GENETIC ALGORITHMS

FROM THEORY TO REAL PROJECTS

$$\text{Fitness}(x) = f(x_1, x_2, \ldots, x_n)$$

$$x = [x_1, x_2, \ldots, x_n]$$

$$x_i \in D_i$$

$$\max f(x)$$

```
while (not termination) {
   evaluate(population);
   selection(population);
   crossover(population);
   mutation(population);
}
```

OPTIMIZE.
EVOLVE.
SOLVE.

KEREM AKTUG

Genetic Algorithms from Theory to Real Projects

KEREM AKTUĞ

Contents

PREFACE

Genetic algorithms have a strange and beautiful promise: instead of forcing us to write the perfect solution directly, they let us describe what “better” means and then allow solutions to evolve.

This book was written with that idea at its center.

Many explanations of genetic algorithms begin and end with theory: chromosomes, mutation, crossover, selection, fitness functions. These concepts are essential, but they become truly meaningful only when they are used to solve real problems. A chromosome is easier to understand when it becomes a route through cities, a Sudoku board, a set of selected features, a gear train, or the weights of a neural network controlling a game agent.

That is why this book follows a practical path. It begins from scratch, with simple examples that expose every moving part of the algorithm. Then it gradually builds toward a reusable GA.Core library and applies the same core ideas to increasingly different domains: classic optimization, scheduling, routing, electronics, continuous functions, image approximation, puzzles, machine learning, neuroevolution, and policy search.

The goal is not only to teach how genetic algorithms work. The goal is to teach a way of thinking.

A genetic algorithm asks us to break a problem into four questions:

What does a candidate solution look like?

How do we measure whether it is good?

How do we create variation?

How do we keep improving over generations?

Once you learn to ask these questions, many difficult problems become approachable. They may not become easy, but they become searchable.

The examples in this book are intentionally visual wherever possible. Watching a population improve, seeing a path shorten, observing a Sudoku grid settle, or following a neural policy as it learns to land a spacecraft makes the algorithm feel less abstract. Evolution becomes something you can inspect, debug, tune, and reason about.

This book is written for developers, students, engineers, and curious problem-solvers who want to move beyond formulas and build working systems. You do not need to be an expert in evolutionary computation before starting. The chapters are designed to grow step by step, from basic representations to reusable architecture and then to more advanced applications.

Some examples will converge quickly. Some will struggle. Some will show the limits of a naive fitness function or the importance of the right chromosome design. Those moments are part of the learning process. Genetic algorithms are powerful, but they are not magic. Their strength comes from how clearly we define representation, variation, selection pressure, and fitness.

If this book succeeds, you will finish it with more than a collection of examples. You will have a mental framework for turning messy, nonlinear, hard-to-search problems into evolutionary systems.

And hopefully, the next time you face a problem where the perfect solution is hard to design directly, you will ask:

Can I let better solutions evolve?

ABOUT THIS BOOK

Who should read this book?

Genetic Algorithms are powerful, flexible, and surprisingly intuitive once understood. This book is designed for a wide range of readers—from beginners exploring optimization techniques to professionals applying them in real-world problems.

Whether you are learning, building, or optimizing, this book will guide you step by step

Software Developers and Engineers

If you are a developer or engineer looking to solve complex optimization problems, this book offers practical tools and techniques.

You will learn how to:

- design reusable and flexible GA frameworks
- apply GAs to real-world problems such as scheduling, routing, and design optimization

The focus is on **hands-on coding**, not just theory.

Data Scientists and Machine Learning Practitioners

Genetic Algorithms are highly useful in modern AI workflows, especially when traditional methods fall short.

This book will help you:

- optimize hyperparameters
- perform feature selection
- explore alternative approaches to model tuning
- understand evolutionary approaches to AI

If you work with machine learning, this book introduces a powerful complementary tool.

Researchers and Problem Solvers

If you are working on complex or unconventional problems, Genetic Algorithms provide a flexible approach to exploration.

You will benefit from:

- strategies for handling large and nonlinear search spaces
- approaches to multi-objective optimization

- inspiration for applying evolutionary methods in your domain

This book emphasizes **thinking in terms of search and evolution**, not rigid formulas.

