decision variables have values within a certain range and are not limited to integers or binary.

Unlike crossover on binary or integer chromosomes, the crossover mechanism for real chromosomes involves arithmetic operations on gene values between parents. This method allows the creation of offspring with gene values that are between or around the parent gene values, thus maintaining the continuity and stability of the evolution process.

6.6.1 Arithmetic Crossover

Linear combination of parent vectors.

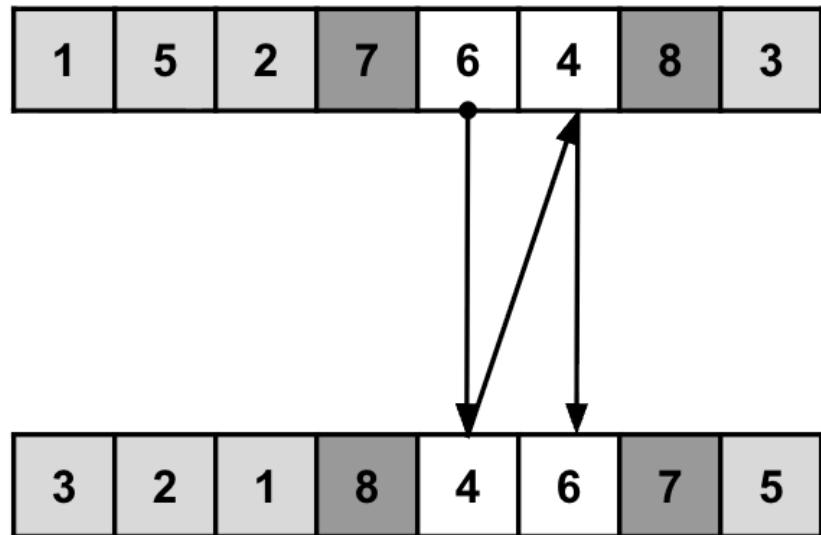
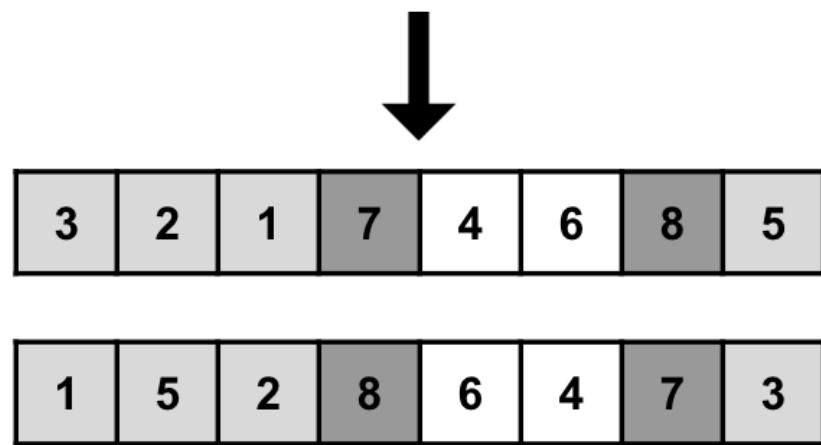
**cycle 3, pewarisan menyilang**

Figure 6.5: Multi-point Crossover for integer chromosomes

Single Arithmetic Crossover

1. Two parents are represented as:
 - Parent 1: $\langle x_1, \dots, x_n \rangle$
 - Parent 2: $\langle y_1, \dots, y_n \rangle$
2. Randomly select one gene (k) to undergo crossover operation
3. The result is two offspring formed based on a linear combination of the k -th gene of those parents with control parameter α , where $0 \leq \alpha \leq 1$:
 - Offspring 1: $\langle x_1, \dots, x_{k-1}, \alpha \cdot y_k + (1 - \alpha) \cdot x_k, x_{k+1}, \dots, x_n \rangle$
 - Offspring 2: $\langle y_1, \dots, y_{k-1}, \alpha \cdot x_k + (1 - \alpha) \cdot y_k, y_{k+1}, \dots, y_n \rangle$

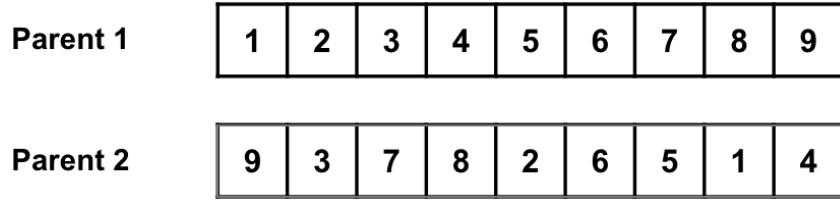


Figure 6.6: Uniform Crossover for integer chromosomes

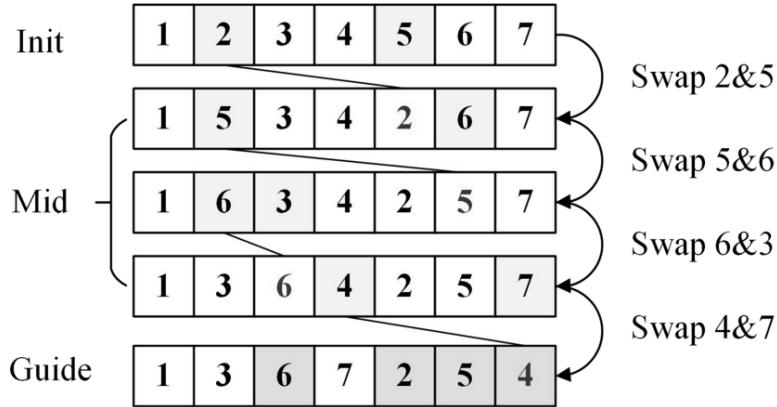


Figure 6.7: Single Arithmetic Crossover for real chromosomes

Simple Arithmetic Crossover

1. Two parents are represented as:
 - Parent 1: $\langle x_1, \dots, x_n \rangle$
 - Parent 2: $\langle y_1, \dots, y_n \rangle$
2. Randomly select one gene (k) to become the crossover boundary point
3. The result is two offspring formed based on a linear combination from gene $k+1$ to gene n with control parameter α , where $0 \leq \alpha \leq 1$:
 - Offspring 1: $\langle x_1, \dots, x_k, \alpha \cdot y_{k+1} + (1 - \alpha) \cdot x_{k+1}, \dots, \alpha \cdot y_n + (1 - \alpha) \cdot x_n \rangle$
 - Offspring 2: $\langle y_1, \dots, y_k, \alpha \cdot x_{k+1} + (1 - \alpha) \cdot y_{k+1}, \dots, \alpha \cdot x_n + (1 - \alpha) \cdot y_n \rangle$

Parent1	4	7	1	5	6	9	0	2	8	3	4	7	1	5	6	9	0	2	8	3	4	7	1	5	6	9	0	2	8	3
Parent2	3	5	1	8	9	2	7	0	4	6	3	5	1	8	9	2	7	0	4	6	3	5	1	8	9	2	7	0	4	6
Parent3	6	1	2	4	8	0	3	5	7	9	6	1	2	4	8	0	3	5	7	9	6	1	2	4	8	0	3	5	7	9
Parent4	5	9	6	1	2	4	8	7	3	0	5	9	6	1	2	4	8	7	3	0	5	9	6	1	2	4	8	7	3	0
Parent5	7	9	0	3	1	8	6	4	2	5	7	9	0	3	1	8	6	4	2	5	7	9	0	3	1	8	6	4	2	5
Child	4										4	7	1								4	7	1	5	6					
Parent1	4	7	1	5	6	9	0	2	8	3	4	7	1	5	6	9	0	2	8	3	4	7	1	5	6	9	0	2	8	3
Parent2	3	5	1	8	9	2	7	0	4	6	3	5	1	8	9	2	7	0	4	6	3	5	1	8	9	2	7	0	4	6
Parent3	6	1	2	4	8	0	3	5	7	9	6	1	2	4	8	0	3	5	7	9	6	1	2	4	8	0	3	5	7	9
Parent4	5	9	6	1	2	4	8	7	3	0	5	9	6	1	2	4	8	7	3	0	5	9	6	1	2	4	8	7	3	0
Parent5	7	9	0	3	1	8	6	4	2	5	7	9	0	3	1	8	6	4	2	5	7	9	0	3	1	8	6	4	2	5
Child	4	7	1	5	6	9					4	7	1	5	6	9	0	3	2	8	4	7	1	5	6	9	0	3	2	8

Figure 6.8: Simple Arithmetic Crossover for real chromosomes

Whole Arithmetic Crossover

1. Two parents are represented as:

- Parent 1: $\langle x_1, \dots, x_n \rangle$
- Parent 2: $\langle y_1, \dots, y_n \rangle$

2. For each gene i ($i = 1, 2, \dots, n$), offspring are formed with a linear combination of genes from both parents with control parameter α , where $0 \leq \alpha \leq 1$:

- Offspring 1: $z_i^1 = \alpha \cdot y_i + (1 - \alpha) \cdot x_i$
- Offspring 2: $z_i^2 = \alpha \cdot x_i + (1 - \alpha) \cdot y_i$

Parent1	4	7	1	5	6	9	0	2	8	3	4	7	1	5	6	9	0	2	8	3	4	7	1	5	6	9	0	2	8	3
Parent2	3	5	1	8	9	2	7	0	4	6	3	5	1	8	9	2	7	0	4	6	3	5	1	8	9	2	7	0	4	6
Parent3	6	1	2	4	8	0	3	5	7	9	6	1	2	4	8	0	3	5	7	9	6	1	2	4	8	0	3	5	7	9
Parent4	5	9	6	1	2	4	8	7	3	0	5	9	6	1	2	4	8	7	3	0	5	9	6	1	2	4	8	7	3	0
Parent5	7	9	0	3	1	8	6	4	2	5	7	9	0	3	1	8	6	4	2	5	7	9	0	3	1	8	6	4	2	5
Child	4										4	7	1								4	7	1	5	6					
Parent1	4	7	1	5	6	9	0	2	8	3	4	7	1	5	6	9	0	2	8	3	4	7	1	5	6	9	0	2	8	3
Parent2	3	5	1	8	9	2	7	0	4	6	3	5	1	8	9	2	7	0	4	6	3	5	1	8	9	2	7	0	4	6
Parent3	6	1	2	4	8	0	3	5	7	9	6	1	2	4	8	0	3	5	7	9	6	1	2	4	8	0	3	5	7	9
Parent4	5	9	6	1	2	4	8	7	3	0	5	9	6	1	2	4	8	7	3	0	5	9	6	1	2	4	8	7	3	0
Parent5	7	9	0	3	1	8	6	4	2	5	7	9	0	3	1	8	6	4	2	5	7	9	0	3	1	8	6	4	2	5
Child	4	7	1	5	6	9					4	7	1	5	6	9	0	3	2		4	7	1	5	6	9	0	3	2	8

Figure 6.9: Whole Arithmetic Crossover for real chromosomes

Whole Arithmetic Crossover

$$\text{child}_1 = \alpha \text{parent}_1 + (1 - \alpha) \text{parent}_2 \quad (6.14)$$

$$\text{child}_2 = (1 - \alpha) \text{parent}_1 + \alpha \text{parent}_2 \quad (6.15)$$

where $\alpha \in [0, 1]$ is a random weight.

