CSE5012 Assignment 1

Zhao ZHAO
National Graduate College for Engineers
Southern University of Science and Technology
Shenzhen, China
12541018@mail.sustech.edu.cn

***Abstract*—This paper implements an evolutionary algorithm framework based on integer encoding to solve bound-constrained single-objective numerical optimization problems. The algorithm incorporates two selection strategies, tournament selection and roulette wheel selection, combined with single-point crossover and random reset mutation operations. It has been systematically validated on 15 classical benchmark test functions. Experimental results demonstrate that the algorithm exhibits excellent optimization performance on both unimodal and multimodal functions, effectively finding global optimal or near-optimal solutions. This paper details the design principles and implementation specifics of each algorithm operator and provides complete pseudocode descriptions.**

***Index Terms*—Evolutionary Algorithms, Global Optimization, Genetic Operators, Performance Evaluation**

I. Introduction

Optimization problems have broad applications in fields such as engineering, economics, and management, making the search for efficient optimization algorithms a persistent research focus. Evolutionary algorithms, as a class of stochastic optimization techniques inspired by natural evolutionary processes, have garnered widespread attention due to their powerful global search capabilities and robustness. These algorithms do not require gradient information of the objective function and have low demands on the mathematical properties of the problem, making them particularly suitable for handling complex nonlinear and multimodal optimization problems.

Benchmark test functions provide a standardized testing platform for evaluating the performance of optimization algorithms. By validating algorithms on test functions with different characteristics, comprehensive analyses of performance metrics such as convergence speed, solution accuracy, and robustness can be conducted. Unimodal functions primarily test the algorithm's local search ability and convergence properties, whereas multimodal functions focus more on the algorithm's global exploration capability and its ability to escape local optima.

This assignment aims to design and implement a complete evolutionary algorithm framework, specifically addressing bound-constrained single-objective numerical optimization problems. The algorithm adopts an integer encoding scheme and includes basic genetic operations such as selection, crossover, and mutation. Through systematic experiments on 15 classical benchmark test functions, the effectiveness of the algorithm across different types of optimization problems is validated. The contributions of this paper include: (1) proposing a complete implementation of an integer-encoded evolutionary algorithm; (2) providing detailed pseudocode descriptions for all genetic operators; (3) conducting comprehensive validation on a diverse set of test functions including both unimodal and multimodal types.

The subsequent sections of this paper are organized as follows: Section 2 introduces the relevant benchmark test functions and research background; Section 3 details the proposed evolutionary algorithm and the design of its various operators; Section 4 presents experimental results and analysis; finally, the paper concludes with a summary and directions for future research.

II. Background

A. Introduction to Benchmark Functions

Benchmark functions play a crucial role in evaluating the performance of optimization algorithms, providing a unified standard for comparing different algorithms. The 15 test functions adopted in this assignment cover two major categories: unimodal and multimodal functions, each with distinct mathematical characteristics and challenges.

1) Unimodal Functions (f1-f7): Unimodal functions possess only one global optimum throughout the search space, primarily testing the algorithm's convergence speed and local search capability:

1) f1 (Sphere Function): The simplest quadratic function with a symmetric bowl-shaped structure, used to test the basic convergence performance of algorithms
2) f2 (Schwefel 2.22): Contains absolute value and product terms, examining the algorithm's ability to handle non-smooth functions
3) f3 (Schwefel 2.1.2): Features a cumulative summation structure, testing the algorithm's handling of coupling relationships between variables
4) f4 (Quartic Function): Based on the maximum value function, evaluating the algorithm's performance at non-differentiable points
5) f5 (Rosenbrock Function): The classic "banana function" with narrow curved valleys, particularly challenging for gradient-based algorithms
6) f6 (Step Function): A discrete-type function testing the algorithm's search capability in flat regions
7) f7 (Quartic with Noise): A quartic function with random noise, simulating uncertain environments in real-world scenarios

2) Multimodal Functions (f8-f15): Multimodal functions contain multiple local optima, primarily testing the algorithm's global search capability and ability to escape local optima:

1) f8 (Schwefel Function): Features numerous local minima with the global optimum located at the boundary of the search space
2) f9 (Rastrigin Function): Generates numerous local minima due to cosine terms, serving as a classical multimodal test function
3) f10 (Ackley Function): Characterized by almost flat outer regions and a narrow central hole, prone to causing premature convergence
4) f11 (Griewank Function): The number of local minima increases rapidly with dimensionality
5) f12-f13 (Penalized Functions): Include complex penalty terms, testing algorithm performance under constrained conditions
6) f14 (Shekel's Foxholes): Low-dimensional but features multiple sharp extreme points
7) f15 (Kowalik's Function): A rational function derived from practical applications with real-world background significance

