Genetic Algorithms

Natural Selection as a Computational Tool

Devyn Miller - MGSC/CS 532

What We'll Cover in This Module

- Genetic algorithms fundamentals
  - Inspired by natural selection
  - Population-based optimization
  - Encoding economic problems
- Core GA operators
  - Selection
  - Crossover
  - Mutation
- Applications in economics
  - The knapsack problem
  - Portfolio optimization
  - Resource allocation
- Advanced concepts
  - Multi-population models
  - Parameter tuning
  - Best practices

Welcome to our module on Genetic Algorithms in Computational Economics!

Over several sessions, we'll explore how these nature-inspired algorithms can solve complex optimization problems that traditional methods struggle with.

GAs are particularly valuable when we face: 1. Multiple local optima 2. Non-linear relationships 3. Combinatorial problems with vast solution spaces 4. Problems without analytical solutions

Point out that students will not just learn theory but implement working GA models for economic applications.

Genetic Algorithms: The Big Picture

What Are Genetic Algorithms?

- **Optimization technique** inspired by natural selection
- **Core idea**: Evolution of solutions over time
- **Key elements**:
- – Population of candidate solutions
  – Fitness-based selection
  – Recombination of successful solutions
  – Random mutation

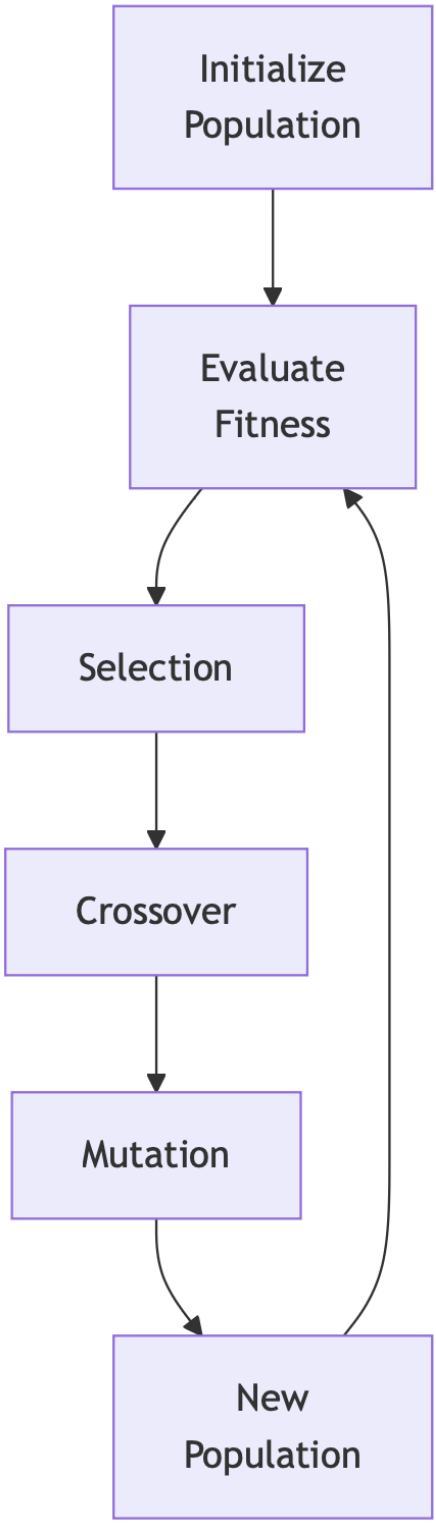


Figure 1: Genetic Algorithm Diagram

Throwback: High School Biology

- **Remember these concepts?**
  - Population of organisms
  - DNA & chromosomes
  - Genes and alleles
  - Fitness: "Survival of the fittest"
  - Natural selection
  - Crossover and mutation
- Evolution = Change in genetic makeup over generations

Natural Selection

1. Phenotypic variation exists within the population.

Originial 

Mutated 

✓ Eg: Variation in beetle shell markings.

2. Specific traits confer fitness advantages.

✓ Predator fears mutated eats original form.

3. Individuals with those traits benefit from increased reproductive success.

Reproduction

✓ Mutated form out-reproduces original form.

Evolution in Nature vs. Genetic Algorithms

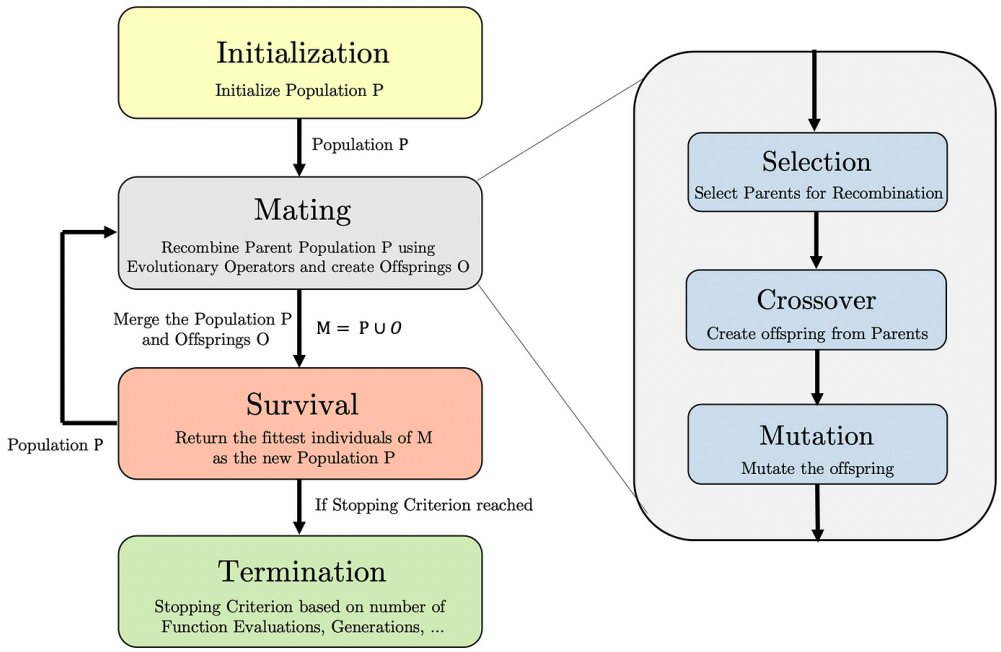
Biological Evolution

- Population of organisms
- DNA/chromosomes = traits
- Fitness = reproductive success
- Natural selection (survival of fittest)
- Crossover & mutation
- Next generation: most fit survive

Genetic Algorithms

- Population of solutions
- Encoded "chromosomes" (e.g., bitstrings)
- Fitness function (objective value)
- Selection of best solutions
- Crossover & mutation operators
- Next generation: better solutions

Genetic Algorithm Diagram



Genetic Algorithms were developed by John Holland in the 1960s and popularized by his student David Goldberg.

