Technical Documentation: Improved Binary Genetic Algorithm with Gray Coding and Hybrid Optimization

Abstract

This document provides comprehensive technical documentation for an advanced Binary Genetic Algorithm implementation. The code features Gray encoding for improved search space continuity, non-uniform mutation with dynamic rate adjustment, hybrid optimization combining genetic operators with local search (Hill Climbing), and extensive grid search capabilities for parameter tuning across multiple benchmark functions. This documentation covers all functions, operators, optimization strategies, and experimental framework.

Contents

1 Overview

1.1 Algorithm Purpose

This implementation is a **Hybrid Binary Genetic Algorithm (BGA)** designed for continuous optimization problems. It combines:

- **Binary representation** with Gray encoding
- **Genetic operators** (crossover, mutation, selection)
- **Hill Climbing** local search
- **Adaptive mechanisms** (non-uniform mutation, stagnation recovery)
- **Automated experimentation** with grid search

1.2 Key Features

1. **Gray Coding:** Reduces Hamming cliff problem
2. **Non-Uniform Mutation:** Decreases mutation rate over generations
3. **Hybrid Search:** Combines global (GA) and local (HC) search
4. **Elitism:** Preserves best solutions
5. **Stagnation Recovery:** Reinitializes population when stuck
6. **Multiple Selection Methods:** Tournament and Roulette Wheel
7. **Grid Search:** Automated parameter tuning

1.3 Benchmark Functions

The code tests on four classic optimization functions:

Function	Domain	Global Minimum
Rastrigin	$[-5.12, 5.12]^n$	$f(\mathbf{0}) = 0$
Griewangk	$[-600, 600]^n$	$f(\mathbf{0}) = 0$
Rosenbrock	$[-5, 10]^n$	$f(\mathbf{1}) = 0$
Michalewicz	$[0, \pi]^n$	$f(\mathbf{x}^*) \approx -1.8$ (n=2)

Table 1: Benchmark optimization functions

2 Test Functions

2.1 Rastrigin Function

```
def rastrigin(x: np.ndarray) -> float:
    A = 10
    n_dim = x.shape[1] if x.ndim > 1 else len(x)
    return A * n_dim + np.sum(x**2 - A * np.cos(2 * math.pi * x),
                              axis=1 if x.ndim > 1 else None)
```

Mathematical Formula:

$$f(\mathbf{x}) = 10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i)\right]$$

Characteristics:

- Highly multimodal with many local minima
- Regular structure of peaks
- Global minimum at origin: $f(\mathbf{0}) = 0$
- Difficulty increases with dimensionality

2.2 Griewangk Function

```python
def griewangk(x: np.ndarray) -> float:
    n_dim = x.shape[1] if x.ndim > 1 else len(x)
    indices = np.arange(1, n_dim + 1)
    sum_part = np.sum(x**2 / 4000.0, axis=1 if x.ndim > 1 else None)
    prod_part = np.prod(np.cos(x / np.sqrt(indices)),
                        axis=1 if x.ndim > 1 else None)
    return sum_part - prod_part + 1
```

Mathematical Formula:

$$f(\mathbf{x}) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

Characteristics:

- Product term creates interdependence between variables
- Wide basin of attraction around global minimum
- Many local minima superimposed
- Global minimum: $f(\mathbf{0}) = 0$

2.3 Rosenbrock Function

```python
def rosenbrock(x: np.ndarray) -> float:
    if x.ndim > 1:
        return np.array([rosenbrock(ind) for ind in x])
    return np.sum(100.0 * (x[1:] - x[:-1]**2)**2 + (1 - x[:-1])**2)
```

Mathematical Formula:

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right]$$

Characteristics:

- Narrow, parabolic-shaped valley
- Global minimum along $x_i = 1$: $f(\mathbf{1}) = 0$
- Difficult due to valley shape, not multimodality
- Tests algorithm's ability to navigate narrow valleys