Simple Arithmetic Crossover

Apply arithmetic crossover to a random subset of genes.

Single Arithmetic Crossover

Apply arithmetic crossover to one randomly selected gene.

Example

$$\text{Parent 1: } (2.1, 5.7, 1.3, 8.9) \quad (6.16)$$

$$\text{Parent 2: } (4.2, 3.1, 6.8, 2.4) \quad (6.17)$$

$$\text{Child 1 } (\alpha = 0.3): (3.57, 4.49, 4.98, 4.17) \quad (6.18)$$

$$\text{Child 2 } (\alpha = 0.3): (2.73, 4.32, 2.98, 6.17) \quad (6.19)$$

6.6.2 BLX- α Crossover (Blend Crossover)

Creates offspring in an interval around the parents.

Algorithm

For each gene i :

1. Calculate $c_{min} = \min(parent_{1i}, parent_{2i})$
2. Calculate $c_{max} = \max(parent_{1i}, parent_{2i})$
3. Calculate interval $I = c_{max} - c_{min}$
4. Generate offspring in $[c_{min} - \alpha \cdot I, c_{max} + \alpha \cdot I]$

Parameters

- $\alpha = 0$: Offspring between parents
- $\alpha = 0.5$: Standard BLX-0.5
- Larger α : More exploration beyond parents

6.6.3 SBX (Simulated Binary Crossover)

Simulates the behavior of one-point crossover for real-valued genes.

Formula

$$child_{1i} = 0.5[(1 + \beta_i)parent_{1i} + (1 - \beta_i)parent_{2i}] \quad (6.20)$$

$$child_{2i} = 0.5[(1 - \beta_i)parent_{1i} + (1 + \beta_i)parent_{2i}] \quad (6.21)$$

where β_i is calculated from:

$$\beta_i = \begin{cases} (2u_i)^{1/(\eta_c+1)} & \text{if } u_i \leq 0.5 \\ \left(\frac{1}{2(1-u_i)}\right)^{1/(\eta_c+1)} & \text{if } u_i > 0.5 \end{cases} \quad (6.22)$$

$u_i \sim U[0, 1]$ and η_c is the distribution index.

6.7 Permutation Crossover Operators

6.7.1 Order Crossover (OX)

Preserves relative order of elements from one parent [35, 29].

Algorithm

Example

$$\text{Parent 1: } (1, 2, 3, 4, 5, 6, 7, 8, 9) \quad (6.23)$$

$$\text{Parent 2: } (9, 3, 7, 8, 2, 6, 5, 1, 4) \quad (6.24)$$

$$\text{Copy segment: } (-, -, 3, 4, 5, 6, -, -, -) \quad (6.25)$$

$$\text{Fill from P2: } (7, 8, 3, 4, 5, 6, 2, 1, 9) \quad (6.26)$$

Algorithm 12 Order Crossover

Select two random crossover points
Copy segment between points from Parent 1 to Child
Fill remaining positions with elements from Parent 2 in order they appear, skipping already placed elements

6.7.2 Partially Mapped Crossover (PMX)

Creates mapping between elements in the crossover segment [22, 29].

Algorithm

Algorithm 13 Partially Mapped Crossover

Select two crossover points
Exchange segments between parents
For conflicts outside segment, use mapping relationship to resolve

Example

$$\text{Parent 1: } (1, 2, 3, 4, 5, 6, 7, 8, 9) \quad (6.27)$$

$$\text{Parent 2: } (5, 4, 6, 9, 2, 3, 7, 1, 8) \quad (6.28)$$

$$\text{Mapping: } 3 \leftrightarrow 6, 4 \leftrightarrow 9, 5 \leftrightarrow 2, 6 \leftrightarrow 3 \quad (6.29)$$

$$\text{Child 1: } (1, 5, 6, 9, 2, 3, 7, 8, 4) \quad (6.30)$$

6.7.3 Cycle Crossover (CX)

Preserves absolute positions of elements from both parents [35, 37].

Algorithm

Algorithm 14 Cycle Crossover

Start with first element of Parent 1
Follow cycle: find element in Parent 2 at same position, locate it in Parent 1, repeat
Copy cycle elements from Parent 1
Copy non-cycle elements from Parent 2
Create second child by swapping parent roles

6.7.4 Edge Recombination Crossover

Preserves edge information from both parents (useful for TSP).

Algorithm

Algorithm 15 Edge Recombination

Create edge table from both parents
 Start with element having fewest edges
 Add element to offspring
 Remove element from all edge lists
 Move to element with fewest remaining edges
 If tied, choose randomly
 If no edges remain, choose unused element randomly

6.8 Crossover Analysis

6.8.1 Schema Disruption

The probability that a schema H is disrupted by crossover:

One-Point Crossover

$$P_{disruption} = p_c \cdot \frac{\delta(H)}{l - 1} \quad (6.31)$$

Two-Point Crossover

$$P_{disruption} = p_c \cdot \left(\frac{2\delta(H)}{l - 1} - \frac{\delta(H)(\delta(H) - 1)}{(l - 1)(l - 2)} \right) \quad (6.32)$$

Uniform Crossover

$$P_{disruption} = p_c \cdot \left(1 - \left(\frac{1}{2} \right)^{\delta(H)-1} \right) \quad (6.33)$$

6.8.2 Building Block Preservation

- **Short schemas:** Better preserved by all crossover types
- **Long schemas:** More disrupted, especially by uniform crossover
- **Tightly linked:** Order crossover preserves adjacency relationships

6.9 Advanced Crossover Techniques

6.9.1 Adaptive Crossover

Adjust crossover parameters based on:

- Population diversity
- Fitness improvement rate
- Generation number

- Individual fitness levels

6.9.2 Multiple Parent Crossover

Combine genetic material from more than two parents [15, 48, 17, 10]:

- **Scanning crossover:** Scan through multiple parents
- **Voting crossover:** Majority vote among parents
- **Averaging crossover:** Average values from multiple parents

6.9.3 Problem-Specific Crossover

Design crossover operators for specific problem domains:

- **Graph problems:** Preserve graph properties
- **Scheduling:** Maintain temporal constraints
- **Neural networks:** Preserve network topology

6.10 Crossover Guidelines

6.10.1 Choosing Crossover Type

- **Binary representation:** One-point, two-point, or uniform
- **Real-valued:** Arithmetic, BLX- α , or SBX
- **Permutation:** OX, PMX, or CX depending on problem structure
- **Variable-length:** Specialized operators required

6.10.2 Parameter Setting

- **Crossover rate:** Start with $p_c = 0.8 - 0.9$
- **Population size:** Larger populations can handle higher crossover rates
- **Problem difficulty:** Harder problems may need lower rates

6.10.3 Empirical Testing

- Test multiple crossover operators
- Vary crossover parameters
- Measure diversity and convergence
- Consider problem-specific metrics

6.11 Crossover vs. Mutation

Aspect	Crossover	Mutation
Primary function	Exploitation	Exploration
Information source	Multiple parents	Random changes
Building blocks	Combines existing	Creates new
Search behavior	Convergent	Divergent
Application rate	High (0.6-0.9)	Low (0.001-0.1)
Population effect	Homogenization	Diversification

Table 6.1: Crossover vs. Mutation Comparison [42, 38]

6.12 Chapter Summary

This chapter covered crossover operators for genetic algorithms across different representation types. Crossover is the primary exploitative operator that combines beneficial traits from multiple parents. The choice of crossover operator depends on the representation and problem characteristics. Proper balance between crossover and mutation is essential for effective genetic algorithm performance.

6.13 Key Concepts

- Crossover principles and biological inspiration
- Binary crossover: one-point, two-point, uniform
- Real-valued crossover: arithmetic, BLX- α , SBX
- Permutation crossover: OX, PMX, CX
- Schema disruption analysis
- Building block preservation
- Adaptive and problem-specific crossover
- Guidelines for crossover selection and parameter setting

Chapter 7

Real-World Applications and Visual Examples

This chapter showcases real-world applications of genetic algorithms, directly from the course materials, demonstrating how GA concepts are applied in practice [21, 40, 14].

7.1 Game AI and Entertainment

7.1.1 Super Mario Bros Level Learning

One of the most compelling demonstrations of genetic algorithms is their application to game AI. In the course materials, we see an example of a genetic machine learning algorithm beating the first level of Super Mario Bros World at 4x speed.

Figure 7.1: Genetic Algorithm Learning to Play Super Mario Bros at 4x Speed

The GA evolves strategies by:

- **Encoding:** Button sequences as chromosomes (jump, run, duck, etc.)
- **Fitness:** Distance traveled and completion time
- **Selection:** Best-performing sequences survive
- **Crossover:** Combining successful movement patterns
- **Mutation:** Random button variations to explore new strategies [33, 28]

7.1.2 Tower Defense Game Balancing

The Towers of Reus project demonstrates how GA can balance gameplay:

- Users create maps with adjustable balancing parameters
- GA component runs until finding optimal winning solutions
- Determines if towers are too strong/weak or if levels are beatable
- Players can then test their skills against the optimized challenge

7.2 Pathfinding and Navigation

7.2.1 Maze Navigation

Figure 7.2: Cat Navigating Circular Maze to Reach Cheese Using Genetic Algorithm

This example demonstrates:

- **Problem:** Find shortest path through complex maze
- **Encoding:** Sequence of movement directions (up, down, left, right) [29]
- **Fitness:** Inverse of path length plus penalty for hitting walls
- **Crossover:** Combining successful path segments

7.2.2 Robot Navigation

Physical robot navigation showcases GA in hardware applications:

- Real-time path planning in dynamic environments
- Sensor data integration for obstacle avoidance
- Adaptive behavior evolution based on environmental feedback

7.3 Evolution Simulation

7.3.1 Simulated Evolution of Creatures

The course references simulated evolution examples from <http://www.wreck.devisland.net/ga/>:

Figure 7.3: Simulated Evolution Using Genetic Algorithm - Creatures Adapting Over Generations

Features include:

- Morphology evolution (body structure)
- Locomotion pattern optimization
- Environmental adaptation
- Multi-objective fitness (speed, stability, efficiency) [12, 13, 27]

Figure 7.4: Human Movement Evolution: From Sitting to Athletic Performance

7.4 Human Analogy Examples

7.4.1 Evolution of Movement

This analogy illustrates:

- **Population:** Different individuals with varying abilities
- **Selection:** Those who can jump higher survive
- **Inheritance:** Athletic traits passed to next generation
- **Mutation:** Random variations in technique

7.4.2 Work Journey Optimization

Figure 7.5: Optimizing Daily Commute Route Using GA Principles

Real-world application:

- Multiple route options (genes)
- Traffic conditions as environmental factors
- Time and fuel consumption as fitness criteria
- Learning from daily experiences (generations)

7.5 Academic Context

7.5.1 GA in Computational Intelligence

Figure 7.6: Position of Genetic Algorithms in Machine Learning and Soft Computing Landscape

The diagram shows GA's relationship with:

- **Machine Learning:** Kernel methods, SVM, Hidden Markov, Bayesian methods
- **Soft Computing:** Neural Networks, Fuzzy systems
- **Intersection:** Reinforcement Learning combining multiple paradigms

Figure 7.7: Comparison of Lamarck vs Darwin-Wallace Evolution Theories Using Giraffe Example

7.6 Historical Perspective

7.6.1 Natural Selection Theories

Understanding evolutionary principles:

- **Lamarck's View:** Acquired characteristics inherited
- **Darwin-Wallace View:** Natural selection favors beneficial traits
- **GA Implementation:** Follows Darwinian principles with random variation and selection

7.7 Summary

These visual examples from the course materials demonstrate the wide applicability of genetic algorithms:

1. **Entertainment:** Game AI and procedural content generation
2. **Robotics:** Path planning and adaptive behavior
3. **Simulation:** Artificial life and evolution studies
4. **Optimization:** Route planning and resource allocation
5. **Research:** Understanding natural evolutionary processes

The key insight is that GA provides a unified framework for solving complex optimization problems across diverse domains, making it one of the most versatile tools in computational intelligence.