The power of GAs lies in their ability to:

1. Explore vast solution spaces efficiently
2. Work with minimal problem information
3. Find "good enough" solutions to intractable problems

Analogies to help students understand: - Like evolution, GAs don't guarantee finding the perfect solution, but they're excellent at finding very good ones - Think of it as "survival of the fittest" for potential solutions to our problem

Ask students: "What other natural processes might inspire computational methods?"

Why Use Genetic Algorithms in Economics?

Strengths:

- Handle complex, nonlinear problems
- No need for derivatives or gradients
- Work with discrete or continuous variables
- Parallelize naturally
- Find multiple solutions

Economic Applications:

- Portfolio optimization
- Resource allocation
- Game theory strategies
- Agent-based modeling
- Market structure evolution
- Policy optimization

Economics is full of complex optimization problems that often:

1. Have multiple local optima
2. Include both discrete and continuous variables
3. Involve constraints that are hard to model analytically
4. Feature interdependencies between variables

Traditional methods like calculus-based optimization often struggle with these challenges.

Point out that GAs don't replace traditional methods but complement them - we use the right tool for the job.

Ask students: "What economic problems have you encountered that might benefit from GA approaches?"

GAs vs. Traditional Optimization

Aspect	Traditional Methods	Genetic Algorithms
Approach	Deterministic	Stochastic
Search	Local, gradient-based	Global, population-based
Solution	Usually single best	Multiple good solutions
Constraints	Often hard to incorporate	Flexible through fitness
Complexity	Struggle with NP-hard problems	Excel with complex landscapes
Derivatives	Often required	Not needed
Parallelization	Often difficult	Natural fit

Traditional optimization methods include: - Linear programming - Gradient descent - Newton-Raphson method - Dynamic programming

Each has strengths but also limitations that GAs can help overcome.

Emphasize that GAs shine when: 1. The fitness landscape is rugged with many local optima 2. The problem has combinatorial complexity 3. The relationships between variables are poorly understood

A good example to discuss: Portfolio optimization with integer constraints on holdings is challenging for traditional methods but well-suited for GAs.

Ask students: "Can you think of an economic problem where you'd prefer traditional optimization? What about one where GAs might be better?"

Classroom Discussion

Question: How would you encode these economic problems for a GA?

1. Portfolio selection (choose 10 stocks from universe of 500)
2. Optimal bidding strategy for auctions
3. Production scheduling with resource constraints
4. Optimal monetary policy rule calibration

Consider:

- What would a "chromosome" represent?
- How would you measure fitness?

- What constraints exist?

Guide students through thinking about encoding strategies:

For portfolio selection: - Binary encoding (1=include stock, 0=exclude) - Fitness could be risk-adjusted return - Constraints include exactly 10 stocks and sector diversification

For auction bidding: - Real-valued encoding of bid amounts for different items - Fitness could be expected profit - Constraints include budget limitations

For production scheduling: - Integer encoding of production quantities by time period - Fitness is profit minus costs - Constraints include production capacity and demand fulfillment

For monetary policy rules: - Real-valued encoding of rule parameters - Fitness based on macroeconomic outcomes - Constraints include bounds on parameter values

Encourage students to debate trade-offs between different encoding schemes.

Core Concepts & Workflow

Solution Encoding: Genotype & Phenotype

Encoding Types:

- **Binary**: [1,0,1,1,0,0]
  - Knapsack, selection problems
- **Integer**: [3,1,4,1,5]
  - Scheduling, resource allocation
- **Real**: [0.35, 1.92, 0.84]
  - Portfolio weights, policy parameters

Example: Knapsack Problem

```
# Binary encoding for knapsack
# 1 = include item, 0 = exclude item
solution = [0,1,0,0,0,1,1]

# This means take items 2, 7, and 8
# and leave the rest behind
```

Challenge: Choose the right encoding for your problem!

Genotype = the encoding representation (e.g., binary string) Phenotype = what the encoding actually means in the problem context

When choosing an encoding scheme, consider: 1. Problem structure - what naturally represents a solution? 2. Efficiency - how compact can the representation be? 3. Completeness - can all possible solutions be represented? 4. Locality - do similar genotypes produce similar phenotypes?

Advanced topic: Some problems benefit from specialized encodings: - Permutation problems (like TSP) need special crossover operators - Tree-based encodings for evolving programs or decision trees

Ask students: "How might the choice of encoding affect the difficulty of the optimization problem?"

Population Initialization

Key Decisions:

- Population size (N)
- Random vs. heuristic seeding
- Diversity considerations

Tradeoffs:

- Large populations → better coverage
- Small populations → faster evolution
- Initialization quality vs. diversity

```
import random

def random_knapsack_solution(num_items):
  return [random.randint(0,1)
          for _ in range(num_items)]

# Generate initial population
POPULATION_SIZE = 20
population = [
    random_knapsack_solution(len(items))
    for _ in range(POPULATION_SIZE)
]
```

Population size is a key parameter: - Too small: Limited genetic diversity, premature convergence - Too large: Computational overhead, slow convergence

Rule of thumb: Population size often between 50-200, but depends on problem complexity

Initialization strategies: 1. Pure random (most common) 2. Heuristically guided (incorporate domain knowledge) 3. Partially seeded with known good solutions

For economic problems, consider: - Incorporating feasible solutions from existing approaches - Using domain constraints to guide initialization - Creating diverse initial solutions across the parameter space

Discuss with students: "What happens if we initialize with all identical solutions? With completely random ones?"