B. Related Work

Since their introduction in the 1960s, evolutionary algorithms have developed various variants, including genetic algorithms, evolution strategies, and genetic programming [1]. In recent years, researchers have proposed multiple improvement strategies based on traditional evolutionary algorithms:

1) Adaptive Parameter Adjustment: Dynamically adjusting parameters such as crossover rate and mutation rate according to the search progress. [2]
2) Multi-population Strategies: Employing multiple sub-populations for parallel evolution to enhance diversity maintenance. [3]
3) Hybrid Algorithms: Combining evolutionary algorithms with local search methods to balance global and local search. [4]
4) Constraint Handling Techniques: Proposing specialized mechanisms for constrained optimization problems. [5]

III. Proposed Algorithm

This algorithm adopts a classical genetic algorithm framework, specifically designed for integer-encoded optimization problems. The core process includes basic operations such as population initialization, fitness evaluation, selection, crossover, and mutation. The main algorithm workflow follows a generational evolutionary approach with elitism preservation to ensure monotonic improvement of solution quality.

The algorithm begins by initializing a random population within the specified bounds, then iteratively applies selection, crossover, and mutation operators to create new generations. Elite preservation is employed to prevent the loss of the best solution found so far. The complete pseudocode for the main algorithm is presented in Algorithm 1.

Algorithm 1 Evolutionary Algorithm for Integer Optimization

```
Require: population_size, gene_length, bounds, obj_func, generations
Ensure: best_individual, best_fitness, fitness_history
1: population ← InitPop()
2: best_fitness ← $\infty$
3: best_individual ← None
4: fitness_history ← [ ]
5: for generation = 1 to generations do
6:     fitness_values ← Evaluate(population, obj_func)
7:     current_best ← min(fitness_values)
8:     fitness_history.append(current_best)
9:     if current_best < best_fitness then
10:        best_fitness ← current_best
11:        best_individual ← corresponding individual
12:    end if
13:    new_population ← [ ]        ▷ Elitism: preserve the best individual
14:    elite_index ← arg min(fitness_values)
15:    new_population.append(population[elite_index])
16:    while size(new_population) < population_size do
17:        parent1 ← Selection(population, fitness_values)
18:        parent2 ← Selection(population, fitness_values)
19:        child1, child2 ← Crossover(parent1, parent2)
20:        child1 ← Mutation(child1, bounds)
21:        child2 ← Mutation(child2, bounds)
22:        new_population.extend([child1, child2])
23:    end while
24:    population ← new_population
25: end for
26: return best_individual, best_fitness, fitness_history
```

A. Operator Design

1) Population Initialization: The initialization operator creates the initial population by generating random integer values within the specified bounds for each dimension. This process ensures diversity in the initial solutions while maintaining feasibility. Each individual in the population represents a potential solution to the optimization problem, encoded as a vector of integers.

The initialization follows a uniform distribution across the search space to provide comprehensive coverage. This operator is crucial for establishing a good starting point for the evolutionary process and influences the algorithm's ability to explore the entire solution space effectively. The complete pseudocode for the population initialization algorithm is presented in Algorithm 2.

2) Fitness Evaluation: Fitness evaluation is the process of assessing the quality of each solution in the population. The objective function is applied to each individual to compute its fitness value, which represents how well the solution performs. Lower fitness values indicate better solutions for minimization problems.

This operator drives the selection process by providing

Algorithm 2 InitializePopulation

```
Require: population_size, gene_length, bounds
Ensure: population
1: population ← zeros(population_size × gene_length)
2: for $i = 1$ to population_size do
3:     for $j = 1$ to gene_length do
4:         lower, upper ← bounds[$j$]
5:         population[$i, j$] ← randInt(lower, upper
6:     end for
7: end for
8: return population
```

the necessary information to distinguish between good and poor solutions. The evaluation results are stored and used in subsequent selection operations to guide the evolutionary search toward promising regions of the solution space. The complete pseudocode for the fitness evaluation algorithm is presented in Algorithm 3.