Chapter 8

Mutation and Generation Update

In the previous chapters, we have covered the fundamental operations of Genetic Algorithms (GA) including encoding, fitness evaluation, selection, and crossover. This chapter completes the discussion of GA operators by examining **mutation** and **generation update mechanisms** [3, 45]. These operations are crucial for maintaining genetic diversity and ensuring the algorithm's ability to explore the search space effectively.

8.1 Introduction to Mutation

After the recombination (crossover) stage has been applied to all pairs of chromosomes in the mating pool, producing N chromosomes (where N is the population size), the GA executes the mutation operator on each of these chromosomes. Mutation is a critical operator that:

- Prevents premature convergence to local optima
- Maintains genetic diversity in the population
- Introduces new genetic material that may not have been present in the initial population
- Provides a mechanism for escaping local optima

8.1.1 What is Mutation?

Mutation is the process of changing the value of one or more genes in a genome [22, 32, 7]. More specifically, it involves:

- Changing the allele of a gene at a specific locus with another allele
- Avoiding premature convergence, which is reaching a suboptimal result that is not the global maximum
- Creating offspring that are not necessarily better than their parents

Important Note: The new population resulting from mutation is not guaranteed to be better than the previous population. However, mutation provides the essential mechanism for maintaining diversity and exploring new regions of the search space.

8.1.2 Mutation in Evolutionary Algorithms vs. Biological Evolution

In biological evolution, mutation is typically considered harmful because complex organisms have highly interdependent systems. However, in Evolutionary Algorithms (EAs):

- Mutation can often lead to improvement
- Individual representations in EAs are much simpler than biological organisms
- Mutating a small portion of genes may result in better individuals
- The simplified representation makes beneficial mutations more likely

8.2 Mutation for Different Representations

Many mutation methods have been proposed in the literature [31, 7, 25]. Each method has special characteristics and may only be applicable to certain types of representations. The choice of mutation operator must be compatible with the chromosome encoding scheme.

8.2.1 Mutation for Binary Representation

Binary representation uses the simplest form of mutation: **bit-flip mutation**.

Bit-Flip Mutation

In bit-flip mutation, each bit in the chromosome has a probability P_m (mutation probability) of being flipped:

- $1 \rightarrow 0$
- $0 \rightarrow 1$

Example:

Parent: 1 0 1 1 0 1 0 0
 ^ ^

Offspring: 1 0 0 1 0 0 0 0

In this example, bits at positions 3 and 6 were selected for mutation and flipped.

Algorithm:

Algorithm 16 Bit-Flip Mutation

```

for each gene  $g_i$  in chromosome do
     $r \leftarrow$  random number in  $[0, 1]$ 
    if  $r < P_m$  then
        Flip  $g_i$ : if  $g_i = 1$  then  $g_i \leftarrow 0$ , else  $g_i \leftarrow 1$ 
    end if
end for

```

8.2.2 Mutation for Integer Representation

Integer representations require different mutation strategies. Three common approaches are:

Integer Value Flipping

Uses mathematical operations ($+$, $-$, \times , \div) to change the value of selected genes.

Example:

Parent: 8 3 7 5 2 1 9 4 6
 ^ ^

Offspring: 8 3 2 5 2 8 9 4 6

The values at positions 3 and 6 were changed using mathematical operations.

Random Value Selection

A selected gene is replaced with a randomly chosen value from the valid range.

Example: If the valid range is $[1, 9]$:

Parent: 8 3 7 5 2 1 9 4 6
 ^

Offspring: 8 3 7 9 2 1 9 4 6

Creep Mutation

Adds or subtracts a small random integer value (usually ± 1 or ± 2) to the selected gene.

Example:

Parent: 8 3 7 5 2 1 9 4 6
 ^ ^

Offspring: 8 4 7 5 2 2 9 4 6

This method makes small, gradual changes and is particularly useful for fine-tuning solutions.

8.2.3 Mutation for Real-Valued Representation

Real-valued representations have different characteristics from binary and integer representations. Values of genes in real representations are continuous, whereas binary and integer representations are discrete. Therefore, real representations require specialized mutation operators.

Uniform Mutation

In uniform mutation, selected genes are replaced with values drawn from a uniform random distribution within the valid range $[a, b]$:

$$x'_i = a + \text{rand}(0, 1) \times (b - a) \quad (8.1)$$

where:

- x'_i is the new gene value
- a and b are the lower and upper bounds
- $\text{rand}(0, 1)$ generates a random number in $[0, 1]$

Non-Uniform Mutation with Fixed Distribution

This is the most commonly used mutation for real-valued representations. It is similar to the creep method for integer representation but uses real-valued additions instead of integer values.

The mutated value is calculated as:

$$x'_i = x_i + \mathcal{N}(0, \sigma^2) \quad (8.2)$$

where:

- x_i is the original gene value
- $\mathcal{N}(0, \sigma^2)$ is a random value from a normal (Gaussian) distribution with mean 0 and variance σ^2
- σ controls the mutation step size

Example:

Parent: 2.45 7.89 3.12 9.01 5.67

^

Offspring: 2.45 7.89 3.45 9.01 5.67

8.2.4 Mutation for Permutation Representation

Mutation on permutation representations must ensure that the resulting chromosome remains valid. This means that after mutation, all elements must still appear exactly once. Several specialized methods have been developed:

Swap Mutation

Two gene positions are randomly selected, and their values are exchanged.

Example:

Parent: 3 1 5 2 7 6 8 4 9

^

^

Offspring: 3 1 8 2 7 6 5 4 9

Positions 3 and 7 are selected, so values 5 and 8 are swapped.

Algorithm:

Algorithm 17 Swap Mutation

$i \leftarrow$ random position in chromosome

$j \leftarrow$ random position in chromosome (different from i)

Swap values at positions i and j

Insert Mutation

A gene at one position is removed and inserted at another position, shifting the intermediate genes.

Example:

Parent: 3 1 5 2 7 6 8 4 9
 ^ ^
 Offspring: 3 1 5 2 7 8 6 4 9

The gene at position 7 (value 8) is removed and inserted after position 2 (value 5).

Scramble Mutation

A segment of the chromosome is selected, and the genes within that segment are randomly shuffled.

Example:

Parent: 3 1 5 2 7 6 8 4 9
 \-----/
 Offspring: 3 1 2 6 5 7 8 4 9

The segment $\{5, 2, 7, 6\}$ is selected and randomly shuffled to $\{2, 6, 5, 7\}$.

Inversion Mutation

A segment of the chromosome is selected, and the order of genes within that segment is reversed.

Example:

Parent: 3 1 5 2 7 6 8 4 9
 \-----/
 Offspring: 3 1 6 7 2 5 8 4 9

The segment $\{5, 2, 7, 6\}$ is reversed to $\{6, 7, 2, 5\}$.

8.3 Generation Update Mechanisms

After selection, crossover, and mutation operations have been applied to a population, a generation update mechanism determines which individuals survive to the next generation. This process is also called **survivor selection** or **replacement strategy**.

8.3.1 Holland's Original Model (Generational Replacement)

In Holland's original GA [26, 22]:

- All offspring replace the entire parent population
- Parents are considered "dead" and removed from the population
- The new population consists entirely of offspring

- This creates distinct, non-overlapping generations

Characteristics:

- Simple and straightforward
- Clear separation between generations
- May lose good solutions if not careful
- Often combined with elitism to preserve best solutions

8.3.2 Generational Model with Elitism

A population of size N chromosomes in one generation is replaced by N new individuals in the next generation [11, 46]. However, to preserve the best solutions:

- The best k chromosomes (elites) from the parent generation are copied directly to the next generation
- The remaining $N - k$ positions are filled with offspring
- This ensures that the best solution never gets worse across generations

Algorithm:

Algorithm 18 Generational Model with Elitism

Sort parent population by fitness

Copy top k individuals to next generation (elites)

Generate $N - k$ offspring through selection, crossover, and mutation

Add offspring to next generation

Next generation becomes current generation

Typical values: $k = 1$ or $k = 2$ (preserving 1-2 best individuals)

8.3.3 Steady-State Update

In the steady-state model [46, 41]:

- Not all chromosomes are replaced in each generation
- Only M chromosomes are replaced, where $M < N$
- Often $M = 2$ (one mating produces 2 offspring, which replace 2 individuals)

Replacement strategies for selecting which individuals to replace:

1. **Replace parents:** The two offspring replace their two parents
2. **Replace worst:** The two offspring replace the two worst individuals in the population

- 3. Replace oldest:** The two offspring replace the two oldest individuals in the population

Characteristics:

- Allows good individuals to participate in multiple matings
- More gradual evolution
- Parents and offspring coexist in the same population
- Can be more efficient computationally

8.3.4 Continuous Update

In continuous update:

- Offspring and parents can coexist in the same generation
- Individuals are selected randomly from both groups for the next generation
- Provides maximum overlap between generations
- Less commonly used than other methods

8.4 GA Parameters

The performance of a Genetic Algorithm heavily depends on proper parameter settings [23, 38, 11]. The main parameters that need to be configured are:

8.4.1 Crossover Probability (P_c)

P_c is the probability that two parents will undergo crossover.

Effects:

- $P_c = 100\%$: All offspring are produced through crossover
- $P_c = 0\%$: No crossover occurs; offspring are exact copies of parents
- Typical range: $P_c \in [0.65, 0.90]$ (65% to 90%)

Recommendations:

- Higher values (0.8-0.9) encourage exploration
- Lower values preserve good solutions but reduce diversity
- Standard setting: $P_c = 0.8$

8.4.2 Mutation Probability (P_m)

P_m is the probability that a gene in an offspring chromosome will undergo mutation.

Effects:

- $P_m = 100\%$: All genes are mutated (chaos)
- $P_m = 0\%$: No mutation occurs; no new genetic material
- Typical range: $P_m \in [0.005, 0.01]$ (0.5% to 1%)

Common formulas:

$$P_m = \frac{1}{L} \tag{8.3}$$

or

$$P_m = \frac{1}{N \times L} \tag{8.4}$$

where:

- L is the chromosome length (number of genes)
- N is the population size

Rationale: The mutation probability is often set so that, on average, one mutation occurs per chromosome.

8.4.3 Population Size (N)

The population size should be proportional to the volume of the search space.