Fitness Functions: Measuring Quality

The fitness function:

- Evaluates solution quality
- Drives selection process
- Maps solutions to numeric values
- Incorporates constraints

Design principles:

- Higher values = better solutions
- Capture true objectives
- Handle constraints appropriately
- Computational efficiency

```
# Knapsack fitness function
def knapsack_fitness(solution, items, max_weight):
  total_weight = 0
  total_value = 0

  # Calculate total weight and value
  for choice, item in zip(solution, items):
    total_weight += choice * item.weight
    total_value += choice * item.value

  # Return 0 if weight constraint violated
  return total_value if total_weight <= max_weight else 0
```

The fitness function is the heart of the GA - it defines what we're actually trying to optimize.

For economic problems, fitness functions often need to: 1. Balance multiple objectives (e.g., risk vs. return) 2. Handle complex constraints (e.g., budget, diversification) 3. Reflect true economic value (not just mathematical convenience)

Constraint handling approaches: 1. Penalty methods (reduce fitness for infeasible solutions) 2. Repair operators (modify infeasible solutions to make them feasible) 3. Special operators (ensure solutions remain feasible)

For the knapsack example, we use a simple approach: zero fitness for infeasible solutions.

Ask students: "How would you design a fitness function for portfolio optimization that balances return and risk?"

Selection: Choosing Parents

Selection methods:

- Roulette wheel (fitness-proportional)
- Rank-based selection
- Tournament selection
- Elitism

Selection pressure:

- Controls convergence rate
- Balances exploration vs. exploitation

```
# Weighted roulette wheel selection
def weighted_selection(population, max_weight):
  # Calculate fitness for each solution
  pop_fitness = [knapsack_fitness(sol, items, max_weight)
                 for sol in population]

  # Use fitness values as weights
  weights = [max(0.1, fit) for fit in pop_fitness]

  # Select two parents based on fitness
  return random.choices(
    population,
    weights=weights,
    k=2
  )
```

Selection is about giving better solutions more chances to reproduce.

Different selection methods have different characteristics: - Roulette wheel: Simple but can lead to premature convergence if fitness varies widely - Rank-based: More stable, less affected by fitness scaling - Tournament: Controllable selection pressure, good for parallel implementation - Elitism: Preserves best solutions, prevents regression

Selection pressure is a crucial parameter: - Too high: Quick convergence but may get stuck in local optima - Too low: Maintains diversity but slow convergence

In economic applications, consider: - Risk of premature convergence to local optima - Need to explore multiple viable strategies - Balance between quick solutions and optimal ones

Ask students: "What selection method would you choose for a highly multimodal economic problem? Why?"

Crossover: Combining Solutions

Crossover types:

- One-point crossover
- Two-point crossover
- Uniform crossover
- Problem-specific crossovers

Key parameters:

- Crossover rate (probability)
- Number and position of crossover points

```
def one_point_crossover(parent1, parent2):
  # Choose random crossover point
  point = random.randint(1, len(parent1)-1)

  # Create children by combining parents
  child1 = parent1[:point] + parent2[point:]
  child2 = parent2[:point] + parent1[point:]

  return child1, child2

# Apply crossover with 70% probability
if random.random() < 0.7:
  child1, child2 = one_point_crossover(parent1, parent2)
```

Crossover (recombination) is the main way GAs explore the solution space by combining successful solutions.

Crossover operators should: 1. Preserve building blocks (good subsolutions) 2. Enable meaningful mixing of parent traits 3. Respect the solution encoding

Different crossover types suit different problems: - One-point: Simple, works well when related traits are adjacent - Two-point: Better preservation of central segments - Uniform: More disruptive, good for breaking unwanted correlations

Crossover rate typically 60-90%: - Higher rates increase exploration - Lower rates preserve good solutions but slow evolution

For economic problems, consider: - Whether parts of solutions can be meaningfully recombined - Whether problem structure suggests certain crossover points are better than others

Demonstrate visually how one-point crossover works.

Mutation: Introducing Variation

Mutation operations:

- Bit flip (binary)
- Random replacement (integer/real)
- Creep mutation (small changes)
- Swap mutation (permutation)

Key parameters:

- Mutation rate
- Mutation magnitude (for real values)

```
def binary_mutation(solution, mutation_rate=0.01):
  # Create a copy to mutate
  mutated = solution.copy()

  # Consider each position for mutation
  for i in range(len(mutated)):
    if random.random() < mutation_rate:
      # Flip the bit (0->1, 1->0)
      mutated[i] = 1 - mutated[i]

  return mutated

child = binary_mutation(child, 0.05)
```

Mutation prevents genetic stagnation and allows exploration of new regions in the search space.

Mutation serves several critical functions: 1. Maintains genetic diversity 2. Helps escape local optima 3. Recovers lost genetic material 4. Enables fine-tuning of solutions

Mutation rate is typically low (0.1% to 5%): - Too high: Disrupts good solutions, approaches random search - Too low: Limited exploration ability, potential premature convergence

For real-valued problems, consider: - Gaussian mutation (add random value from normal distribution) - Non-uniform mutation (decreasing magnitude over generations)

Economic context: Mutation can represent small changes in strategy, policy adjustments, or asset allocation shifts.

Ask students: "How might mutation rate change as the algorithm progresses? Should it be static or dynamic?"

Population Management: Survival Strategies

Generational models:

- Replace entire population each generation
- (,): Generate children, select best
- (+): Combine parents and children, select best

Steady-state models:

- Replace only a few individuals
- Maintain population stability
- Often used with elitism

```
# Generational replacement with elitism
def next_generation(population, elite_count=2):
  # Sort by fitness (descending)
  pop_with_fitness = [(p, knapsack_fitness(p, items, max_weight))
                      for p in population]
  sorted_pop = [p for p, f in sorted(
      pop_with_fitness,
      key=lambda x: x[1],
      reverse=True)]

  # Keep elite individuals
  new_pop = sorted_pop[:elite_count]
```

```
  # Fill rest with offspring
  while len(new_pop) < POPULATION_SIZE:
    p1, p2 = selection(sorted_pop)
    c1, c2 = crossover(p1, p2)
    new_pop.extend([mutation(c1), mutation(c2)])

  return new_pop[:POPULATION_SIZE]
```

Population management determines how new solutions replace old ones in the evolutionary process.