Curious Minds and Creative Thinkers

You don't have to be an engineer or a scientist to appreciate Genetic Algorithms.

If you are curious about:

- how nature-inspired systems can solve problems
- how complex solutions can emerge from simple rules
- how algorithms can be used creatively

then this book will offer a new perspective.

What You Will Gain

By the end of this book, you will:

- understand the core principles of Genetic Algorithms
- build your own GA implementations
- apply them to practical problems
- think differently about problem-solving

Final Note

This book is not just about learning an algorithm.

It is about learning a **way of thinking**:

Instead of trying to construct the perfect solution directly, let better solutions evolve over time

Source Code and Implementation

Genetic Algorithms are best understood through practice. For this reason, this book is designed as a hands-on learning experience, where every concept is supported by real, working code examples.

All implementations in this book are written in .NET / C#. The projects are organized step by step, starting from simple from-scratch implementations and gradually moving toward reusable libraries, WPF visualizations, engineering examples, machine learning applications, and neuroevolution projects.

The complete source code is available on GitHub:

https://github.com/keremaktug/Genetic-Algorithms-From-Theory-to-Real-Projects

Why .NET and C#?

This book focuses on .NET because it provides a strong balance between clarity, structure, performance, and practical application development.

C# is used for:

- Strongly typed and readable code
- Clear object-oriented design
- Reusable library architecture
- High-performance application development
- WPF-based visual demonstrations
- Practical engineering and desktop applications
- Machine learning examples with ML.NET

Using a single technology stack also keeps the learning path consistent. Instead of switching between languages, the reader can focus on the genetic algorithm concepts themselves:

- How chromosomes are represented
- How fitness functions are designed
- How selection, crossover, mutation, and elitism work
- How the same GA.Core library can be reused across different problems
- How visual applications make evolution easier to understand

The goal is not only to provide code that works, but code that can be studied, modified, extended, and reused in real projects.

Code Structure

This book is organized as a gradual journey from simple genetic algorithm mechanics to complete real-world applications. The code is not presented as isolated snippets. Each project builds a specific idea, and together they show how genetic algorithms can be designed, reused, visualized, and applied across different domains.

Chapter 1 – From Scratch Implementation

You will first build a genetic algorithm without any reusable framework.

The goal is to understand the algorithm from the inside:

- Genes

- Chromosomes
- Population
- Fitness evaluation
- Selection
- Crossover
- Mutation
- Elitism
- Generation loop

The first phrase evolution example is intentionally simple. A chromosome is a string of characters, and the algorithm tries to evolve a target phrase. Because the problem is easy to inspect, every part of the GA becomes visible.

You will see:

- How random chromosomes are created
- How fitness is calculated
- How better candidates survive
- How crossover combines parents
- How mutation introduces variation
- How the best solution improves generation by generation

Chapter 2 – From Scratch GUI Visualization

After the console version, the same phrase evolution idea is rebuilt with WPF.

This chapter shows why visualization matters. Instead of only reading console output, you can watch the population, best chromosome, average fitness, and fitness curve evolve in real time.

You will learn how to present GA behavior through:

- Current best chromosome
- Population preview
- Fitness-over-generation chart
- Start, pause, step, and reset controls
- Parameter inputs for population size, mutation rate, elitism rate, and generation delay

This makes the algorithm easier to understand, debug, and explain.

Chapter 3 – Building GA.Core

Once the basic mechanics are clear, the from-scratch logic is refactored into a reusable library.

The purpose of GA.Core is to separate the genetic algorithm engine from the problem being solved.

The reusable library introduces:

- Chromosome<T>
- IGeneticProblem<T>
- GeneticSolver<T>
- SolverOptions
- Selection operators
- Crossover operators
- Mutation operators
- Generation results
- Minimization and maximization support

Here, you learn how to:

- Separate problem definition from solver logic
- Reuse the same algorithm across many domains
- Plug different operators into the same engine
- Keep examples small while sharing common infrastructure

Chapter 4 – Rebuilding Phrase Evolution with GA.Core

The phrase example is then implemented again, this time using the reusable GA.Core library. This chapter connects the from-scratch version to the framework-based version.