2.4 Michalewicz Function

```
def michalewicz(x: np.ndarray, m: int = 10) -> float:
    n_dim = x.shape[1] if x.ndim > 1 else len(x)
    i = np.arange(1, n_dim + 1)
    return -np.sum(np.sin(x) * (np.sin(i * x**2 / math.pi))**(2*m),
                   axis=1 if x.ndim > 1 else None)
```

Mathematical Formula:

$$f(\mathbf{x}) = -\sum_{i=1}^{n} \sin(x_i) \left[\sin\left(\frac{ix_i^2}{\pi}\right)\right]^{2m}$$

Characteristics:

- Parameter m defines steepness of valleys
- $m = 10$ creates very steep valleys
- Number of local minima increases with n
- For $n = 2$: $f(\mathbf{x}^*) \approx -1.8013$

3 Gray Coding Implementation

3.1 Why Gray Coding?

Problem with Standard Binary: Adjacent real values may have large Hamming distances.
Example:

```
Decimal 7:  0111
Decimal 8:  1000  (Hamming distance = 4!)
```

Solution - Gray Coding: Adjacent values differ by exactly 1 bit.

```
Decimal 7:  0100
Decimal 8:  1100  (Hamming distance = 1)
```

3.2 Bit Precision Calculation

```
def calculate_n_bits(bounds, precision=5):
    lower_bound, upper_bound = bounds
    num_steps = (upper_bound - lower_bound) * (10 ** precision)
    n_bits = math.ceil(math.log2(num_steps))
    return int(n_bits)
```

Purpose: Determines how many bits needed for desired decimal precision.
Algorithm:

1. Calculate total steps: $(upper - lower) \times 10^{precision}$
2. Find minimum bits: $\lceil \log_2(steps) \rceil$

Example:

- Bounds: $[-5.12, 5.12]$, Precision: 4
- Steps: $(5.12 - (-5.12)) \times 10^4 = 102,400$
- Bits: $\lceil \log_2(102400) \rceil = 17$ bits

3.3 Gray to Binary Conversion

```
def gray_to_binary_vectorized(gray_codes):
    mask = gray_codes >> 1
    binary_codes = gray_codes.copy()
    while np.any(mask):
        binary_codes ^= mask
        mask >>= 1
    return binary_codes
```

Purpose: Converts Gray code integers to standard binary integers.
Algorithm:

1. Start with Gray code value
2. XOR with right-shifted version
3. Repeat until no bits left to process

Example Conversion:

```
Gray code:     1100 (decimal 12 in Gray)
Step 1: XOR    0110
Result:        1010
Step 2: XOR    0101
Result:        1111
Step 3: XOR    0111
Result:        1000 (decimal 8 in binary)
```

3.4 Population Decoding with Gray

```
def decode_population_gray_vectorized(bounds, n_bits, n_dim,
                                      population):
    lower_bound, upper_bound = bounds
    pop_reshaped = population.reshape(population.shape[0],
                                      n_dim, n_bits)

    # Convert binary to Gray integers
    powers_of_2 = 2 ** np.arange(n_bits - 1, -1, -1)
    gray_integer_values = pop_reshaped @ powers_of_2

    # Convert Gray to binary integers
    binary_integer_values = gray_to_binary_vectorized(
                                gray_integer_values)

    # Scale to real values
    max_integer = 2 ** n_bits - 1
    scaled_values = lower_bound + (binary_integer_values /
                    max_integer) * (upper_bound - lower_bound)

    return scaled_values
```

Purpose: Converts entire population from binary chromosomes to real-valued vectors.
Steps:

1. Reshape population: (pop_size, n_dim × n_bits) → (pop_size, n_dim, n_bits)
2. Convert each gene to Gray integer
3. Convert Gray integers to binary integers

4. Scale to $[lower\_bound, upper\_bound]$

Scaling Formula:

$$x_{real} = lower + \frac{x_{int}}{2^{n_{bits}} - 1} \times (upper - lower)$$

4 Genetic Operators

4.1 Uniform Crossover

```python
def crossover_uniform_vectorized(p1, p2, r_cross):
    if np.random.rand() >= r_cross:
        return p1.copy(), p2.copy()
    mask = np.random.rand(len(p1)) < 0.5
    c1 = np.where(mask, p2, p1)
    c2 = np.where(mask, p1, p2)
    return c1, c2
```