Key considerations: 1. Elitism - preserving best solutions (typically 1-5% of population) 2. Replacement rate - how many individuals to replace each generation 3. Selection pressure - how strongly better solutions are favored

Generational vs. steady-state: - Generational: Complete replacement each cycle (with or without elitism) - Steady-state: Replace only a few individuals each cycle

For economic applications: - Elitism ensures we never lose our best solutions (e.g., profitable strategies) - Steady-state can be good for ongoing optimization of economic models - Balance between maintaining diversity and optimizing current best solutions

Ask students: "In an economic context, what might 'elitism' represent? What are the trade-offs?"

Termination: When to Stop

Common termination criteria:

- Fixed number of generations
- Fitness threshold reached
- Convergence (lack of improvement)
- Time limit exceeded
- Combination of criteria

```
def run_genetic_algorithm():
  # Initialize population
  population = initialize_population()

  # Track best solution and fitness
  best_solution = None
  best_fitness = 0
  stagnant_generations = 0
```

```
  # Main loop
  for generation in range(MAX_GENERATIONS):
    # Evaluate population
    for solution in population:
      fitness = calculate_fitness(solution)
      if fitness > best_fitness:
        best_fitness = fitness
        best_solution = solution
        stagnant_generations = 0
      else:
        stagnant_generations += 1

    # Check termination conditions
    if stagnant_generations >= CONVERGENCE_THRESHOLD:
      print("Converged!")
      break

    # Create next generation
    population = next_generation(population)

  return best_solution, best_fitness
```

Termination criteria determine when to stop the evolutionary process.

Common strategies: 1. Fixed computation budget (generations or evaluations) 2. Convergence detection (no improvement for N generations) 3. Target fitness achieved 4. Diversity below threshold (population too similar)

For economic applications: - Consider computational resources vs. solution quality - May need multiple runs to ensure robust solutions - For time-sensitive applications, set hard time limits - For critical applications, use multiple criteria

Often in economic modeling, we're looking for "good enough" solutions within resource constraints.

Ask students: "What termination criteria would you use for a time-sensitive economic decision vs. a long-term planning problem?"

Hands-on Demonstration

Simple GA Implementation: Function Maximization

Let's maximize:

$$f(x) = x \sin(10\pi x) + 2.0$$

where: $x \in [0, 1]$

This function has multiple local optima, making it a good GA test case.

Key steps:

1. Binary encoding of x values
2. Initialize random population
3. Run GA operations
4. Track best solution

```
import numpy as np
import matplotlib.pyplot as plt

# Target function to maximize
def f(x):
    return x * np.sin(10 * np.pi * x) + 2.0

# Plot the function
x = np.linspace(0, 1, 1000)
y = f(x)
plt.figure(figsize=(8, 4))
plt.plot(x, y)
plt.title("Function to Maximize")
plt.grid(True)
plt.show()

# Global maximum is around x   0.85
# with f(x)   2.85
```

This is a classic test function with multiple local maxima, making it challenging for traditional hill-climbing methods.

The function has: - A global maximum around x 0.85 with value 2.85 - Multiple local maxima due to the sine term - Increasing trend due to the linear term

As we implement the GA, highlight: 1. How we encode real values as binary strings 2. How the population explores different peaks 3. How selection pressure and mutation balance exploration vs. exploitation

This example is simpler than economic problems but illustrates core GA mechanics that transfer to more complex domains.

Encourage students to think about: "How would traditional optimization methods handle this function? Why might they struggle?"

Binary Encoding for Real Values

Binary encoding:

- Map binary string to real value
- More bits = higher precision
- Range mapped to $[0, 2^n - 1]$

Example:

8-bit encoding for $x \in [0, 1]$:

- `00000000` $\rightarrow 0.0$
- `10000000` $\rightarrow 0.5$
- `11111111` $\rightarrow 0.996$

```
def binary_to_real(binary, min_val, max_val):
    # Convert binary array to integer
    int_val = 0
    for i, bit in enumerate(reversed(binary)):
        int_val += bit * (2 ** i)

    # Map to range [min_val, max_val]
    n_bits = len(binary)
    return min_val + (max_val - min_val) * int_val / (2**n_bits - 1)

# Example
chromosome = [1, 0, 1, 1, 0, 0, 1, 0]
x = binary_to_real(chromosome, 0, 1)
print(f"Chromosome {chromosome} represents x = {x:.4f}")
```

Binary encoding is a classic approach for representing real values in GAs.

Key points to emphasize: 1. Precision depends on the number of bits (n bits gives 2^n possible values) 2. We're mapping the integer range [0, 2^n-1] to our desired range [min, max] 3. The mapping is linear, so we get uniform coverage of the range

For economic variables: - Interest rates might need only a few bits (e.g., 0% to 5% in 0.1% increments) - Asset allocation percentages might need more precision - Some problems benefit from direct real-value encoding instead

Trade-off: More bits means higher precision but larger search space.

Demonstrate the concept by showing how different binary strings map to different x-values, and how changing individual bits can make large or small changes to the value depending on their position.