You will see that the problem logic becomes smaller and clearer:

- The problem defines how to create chromosomes
- The problem defines how to calculate fitness
- GA.Core handles population management, selection, crossover, mutation, elitism, and generation flow

This helps you understand what a reusable solver gives you, and what still belongs inside each problem.

Chapter 5 – Classic Optimization Problems

The book then moves into classic genetic algorithm problems.

These examples show how representation changes from problem to problem:

- 8-Queens uses integer genes where each gene represents a queen row.
- Knapsack uses binary genes where each gene decides whether an item is selected.
- Traveling Salesman uses permutation chromosomes where each gene is a city.
- Sudoku uses genes for editable cells while fixed puzzle values remain unchanged.
- Timetabling uses assignment genes for course, room, and time-slot combinations.
- Rectangle Packing uses mixed integer genes for position and rotation.
- Graph Coloring uses color-index genes for graph nodes.

In these chapters, you learn that the most important design decision is often the chromosome representation.

Chapter 6 – Logistics and Planning

The next group focuses on path, routing, and planning behavior.

These examples include:

- Vehicle Routing
- Maze Solver

Vehicle Routing extends the TSP idea by adding capacity constraints and multiple routes. Maze Solver introduces action-sequence evolution, where a chromosome represents movement commands rather than a final static solution.

These projects teach:

- Route encoding
- Constraint penalties
- Sequence simulation
- Collision handling
- Distance-based fitness shaping

Chapter 7 – Engineering and Continuous Search

The engineering examples show how genetic algorithms can optimize physical or mathematical systems.

These examples include:

- Analog RC Filter
- Analog Op-Amp Gain
- Rastrigin Function Minimization

- Gear Train Optimization

Here, chromosomes may represent:

- Resistor and capacitor catalog indices
- Feedback resistor choices
- Real-valued x and y coordinates
- Gear tooth counts

These chapters demonstrate both discrete engineering search and continuous numerical optimization.

Chapter 8 – Puzzle and Creative Search

This group contains more visual and exploratory problems:

- Image Approximation
- Rubik's Cube Solver

Image Approximation evolves colored circles to match a target image. Rubik's Cube evolves short move sequences and visualizes the result in a 3D viewer.

These examples show:

- Simulation-based fitness
- Visual approximation
- Move-sequence chromosomes
- 3D state visualization
- The limits of simple genetic search on very large state spaces

Chapter 9 – Machine Learning Applications

The machine learning examples show how genetic algorithms can support model development.

These examples include:

- ML.NET Hyperparameter Optimization
- Random Forest Hyperparameter Optimization
- Feature Selection

In these chapters, GA is used to search over machine learning decisions:

- Learning rate
- Number of iterations
- Tree count

- Tree depth
- Feature subset selection

The important lesson is that a genetic algorithm can optimize decisions around a model, even when the model itself is trained by another library.

Chapter 10 – Neuroevolution and Policy Search

The final group explores genetic algorithms as a way to evolve behavior.

These examples include:

- Lunar Lander Policy Search
- Neural Architecture Search
- Snake Agent With Neuroevolution

Here, the chromosome no longer represents a direct answer. Instead, it represents a neural network policy.

You will learn how to evolve:

- Neural network weights
- Control policies
- Game agents
- Network architecture choices
- Behavior through simulation feedback

These examples introduce an important idea: a genetic algorithm can train a decision-maker, not just search for a single static solution.

Common Pattern Across Projects

Each project follows the same structure:

1. Problem definition
2. Chromosome representation
3. Fitness function
4. Genetic operators
5. Solver configuration
6. Visualization or output
7. Interpretation of results

This repeated structure makes the book easier to follow. Even when the problem domain changes, the genetic algorithm design process remains familiar.

How to Use the Code

You can use the projects in several ways:

- Run each example as-is
- Change population size, mutation rate, and elitism rate
- Observe how fitness curves change
- Modify the fitness function
- Try different chromosome representations
- Add new mutation or crossover operators
- Adapt GA.Core to your own optimization problems

The code is intentionally written to be:

- Readable
- Modular
- Visual where useful
- Easy to experiment with
- Close to the concepts explained in the book

Practical Tip

Do not only read the code.

Run it.

Pause it.

Change the parameters.