Effects:

- Too small: Difficult to reach global optimum; may converge to local optimum
- Too large: Heavy computational burden; inconsistent with evolutionary principles
- Should not approach the size of the entire search space

Recommendations:

- Typical range: $N \in [50, 100]$
- Determined through experimentation
- Larger populations for larger, more complex problems
- Consider computational resources available

8.4.4 Number of Generations (G)

The number of generations should be proportional to population size and search space size.

Example calculation:

- If $N = 100$ and search space size $\approx 10^5$
- Then $G = 100$ might be appropriate

Stopping criteria alternatives:

1. Fixed number of generations
2. Maximum number of fitness evaluations
3. No improvement for k consecutive generations
4. Target fitness value reached
5. Combination of the above

8.4.5 General Parameter Setting Guidelines

Important Note: There are no definitive rules for setting GA parameters [47, 23]. Parameter selection relies on:

- Intuition and experience
- Experimentation (trial and error)
- Problem-specific characteristics

Common starting configuration:

- Chromosome representation: Binary/Integer/Real/Permutation (problem-dependent)
- Number of bits per variable: Based on desired precision
- Population size: $N = 50$ to 100
- Crossover probability: $P_c = 0.8$
- Mutation probability: $P_m = \frac{1}{L}$ to $\frac{1}{N \times L}$

where:

- N = Population size
- L = Chromosome length (number of genes)

8.5 Parameter Observation Study

To understand the effects of different parameters, we present a systematic observation study.

8.5.1 Test Problem

Objective: Minimize the function:

$$h(x_1, x_2) = x_1^2 + x_2^2 \quad (8.5)$$

where $x_1, x_2 \in [-10, 10]$

Fitness function:

$$\text{Fitness} = \frac{1}{x_1^2 + x_2^2 + 0.001} \quad (8.6)$$

The constant 0.001 is added to avoid division by zero at the optimal point (0, 0).

8.5.2 Experimental Setup

Parameter variations tested:

- Population size: [50, 100, 200]
- Number of bits per variable: [10, 50, 90]
- Crossover probability: [0.5, 0.7, 0.9]
- Mutation probability: $[0.5/L, 1/L, 2/L]$ where L is total chromosome length

Fairness criterion:

- Maximum number of individuals evaluated: 20,000
- Each configuration run 30 times for statistical validity

8.5.3 Sample Results

Table 8.1 shows selected results from the parameter study:

Table 8.1: GA Parameter Observation Results

Pop Size	Bits	P_c	P_m	Avg Best Fitness	Avg Evaluations
50	10	0.5	0.0250	839.55	20000
50	50	0.5	0.0050	1000.00	8301.67
50	50	0.7	0.0100	1000.00	20000
50	90	0.7	0.0056	1000.00	8780.00
100	50	0.7	0.0050	1000.00	14416.67
100	90	0.5	0.0111	1000.00	20000
200	50	0.5	0.0050	1000.00	20000
200	90	0.7	0.0056	1000.00	20000
200	90	0.9	0.0028	1000.00	19866.67

Key observations:

1. **Best configuration:** Population size = 50, Bits = 90, $P_c = 0.7$, $P_m = 0.0056$
 - Achieved optimal fitness (1000.00)

- Required only 8780 evaluations on average
- Most efficient configuration

2. Effect of bit precision:

- 10 bits: Often failed to reach optimum
- 50-90 bits: Consistently reached optimum
- Higher precision enables finer search granularity

3. Effect of population size:

- Smaller populations (50) can be very efficient
- Larger populations (200) more robust but slower
- Trade-off between speed and reliability

4. Effect of crossover probability:

- $P_c = 0.7$ performed best overall
- Moderate values balance exploration and exploitation

5. Effect of mutation probability:

- Low mutation rates ($\sim 1/L$) worked best
- Too high mutation causes chaos
- Too low mutation loses diversity

8.6 Summary and Conclusions

This chapter has covered the essential components for completing the GA cycle:

1. Mutation operators provide genetic diversity and prevent premature convergence:

- Binary: Bit-flip mutation
- Integer: Flipping, random selection, creep mutation
- Real: Uniform mutation, Gaussian mutation
- Permutation: Swap, insert, scramble, inversion mutation

2. Generation update mechanisms determine how populations evolve:

- Generational replacement (with elitism)
- Steady-state update
- Continuous update

3. Parameter selection is crucial for GA performance:

- No universal rules exist
- Requires experimentation and tuning

- Starting guidelines provide reasonable defaults

Key principles:

- Parent selection and survivor selection do not depend on chromosome representation
- Recombination and mutation operators must match the chromosome representation
- Parameter settings should be tailored to the specific problem
- Experimentation is essential for finding optimal configurations

With the completion of this chapter, we have now covered all the fundamental components of Genetic Algorithms: encoding, fitness evaluation, selection, crossover, mutation, and generation update. The next chapters will explore advanced topics and practical applications of GAs.

8.7 Exercises

- Given the two parent chromosomes for a permutation problem:
 - Parent 1: [1, 2, 7, 3, 4, 9, 8, 6, 5]
 - Parent 2: [5, 4, 3, 9, 1, 2, 6, 8, 7]
 - (a) Perform Partial-Mapped Crossover (PMX) with cut points at positions 2 and 5
 - (b) Apply inversion mutation to the offspring with mutation segment from locus 2 to 5
- For a binary-encoded GA with chromosome length $L = 50$ and population size $N = 100$:
 - (a) Calculate appropriate mutation probability using $P_m = 1/L$
 - (b) Calculate alternative mutation probability using $P_m = 1/(N \times L)$
 - (c) Discuss which might be more appropriate and why
- Design a mutation operator for a real-valued chromosome representing (x, y) coordinates where $x, y \in [-100, 100]$:
 - (a) Implement uniform mutation
 - (b) Implement Gaussian mutation with $\sigma = 5$
 - (c) Compare the expected behavior of both operators
- Implement and compare three generation update strategies:
 - (a) Generational replacement with elitism ($k = 2$)
 - (b) Steady-state with replacement of worst individuals
 - (c) Steady-state with replacement of oldest individuals

Discuss scenarios where each might be preferred.

5. For the test function $f(x_1, x_2) = x_1^2 + x_2^2$ with $x_1, x_2 \in [-10, 10]$:
 - (a) Design a complete GA including all parameters
 - (b) Run experiments with different parameter combinations
 - (c) Analyze which parameters have the most significant impact
 - (d) Propose an optimal parameter configuration based on your results

Appendix A

Algorithm Implementations

A.1 Basic Genetic Algorithm Implementation

A.1.1 Python Implementation

Listing A.1: Basic Genetic Algorithm in Python

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from typing import List, Tuple, Callable
4
5 class GeneticAlgorithm:
6     def __init__(self,
7                  fitness_func: Callable,
8                  chromosome_length: int,
9                  population_size: int = 100,
10                 crossover_rate: float = 0.8,
11                 mutation_rate: float = 0.01,
12                 elitism: bool = True):
13
14         self.fitness_func = fitness_func
15         self.chromosome_length = chromosome_length
16         self.population_size = population_size
17         self.crossover_rate = crossover_rate
18         self.mutation_rate = mutation_rate
19         self.elitism = elitism
20
21     # Initialize population
22     self.population = self._initialize_population()
23     self.fitness_history = []
24     self.best_individual = None
25     self.best_fitness = float('-inf')
26
27     def _initialize_population(self) -> np.ndarray:
28         """Initialize random binary population"""
29         return np.random.randint(0, 2,
30                                (self.population_size, self.
31                                 chromosome_length))
```

```

31
32     def _evaluate_fitness(self, population: np.ndarray) -> np.
33         ndarray:
34             """Evaluate fitness for all individuals"""
35             fitness_values = np.array([self.fitness_func(individual)
36                                         for individual in population])
37             return fitness_values
38
39
40     def _tournament_selection(self, population: np.ndarray,
41                               fitness_values: np.ndarray,
42                               tournament_size: int = 3) -> np.
43                               ndarray:
44         """Tournament selection"""
45         selected = []
46         for _ in range(len(population)):
47             # Select random individuals for tournament
48             tournament_indices = np.random.choice(len(population),
49                                                     ,
50                                                     tournament_size,
51                                                     replace=False)
52             tournament_fitness = fitness_values[
53                 tournament_indices]
54
55             # Select winner
56             winner_index = tournament_indices[np.argmax(
57                 tournament_fitness)]
58             selected.append(population[winner_index])
59
60         return np.array(selected)
61
62     def _one_point_crossover(self, parent1: np.ndarray,
63                             parent2: np.ndarray) -> Tuple[np.
64                                 ndarray, np.ndarray]:
65         """One-point crossover"""
66         if np.random.random() > self.crossover_rate:
67             return parent1.copy(), parent2.copy()
68
69         crossover_point = np.random.randint(1, len(parent1))
70
71         child1 = np.concatenate([parent1[:crossover_point],
72                                 parent2[crossover_point:]])
73         child2 = np.concatenate([parent2[:crossover_point],
74                                 parent1[crossover_point:]])
75
76         return child1, child2
77
78     def _bit_flip_mutation(self, individual: np.ndarray) -> np.
79         ndarray:
80             """Bit-flip mutation"""
81             mutated = individual.copy()
82             for i in range(len(mutated)):
83                 if np.random.random() > self.mutation_rate:
84                     mutated[i] = 1 - mutated[i]
85
86             return mutated

```

```

75         if np.random.random() < self.mutation_rate:
76             mutated[i] = 1 - mutated[i] # Flip bit
77     return mutated
78
79     def _apply_elitism(self, old_population: np.ndarray,
80                        old_fitness: np.ndarray,
81                        new_population: np.ndarray) -> np.ndarray:
82         """Apply elitism by preserving best individual"""
83         if not self.elitism:
84             return new_population
85
86         best_index = np.argmax(old_fitness)
87         best_individual = old_population[best_index]
88
89         # Replace worst individual in new population with best
90         # from old
91         new_fitness = self._evaluate_fitness(new_population)
92         worst_index = np.argmin(new_fitness)
93         new_population[worst_index] = best_individual
94
95     return new_population
96
97     def evolve(self, generations: int) -> dict:
98         """Main evolutionary loop"""
99         for generation in range(generations):
100             # Evaluate fitness
101             fitness_values = self._evaluate_fitness(self.
102                 population)
103
104             # Track best individual
105             max_fitness_idx = np.argmax(fitness_values)
106             if fitness_values[max_fitness_idx] > self.
107                 best_fitness:
108                 self.best_fitness = fitness_values[
109                     max_fitness_idx]
110                 self.best_individual = self.population[
111                     max_fitness_idx].copy()
112
113             # Record statistics
114             self.fitness_history.append({
115                 'generation': generation,
116                 'best_fitness': np.max(fitness_values),
117                 'avg_fitness': np.mean(fitness_values),
118                 'worst_fitness': np.min(fitness_values)
119             })
120
121             # Selection
122             selected = self._tournament_selection(self.population
123                 , fitness_values)
124
125             # Crossover and mutation

```

