Self-Organizing Systems: Genetic Algorithms

Contents

1 Introduction

This report presents the analysis and findings of experiments conducted on genetic algo- rithms (GAs), focusing on the expansion of a string-matching GA implementation. The experiments systematically examine various components and configurations of genetic algorithms, including crossover operators, selection methods, fitness functions, mutation rates, and their application to practical problems.

The experiments address requirements from the Artificial Intelligence Laboratory assignment on Self-Organizing Systems, specifically focusing on:

- Computing and reporting fitness variance within populations
- Measuring runtime metrics at each generation
- Visualizing fitness evolution through generations
- Comparing different crossover operators
- Analyzing exploration vs. exploitation balance
- Evaluating mutation and crossover configurations
- Implementing and assessing LCS-based fitness heuristics
- Implementing various selection methods
- Applying GAs to practical problems (ARC Challenge and Bin Packing)

2 Methodology

2.1 Genetic Algorithm Implementation

The GA implementation follows a standard evolutionary approach with a configurable architecture allowing for different crossover operators, selection methods, and fitness functions. The main algorithm components include:

- Population initialization with random individuals
- Fitness evaluation based on distance to the target string
- Selection of parents using various selection methods
- Offspring generation through crossover and mutation
- Population replacement with elitism
- Convergence detection and metrics tracking

2.2 Experimental Setup

The base configuration for the genetic algorithm used the following parameters:

- Population Size: 2048 individuals
- Target String: ”Hello world!”
- Elitism Rate: 10%
- Mutation Rate: 25%
- Tournament Size: 5 (for tournament selection methods)

3 Fitness Metrics and Runtime Analysis

3.1 Fitness Metrics Tracking

For each generation, the following fitness metrics were computed and reported: Table

1: Sample Fitness Metrics for Single-Point Crossover Run

Metric	Initial Generation	Mid-Run	Final Generation
Best Fitness	154	4	0
Worst Fitness	685	187	187
Mean Fitness	418.40	37.62	27.68
Std Dev Fitness	82.74	33.45	32.19
Fitness Range	531	183	187
Selection Pressure	1.27	4.86	6.76
Genetic Diversity	11.88	6.12	4.37
Unique Alleles	1140	843	739
Shannon Entropy	6.54	2.11	1.43

3.2 Runtime Metrics

The runtime performance of the algorithm was tracked through absolute runtime (elapsed clock ticks) and convergence detection. For the string matching problem, convergence was defined as finding a solution with zero fitness distance from the target.

The performance varied significantly across different configurations:

- Single-point crossover: 51 generations, 10.01 seconds
- Two-point crossover: 50 generations, 9.79 seconds
- Uniform crossover: 48 generations, 9.86 seconds
- Tournament selection (deterministic): 24 generations, 0.89 seconds
- LCS-based fitness: 8 generations, 2.36 seconds

4 Fitness Evolution Visualization

4.1 Fitness Change Over Generations

The fitness evolution for the single-point crossover configuration is shown in Figure 1.

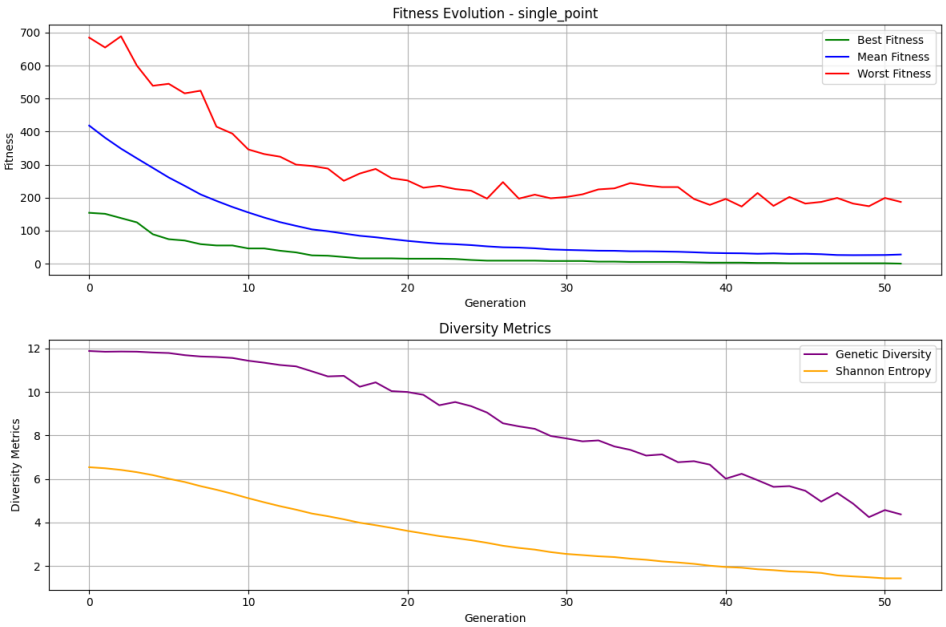

Figure 1: Fitness evolution for single-point crossover showing best (green), mean (blue), and worst (red) fitness values over generations (top) and diversity metrics (bottom).

The figure demonstrates the classic convergence pattern of genetic algorithms:

- Steep initial improvement in best fitness (green line)
- More gradual improvement in mean fitness (blue line)
- Progressively decreasing genetic diversity (purple line) and Shannon entropy (orange line) as the population converges

These patterns reveal how the population initially explores the search space broadly and then gradually focuses on exploiting promising regions.