Make the mutation rate too high. Make the population too small. Remove elitism. Change the fitness function. Watch what breaks and what improves.

Genetic algorithms are best understood by observing their behavior over generations.

Final Note

The goal of this book is not only to explain genetic algorithms. It is to help you develop an evolutionary way of thinking.

By the end, you will have seen the same core algorithm applied to strings, boards, routes, schedules, circuits, images, mathematical functions, puzzles, machine learning models, and neural agents.

The code is not an appendix to the book.

It is the book’s laboratory.

FOUNDATIONS

Definition and History

Genetic Algorithms (GAs) are a class of optimization techniques inspired by the process of natural selection. They belong to the broader field of evolutionary computation, which models computational problem-solving based on mechanisms observed in biological evolution.

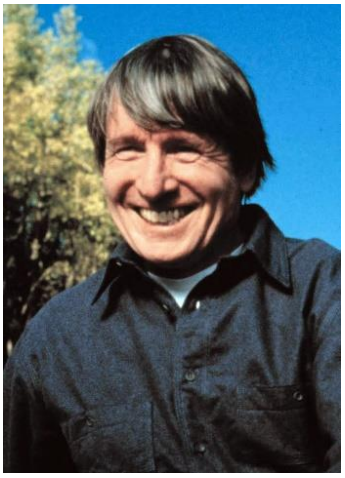
At their core, genetic algorithms operate on a simple yet powerful principle:

Solutions evolve over time through selection, recombination, and variation

Rather than following a deterministic path toward a solution, genetic algorithms explore a population of candidate solutions and iteratively improve them. This population-based approach allows the algorithm to search complex and high-dimensional spaces more effectively than many traditional optimization methods.

The conceptual roots of genetic algorithms can be traced back to the 1950s and 1960s, when researchers began exploring the idea of simulating evolutionary processes using computers. Early work in this area focused on evolutionary strategies and adaptive systems, laying the groundwork for later developments.

However, these early efforts lacked a unified theoretical framework.



John Holland and Formalization

The formal development of genetic algorithms is primarily credited to John Holland in the 1970s. As a professor at the University of Michigan, Holland developed the first comprehensive theory of genetic algorithms and their application to optimization problems.

In his seminal 1975 book Adaptation in Natural and Artificial Systems, Holland introduced the key components of genetic algorithms, including:

- Representation of solutions as chromosomes
- Evaluation through a fitness function
- Evolutionary operators such as selection, crossover and mutation

Holland's work demonstrated that these biologically inspired mechanisms could be used to solve complex computational problems without requiring explicit mathematical models of the search space.

Growth and Popularization

During the 1980s and 1990s, genetic algorithms gained significant attention across multiple disciplines. Researchers and practitioners began applying GAs to real-world problems, particularly in areas where traditional optimization techniques were ineffective.

One of the key figures in popularizing genetic algorithms was David E. Goldberg, whose work in engineering optimization demonstrated their practical value. His book Genetic Algorithms in Search, Optimization, and Machine Learning became a widely referenced resource in the field.

During this period, genetic algorithms were successfully applied in:

- Engineering design optimization
- Control systems
- Scheduling and planning
- Artificial intelligence

This era marked the transition of genetic algorithms from a theoretical concept to a practical problem-solving tool.

Modern Developments

In recent years, genetic algorithms have continued to evolve and remain relevant, particularly in the context of modern computational challenges. With the rise of machine learning and data-driven systems, GAs are increasingly used for:

- Hyperparameter optimization
- Feature selection
- Neural architecture search

Additionally, advances in computing power and parallel processing have significantly improved the scalability and efficiency of genetic algorithms.

Today, genetic algorithms are often combined with other techniques, forming hybrid approaches that leverage the strengths of multiple optimization strategies.

Summary

Genetic algorithms have evolved from early theoretical ideas into a powerful and widely used optimization framework. Rooted in biological principles and formalized through rigorous research, they provide a flexible approach to solving complex problems across diverse domains.

Their continued relevance highlights a key insight:

Nature-inspired computation remains one of the most effective ways to tackle challenging optimization problems.