Complete GA for Function Maximization

```
# Parameters
POPULATION_SIZE = 50
CHROMOSOME_LENGTH = 16
MUTATION_RATE = 0.01
MAX_GENERATIONS = 100
MIN_X, MAX_X = 0.0, 1.0

# Initialize population
def initialize_population():
    return [[random.randint(0, 1) for _ in range(CHROMOSOME_LENGTH)]
            for _ in range(POPULATION_SIZE)]

# Calculate fitness
def calculate_fitness(chromosome):
    x = binary_to_real(chromosome, MIN_X, MAX_X)
    return f(x)  # Our target function

# Selection (tournament selection)
def selection(population, fitnesses, tournament_size=3):
    selected = []
    for _ in range(2):
        contestants = random.sample(range(POPULATION_SIZE), tournament_size)
        winner = max(contestants, key=lambda i: fitnesses[i])
        selected.append(population[winner])
    return selected[0], selected[1]
```

```
# Run the algorithm
population = initialize_population()
best_x, best_fitness = 0, 0

for generation in range(MAX_GENERATIONS):
    # Evaluate fitness
    fitnesses = [calculate_fitness(chromosome) for chromosome in population]

    # Track best solution
    best_idx = fitnesses.index(max(fitnesses))
    x_value = binary_to_real(population[best_idx], MIN_X, MAX_X)
    if fitnesses[best_idx] > best_fitness:
        best_fitness = fitnesses[best_idx]
        best_x = x_value

    # Create new population
    new_population = []
    while len(new_population) < POPULATION_SIZE:
        # Selection
        parent1, parent2 = selection(population, fitnesses)

        # Crossover
        if random.random() < 0.7:
            point = random.randint(1, CHROMOSOME_LENGTH-1)
            child1 = parent1[:point] + parent2[point:]
            child2 = parent2[:point] + parent1[point:]
        else:
            child1, child2 = parent1.copy(), parent2.copy()

        # Mutation
        for i in range(CHROMOSOME_LENGTH):
            if random.random() < MUTATION_RATE:
                child1[i] = 1 - child1[i]
            if random.random() < MUTATION_RATE:
                child2[i] = 1 - child2[i]

        new_population.extend([child1, child2])

    population = new_population[:POPULATION_SIZE]

print(f"Best solution: x = {best_x:.6f}, f(x) = {best_fitness:.6f}")
```

This is a complete GA implementation for our function maximization problem.

Key points to highlight: 1. Population initialization (random binary strings) 2. Fitness calculation (evaluating our function) 3. Selection (tournament selection chooses better solutions) 4. Crossover (one-point, combining parent solutions) 5. Mutation (bit flips to maintain diversity) 6. Generational replacement

Walk through a couple of generations to show: - How initial population covers the search space - How selection favors better solutions - How crossover combines good solutions - How mutation provides small variations

This same structure applies to economic optimization problems - only the encoding and fitness function would change based on the specific problem.

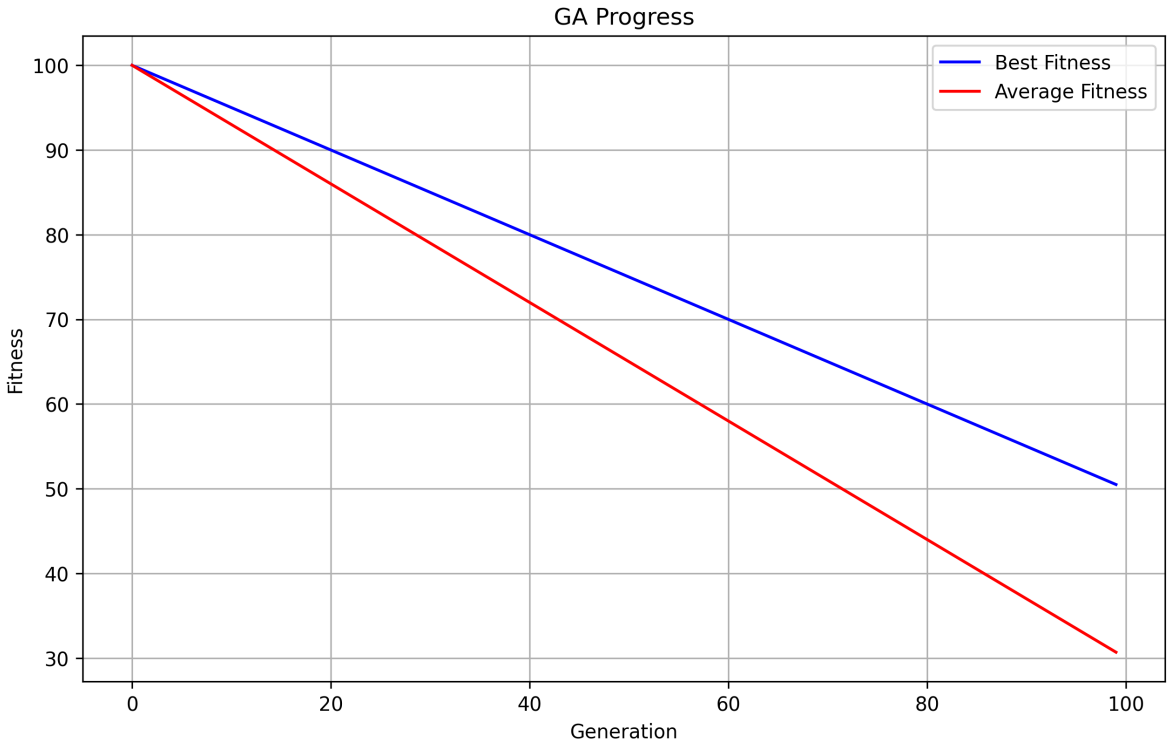
Ask students: "Where in this algorithm might we adjust parameters to balance exploration vs. exploitation?"