Purpose: Creates offspring by exchanging genes between parents.
Algorithm:

1. With probability r_{cross}, perform crossover
2. Generate random mask (0.5 probability for each bit)
3. Child 1: Take bits from P2 where mask=True, P1 otherwise
4. Child 2: Inverse of Child 1

Example:

```
Parent 1:  1 0 1 1 0 1
Parent 2:  0 1 0 0 1 0
Mask:      T F T F T F
Child 1:   0 0 0 1 1 1  (P2 where T, P1 where F)
Child 2:   1 1 1 0 0 0  (opposite)
```

4.2 Bit-Flip Mutation

```python
def mutation_vectorized(chromosome, r_mut):
    mask = np.random.rand(len(chromosome)) < r_mut
    chromosome[mask] = 1 - chromosome[mask]
    return chromosome
```

Purpose: Introduces random variations by flipping bits.
Algorithm:

1. For each bit, generate random number $r \in [0, 1]$
2. If $r < r_{mut}$, flip bit (0→1 or 1→0)

Example:

```
Chromosome:  1 0 1 1 0 1
r_mut = 0.1
Random:     0.05 0.8 0.03 0.5 0.9 0.2
Flip?:       Yes  No  Yes  No  No  No
Result:      0 0 0 1 0 1
```

5 Selection Methods

5.1 Tournament Selection

```
def selection_tournament(pop, scores, k=3):
    pop_indices = np.arange(len(pop))
    tournament_ix = np.random.choice(pop_indices, size=k,
                                     replace=False)
    winner_local_ix = np.argmin(scores[tournament_ix])
    winner_global_ix = tournament_ix[winner_local_ix]
    return pop[winner_global_ix]
```

Purpose: Selects individuals through competition.
Algorithm:

1. Randomly select k individuals (default: $k = 3$)
2. Compare their fitness values
3. Return the best individual from the tournament

Characteristics:

- **Selection pressure:** Controlled by k
- Larger k = stronger pressure toward best solutions
- Smaller k = more exploration, less exploitation
- Computationally efficient

Example:

```
Population indices: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Tournament (k=3): Select [2, 7, 5]
Scores: [45.2, 38.1, 51.3]
Winner: Individual 7 (score 38.1)
```

5.2 Roulette Wheel Selection

```
def selection_roulette_wheel(pop, scores):
    inverted_scores = 1.0 / (scores - np.min(scores) + 1e-9)
    total_fitness = np.sum(inverted_scores)
    if total_fitness == 0:
        probabilities = np.full(len(pop), 1 / len(pop))
    else:
        probabilities = inverted_scores / total_fitness
    selected_ix = np.random.choice(len(pop), p=probabilities)
    return pop[selected_ix]
```

Purpose: Probabilistic selection based on fitness.
Algorithm:

1. Invert scores (for minimization): $fitness = \frac{1}{score - min(scores) + \epsilon}$
2. Normalize to probabilities: $p_i = \frac{fitness_i}{\sum fitness}$
3. Select individual proportional to probability

Why invert? Lower score = better solution, but needs higher selection probability.
Example:

```
Scores:      [10, 20, 30, 40]
Min score:   10
Adjusted:    [0, 10, 20, 30]
Inverted:    [inf, 0.1, 0.05, 0.033]
Normalized:  [0.0, 0.54, 0.27, 0.18]
Selection:   Individual 1 has 54% chance
```

6 Hill Climbing Integration

6.1 Hill Climbing with Gray Coding

```python
def hill_climbing_gray(objective_func, bounds, n_bits, n_dim,
                       chromosome, max_attempts):
    def decode_single_gray(chrom):
        pop_matrix = chrom.reshape(1, -1)
        return decode_population_gray_vectorized(
                   bounds, n_bits, n_dim, pop_matrix)[0]

    current_best_chromosome = chromosome.copy()
    current_best_score = objective_func(
                            decode_single_gray(current_best_chromosome))

    bit_indices = np.random.permutation(len(chromosome))
    for i in range(min(max_attempts, len(bit_indices))):
        bit_to_flip = bit_indices[i]
        neighbor = current_best_chromosome.copy()
        neighbor[bit_to_flip] = 1 - neighbor[bit_to_flip]

        neighbor_score = objective_func(decode_single_gray(neighbor))

        if neighbor_score < current_best_score:
            current_best_chromosome = neighbor
            current_best_score = neighbor_score

    return current_best_chromosome
```