Usage Fields

At first glance, Genetic Algorithms may seem like an abstract idea—an elegant concept inspired by nature. But their true power reveals itself when they are applied to real-world problems. Across industries and disciplines, Genetic Algorithms quietly solve problems that are too complex, too nonlinear, or simply too vast for traditional methods. In this section, we explore where these algorithms truly come to life.

Engineering: Shaping the Physical World

Imagine designing a bridge, an aircraft component, or even an antenna. There isn't just one "correct" solution—there are thousands, sometimes millions, each with trade-offs.

This is where Genetic Algorithms thrive.

Instead of searching for a perfect answer directly, they evolve better designs over time:

- lighter structures that remain strong
- more efficient mechanical systems
- finely tuned control parameters
- optimized electronic and antenna designs

In engineering, GAs don't just find solutions—**they discover better designs than intuition alone could reach.**

Finance: Navigating Uncertainty

Financial systems are unpredictable, dynamic, and often chaotic.

- How do you balance risk and return?
- How do you build a portfolio that survives uncertainty?

Genetic Algorithms approach this like evolution:

- strategies compete
- poor performers are eliminated
- better ones survive and improve

Over generations, portfolios evolve, trading strategies adapt, and decision-making becomes more robust.

In a world of uncertainty, GAs provide a way to **search intelligently without knowing the future.**

Environment: Understanding Complex Systems

Environmental systems are vast and interconnected. Air pollution, climate models, and geophysical processes involve countless variables interacting in ways we cannot fully observe.

Genetic Algorithms help by exploring possibilities:

- estimating pollution sources
- tuning environmental models
- solving inverse geophysical problems
- optimizing spatial resource use

They allow us to work with incomplete data and still move toward meaningful answers. In these domains, GAs become tools for **understanding systems we cannot directly control**.

Machine Learning & AI: Improving Intelligence

Modern AI systems are powerful—but also complex and sensitive. Small changes in parameters can lead to huge differences in performance.

Genetic Algorithms step in as intelligent explorers:

- tuning hyperparameters
- selecting the most relevant features
- evolving neural network architectures
- optimizing model combinations

When gradients fail or the search space becomes too irregular, GAs continue searching. They don't rely on perfect information—they rely on **progress through iteration.**

Biology & Medicine: Returning to Their Roots

It is no coincidence that Genetic Algorithms shine in biological domains.

After all, they are inspired by life itself.

They are used to:

- model genes and proteins
- explore drug candidates
- analyze biological sequences
- support medical decision systems

These problems often involve enormous search spaces—far beyond brute-force methods.

Here, GAs feel almost natural—**an algorithm inspired by evolution, solving problems of life**.

Operations & Scheduling: Taming Complexity

Some problems are not continuous—they are combinatorial.

Scheduling tasks, routing vehicles, organizing resources—these are problems where possibilities explode exponentially.

Genetic Algorithms offer a practical approach:

- building efficient schedules
- optimizing delivery routes
- improving logistics systems
- balancing resource allocation

Exact solutions may be impossible to compute, but GAs provide solutions that are **good, fast, and practical**.

Creative Fields: Beyond Optimization

Perhaps the most surprising applications of Genetic Algorithms are in creativity.

Instead of optimizing numbers, they evolve ideas.

- game behaviors that adapt over time
- procedurally generated worlds
- music and visual art
- evolving designs and patterns

In these areas, the goal is not always perfection—it is exploration.

Genetic Algorithms become tools not just for solving problems, but for **creating new possibilities**.

Scientific Discovery: Searching the Unknown

In research, some questions don't have clear answers—or even clear paths.

Genetic Algorithms help explore:

- symbolic regression (discovering equations)
- multi-objective optimization
- simulation tuning
- model discovery

They allow scientists to search through immense spaces of possibilities, often revealing solutions that were not previously considered.

In this sense, GAs are not just optimization tools—they are **engines of discovery**.

Closing Thought

Across all these domains, one pattern remains the same:

Genetic Algorithms do not require perfect knowledge. They require only a way to evaluate progress.

From engineering structures to creative art, from financial strategies to scientific discovery, they offer a powerful idea:

Instead of solving a problem directly, evolve the solution

Biological Background

Genetic Algorithms are inspired by one of the most powerful processes in nature: **evolution by natural selection**.