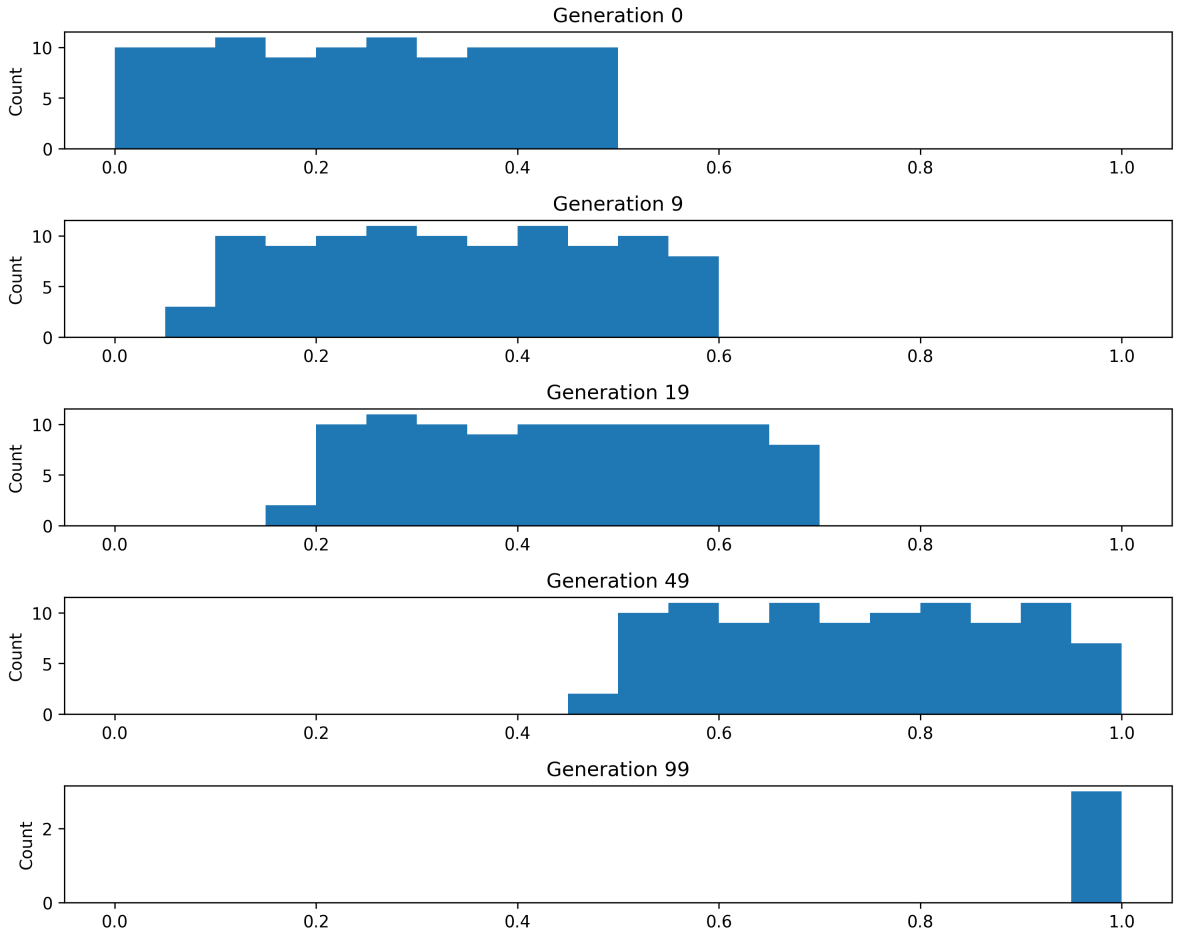
Visualizing GA Progress (1/2): History + Setup

```
# Store history for visualization
history = []

# Inside the GA loop, after fitness evaluation:
history.append({
    'generation': generation,
    'best_fitness': best_fitness,
    'avg_fitness': sum(fitnesses) / len(fitnesses),
    'best_x': best_x,
    'population': [binary_to_real(chrom, MIN_X, MAX_X) for chrom in population]
})

# After the algorithm completes:
generations = [h['generation'] for h in history]
best_fitnesses = [h['best_fitness'] for h in history]
avg_fitnesses = [h['avg_fitness'] for h in history]
```

Visualizing GA Progress (2/2): Generate Plots

```
import matplotlib.pyplot as plt

# Generate mock history data
history = [
    {
        'generation': i,
        'best_fitness': 100 - i * 0.5,
        'avg_fitness': 100 - i * 0.7,
        'population': [i / 100 + j * 0.005 for j in range(100)]
    }
    for i in range(100)
]

generations = [h['generation'] for h in history]
best_fitnesses = [h['best_fitness'] for h in history]
avg_fitnesses = [h['avg_fitness'] for h in history]

# Plot fitness progress
plt.figure(figsize=(10, 6))
plt.plot(generations, best_fitnesses, 'b-', label='Best Fitness')
plt.plot(generations, avg_fitnesses, 'r-', label='Average Fitness')
plt.xlabel('Generation')
plt.ylabel('Fitness')
plt.title('GA Progress')
plt.legend()
plt.grid(True)
plt.show()

# Plot population distribution at selected generations
plt.figure(figsize=(10, 8))
for i, gen in enumerate([0, 9, 19, 49, 99]):
    plt.subplot(5, 1, i+1)
    plt.hist(history[gen]['population'], bins=20, range=(0, 1))
    plt.title(f"Generation {gen}")
    plt.ylabel("Count")
plt.tight_layout()
plt.show()
```


GA Progress
Best Fitness
Average Fitness
Fitness
Generation
100
90
80
70
60
50
40
30
0
20
40
60
80
100



Visualizing GA progress helps understand the evolutionary process and diagnose issues.

Key visualizations to explain: 1. Fitness over time (best and average) - Shows convergence rate - Gap between best and average indicates diversity

2. Population distribution

   - Initial: Wide distribution (exploration phase)
   - Middle: Clustering around promising areas
   - Final: Concentrated at optimal solution(s)

Look for patterns: - Rapid initial improvement followed by slower progress - Premature convergence (flat line too early) - Stagnation (no improvement for many generations) - Population diversity loss (all individuals too similar)

These visualizations help tune parameters: - If converging too quickly, increase population size or decrease selection pressure - If converging too slowly, increase selection pressure or decrease mutation rate

Ask students: "What patterns would you look for to determine if the GA is working effectively?"

Coding Challenge: Parameter Effects (1/3)

Challenge: Modify the GA to investigate these parameters:

1. Population size (`POPULATION_SIZE`)
2. Mutation rate (`MUTATION_RATE`)
3. Selection pressure (tournament size)
4. Crossover rate

Coding Challenge: Parameter Effects (2/3)

For each parameter:

- Run 5 experiments with different values
- Track best fitness and generations to convergence
- Plot results
- Which setting worked best? Why?

Coding Challenge: Parameter Effects (3/3)

```
# Parameter ranges to test
pop_sizes = [10, 25, 50, 100, 200]
mutation_rates = [0.001, 0.01, 0.05, 0.1, 0.2]
tournament_sizes = [2, 3, 5, 7, 10]
crossover_rates = [0.5, 0.6, 0.7, 0.8, 0.9]

# Function to run experiment with given parameters
def run_experiment(pop_size, mut_rate, tourn_size, cross_rate):
    # Your GA implementation here
    # Return best_fitness, generations_to_converge
    pass
```

This challenge helps students understand parameter sensitivity in GAs.