Purpose: Local refinement of elite solution through bit-flip neighborhood search.
Algorithm:

1. Start with elite chromosome
2. Randomly permute bit indices
3. For each bit (up to max_attempts):
   - Flip the bit
   - Decode and evaluate
   - Keep if better
4. Return best chromosome found

Key Features:

- **First Improvement:** Accepts first better neighbor
- **Random order:** Prevents bias toward specific bits
- **Limited attempts:** Controls computational cost

- **Greedy:** Always improves, never worsens

Example Execution:

```
Initial chromosome: 101101 (fitness: 45.2)
Attempt 1: Flip bit 3 -> 101001 (fitness: 43.8) ACCEPT
Attempt 2: Flip bit 0 -> 001001 (fitness: 44.1) REJECT
Attempt 3: Flip bit 4 -> 111001 (fitness: 42.5) ACCEPT
...
Final: 111001 (fitness: 42.5)
```

7 Main Algorithm: BGA Improved

7.1 Algorithm Overview

The main function `bga_improved` orchestrates all components into a complete optimization algorithm.

7.2 Non-Uniform Mutation Rate

Key Innovation: Mutation rate decreases over generations.

Formula:

$$r_{mut}(gen) = r_{final} + (r_{initial} - r_{final}) \times \left(1 - \frac{gen}{n_{iter}}\right)^{decay}$$

where:

- $r_{initial}$: Starting mutation rate (high exploration)
- $r_{final} = r_{initial}/100$: Final rate (exploitation)
- $decay = 4.0$: Controls transition speed
- gen: Current generation
- n_{iter}: Total generations

Rationale:

- **Early generations:** High mutation for exploration
- **Late generations:** Low mutation for fine-tuning
- **Smooth transition:** Power decay function

7.3 Stagnation Recovery

Problem: Population may converge prematurely to local optimum.

Solution: Re-initialize population while keeping elite solutions.

Algorithm:

1. Track generations without improvement
2. If stagnation limit reached:
   - Keep top 10% (elite)
   - Generate 90% new random individuals
   - Reset stagnation counter

Benefits:

- Escapes local optima
- Maintains diversity
- Preserves best solutions found

7.4 Main Loop Structure

```
for gen in range(n_iter):
    # 1. Calculate adaptive mutation rate
    progress = gen / n_iter
    effective_r_mut = r_mut_final + (r_mut_initial - r_mut_final)
                      * (1 - progress) ** decay_factor

    # 2. Decode and Evaluate
    decoded_pop = decode_population_gray_vectorized(...)
    scores = objective_func(decoded_pop)

    # 3. Update best solution
    # 4. Check stagnation
    # 5. Selection
    # 6. Crossover and Mutation
    # 7. Elitism + Hill Climbing
    # 8. Replacement
```

Flow Diagram:

```
Start → Initialize Population (binary) →
  ↓
  Loop (n_iter generations):
    1. Calculate r_mut(gen)
    2. Decode (Gray) → Real values
    3. Evaluate fitness
    4. Update best
    5. Stagnation check?
       Yes → Reinitialize 90%
    6. Selection (Tournament/Roulette)
    7. Crossover (Uniform)
    8. Mutation (Bit-flip, adaptive rate)
    9. Hill Climbing on elite
    10. Replace worst with refined elite
  ↓
End → Return best solution
```

8 Grid Search Framework

8.1 Parameter Grid

The code performs exhaustive testing across:

Parameter	Values Tested
Functions	Rastrigin, Griewangk, Rosenbrock, Michalewicz
Dimensions	2, 30, 100
Selection	Tournament, Roulette Wheel
Crossover rate	0.7, 0.8, 0.9
Mutation multiplier	1.0, 2.0, 4.0
Total experiments	$4 \times 3 \times 2 \times 3 \times 3 = 216$
Runs per experiment	30
Total evaluations	6,480 runs