For millions of years, living organisms have adapted to their environments through a simple yet remarkably effective mechanism. Individuals that are better suited to their surroundings are more likely to survive and reproduce. Over time, beneficial traits are preserved, while less effective ones gradually disappear.

This process—slow, iterative, and decentralized—has produced the incredible diversity and complexity of life we observe today.

Genetic Algorithms borrow these core principles and translate them into a computational framework.

Natural Selection: Survival of the Fittest

In nature, not all individuals are equally successful. Some are faster, stronger, or better adapted to their environment. These individuals are more likely to survive and pass their traits to the next generation.

In Genetic Algorithms, this idea is represented through a fitness function. Each candidate solution is evaluated based on how well it solves a problem. The better the solution, the higher its fitness—and the more likely it is to contribute to future generations. Over time, weaker solutions are discarded, and stronger ones dominate the population.

Genes and Chromosomes

Biological organisms store information in the form of **genes**, which are organized into **chromosomes**.

These genes determine observable traits such as eye color, height, or resistance to disease.

In Genetic Algorithms, solutions are encoded in a similar way:

- A **chromosome** represents a complete solution
- **Genes** represent individual components of that solution

For example, in a simple problem, a chromosome might be a string of characters. In more complex cases, it could represent numbers, parameters, or even neural network structures. This encoding allows solutions to be manipulated, combined, and evolved over time.

Reproduction and Crossover

In biological reproduction, offspring inherit genetic material from their parents. This process mixes traits, sometimes producing better combinations than either parent alone.

Genetic Algorithms mimic this process through **crossover (recombination)**.

Two parent solutions are selected, and parts of their structure are combined to create a new solution. This allows useful traits from different individuals to be merged, potentially producing stronger offspring.

Crossover is one of the key mechanisms that enables exploration of the search space.

Mutation: Introducing Diversity

While inheritance passes down traits, evolution would quickly stagnate without variation.

This variation is introduced through **mutation**—random changes in genetic material.

In nature, mutations can be beneficial, neutral, or harmful. However, even small random changes can occasionally lead to significant improvements.

In Genetic Algorithms, mutation plays a similar role:

- it prevents the population from becoming too similar
- it helps escape local optima
- it introduces new possibilities into the search process

Without mutation, the algorithm would lose diversity and converge too early.

Population and Evolution Over Generations

Evolution does not occur in a single individual—it happens across **populations over generations**.

A population of organisms evolves as:

- Individuals are evaluated (fitness)
- The best are selected
- New individuals are created through reproduction
- Random variations are introduced

Genetic Algorithms follow the same cycle. Each iteration (generation) improves the overall quality of solutions. While not every step leads to improvement, the overall trend is toward better and better solutions.

From Biology to Computation

What makes this biological analogy so powerful is its simplicity. There is no central controller. There is no exact formula for the perfect solution. There is only a process:

- variation
- selection
- inheritance

From these simple rules, complex and highly effective solutions emerge.

Genetic Algorithms capture this idea and apply it to computational problems.

A Different Way of Thinking

Traditional algorithms often try to **construct a solution directly**.

Genetic Algorithms take a different approach:

They start with many imperfect solutions and let them evolve.

Instead of solving the problem step by step, they **search through possibilities**, guided by feedback.

This makes them especially useful when:

- the search space is large
- the problem is nonlinear
- or the optimal solution is unknown

Closing Perspective

By mimicking the principles of natural evolution, Genetic Algorithms provide a powerful and flexible framework for solving complex problems.

They remind us of an important idea:

Complex solutions do not always need to be designed—they can emerge.

CORE CONCEPTS

Genes

A gene is the smallest unit of information in a Genetic Algorithm. It represents one changeable part of a candidate solution.

A chromosome is made of many genes. The chromosome is evaluated as a whole, but evolution acts by changing, mixing, and preserving individual genes.

Genes as Building Blocks of Solutions

Instead of constructing a full solution directly, a Genetic Algorithm breaks the solution into smaller pieces. Each piece can be copied, mutated, or recombined with pieces from another solution.