Guidance for students: 1. Focus on one parameter at a time, keeping others constant 2. Run multiple trials for each parameter value (results vary due to randomness) 3. Look for both the best fitness achieved and how quickly it converges 4. Consider the tradeoff between solution quality and computational effort

Expected outcomes: - Population size: Larger populations explore more but take longer - Mutation rate: Too low = premature convergence, too high = disruption - Tournament size: Higher values increase selection pressure - Crossover rate: Higher values increase exploration

Economic context: Parameter tuning is like calibrating an economic model - we're looking for the settings that give the best performance for our specific problem.

After students complete the exercise, discuss how different economic problems might require different parameter settings.

Key Parameters & Tuning

Population Size: Balancing Coverage & Speed

Population size effects:

- **Small** (20-50)
  - Fast execution
  - Limited diversity
  - Quick convergence
  - May miss optimal solutions
- **Large** (100-1000+)
  - Better coverage of search space
  - Higher diversity
  - Slower convergence
  - Higher computational cost

```
# Common scaling approaches

# Scale with problem complexity
POPULATION_SIZE = 4 * CHROMOSOME_LENGTH

# Scale with problem size
# (for combinatorial problems)
POPULATION_SIZE = 10 * NUMBER_OF_VARIABLES
```

```
# Dynamic sizing
# (reduce size over time)
gen_factor = 1.0 - (generation / MAX_GENERATIONS) * 0.5
current_pop_size = int(INITIAL_POP * gen_factor)
```

Population size is one of the most critical parameters in GA performance.

Rules of thumb for sizing: 1. Small problems (few variables): 30-50 individuals 2. Medium problems: 50-200 individuals 3. Large/complex problems: 200-1000+ individuals

The ideal size depends on: - Problem complexity and dimensionality - Available computational resources - Required solution quality - Time constraints

For economic applications: - Portfolio optimization with many assets benefits from larger populations - Simple policy optimization might work well with smaller populations - Consider computational budget vs. solution quality tradeoffs

Some advanced approaches use variable population sizes: - Starting large for exploration, then reducing - Adding new random individuals when diversity drops - Adaptive sizing based on progress

Ask students: "For a portfolio optimization problem, how might the number of assets affect your choice of population size?"

Selection Pressure: Exploration vs. Exploitation

Controlling selection pressure:

- Tournament size
- Ranking schemes
- Fitness scaling
- Elitism percentage

Effects:

- **High pressure**: Quick convergence, risk of local optima
- **Low pressure**: Slow convergence, better exploration

```
# Tournament selection with adjustable pressure
def tournament_selection(population, fitnesses, size):
    """
    Higher size = higher selection pressure
    """
    contestants = random.sample(range(len(population)), size)
```

```
    winner_idx = max(contestants, key=lambda i: fitnesses[i])
    return population[winner_idx]

# Fitness scaling to adjust pressure
def scale_fitness(fitnesses, pressure=1.5):
    """
    Higher pressure value increases selectivity
    """
    ranked = sorted(range(len(fitnesses)),
                    key=lambda i: fitnesses[i])
    scaled = []
    for i in range(len(fitnesses)):
        rank = ranked.index(i)
        scaled.append(2 - pressure +
                      2*(pressure-1)*rank/(len(fitnesses)-1))
    return scaled
```

Selection pressure is the degree to which better solutions are favored over worse ones.

Think of selection pressure as: - High pressure = "survival of only the very fittest" - Low pressure = "most solutions get some chance to reproduce"

Selection pressure affects: 1. Convergence speed 2. Diversity maintenance 3. Exploration vs. exploitation balance 4. Likelihood of finding global vs. local optima

Tournament selection is popular because it's easy to adjust: - Tournament size 2 = low pressure - Tournament size 7+ = high pressure

For economic problems: - Complex landscapes (e.g., macro policy optimization) may need lower pressure - Simple problems with few local optima can use higher pressure - Consider starting with moderate pressure and adjusting based on results

Ask students: "In an economic context, what risks might come from too high selection pressure? Too low?"

Mutation & Crossover Rates: Variation Control

Mutation rate:

- Typical range: 0.1% - 5% per bit
- Too high: Random search
- Too low: Premature convergence
- Consider 1/chromosome_length

Crossover rate:

- Typical range: 60% - 95%
- Higher rate: More exploration
- Lower rate: More exploitation

```
# Adaptive mutation rate
def adaptive_mutation_rate(generation, max_gen):
    """
    Start high, decrease over time
    """
    start_rate = 0.05  # 5%
    end_rate = 0.001   # 0.1%
    return start_rate - (start_rate - end_rate) * (generation / max_gen)

# Adaptive crossover
def adaptive_crossover_rate(avg_fitness, max_fitness):
    """
    Increase crossover when population converging
    """
    fitness_ratio = avg_fitness / max_fitness
    return 0.5 + 0.5 * fitness_ratio  # Range: 0.5 - 1.0
```

Mutation and crossover rates control how much variation is introduced in each generation.

Mutation guidelines: - Binary encoding: Often 1/L where L is chromosome length - Real-valued: Often 5-20% with smaller magnitude changes - Permutation problems: Often 1-5% swap mutations

Crossover considerations: - Rate determines % of population that undergoes crossover - 60-90% is common, with 70-80% being typical - Lower rates preserve more solutions intact - Higher rates create more new combinations

Adaptive rates can improve performance: - Decrease mutation rate as algorithm converges - Increase mutation when population diversity is low - Adjust crossover based on fitness convergence

For economic problems: - When solution precision matters (e.g., exact portfolio weights), consider lower mutation rates later in the run - For exploring diverse economic strategies, maintain higher rates longer

Ask students: “How would you adjust mutation and crossover rates for a rapidly changing economic environment vs. a stable one?”

Convergence Criteria: When Is "Good Enough"?