Table 2: Grid search configuration

8.2 Adaptive Parameters by Dimension

Low dimensions (2D):

- Iterations: 200
- Population: 100
- HC attempts: 15
- Stagnation: Disabled

High dimensions (30D, 100D):

- Iterations: 400
- Population: 150
- HC attempts: 20
- Stagnation limit: 50 generations

Rationale:

- Higher dimensions require more exploration
- Larger populations reduce premature convergence
- More iterations allow sufficient convergence time
- Stagnation recovery prevents getting stuck

8.3 Output Structure

For each experiment, the code generates:

1. **summary.txt**: Statistical results
   - Best, mean, std of 30 runs
   - Best solution vector
   - Algorithm parameters
   - Execution time
2. **convergence.png**: Convergence plot
   - X-axis: Generation
   - Y-axis: Best score
   - Shows optimization trajectory
3. **boxplot.png**: Score distribution
   - Visualizes 30 final scores
   - Shows median, quartiles, outliers
   - Assesses algorithm consistency

Directory structure:

```
GRID_SEARCH_RESULTS_BINAR_IMBUNATATIT/
 Rastrigin_D2_Tournament_C0.7_M1.0/
    summary.txt
    convergence.png
    boxplot.png
 Rastrigin_D2_Tournament_C0.7_M2.0/
    ...
 ...
```

9 Visualization Functions

9.1 Convergence Plot

```
def plot_convergence(history, title, output_dir):
    plt.figure(figsize=(10, 6))
    plt.plot(history)
    plt.title(f"Convergenta - {title}")
    plt.xlabel("Generatie")
    plt.ylabel("Cel mai bun scor")
    plt.grid(True)
    plt.savefig(os.path.join(output_dir, "convergence.png"))
    plt.close()
```

Purpose: Shows how best solution improves over generations.
Interpretation:

- **Steep decline:** Rapid improvement (good)
- **Plateau:** Convergence or stagnation
- **Oscillations:** Exploration vs exploitation
- **Final value:** Solution quality

9.2 Box Plot

```
def plot_box_scores(all_scores, title, output_dir):
    plt.figure(figsize=(8, 6))
    plt.boxplot(all_scores)
    plt.title(f"Distributia Scorurilor - {title}")
    plt.ylabel("Scor Final")
    plt.xticks([1], ['BGA Imbunatatit'])
    plt.grid(True)
    plt.savefig(os.path.join(output_dir, "boxplot.png"))
    plt.close()
```

Purpose: Visualizes algorithm consistency across 30 runs.
Box plot components:

- **Box:** Interquartile range (IQR) - 25th to 75th percentile
- **Line in box:** Median score
- **Whiskers:** Min and max within 1.5*IQR
- **Dots:** Outliers beyond whiskers

Interpretation:

- **Narrow box:** Consistent performance
- **Wide box:** High variability
- **Low median:** Generally good results
- **Few outliers:** Reliable algorithm