For a simple expression problem, each variable can be represented as one gene:

```
int x = 3;
int y = 7;
int z = 2;
int t = 5;

int target = 20;
int value = x + y + z + t;
```

The same idea becomes more useful when the genes are stored together inside a chromosome:

```
int[] chromosome = [3, 7, 2, 5];
int value = chromosome.Sum();
int fitness = Math.Abs(target - value);
```

Here, the first gene stores x, the second gene stores y, the third gene stores z, and the fourth gene stores t. If the fitness is lower, this chromosome is closer to the target.

Mutation Changes Genes

Mutation is a small random change applied to a gene. It gives the algorithm a way to explore values that do not already exist in the current population.

```
var random = new Random();
int geneIndex = random.Next(chromosome.Length);
chromosome[geneIndex] = random.Next(0, 11);
```

Only one part of the solution changes. The chromosome remains mostly the same, but the algorithm gets a new candidate to test.

Chromosomes

While a gene represents one unit of information, a chromosome represents a complete candidate solution.

A chromosome is the structure that the Genetic Algorithm evaluates, selects, crosses over, and mutates. It is the object that competes inside the population.

From Individual Genes to a Complete Solution

A single gene rarely solves a problem by itself. Genes become meaningful when they are arranged together as a chromosome.

```
int[] genes = [3, 7, 2, 5];
// x = 3, y = 7, z = 2, t = 5
// Together they form one candidate solution.
```

In this example, the chromosome is the full array. Each position has a meaning, and the whole array can be evaluated by a fitness function.

A Minimal Chromosome Type

In code, a chromosome usually stores two things: the genes and the fitness value calculated from those genes.

```
public sealed class Chromosome<TGene>
{
    public Chromosome(IReadOnlyList<TGene> genes)
    {
        Genes = genes.ToArray();
    }

    public TGene[] Genes { get; }

    public double Fitness { get; set; }
}
```

This is the same core idea used by the GA.Core project. The gene type is generic, so the same chromosome structure can hold integers, characters, doubles, booleans, or custom values.

Evaluating a Chromosome

A chromosome becomes useful only after it is evaluated. The fitness function converts the genes into a score.

```
int target = 20;
var chromosome = new Chromosome<int>([3, 7, 2, 5]);
int value = chromosome.Genes.Sum();
chromosome.Fitness = Math.Abs(target - value);
```

For a minimization problem, a lower fitness value means a better chromosome. For a maximization problem, a higher value would be better.

Chromosomes as Search Space Explorers

A Genetic Algorithm does not test one chromosome only. It creates a population, where each chromosome explores a different point in the search space.

```
var population = new List<Chromosome<int>>
{
    new([3, 7, 2, 5]),
    new([6, 1, 9, 4]),
    new([2, 8, 3, 7]),
    new([5, 5, 5, 5])
};
```

Selection keeps the stronger chromosomes, crossover mixes them, and mutation changes some of their genes.

Crossover Creates New Chromosomes

Crossover combines parts of two parent chromosomes to produce a child chromosome.

```
int[] parentA = [3, 7, 2, 5];
int[] parentB = [6, 1, 9, 4];

int[] child = [parentA[0], parentA[1], parentB[2], parentB[3]];
// child = [3, 7, 9, 4]
```

The child is not copied from one parent completely. It inherits some genes from one parent and some genes from the other.

Different Chromosome Structures

The structure of a chromosome depends on the problem:

- Array or list: common for numeric and symbolic problems.
- Binary string: useful for on/off decisions and feature selection.
- Permutation: useful for routes, schedules, and ordering problems.
- Real-valued vector: useful for engineering parameters and neural network weights.
- Custom structure: useful when a solution is naturally a tree, graph, layout, or policy.

```
int[] queens = [0, 4, 7, 5, 2, 6, 1, 3];
bool[] selectedFeatures = [true, false, true, true];
double[] policyWeights = [0.12, -0.44, 1.08, 0.31];
char[] phrase = ['H', 'e', 'l', 'l', 'o'];
```

Closing Insight

Genes define what can change. Chromosomes define what is being optimized.

A good chromosome representation makes evaluation, crossover, and mutation natural. When the chromosome structure matches the problem, the Genetic Algorithm can search more effectively.

Population

A population is a collection of chromosomes. Each chromosome represents a different candidate solution to the same problem.

Genetic Algorithms do not improve one solution in isolation. They improve a group of solutions over many generations.