Common criteria:

- Fitness threshold achieved
- No improvement for N generations
- Population diversity below threshold
- Fixed computational budget
- Combination of criteria

Economic considerations:

- Solution quality requirements
- Time constraints
- Computational resources
- Robustness needs

```
# Track convergence indicators
def check_convergence(history, current_gen):
    # Has best fitness improved recently?
    stagnant_gens = 0
    if current_gen > 0 and history[current_gen]['best_fitness'] <= history[current_gen-1]['be
        stagnant_gens += 1
    else:
        stagnant_gens = 0

    # Check if convergence criteria met
    if stagnant_gens >= STAGNATION_THRESHOLD:
        return True
    return False

# Example usage in GA loop
for generation in range(MAX_GENERATIONS):
    # Evaluate population
    fitnesses = [calculate_fitness(chromosome) for chromosome in population]

    # Track best solution
    best_idx = fitnesses.index(max(fitnesses))
    best_fitness = fitnesses[best_idx]
    best_x = binary_to_real(population[best_idx], MIN_X, MAX_X)

    # Store history
    history.append({'generation': generation, 'best_fitness': best_fitness})
```

```
    # Check for convergence
    if check_convergence(history, generation):
        print(f"Converged at generation {generation}")
        break

    # Create next generation
    population = next_generation(population)

print(f"Best solution: x = {best_x:.6f}, f(x) = {best_fitness:.6f}")
```

Convergence criteria are crucial for determining when to stop the GA process efficiently.

Key considerations: 1. Balance between computational cost and solution quality 2. Avoiding premature convergence while ensuring timely results 3. Using multiple criteria can provide robustness

For economic applications, consider: - The cost of computation vs. the value of improved solutions - The need for timely decisions in dynamic markets - The importance of robustness in policy recommendations

Ask students: "What factors would influence your choice of convergence criteria in a real-world economic scenario?"

Application: The Knapsack Problem

Solving the Knapsack Problem with GAs

Problem setup:

- Maximize total value of items in knapsack
- Subject to weight constraint

GA approach:

1. Binary encoding of item inclusion
2. Fitness = total value (penalize overweight solutions)
3. Use selection, crossover, mutation to evolve solutions

Knapsack GA Code Example

```
# Example knapsack items
items = [
    Item('Item 1', 2, 100),
    Item('Item 2', 3, 120),
    Item('Item 3', 4, 150),
    Item('Item 4', 5, 180),
    Item('Item 5', 6, 200),
    Item('Item 6', 7, 210),
    Item('Item 7', 8, 220),
    Item('Item 8', 50, 800),
]

# Run GA to solve knapsack
best_solution, best_value = run_knapsack_ga(items, max_weight=15)
print(f"Best solution: {best_solution}, Total value: {best_value}")
```

The knapsack problem is a classic combinatorial optimization problem, ideal for demonstrating GAs.

Key points: 1. Binary encoding naturally fits the problem structure 2. Fitness function must balance value and weight constraints 3. GAs can efficiently explore large solution spaces

Discuss with students: - How would you adjust the GA parameters for different item sets? - What real-world problems resemble the knapsack problem?

Application: Optimization in Economics

Portfolio Optimization with GAs

Problem setup:

- Maximize return, minimize risk
- Subject to budget and diversification constraints

GA approach:

1. Real-valued encoding of asset weights
2. Fitness = risk-adjusted return
3. Use selection, crossover, mutation to optimize portfolio

```
# Example portfolio optimization
assets = ['Stock A', 'Stock B', 'Stock C']
returns = [0.1, 0.2, 0.15]
risks = [0.05, 0.1, 0.07]

# Run GA to optimize portfolio
best_portfolio, best_return = run_portfolio_ga(assets, returns, risks)
print(f"Best portfolio: {best_portfolio}, Expected return: {best_return}")
```

Portfolio optimization is a key application of GAs in finance.

Key points: 1. Real-valued encoding allows precise control of asset weights 2. Fitness function must balance return and risk 3. GAs can handle complex constraints and multiple objectives

Discuss with students: - How would you incorporate additional constraints (e.g., sector limits)? - What are the advantages of using GAs over traditional methods?

Advanced Variants

Multi-population GAs and Parallelization

Concepts:

- Multiple subpopulations (islands)
- Periodic migration of individuals
- Enhances diversity, prevents premature convergence

Benefits:

- Better exploration of solution space
- Scalability with parallel computing

```
# Example multi-population GA setup
num_islands = 5
migration_interval = 10

# Run multi-population GA
best_solutions = run_multi_population_ga(num_islands, migration_interval)
print(f"Best solutions from islands: {best_solutions}")
```

Multi-population GAs (island models) are powerful for maintaining diversity and exploring large solution spaces.

Key points: 1. Subpopulations evolve independently, enhancing exploration 2. Migration allows sharing of good solutions 3. Suitable for parallel and distributed computing environments

Discuss with students: - How does migration frequency affect performance? - What types of economic problems might benefit from this approach?

Pitfalls and Best Practices

Avoiding Common GA Pitfalls

Common issues:

- Premature convergence
- Loss of diversity
- Overfitting to training data

Best practices:

- Maintain diversity (mutation, crossover)
- Use adaptive parameters
- Validate with unseen data

```
# Example strategies to maintain diversity
mutation_rate = adaptive_mutation_rate(generation, MAX_GENERATIONS)
crossover_rate = adaptive_crossover_rate(avg_fitness, max_fitness)

# Validate GA solutions
validate_solutions(best_solutions, validation_data)
```

Avoiding common pitfalls ensures robust GA performance.

Key strategies: 1. Use adaptive mutation and crossover rates to maintain diversity 2. Validate solutions on unseen data to prevent overfitting 3. Monitor convergence and adjust parameters as needed

Discuss with students: - What signs indicate premature convergence? - How can you ensure your GA solutions generalize well?

Discussion and Exploration

Open Questions and Future Directions

- How can GAs be combined with other optimization techniques?
- What are the limitations of GAs in economic modeling?
- How can GAs be adapted for real-time decision-making?

Student Challenge: Modify a GA parameter and observe the impact on convergence and solution quality.

Encourage students to think critically about the role of GAs in economic modeling.

Key discussion points: 1. Hybrid approaches (e.g., combining GAs with local search) 2. Limitations in terms of scalability and interpretability 3. Potential for real-time applications in dynamic markets

Challenge students to experiment with GA parameters and share their findings.