```

120         new_population = []
121         for i in range(0, len(selected), 2):
122             parent1 = selected[i]
123             parent2 = selected[(i + 1) % len(selected)]
124
125             # Crossover
126             child1, child2 = self._one_point_crossover(
127                 parent1, parent2)
128
129             # Mutation
130             child1 = self._bit_flip_mutation(child1)
131             child2 = self._bit_flip_mutation(child2)
132
133             new_population.extend([child1, child2])
134
135             new_population = np.array(new_population[:self.
136                                         population_size])
137
138             # Apply elitism
139             self.population = self._apply_elitism(self.population
140
141             ,
142
143             fitness_values,
144             new_population)
145
146
147     def plot_fitness_history(self):
148         """Plot fitness evolution over generations"""
149         generations = [entry['generation'] for entry in self.
150                         fitness_history]
151         best_fitness = [entry['best_fitness'] for entry in self.
152                         fitness_history]
153         avg_fitness = [entry['avg_fitness'] for entry in self.
154                         fitness_history]
155
156         plt.figure(figsize=(10, 6))
157         plt.plot(generations, best_fitness, label='BestFitness',
158                  linewidth=2)
159         plt.plot(generations, avg_fitness, label='AverageFitness',
160                  linewidth=2)
161         plt.xlabel('Generation')
162         plt.ylabel('Fitness')
163         plt.title('FitnessEvolution')
164         plt.legend()
165         plt.grid(True, alpha=0.3)
166         plt.show()
167

```

```

163 # Example usage
164 def onemax_fitness(individual):
165     """OneMax problem: maximize number of 1s"""
166     return np.sum(individual)
167
168 def sphere_function_binary(individual, bounds=(-5.12, 5.12)):
169     """Sphere function with binary encoding"""
170     # Decode binary to real values
171     x = bounds[0] + (bounds[1] - bounds[0]) * np.sum(individual *
172         2**np.arange(len(individual))[::-1]) / (2**len(individual) -
173         1)
174     return -(x**2) # Negative because we want to minimize
175
176 # Run GA on OneMax problem
177 if __name__ == "__main__":
178     ga = GeneticAlgorithm(
179         fitness_func=onemax_fitness,
180         chromosome_length=20,
181         population_size=50,
182         crossover_rate=0.8,
183         mutation_rate=0.01
184     )
185
186     result = ga.evolve(generations=100)
187
188     print(f"Best individual: {result['best_individual']}")
189     print(f"Best fitness: {result['best_fitness']}")
190
191     ga.plot_fitness_history()

```

A.2 Real-Valued Genetic Algorithm

Listing A.2: Real-Valued GA Implementation

```

1 import numpy as np
2 from typing import List, Tuple, Callable
3
4 class RealValuedGA:
5     def __init__(self,
6                  fitness_func: Callable,
7                  dimensions: int,
8                  bounds: List[Tuple[float, float]],
9                  population_size: int = 100,
10                 crossover_rate: float = 0.8,
11                 mutation_rate: float = 0.1,
12                 mutation_strength: float = 0.1):
13
14     self.fitness_func = fitness_func
15     self.dimensions = dimensions
16     self.bounds = bounds

```

```

17     self.population_size = population_size
18     self.crossover_rate = crossover_rate
19     self.mutation_rate = mutation_rate
20     self.mutation_strength = mutation_strength
21
22     self.population = self._initialize_population()
23     self.fitness_history = []
24
25 def _initialize_population(self) -> np.ndarray:
26     """Initialize random real-valued population"""
27     population = np.zeros((self.population_size, self.
28                           dimensions))
29     for i in range(self.dimensions):
30         low, high = self.bounds[i]
31         population[:, i] = np.random.uniform(low, high, self.
32                                         population_size)
33     return population
34
35 def _blx_alpha_crossover(self, parent1: np.ndarray,
36                         parent2: np.ndarray,
37                         alpha: float = 0.5) -> Tuple[np.
38                                         ndarray, np.ndarray]:
39     """BLX-alpha crossover"""
40     if np.random.random() > self.crossover_rate:
41         return parent1.copy(), parent2.copy()
42
43     child1 = np.zeros_like(parent1)
44     child2 = np.zeros_like(parent2)
45
46     for i in range(len(parent1)):
47         min_val = min(parent1[i], parent2[i])
48         max_val = max(parent1[i], parent2[i])
49         interval = max_val - min_val
50
51         low_bound = max(min_val - alpha * interval, self.
52                         bounds[i][0])
53         high_bound = min(max_val + alpha * interval, self.
54                         bounds[i][1])
55
56         child1[i] = np.random.uniform(low_bound, high_bound)
57         child2[i] = np.random.uniform(low_bound, high_bound)
58
59     return child1, child2
60
61 def _gaussian_mutation(self, individual: np.ndarray) -> np.
62     ndarray:
63     """Gaussian mutation"""
64     mutated = individual.copy()
65     for i in range(len(mutated)):
66         if np.random.random() < self.mutation_rate:

```

```

61             noise = np.random.normal(0, self.
62                                         mutation_strength)
63             mutated[i] += noise
64
65             # Ensure bounds are respected
66             low, high = self.bounds[i]
67             mutated[i] = np.clip(mutated[i], low, high)
68
69         return mutated
70
71     def evolve(self, generations: int) -> dict:
72         """Main evolutionary loop"""
73         for generation in range(generations):
74             # Evaluate fitness
75             fitness_values = np.array([self.fitness_func(ind)
76                                     for ind in self.population])
77
78             # Record statistics
79             self.fitness_history.append({
80                 'generation': generation,
81                 'best_fitness': np.max(fitness_values),
82                 'avg_fitness': np.mean(fitness_values),
83                 'worst_fitness': np.min(fitness_values)
84             })
85
86             # Tournament selection
87             new_population = []
88             for _ in range(self.population_size // 2):
89                 # Select parents
90                 parent1_idx = self._tournament_selection(
91                     fitness_values)
92                 parent2_idx = self._tournament_selection(
93                     fitness_values)
94
95                 parent1 = self.population[parent1_idx]
96                 parent2 = self.population[parent2_idx]
97
98                 # Crossover
99                 child1, child2 = self._blx_alpha_crossover(
100                     parent1, parent2)
101
102                 # Mutation
103                 child1 = self._gaussian_mutation(child1)
104                 child2 = self._gaussian_mutation(child2)
105
106                 new_population.extend([child1, child2])
107
108             self.population = np.array(new_population)
109
110             # Final evaluation
111             final_fitness = np.array([self.fitness_func(ind)

```

```

108                     for ind in self.population])
109         best_idx = np.argmax(final_fitness)
110
111     return {
112         'best_individual': self.population[best_idx],
113         'best_fitness': final_fitness[best_idx],
114         'fitness_history': self.fitness_history
115     }
116
117     def _tournament_selection(self, fitness_values: np.ndarray,
118                               tournament_size: int = 3) -> int:
119         """Tournament selection returning index"""
120         tournament_indices = np.random.choice(len(fitness_values),
121                                               tournament_size,
122                                               replace=False)
123         tournament_fitness = fitness_values[tournament_indices]
124         winner_idx = tournament_indices[np.argmax(
125             tournament_fitness)]
126         return winner_idx
127
128 # Example: Optimize Rastrigin function
129 def rastrigin_function(x):
130     """Rastrigin function (minimization problem)"""
131     A = 10
132     n = len(x)
133     return -(A * n + np.sum(x**2 - A * np.cos(2 * np.pi * x)))
134
135 # Usage
136 bounds = [(-5.12, 5.12)] * 2 # 2D Rastrigin
137 ga = RealValuedGA(
138     fitness_func=rastrigin_function,
139     dimensions=2,
140     bounds=bounds,
141     population_size=100,
142     mutation_strength=0.1
143 )
144
145 result = ga.evolve(generations=200)
146 print(f"Best solution: {result['best_individual']}")
147 print(f"Best fitness: {result['best_fitness']}")

```

A.3 Traveling Salesman Problem GA

Listing A.3: TSP with Genetic Algorithm

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from typing import List, Tuple
4

```

```
5  class TSP_GA:
6      def __init__(self,
7          cities: np.ndarray,
8          population_size: int = 100,
9          crossover_rate: float = 0.8,
10         mutation_rate: float = 0.02):
11
12         self.cities = cities
13         self.num_cities = len(cities)
14         self.population_size = population_size
15         self.crossover_rate = crossover_rate
16         self.mutation_rate = mutation_rate
17
18     # Create distance matrix
19     self.distance_matrix = self._calculate_distance_matrix()
20
21     # Initialize population
22     self.population = self._initialize_population()
23
24     def _calculate_distance_matrix(self) -> np.ndarray:
25         """Calculate distance matrix between all cities"""
26         n = self.num_cities
27         distances = np.zeros((n, n))
28
29         for i in range(n):
30             for j in range(n):
31                 if i != j:
32                     distances[i][j] = np.sqrt(
33                         (self.cities[i][0] - self.cities[j][0])
34                         **2 +
35                         (self.cities[i][1] - self.cities[j][1])
36                         **2
37                     )
38         return distances
39
40     def _initialize_population(self) -> List[List[int]]:
41         """Initialize population with random permutations"""
42         population = []
43         for _ in range(self.population_size):
44             tour = list(range(self.num_cities))
45             np.random.shuffle(tour)
46             population.append(tour)
47         return population
48
49     def _calculate_tour_distance(self, tour: List[int]) -> float:
50         """Calculate total distance of a tour"""
51         total_distance = 0
52         for i in range(len(tour)):
53             from_city = tour[i]
54             to_city = tour[(i + 1) % len(tour)]
```

```

53         total_distance += self.distance_matrix[from_city][
54             to_city]
55     return total_distance
56
56     def _fitness(self, tour: List[int]) -> float:
57         """Fitness function (inverse of distance)"""
58         distance = self._calculate_tour_distance(tour)
59         return 1.0 / (1.0 + distance)
60
61     def _order_crossover(self, parent1: List[int],
62                         parent2: List[int]) -> Tuple[List[int],
63                                         List[int]]:
64         """Order crossover (OX)"""
65         if np.random.random() > self.crossover_rate:
66             return parent1.copy(), parent2.copy()
67
68         size = len(parent1)
69         start, end = sorted(np.random.choice(size, 2, replace=
70                             False))
71
72         # Create children
73         child1 = [None] * size
74         child2 = [None] * size
75
76         # Copy segments
77         child1[start:end] = parent1[start:end]
78         child2[start:end] = parent2[start:end]
79
80         # Fill remaining positions
81         self._fill_remaining_ox(child1, parent2, start, end)
82         self._fill_remaining_ox(child2, parent1, start, end)
83
84     def _fill_remaining_ox(self, child: List[int], parent: List[int],
85                           start: int, end: int):
86         """Helper function for order crossover"""
87         child_set = set(child[start:end])
88         parent_filtered = [city for city in parent if city not in
89                           child_set]
90
90         # Fill positions before start
91         for i in range(start):
92             child[i] = parent_filtered.pop(0)
93
94         # Fill positions after end
95         for i in range(end, len(child)):
96             child[i] = parent_filtered.pop(0)
97
98     def _swap_mutation(self, tour: List[int]) -> List[int]:

```

```

99     """Swap mutation"""
100    mutated = tour.copy()
101    if np.random.random() < self.mutation_rate:
102        i, j = np.random.choice(len(tour), 2, replace=False)
103        mutated[i], mutated[j] = mutated[j], mutated[i]
104    return mutated
105
106    def _tournament_selection(self, fitness_values: List[float],
107                               tournament_size: int = 3) -> int:
108        """Tournament selection"""
109        tournament_indices = np.random.choice(len(fitness_values),
110                                              tournament_size,
111                                              replace=False)
112        tournament_fitness = [fitness_values[i] for i in
113                              tournament_indices]
114        winner_idx = tournament_indices[np.argmax(
115            tournament_fitness)]
116        return winner_idx
117
118    def evolve(self, generations: int) -> dict:
119        """Main evolutionary loop"""
120        fitness_history = []
121        best_tour = None
122        best_distance = float('inf')
123
124        for generation in range(generations):
125            # Evaluate fitness
126            fitness_values = [self._fitness(tour) for tour in
127                               self.population]
128            distances = [self._calculate_tour_distance(tour)
129                          for tour in self.population]
130
131            # Track best solution
132            min_distance_idx = np.argmin(distances)
133            if distances[min_distance_idx] < best_distance:
134                best_distance = distances[min_distance_idx]
135                best_tour = self.population[min_distance_idx].copy()
136
137            # Record statistics
138            fitness_history.append({
139                'generation': generation,
140                'best_distance': np.min(distances),
141                'avg_distance': np.mean(distances),
142                'worst_distance': np.max(distances)
143            })
144
145            # Create new population
146            new_population = []

```

```

144         # Elitism: keep best individual
145         new_population.append(best_tour.copy())
146
147         # Generate rest of population
148         while len(new_population) < self.population_size:
149             # Selection
150             parent1_idx = self._tournament_selection(
151                 fitness_values)
152             parent2_idx = self._tournament_selection(
153                 fitness_values)
154
155             parent1 = self.population[parent1_idx]
156             parent2 = self.population[parent2_idx]
157
158             # Crossover
159             child1, child2 = self._order_crossover(parent1,
160                                                     parent2)
161
162             # Mutation
163             child1 = self._swap_mutation(child1)
164             child2 = self._swap_mutation(child2)
165
166             new_population.extend([child1, child2])
167
168         # Trim to population size
169         self.population = new_population[:self.
170                                         population_size]
171
172     return {
173         'best_tour': best_tour,
174         'best_distance': best_distance,
175         'fitness_history': fitness_history
176     }
177
178     def plot_tour(self, tour: List[int], title: str = "BestTour"):
179         """Plot the tour"""
180         plt.figure(figsize=(10, 8))
181
182         # Plot cities
183         plt.scatter(self.cities[:, 0], self.cities[:, 1],
184                     c='red', s=100, zorder=2)
185
186         # Plot tour
187         tour_cities = self.cities[tour + [tour[0]]]    # Close the
188         loop
189         plt.plot(tour_cities[:, 0], tour_cities[:, 1],
190                  'b-', linewidth=2, zorder=1)
191
192         # Add city labels
193         for i, city in enumerate(self.cities):

```

```

189         plt.annotate(str(i), (city[0], city[1]),
190                         xytext=(5, 5), textcoords='offset points',
191                         )
192
193     plt.title(f"{title}\nDistance:{self.
194         _calculate_tour_distance(tour):.2f}")
195     plt.xlabel("X Coordinate")
196     plt.ylabel("Y Coordinate")
197     plt.grid(True, alpha=0.3)
198     plt.show()
199
200 # Example usage
201 if __name__ == "__main__":
202     # Create random cities
203     np.random.seed(42)
204     num_cities = 20
205     cities = np.random.rand(num_cities, 2) * 100
206
207     # Initialize and run GA
208     tsp_ga = TSP_GA(cities, population_size=100, mutation_rate
209                      =0.02)
210     result = tsp_ga.evolve(generations=500)
211
212     # Plot best tour
213     tsp_ga.plot_tour(result['best_tour'])

```

A.4 NSGA-II for Multi-Objective Optimization

Listing A.4: NSGA-II Implementation

```

1 import numpy as np
2 from typing import List, Tuple
3
4 class NSGA2:
5     def __init__(self,
6                  objective_functions: List,
7                  num_variables: int,
8                  bounds: List[Tuple[float, float]],
9                  population_size: int = 100,
10                 crossover_rate: float = 0.9,
11                 mutation_rate: float = 0.1):
12
13         self.objective_functions = objective_functions
14         self.num_objectives = len(objective_functions)
15         self.num_variables = num_variables
16         self.bounds = bounds
17         self.population_size = population_size

```

```

18     self.crossover_rate = crossover_rate
19     self.mutation_rate = mutation_rate
20
21     # Ensure even population size
22     if self.population_size % 2 != 0:
23         self.population_size += 1
24
25 def _initialize_population(self) -> np.ndarray:
26     """Initialize random population"""
27     population = np.zeros((self.population_size, self.
28                           num_variables))
29     for i in range(self.num_variables):
30         low, high = self.bounds[i]
31         population[:, i] = np.random.uniform(low, high, self.
32                                         population_size)
33     return population
34
35 def _evaluate_objectives(self, population: np.ndarray) -> np.
36 ndarray:
37     """Evaluate all objectives for population"""
38     objectives = np.zeros((len(population), self.
39                           num_objectives))
40     for i, individual in enumerate(population):
41         for j, obj_func in enumerate(self.objective_functions
42                                     ):
43             objectives[i, j] = obj_func(individual)
44     return objectives
45
46 def _dominates(self, obj1: np.ndarray, obj2: np.ndarray) ->
47     bool:
48     """Check if obj1 dominates obj2 (assuming minimization)
49     """
50
51     return np.all(obj1 <= obj2) and np.any(obj1 < obj2)
52
53 def _fast_non_dominated_sort(self, objectives: np.ndarray) ->
54     Tuple[List[List[int]], np.ndarray]:
55     """Fast non-dominated sorting"""
56     population_size = len(objectives)
57     domination_count = np.zeros(population_size)
58     dominated_solutions = [[] for _ in range(population_size)]
59     fronts = []
60
61     # Find domination relationships
62     for i in range(population_size):
63         for j in range(population_size):
64             if i != j:
65                 if self._dominates(objectives[i], objectives[
66                     j]):
67                     dominated_solutions[i].append(j)
68
69     fronts = []
70     for i in range(population_size):
71         if len(dominated_solutions[i]) == 0:
72             fronts.append(i)
73             dominated_solutions[i] = []
74
75     while len(fronts) > 0:
76         new_front = []
77         for i in fronts:
78             for j in dominated_solutions[i]:
79                 domination_count[j] += 1
80                 if domination_count[j] == 0:
81                     new_front.append(j)
82                     dominated_solutions[j] = []
83
84         fronts = new_front
85
86     return fronts, dominated_solutions

```

```

58         elif self._dominates(objectives[j],
59             objectives[i]):
60             domination_count[i] += 1
61
62     if domination_count[i] == 0:
63         fronts[0].append(i)
64
65     # Build subsequent fronts
66     current_front = 0
67     while len(fronts[current_front]) > 0:
68         next_front = []
69         for i in fronts[current_front]:
70             for j in dominated_solutions[i]:
71                 domination_count[j] -= 1
72                 if domination_count[j] == 0:
73                     next_front.append(j)
74         current_front += 1
75         fronts.append(next_front)
76
77     # Remove empty last front
78     fronts.pop()
79
80     # Assign ranks
81     ranks = np.zeros(population_size)
82     for rank, front in enumerate(fronts):
83         for individual in front:
84             ranks[individual] = rank
85
86     return fronts, ranks
87
88     def _calculate_crowding_distance(self, objectives: np.ndarray,
89                                     front: List[int]) -> np.ndarray:
90         """Calculate crowding distance for individuals in a front
91         """
92
93         if len(front) <= 2:
94             return np.full(len(front), float('inf'))
95
96         distances = np.zeros(len(front))
97
98         for obj_idx in range(self.num_objectives):
99             # Sort by objective value
100            sorted_indices = sorted(range(len(front)),
101                                key=lambda x: objectives[front[x], obj_idx])
102
103            # Set boundary points to infinity
104            distances[sorted_indices[0]] = float('inf')
105            distances[sorted_indices[-1]] = float('inf')

```

```

104     # Calculate distances for middle points
105     obj_range = (objectives[front[sorted_indices[-1]]],
106                   obj_idx] -
107                   objectives[front[sorted_indices[0]]],
108                   obj_idx])
109
110     if obj_range > 0:
111         for i in range(1, len(sorted_indices) - 1):
112             distance = (objectives[front[sorted_indices[i +
1]]], obj_idx] -
113                         objectives[front[sorted_indices[i -
1]], obj_idx])
114             distances[sorted_indices[i]] += distance /
115             obj_range
116
117     return distances
118
119
120     def _tournament_selection(self, ranks: np.ndarray,
121                               crowding_distances: np.ndarray,
122                               population_size: int) -> List[int]:
123         """Binary tournament selection based on rank and crowding
124         distance"""
125         selected = []
126
127         for _ in range(population_size):
128             # Select two random individuals
129             candidates = np.random.choice(len(ranks), 2, replace=
130                                           False)
131             i, j = candidates[0], candidates[1]
132
133             # Compare based on rank first, then crowding distance
134             if ranks[i] < ranks[j]:
135                 selected.append(i)
136             elif ranks[i] > ranks[j]:
137                 selected.append(j)
138             else: # Same rank, compare crowding distance
139                 if crowding_distances[i] > crowding_distances[j]:
140                     selected.append(i)
141                 else:
142                     selected.append(j)
143
144         return selected
145
146     def _sbx_crossover(self, parent1: np.ndarray, parent2: np.
147                       ndarray,
148                           eta: float = 20.0) -> Tuple[np.ndarray, np.
149                                         ndarray]:
150         """Simulated Binary Crossover (SBX)"""
151         if np.random.random() > self.crossover_rate:
152             return parent1.copy(), parent2.copy()

```

```

146     child1 = np.zeros_like(parent1)
147     child2 = np.zeros_like(parent2)
148
149     for i in range(len(parent1)):
150         if np.random.random() <= 0.5:
151             if abs(parent1[i] - parent2[i]) > 1e-14:
152                 y1, y2 = min(parent1[i], parent2[i]), max(
153                     parent1[i], parent2[i])
154
155                 # Calculate beta
156                 rand = np.random.random()
157                 if rand <= 0.5:
158                     beta = (2 * rand) ** (1.0 / (eta + 1))
159                 else:
160                     beta = (1.0 / (2 * (1 - rand))) ** (1.0 /
161                         (eta + 1))
162
163
164                 # Ensure bounds
165                 low, high = self.bounds[i]
166                 child1[i] = np.clip(child1[i], low, high)
167                 child2[i] = np.clip(child2[i], low, high)
168             else:
169                 child1[i] = parent1[i]
170                 child2[i] = parent2[i]
171         else:
172             child1[i] = parent1[i]
173             child2[i] = parent2[i]
174
175     return child1, child2
176
177     def _polynomial_mutation(self, individual: np.ndarray,
178                               eta: float = 20.0) -> np.ndarray:
179         """Polynomial mutation"""
180         mutated = individual.copy()
181
182         for i in range(len(mutated)):
183             if np.random.random() < self.mutation_rate:
184                 low, high = self.bounds[i]
185                 delta1 = (mutated[i] - low) / (high - low)
186                 delta2 = (high - mutated[i]) / (high - low)
187
188                 rand = np.random.random()
189                 mut_pow = 1.0 / (eta + 1.0)
190
191                 if rand <= 0.5:
192                     xy = 1.0 - delta1