10 Complete Algorithm Pseudocode

```
Algorithm: Improved Binary GA with Gray Coding
Input: objective_func, bounds, n_dim, n_bits, n_iter, n_pop,
       r_cross, r_mut_initial, hc_attempts, selection_func
Output: best_solution, best_score, convergence_history

1. Initialize:
   - chromosome_len = n_bits * n_dim
   - population = random binary matrix (n_pop x chromosome_len)
   - best_score = infinity
   - r_mut_final = r_mut_initial / 100

2. For each generation g from 0 to n_iter:

   a) Calculate adaptive mutation rate:
      progress = g / n_iter
      r_mut = r_mut_final + (r_mut_initial - r_mut_final)
              * (1 - progress)^4

   b) Decode population using Gray coding:
      decoded_pop = decode_population_gray_vectorized(population)

   c) Evaluate fitness:
      scores = objective_func(decoded_pop)

   d) Update best solution:
      if min(scores) < best_score:
         best_solution = population[argmin(scores)]
         best_score = min(scores)
         stagnation_counter = 0
      else:
         stagnation_counter += 1

   e) Stagnation recovery (if enabled):
      if stagnation_counter >= limit:
         Keep top 10% elite
         Generate 90% new random individuals
         Reset stagnation_counter

   f) Selection:
      selected_pop = [selection_func(pop, scores)
                      for _ in range(n_pop)]

   g) Generate offspring:
      For i = 0, 2, 4, ..., n_pop-2:
         p1, p2 = selected_pop[i], selected_pop[i+1]
         c1, c2 = crossover_uniform(p1, p2, r_cross)
         children[i] = mutation(c1, r_mut)
         children[i+1] = mutation(c2, r_mut)

   h) Elitism + Hill Climbing:
      elite = best individual in population
```

```
      refined_elite = hill_climbing_gray(elite, hc_attempts)
      Replace worst child with refined_elite

   i) Replacement:
      population = children

3. Decode final best solution
4. Return best_solution, best_score, history
```

11 Usage Example

11.1 Basic Usage

```python
# Run single experiment
solution, score, history = bga_improved(
    objective_func=rastrigin,
    bounds=[-5.12, 5.12],
    n_dim=30,
    n_bits=17,  # Calculated from precision
    n_iter=400,
    n_pop=150,
    r_cross=0.9,
    r_mut_initial=0.001,
    hc_attempts=20,
    selection_func=selection_tournament,
    stagnation_limit=50
)

print(f"Best solution: {solution}")
print(f"Best score: {score}")
```

11.2 Parameter Guidelines

Crossover rate (r_cross):

- Typical range: 0.7 - 0.9
- Higher = more mixing
- Lower = more preservation

Mutation rate (r_mut_initial):

- Recommended: $\frac{1}{n_{bits} \times n_{dim}}$ to $\frac{4}{n_{bits} \times n_{dim}}$
- Scales with problem size
- Decreases automatically over time

Population size (n_pop):

- Low dim: 50-100
- High dim: 100-200
- Larger = more diversity, slower

Iterations (n_iter):

- Low dim: 100-200
- High dim: 300-500
- Depends on convergence speed

12 Key Advantages

1. **Gray Coding:** Eliminates Hamming cliff problem
   - Smooth transitions in search space
   - Better for real-valued optimization
2. **Adaptive Mutation:**
   - Balances exploration and exploitation
   - Automatic schedule, no manual tuning
3. **Hybrid Approach:**
   - GA for global search
   - HC for local refinement
   - Best of both worlds
4. **Stagnation Recovery:**
   - Escapes local optima
   - Maintains diversity
5. **Comprehensive Testing:**
   - Grid search for parameter tuning
   - Statistical validation (30 runs)
   - Multiple benchmark functions

13 Computational Complexity

13.1 Per Generation Complexity

Operation	**Complexity**
Decoding (Gray)	$O(n_{pop} \times n_{dim} \times n_{bits})$
Evaluation	$O(n_{pop} \times n_{dim})$
Selection	$O(n_{pop} \times k)$ or $O(n_{pop})$
Crossover	$O(n_{pop} \times chromosome\_len)$
Mutation	$O(n_{pop} \times chromosome\_len)$
Hill Climbing	$O(hc\_attempts \times n_{dim})$
Total per generation	$O(n_{pop} \times n_{dim} \times n_{bits})$

Table 3: Complexity analysis per generation

13.2 Full Algorithm Complexity

$$T_{total} = n_{iter} \times O(n_{pop} \times n_{dim} \times n_{bits})$$

Typical execution times:

- 2D, 200 iter, 100 pop: ∼1-2 seconds
- 30D, 400 iter, 150 pop: ∼10-20 seconds
- 100D, 400 iter, 150 pop: ∼30-60 seconds

14 Conclusions

- **Gray encoding** for improved continuity
- **Adaptive mechanisms** for automatic parameter control
- **Hybrid optimization** combining global and local search
- **Robust design** with stagnation recovery
- **Comprehensive evaluation** framework