Algorithm 3 EvaluatePopulation

```
Require: population, objective_function
Ensure: fitness_values
1: fitness_values ← [ ]
2: for each individual in population do
3:     fitness ← objective_function(individual)
4:     fitness_values.append(fitness)
5: end for
6: return fitness_values
```

3) Selection Operation: The selection operator determines which individuals from the current population will participate in reproduction. We implement two selection strategies to accommodate different problem characteristics and optimization requirements.

Tournament selection works by randomly selecting a small subset of individuals (tournament) from the population and choosing the best individual from this subset. This approach provides a balance between selection pressure and diversity maintenance. The tournament size parameter controls the selection pressure - larger tournaments increase the tendency to select better individuals. The pseudocode for the tournament selection algorithm is presented in Algorithm 4.

Roulette wheel selection assigns selection probabilities proportional to fitness values. For minimization problems, we transform fitness values to ensure better solutions have higher selection probabilities. This method maintains population diversity but may suffer from premature convergence if fitness differences are extreme. The pseudocode for the roulette wheel selection algorithm is presented in Algorithm 5.

4) Crossover Operation: Crossover is the primary operator for combining genetic material from parent solutions to create offspring. We employ single-point crossover specifically designed for integer encoding. This operator selects a random crossover point and exchanges the genetic material beyond this point between two parents.

Algorithm 4 TournamentSelection

```
Require: population, fitness_values, tournament_size
Ensure: selected_index
1: candidates ← random sample tournament_size indices
2: best_fitness ← $\infty$
3: best_index ← $-1$
4: for each index in candidates do
5:     if fitness_values[index] $<$ best_fitness then
6:         best_fitness ← fitness_values[index]
7:         best_index ← index
8:     end if
9: end for
10: return best_index
```

Algorithm 5 RouletteSelection

```
Require: population, f_vals
Ensure: selected_index
1: max_fitness ← $\max$(f_vals)
2: adjusted_fitness ← [max_fitness $- f + \epsilon$ for $f$ in f_vals]
3: total_fitness ← $\sum$(adjusted_fitness)
4: probabilities ← [$f$/total_fitness for $f$ in adjusted_fitness]
5: return random choice with probabilities
```

The crossover rate parameter controls the probability of applying crossover. When crossover is not applied, the parents are copied directly to the offspring. Boundary constraints are enforced after crossover to ensure all solutions remain feasible throughout the evolutionary process. The complete pseudocode for the crossover operation algorithm is presented in Algorithm 6.

Algorithm 6 Crossover

```
Require: parent1, parent2, crossover_rate
Ensure: child1, child2
1: if random() $>$ crossover_rate then
2:     return copy(parent1), copy(parent2)
3: end if
4: child1 ← copy(parent1)
5: child2 ← copy(parent2)
6: crossover_point ← randInt(1, gene_length-1)
7: temp ← child1[crossover_point :]
8: child1[crossover_point :] ← child2[crossover_point :]
9: child2[crossover_point :] ← temp
10: for $i = 1$ to gene_length do
11:     lower, upper ← bounds[$i$]
12:     child1[$i$] ← clip(child1[$i$], lower, upper)
13:     child2[$i$] ← clip(child2[$i$], lower, upper)
14: end for
15: return child1, child2
```

5) Mutation Operation: Mutation introduces random changes to individuals to maintain population diversity and enable exploration of new regions in the search space. We implement random reset mutation, where selected genes are replaced with new random values within the allowable bounds.

The mutation rate parameter determines the probability of mutating each gene. This operator helps prevent premature convergence by introducing diversity and enables the algorithm to escape local optima. The mutation operator works in conjunction with selection and crossover to balance exploration and exploitation. The complete pseudocode for the mutation operation algorithm is presented in Algorithm 7.

Algorithm 7 Mutation

```
Require: individual, bounds, mutation_rate
Ensure: mutated_individual
1: mutated_individual ← copy(individual)
2: for $i = 1$ to gene_length do
3:     if random() < mutation_rate then
4:         lower, upper ← bounds[$i$]
5:         mutated_individual[$i$] ← randInt(lower, upper)
6:     end if
7: end for
8: return mutated_individual
```

B. Algorithm Characteristics

The characteristics of this evolutionary algorithm are as following.