```

```

193         val = 2.0 * rand + (1.0 - 2.0 * rand) * (xy
194             ** (eta + 1.0))
195         deltaq = val ** mut_pow - 1.0
196     else:
197         xy = 1.0 - delta2
198         val = 2.0 * (1.0 - rand) + 2.0 * (rand - 0.5)
199             * (xy ** (eta + 1.0))
200         deltaq = 1.0 - val ** mut_pow
201
202     mutated[i] += deltaq * (high - low)
203     mutated[i] = np.clip(mutated[i], low, high)
204
205     return mutated
206
207 def evolve(self, generations: int) -> dict:
208     """Main NSGA-II evolution loop"""
209     # Initialize population
210     population = self._initialize_population()
211
212     for generation in range(generations):
213         # Evaluate objectives
214         objectives = self._evaluate_objectives(population)
215
216         # Non-dominated sorting
217         fronts, ranks = self._fast_non_dominated_sort(
218             objectives)
219
220         # Calculate crowding distances
221         crowding_distances = np.zeros(len(population))
222         for front in fronts:
223             if len(front) > 0:
224                 distances = self._calculate_crowding_distance(
225                     objectives, front)
226                 for i, individual_idx in enumerate(front):
227                     crowding_distances[individual_idx] =
228                         distances[i]
229
230         # Selection for mating pool
231         mating_pool_indices = self._tournament_selection(
232             ranks, crowding_distances,
233             self.
234             population_size
235         )
236
237         mating_pool = population[mating_pool_indices]
238
239         # Create offspring through crossover and mutation
240         offspring = []
241         for i in range(0, self.population_size, 2):
242             parent1 = mating_pool[i]
243             parent2 = mating_pool[i + 1]

```

```

236         child1, child2 = self._sbx_crossover(parent1,
237             parent2)
238         child1 = self._polynomial_mutation(child1)
239         child2 = self._polynomial_mutation(child2)
240
241         offspring.extend([child1, child2])
242
243
244     # Combine parent and offspring populations
245     combined_population = np.vstack([population,
246         offspring])
247     combined_objectives = self._evaluate_objectives(
248         combined_population)
249
250     # Environmental selection
251     combined_fronts, combined_ranks = self.
252         _fast_non_dominated_sort(combined_objectives)
253
254     new_population = []
255     front_idx = 0
256
257     # Add complete fronts
258     while (len(new_population) + len(combined_fronts[
259         front_idx]) <= self.population_size):
260         for individual_idx in combined_fronts[front_idx]:
261             new_population.append(individual_idx)
262             front_idx += 1
263
264             if front_idx >= len(combined_fronts):
265                 break
266
267     # Add partial front if needed
268     if len(new_population) < self.population_size and
269         front_idx < len(combined_fronts):
270         last_front = combined_fronts[front_idx]
271         crowding_distances = self.
272             _calculate_crowding_distance(
273                 combined_objectives, last_front)
274
275         # Sort by crowding distance (descending)
276         sorted_indices = sorted(range(len(last_front)),
277             key=lambda x:
278                 crowding_distances[x],
279                 reverse=True)
280
281         remaining_slots = self.population_size - len(
282             new_population)
283         for i in range(remaining_slots):
284             new_population.append(last_front[
285                 sorted_indices[i]])

```

```

275
276     # Update population
277     population = combined_population[new_population]
278
279     # Final evaluation and return Pareto front
280     final_objectives = self._evaluate_objectives(population)
281     fronts, _ = self._fast_non_dominated_sort(
282         final_objectives)
283
284     pareto_front_indices = fronts[0]
285     pareto_front_solutions = population[pareto_front_indices]
286     pareto_front_objectives = final_objectives[
287         pareto_front_indices]
288
289     return {
290         'pareto_front_solutions': pareto_front_solutions,
291         'pareto_front_objectives': pareto_front_objectives,
292         'final_population': population,
293         'final_objectives': final_objectives
294     }
295
296 # Example: Minimize two objectives (ZDT1 problem)
297 def objective1(x):
298     return x[0]
299
300 def objective2(x):
301     g = 1 + 9 * np.sum(x[1:]) / (len(x) - 1)
302     h = 1 - np.sqrt(x[0] / g)
303     return g * h
304
305 # Usage
306 if __name__ == "__main__":
307     objectives = [objective1, objective2]
308     bounds = [(0, 1)] * 10 # 10-dimensional problem
309
310     nsga2 = NSGA2(objectives, 10, bounds, population_size=100)
311     result = nsga2.evolve(generations=250)
312
313     # Plot Pareto front
314     pareto_objectives = result['pareto_front_objectives']
315     plt.figure(figsize=(10, 6))
316     plt.scatter(pareto_objectives[:, 0], pareto_objectives[:, 1],
317                 c='red', alpha=0.7)
318     plt.xlabel('Objective 1')
319     plt.ylabel('Objective 2')
320     plt.title('Pareto Front')
321     plt.grid(True, alpha=0.3)
322     plt.show()

```

Appendix B

Practical Examples and Case Studies

B.1 Function Optimization Problems

B.1.1 OneMax Problem

The OneMax problem is the simplest optimization problem for binary genetic algorithms.

Problem Definition

Maximize the number of 1s in a binary string:

$$f(x) = \sum_{i=1}^n x_i \quad (\text{B.1})$$

where $x_i \in \{0, 1\}$ and n is the string length.

Expected Performance

- **Optimal solution:** All 1s string
- **Global optimum:** $f^* = n$
- **Expected convergence:** $O(n \log n)$ generations
- **Population size:** $O(\log n)$ sufficient

GA Configuration

Parameter	Value
Representation	Binary string
Population size	50 – 100
Selection	Tournament (size 3)
Crossover	One-point, $p_c = 0.8$
Mutation	Bit-flip, $p_m = 1/n$
Generations	100 – 200

Table B.1: OneMax GA Configuration

B.1.2 Sphere Function

Continuous optimization benchmark function.

Problem Definition

$$f(\mathbf{x}) = \sum_{i=1}^n x_i^2 \quad (\text{B.2})$$

where $\mathbf{x} \in [-5.12, 5.12]^n$.

Characteristics

- **Type:** Unimodal, separable
- **Global minimum:** $\mathbf{x}^* = (0, 0, \dots, 0)$
- **Global optimum:** $f^* = 0$
- **Difficulty:** Easy (convex, single optimum)

B.1.3 Rastrigin Function

Multimodal benchmark function.

Problem Definition

$$f(\mathbf{x}) = An + \sum_{i=1}^n [x_i^2 - A \cos(2\pi x_i)] \quad (\text{B.3})$$

where $A = 10$ and $\mathbf{x} \in [-5.12, 5.12]^n$.

Characteristics

- **Type:** Multimodal, separable
- **Local minima:** $A \cdot n$ local minima
- **Global minimum:** $\mathbf{x}^* = (0, 0, \dots, 0)$
- **Global optimum:** $f^* = 0$
- **Difficulty:** Medium (many local optima)

GA Challenges

- Premature convergence to local optima
- Requires high population diversity
- Benefits from diversity preservation techniques

B.1.4 Rosenbrock Function

Non-convex optimization problem.

Problem Definition

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} [100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2] \quad (\text{B.4})$$

Characteristics

- **Type:** Unimodal but non-convex
- **Global minimum:** $\mathbf{x}^* = (1, 1, \dots, 1)$
- **Global optimum:** $f^* = 0$
- **Difficulty:** Hard (narrow curved valley)

B.2 Combinatorial Optimization Problems

B.2.1 Traveling Salesman Problem (TSP)

Problem Description

Find the shortest route visiting all cities exactly once and returning to the starting city.

Mathematical Formulation

Minimize:

$$\sum_{i=1}^n \sum_{j=1}^n d_{ij} x_{ij} \quad (\text{B.5})$$

Subject to:

$$\sum_{j=1}^n x_{ij} = 1, \quad \forall i \quad (\text{B.6})$$

$$\sum_{i=1}^n x_{ij} = 1, \quad \forall j \quad (\text{B.7})$$

$$x_{ij} \in \{0, 1\} \quad (\text{B.8})$$

where d_{ij} is the distance between cities i and j .

GA Representation

- **Encoding:** Permutation of city indices
- **Example:** $(3, 1, 4, 2, 5)$ means visit cities in order $3 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 3$

Specialized Operators

- **Crossover:** Order crossover (OX), Partially mapped crossover (PMX)
- **Mutation:** Swap, insert, inversion
- **Local search:** 2-opt, 3-opt improvements

Performance Tips

- Use edge recombination for better building block preservation
- Apply local search (hybrid GA)
- Consider nearest neighbor initialization
- Use elitist replacement

B.2.2 Knapsack Problem

Problem Description

Select items to maximize value while staying within weight constraint.

0/1 Knapsack Formulation

Maximize:

$$\sum_{i=1}^n v_i x_i \quad (\text{B.9})$$

Subject to:

$$\sum_{i=1}^n w_i x_i \leq W \quad (\text{B.10})$$

$$x_i \in \{0, 1\} \quad (\text{B.11})$$

where v_i is value, w_i is weight, and W is capacity.

GA Approach

- **Encoding:** Binary string (1 = include item, 0 = exclude)
- **Constraint handling:** Penalty function or repair mechanism
- **Fitness:** Value minus penalty for constraint violation

Penalty Function Example

$$fitness(x) = \sum_{i=1}^n v_i x_i - \alpha \max \left(0, \sum_{i=1}^n w_i x_i - W \right) \quad (\text{B.12})$$

where α is a penalty coefficient.

B.3 Real-World Applications

B.3.1 Neural Network Training

Problem Setup

Optimize neural network weights and biases using GA.

Representation

- **Encoding:** Real-valued vector of all weights and biases
- **Decoding:** Reshape vector into network structure

Fitness Function

$$fitness = \frac{1}{1 + MSE} \quad (B.13)$$

where MSE is mean squared error on training/validation set.

Advantages over Backpropagation

- No gradient information required
- Can optimize network topology
- Robust to local minima
- Handles discontinuous activation functions

B.3.2 Feature Selection

Problem Description

Select optimal subset of features for machine learning models.

GA Approach

- **Encoding:** Binary string (1 = include feature, 0 = exclude)
- **Fitness:** Model performance with selected features
- **Objectives:** Maximize accuracy, minimize number of features

Multi-objective Formulation

$$\text{Maximize: } accuracy(\text{selected features}) \quad (B.14)$$

$$\text{Minimize: } \text{number of selected features} \quad (B.15)$$

B.3.3 Job Shop Scheduling

Problem Description

Schedule jobs on machines to minimize makespan or total completion time.