Why a Population?

A population gives the algorithm multiple search points at the same time. Some chromosomes may be weak, some may be promising, and some may contain useful partial solutions.

- It explores different regions of the search space.
- It reduces the risk of getting stuck in one poor solution.
- It preserves diversity so crossover has useful material to combine.

Creating an Initial Population

At the beginning, the population is usually generated randomly. The goal is not to start with perfect answers, but to create enough variety for evolution to work with.

```
var random = new Random();
var population = new List<Chromosome<int>>();

for (int i = 0; i < 6; i++)
{
    int[] genes =
    [
        random.Next(0, 11),
        random.Next(0, 11),
        random.Next(0, 11),
        random.Next(0, 11)
    ];

    population.Add(new Chromosome<int>(genes));
}
```

Each chromosome is one possible answer. A larger population usually explores more possibilities, but it also requires more fitness calculations.

Evaluating the Population

After creation, every chromosome must be evaluated. The fitness value allows the algorithm to compare candidates.

```
int target = 20;
foreach (var chromosome in population)
{
    int value = chromosome.Genes.Sum();
    chromosome.Fitness = Math.Abs(target - value);
}
```

For this minimization example, the best chromosome is the one with the smallest fitness value.

```
var best = population
    .OrderBy(chromosome => chromosome.Fitness)
    .First();
```

Diversity: The Key to Exploration

A useful population should not contain identical chromosomes too early. If all candidates become similar, the algorithm may stop discovering new possibilities.

- Selection increases pressure toward better solutions.
- Mutation introduces new gene values.
- Crossover recombines useful parts from different chromosomes.

Population Size Matters

Population size controls the balance between exploration and speed.

- Small population: faster, but less diverse and easier to trap in poor solutions.
- Large population: more diverse, but slower because more chromosomes must be evaluated.

There is no universally correct population size. It depends on the problem, chromosome length, fitness cost, and how much diversity is needed.

Creating the Next Generation

A generation is one full population update. The algorithm evaluates the current population, keeps or selects strong chromosomes, creates children, mutates them, and evaluates again.

```
var nextGeneration = new List<Chromosome<int>>();

nextGeneration.Add(best); // elitism keeps the strongest solution

while (nextGeneration.Count < population.Count)
{
    var parentA = SelectParent(population);
    var parentB = SelectParent(population);
    var child = Crossover(parentA, parentB);

    Mutate(child, mutationRate: 0.05);
    nextGeneration.Add(child);
}

population = nextGeneration;
```

Population as Collective Search

A population is more than a list of chromosomes. It is a distributed search process. Different candidates test different ideas, and evolution gradually shifts the group toward better regions of the search space.

Closing Insight

A Genetic Algorithm does not rely on one lucky guess. It relies on a population of imperfect guesses that compete, mix, and improve over time.

The strength of Genetic Algorithms comes from evolving many candidates together, not from optimizing a single candidate alone.

Selection

Selection is the process of choosing chromosomes that will become parents for the next generation.

The goal is simple: better chromosomes should have a higher chance of producing children. However, weaker chromosomes should not disappear too quickly, because they may still contain useful genes.

Why Selection Is Needed

After fitness evaluation, the algorithm knows which chromosomes perform better. Selection uses this information to guide evolution.

- Strong chromosomes are more likely to pass their genes forward.
- Weak chromosomes are less likely to reproduce.
- Some randomness is preserved so the population does not become identical too early.

Sorting by Fitness

A simple first step is sorting the population. In a minimization problem, lower fitness is better.

```
var sortedPopulation = population
    .OrderBy(chromosome => chromosome.Fitness)
    .ToList();

var best = sortedPopulation[0];
```

Sorting tells us who is best, but selection decides who gets to reproduce.

Tournament Selection

Tournament selection randomly picks a small group of chromosomes and selects the best one from that group. It is simple, fast, and works well in many practical Genetic Algorithms.

```
Chromosome<int> TournamentSelect(
    IReadOnlyList<Chromosome<int>> population,
    int tournamentSize,
    Random random)
{
    var candidates = new List<Chromosome<int>>();

    for (int i = 0; i < tournamentSize; i++)
    {
```