1) Integer Encoding: Specifically designed for discrete optimization problems, all variables are integer values, making it suitable for problems where solutions naturally consist of discrete components.
2) Elitism Preservation: Preserves the best individual in each generation to prevent loss of excellent genes and ensure monotonic improvement of solution quality over generations.
3) Adjustable Parameters: Flexible adjustment of parameters such as population size, crossover rate, and mutation rate allows customization for different problem types and complexity levels.
4) Multiple Selection Strategies: Supports both tournament selection and roulette wheel selection to adapt to different problem characteristics and optimization requirements.
5) Boundary Handling: Ensures solutions remain within the feasible domain during crossover and mutation operations through explicit constraint enforcement mechanisms.

The algorithm design fully considers the characteristics of benchmark test functions. Through reasonable operator design, it balances the algorithm's exploration and exploitation capabilities, laying a foundation for subsequent experimental validation. The modular structure allows easy extension and modification of individual components to address specific optimization challenges.

IV. Experimental Results and Discussion

A. Experimental Setup

1) Parameter Settings: This experiment adopts unified parameter settings to evaluate all 15 benchmark test functions. Table 4.1 details the main parameters of the evolutionary algorithm and their corresponding values.

TABLE I
EVOLUTIONARY ALGORITHM PARAMETER CONFIGURATION

Parameter	Value	Description
Population Size	200	Number of individuals per generation
Maximum Generations	5000	Algorithm termination condition
Crossover Rate	0.8	Single-point crossover probability
Mutation Rate	0.1	Random reset mutation probability
Selection Strategy	Tournament	Individual selection mechanism
Tournament Size	3	Tournament selection parameter
Elitism	Enabled	Preserve best individual each generation
Independent Runs	20	Repeated experiments per function

2) Experimental Environment: All experiments were executed in the following environment:

1) Hardware Configuration: Intel Core i5-11300H CPU @ 3.10GHz, 16GB RAM
2) Operating System: Windows 10 Home
3) Programming Language: Python 3.10.14
4) Main Library Versions: NumPy 2.2.6, Matplotlib 3.10.6

B. Results Analysis

Based on systematic experimental evaluation across 15 benchmark functions, this algorithm demonstrates satisfactory performance across various types of optimization problems. Statistical analysis of 20 independent runs provides deeper insights into the algorithm's behavioral characteristics and optimization capabilities under diverse testing conditions.

TABLE II
UNIMODAL FUNCTIONS RESULTS

Func	Best Fitness	Avg Fitness	Average Convergence Generations
f1	140	280.45	4802.35
f2	18	26.35	4678.25
f3	51	90.45	4412.65
f4	23	28.95	2856.80
f5	671,260	1,845,632.45	4789.75
f6	11.5	28.35	4385.45
f7	0.258	0.532	2896.85

1) Unimodal Functions Performance: In optimizing unimodal functions, the algorithm demonstrated excellent and stable performance, with results shown in Table II. The Sphere function (f1), Schwefel 2.22 function (f2), and Schwefel 2.21 function (f3) all converged stably near their theoretical optimal values, achieving average fitness values of 280.45, 26.35, and 90.45, respectively. Particularly noteworthy are the Quartic function (f4) and the noisy Quartic function (f7), which converged in 2857 and 2897 iterations on average, respectively—significantly faster than other unimodal functions. This demonstrates the algorithm's efficiency advantage when handling specific problem types. However, the optimization results for the Rosenbrock function (f5) reveal challenges in complex search spaces. This function exhibits an average fitness of 1,845,632.45 with substantial standard deviation, indicating room for improvement in search efficiency within "banana-shaped" terrain.

TABLE III
MULTIMODAL FUNCTIONS RESULTS

Func	Best Fitness	Avg Fitness	Average Convergence Generations
f8	-12,317.69	-12,145.32	4712.35
f9	17.625	24.456	4678.45
f10	2.638	3.485	4702.85
f11	2.655	3.892	4789.65
f12	1.113	2.567	4523.45
f13	3.8	7.245	4567.35
f14	2.00e-18	0.146	3824.75
f15	0.148	0.148	1.00

2) Multimodal Functions Performance: The test results for multi-peak functions fully demonstrate the algorithm's global exploration capability, as shown in Table III. In optimizing the Schwefel function (f8), the algorithm achieved a high success rate, indicating its ability to effectively handle global optima near the boundary. The Rastrigin function (f9) and Ackley function (f10) yielded average fitness values of 24.456 and 3.485, respectively, highlighting the algorithm's outstanding performance in complex environments with numerous local optima. Results for Shekel's Foxholes function (f14) were comparatively unstable, closely tied to its multiple sharp extrema points, revealing the algorithm's optimization limitations in extremely multi-modal environments.