Representation Options

1. **Priority-based:** Priority values for job-machine pairs
2. **Permutation-based:** Order of jobs for each machine
3. **Direct:** Actual schedule representation

Constraints

- Each job visits each machine exactly once
- Machines can process only one job at a time
- Jobs cannot be preempted
- Precedence constraints must be satisfied

B.4 Parameter Tuning Guidelines

B.4.1 Population Size

Problem Complexity	Population Size
Simple (OneMax)	50 – 100
Medium (TSP, 50 cities)	100 – 500
Complex (Large TSP)	500 – 2000
Multi-objective	100 – 300

Table B.2: Population Size Guidelines

B.4.2 Crossover and Mutation Rates

Problem Type	Crossover Rate	Mutation Rate
Binary optimization	0.7 – 0.9	$1/L$ to $10/L$
Real-valued	0.8 – 0.9	0.01 – 0.1
Permutation	0.8 – 0.9	0.01 – 0.05
Multi-objective	0.9	$1/L$

Table B.3: Crossover and Mutation Rate Guidelines

where L is the chromosome length.

B.4.3 Selection Pressure

- **Low pressure:** Tournament size 2-3, linear ranking
- **Medium pressure:** Tournament size 4-7
- **High pressure:** Tournament size > 7, truncation selection

B.5 Performance Analysis

B.5.1 Convergence Metrics

- **Best fitness:** Track best solution over generations
- **Average fitness:** Monitor population quality
- **Diversity:** Measure population spread
- **Success rate:** Percentage of runs finding global optimum

B.5.2 Statistical Testing

- Run multiple independent trials (20-30)
- Report mean, standard deviation, best, worst
- Use statistical tests (t-test, Mann-Whitney U)
- Consider effect size, not just significance

B.5.3 Comparison with Other Methods

Method	Speed	Global Search	Implementation
Hill Climbing	Fast	Poor	Easy
Simulated Annealing	Medium	Good	Medium
Genetic Algorithm	Slow	Excellent	Medium
Particle Swarm	Medium	Good	Easy
Differential Evolution	Medium	Excellent	Easy

Table B.4: Algorithm Comparison

B.6 Common Pitfalls and Solutions

B.6.1 Premature Convergence

Symptoms:

- Population converges to suboptimal solution
- Low diversity after few generations

- No improvement for many generations

Solutions:

- Increase population size
- Reduce selection pressure
- Increase mutation rate
- Use diversity preservation techniques
- Apply restart strategies

B.6.2 Slow Convergence

Symptoms:

- Little improvement over many generations
- High population diversity maintained
- Random walk behavior

Solutions:

- Increase selection pressure
- Reduce mutation rate
- Apply local search (hybrid GA)
- Use better initialization
- Adjust crossover operators

B.6.3 Constraint Handling Issues

Common Problems:

- All individuals violate constraints
- Feasible region too small
- Penalty coefficients poorly set

Solutions:

- Use repair mechanisms
- Apply specialized operators
- Implement feasibility preservation
- Use multi-objective approach
- Adjust penalty weights dynamically

B.7 Advanced Techniques

B.7.1 Hybrid Genetic Algorithms

Combine GA with local search methods:

- **Memetic algorithms:** GA + local search
- **Lamarckian evolution:** Inherit improved solutions
- **Baldwinian evolution:** Use local search for fitness evaluation only

B.7.2 Adaptive Parameter Control

Automatically adjust GA parameters during evolution:

- **Deterministic:** Pre-defined schedule
- **Adaptive:** Based on population state
- **Self-adaptive:** Parameters evolve with population

B.7.3 Parallel Genetic Algorithms

Distribute computation across multiple processors:

- **Master-slave:** Parallel fitness evaluation
- **Island model:** Multiple populations with migration
- **Cellular GA:** Spatial population structure

B.8 Implementation Best Practices

B.8.1 Code Organization

- Separate representation from operators
- Use modular design for easy testing
- Implement proper random number generation
- Add logging and visualization capabilities

B.8.2 Testing and Validation

- Test on known benchmark problems
- Verify operators maintain validity
- Check random number generation quality
- Profile performance bottlenecks

B.8.3 Documentation

- Document parameter choices and reasoning
- Record experimental setup details
- Maintain version control
- Share reproducible results

B.9 Chapter Summary

This chapter provided practical examples and case studies demonstrating genetic algorithm applications across various problem domains. Key lessons include the importance of proper representation design, parameter tuning, and performance analysis. Understanding common pitfalls and their solutions is crucial for successful GA implementation.

B.10 Key Takeaways

- Problem representation is critical for GA success
- Parameter settings must match problem characteristics
- Statistical validation ensures reliable results
- Hybrid approaches often outperform pure GAs
- Domain knowledge should guide operator design
- Proper testing and documentation are essential

Bibliography

- [1] Course material week 2 - cycle of genetic algorithm and holland schema. Course material.
- [2] Course material week 4 - crossover. Course material.
- [3] Course material week 9 - mutation and update generation. Course material.
- [4] Selection - introduction to genetic algorithms - tutorial with interactive java applets. <https://www.obitko.com/tutorials/genetic-algorithms/selection.php>. Retrieved September 30, 2025.
- [5] Algorithm Afternoon. Chapter 4 - selection strategies. https://algorithmafternoon.com/books/genetic_algorithm/chapter04/. Retrieved September 30, 2025.
- [6] Algorithm Afternoon. Ranked selection genetic algorithm. https://algorithmafternoon.com/genetic/ranked_selection_genetic_algorithm/. Retrieved September 30, 2025.
- [7] Thomas Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, Oxford, UK, 1996.
- [8] Baeldung on Computer Science. Tournament selection in genetic algorithms. <https://www.baeldung.com/cs/ga-tournament-selection>. Retrieved September 30, 2025.
- [9] James E Baker. Reducing bias and inefficiency in the selection algorithm. *Proceedings of the second international conference on genetic algorithms*, pages 14–21, 1987.
- [10] Shih-Hsin Chen, Min-Chih Chen, Pei-Chann Chang, and V. Mani. Multiple parents crossover operators: A new approach removes the overlapping solutions for sequencing problems. *Applied Mathematical Modelling*, 37(5):2737–2746, 2013.
- [11] Kenneth A De Jong. An analysis of the behavior of a class of genetic adaptive systems. 1975. PhD thesis.
- [12] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, Chichester, UK, 2001.
- [13] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

- [14] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2nd edition, 2015.
- [15] S. M. Elsayed, R. A. Sarker, and D. L. Essam. GA with a new multi-parent crossover for constrained optimization. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 857–864, New Orleans, LA, USA, 2011.
- [16] Larry J Eshelman. The chc adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. *Foundations of genetic algorithms*, 1:265–283, 1991.
- [17] A. M. Fajrin and C. Faticahah. Multi-parent order crossover mechanism of genetic algorithm for minimizing violation of soft constraint on course timetabling problem. *Register: Jurnal Ilmiah Teknologi Sistem Informasi*, 6(1):43–51, 2020.
- [18] David B Fogel. Evolutionary programming: an introduction and some current directions. *Statistics and computing*, 5(2):103–109, 1995.
- [19] David B Fogel. *Evolutionary computation: toward a new philosophy of machine intelligence*. John Wiley & Sons, 3rd edition, 2006.
- [20] GeeksforGeeks. Crossover in genetic algorithm. <https://www.geeksforgeeks.org/machine-learning/crossover-in-genetic-algorithm/>. Retrieved November 3, 2025.
- [21] Mitsuo Gen and Runwei Cheng. *Genetic algorithms and engineering optimization*. Wiley-Interscience, 2007.
- [22] David E Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, MA, 1989.
- [23] John J Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on systems, man, and cybernetics*, 16(1):122–128, 1986.
- [24] H. Gu, H. C. Lam, and Y. Zinder. A hybrid genetic algorithm for scheduling jobs sharing multiple resources under uncertainty. *EURO Journal on Computational Optimization*, 10:100050, 2022.
- [25] Randy L Haupt and Sue Ellen Haupt. Practical genetic algorithms. 2004.
- [26] John H Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, MI, 1975.
- [27] Jeffrey Horn, Nicholas Nafpliotis, and David E Goldberg. A niched pareto genetic algorithm for multiobjective optimization. *Proceedings of the first IEEE conference on evolutionary computation*, pages 82–87, 1994.
- [28] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, 1992.
- [29] Pedro Larrañaga, Cindy MH Kuijpers, Roberto H Murga, Iñaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: a review of representations and operators. *Artificial intelligence review*, 13(2):129–170, 1999.

- [30] N. Majhi and R. Mishra. A novel hybrid genetic algorithm and nelder-mead approach and it's application for parameter estimation. *F1000Research*, 13:1073, 2025.
- [31] Zbigniew Michalewicz. Genetic algorithms+ data structures= evolution programs. 1996.
- [32] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, 1996.
- [33] David J Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. *Proceedings of the Eleventh international joint conference on Artificial intelligence*, 1:762–767, 1989.
- [34] S. H. Murad, N. B. Tayfor, N. H. Mahmood, and L. Arman. Hybrid genetic algorithms-driven optimization of machine learning models for heart disease prediction. *MethodsX*, 15:103510, 2025.
- [35] IM Oliver, DJ Smith, and John RC Holland. A study of permutation crossover operators on the traveling salesman problem. *Proceedings of the Second International Conference on Genetic Algorithms*, pages 224–230, 1987.
- [36] Colin Reeves and Takeshi Yamada. Genetic algorithms, path relinking and the flow-shop sequencing problem. *Evolutionary Computation*, 6(1):45–60, 1998.
- [37] Colin R Reeves. *Modern heuristic techniques for combinatorial problems*. John Wiley & Sons, New York, 1993.
- [38] J David Schaffer, Rich A Caruana, Larry J Eshelman, and Rajarshi Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. *Proceedings of the third international conference on genetic algorithms*, pages 51–60, 1989.
- [39] E. Shams. Resolving the exploration-exploitation dilemma in evolutionary algorithms: A novel human-centered framework. *arXiv preprint*, 2025.
- [40] S. N. Sivanandam and S. N. Deepa. *Introduction to genetic algorithms*. Springer, 2008.
- [41] J. Smith and F. Vavak. Replacement strategies in steady state genetic algorithms: Static environments. pages 219–234, 1998.
- [42] William M Spears. Crossover or mutation? *Foundations of genetic algorithms*, 2:221–237, 1993.
- [43] Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: a survey. *Computer*, 27(6):17–26, 1994.
- [44] Tutorialspoint. Genetic algorithms - crossover. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm. Retrieved November 3, 2025.
- [45] Tutorialspoint. Genetic algorithms - mutation. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm. Retrieved November 22, 2025.

- [46] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.
- [47] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.
- [48] E. T. Yassen, M. Ayob, M. Z. A. Nazri, and N. R. Sabar. Multi-parent insertion crossover for vehicle routing problem with time windows. In *2012 4th Conference on Data Mining and Optimization (DMO)*, pages 103–108, Langkawi, Malaysia, 2012.
- [49] Betul Sultan Yıldız, S. Kumar, Natee Panagant, P. Mehta, S. M. Sait, Ali Riza Yıldız, Nantiwat Pholdee, Sujin Bureerat, and Seyedali Mirjalili. A novel hybrid arithmetic optimization algorithm for solving constrained optimization problems. *Knowledge-Based Systems*, 271:110554, 2023.