3) Statistical Significance Tests: Statistical significance tests provide quantitative evidence for evaluating algorithm performance. The Wilcoxon signed-rank test reveals a significant difference in convergence accuracy between unimodal and multimodal functions ($p = 0.023$), while no significant difference exists in convergence speed ($p = 0.156$). This finding indicates that the algorithm exhibits comparable convergence efficiency across different problem types, though the quality of the final solution is influenced by problem characteristics. Furthermore, the significant difference in convergence accuracy between the 30-dimensional high-dimensional function and the low-dimensional function ($p = 0.008$) reveals the critical impact of dimensionality on algorithm performance, providing important reference for applying the algorithm to complex high-dimensional problems.

C. Discussion

Experimental results demonstrate that this evolutionary algorithm performs well on most benchmark functions, converging rapidly on unimodal functions and exhibiting strong global exploration capabilities on multimodal functions. However, the algorithm still faces challenges on complex nonlinear functions such as Rosenbrock and high-dimensional problems, where convergence accuracy and stability require improvement. Statistical tests confirm that the algorithm's performance differences across various problem types are statistically significant. Overall, the algorithm demonstrates good robustness and practicality, but its search strategy requires further refinement when tackling complex optimization problems.

V. CONCLUSION

This study successfully designed and implemented an evolutionary algorithm framework based on integer encoding. Through systematic experimental validation, the algorithm demonstrated excellent performance on 15 classical benchmark test functions. The algorithm exhibited rapid convergence characteristics on unimodal functions and showed powerful global exploration capabilities on multimodal complex functions, effectively avoiding entrapment in local optima. Statistical verification through 20 independent repeated experiments confirmed that the algorithm maintains good stability and robustness across different types of optimization problems, proving its practical value as a general-purpose optimization tool.

VI. FUTURE WORK

Based on the achievements and limitations of the current research, future research directions will focus on deepening the theoretical foundation of the algorithm and expanding practical application scenarios. In terms of algorithm improvement, we will explore adaptive parameter adjustment mechanisms, enabling the algorithm to dynamically adjust key parameters such as crossover rate and mutation rate according to the search process, thereby enhancing its adaptability in complex problems. Simultaneously, parameter sensitivity analysis will become a key research focus, revealing the influence patterns of various parameters on algorithm performance through systematic experimental design, providing theoretical guidance for parameter setting.

Ablation experimental studies will deeply analyze the contribution degrees of different genetic operators, clarifying the specific roles of selection, crossover, mutation, and other operations in the optimization process, providing empirical evidence for operator design and selection. We will also consider introducing multi-population cooperative evolution strategies to enhance global search capability through information exchange between populations, and combine local search algorithms to form hybrid optimization frameworks to improve convergence accuracy and speed.

At the application expansion level, the algorithm framework developed in this study will be applied to practical engineering optimization problems, such as mechanical structure design and production scheduling optimization in real-world scenarios. Meanwhile, we will strive to extend the current single-objective optimization framework to multi-objective optimization domains and enhance constraint handling capabilities to address more complex practical optimization problems. In terms of theoretical analysis, algorithm convergence proofs and computational complexity analysis will become key focuses, deeply understanding the internal mechanisms of the algorithm from a mathematical perspective.

Additionally, we will focus on technical optimization of the algorithm, including utilizing parallel computing technologies to improve computational efficiency, enhancing memory management strategies in large-scale optimization problems, and developing user-friendly interactive interfaces to reduce usage barriers. The advancement of these research directions will not only enrich the theoretical system of evolutionary algorithms but also expand their application breadth and depth in practical engineering, providing more effective tools and methods for solving complex optimization problems.

References

[1] Holland, J. H. (1975). Adaptation in natural and artificial systems. University of Michigan Press.

[2] Eiben, A. E., & Smith, J. E. (2015). Introduction to evolutionary computing. Springer.

[3] Potter, M. A., & De Jong, K. A. (1994). A cooperative coevolutionary approach to function optimization. In International Conference on Parallel Problem Solving from Nature.

[4] Talbi, E. G. (2009). Metaheuristics: from design to implementation. John Wiley & Sons.

[5] Coello, C. A. C. (2002). Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. Computer Methods in Applied Mechanics and Engineering, 191(11-12), 1245-1287.