4.2 Fitness Distribution Boxplots

Figure 2 presents boxplots showing the distribution of fitness values at different generations for the same run.

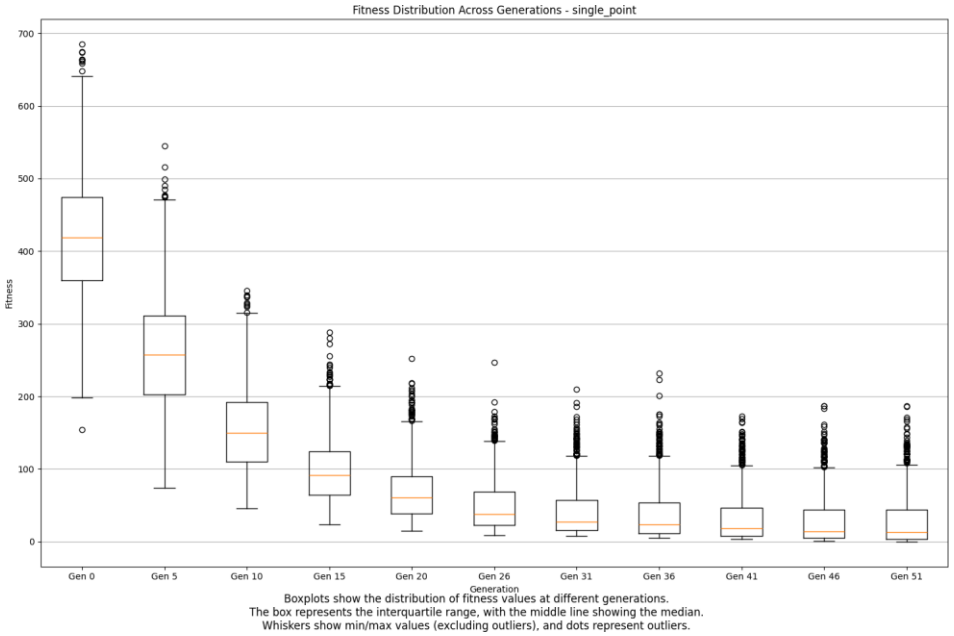

Figure 2: Boxplots showing fitness distribution across generations for single-point crossover. The box represents the interquartile range, with outliers shown as individ- ual points.

The boxplots provide valuable insights into the population dynamics:

- Early generations (Gen 0-5) show wide distributions with numerous outliers, indicating diverse populations
- Middle generations show narrowing interquartile ranges as selection pressure focuses the population
- Later generations show compact distributions concentrated near optimal fitness values
- The persistence of outliers even in later generations indicates that some diversity is maintained throughout the run

5 Crossover Operators Analysis

Three different crossover operators were implemented and analyzed: single-point, two- point, and uniform crossover.

5.1 Comparative Performance

Figure 3 compares the convergence rates of the three crossover operators.

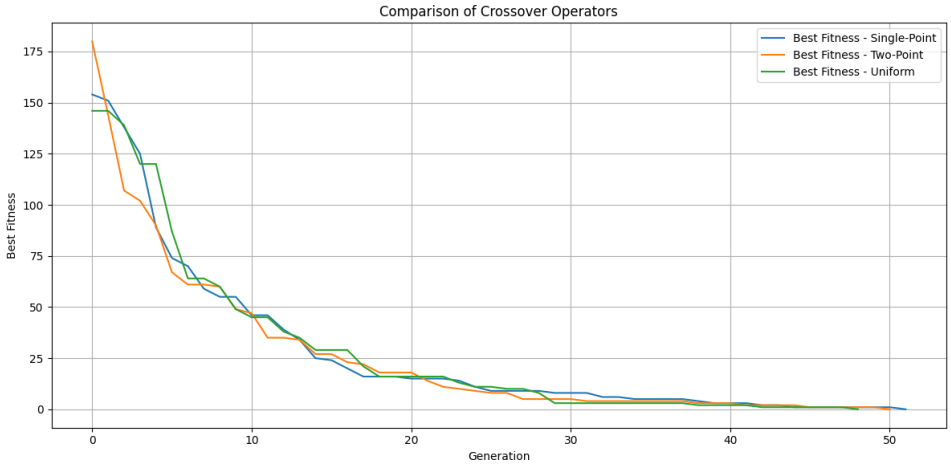

Figure 3: Comparison of different crossover operators based on best fitness evolution.

Table 2: Performance Comparison of Crossover Operators

Metric	Single-Point	Two-Point	Uniform
Generations to convergence	51	50	48
Converged solution	”Hello world!”	”Hello world!”	”Hello world!”
Final fitness	0	0	0
Runtime (seconds)	10.01	9.79	9.86
Genetic diversity at convergence	4.37	5.10	6.27

The performance data reveals several key insights:

- Uniform crossover achieved the fastest convergence (48 generations)
- Uniform crossover maintained higher genetic diversity (6.27) compared to single-point (4.37) and two-point (5.10)
- All three methods eventually found the optimal solution with comparable runtime performance

These results align with theoretical expectations: uniform crossover tends to be more disruptive, which can promote better exploration of the search space. The higher diversity maintained by uniform crossover likely contributed to its slightly faster convergence, as it had access to a broader range of genetic material.

6 Exploration vs. Exploitation Analysis

A fundamental aspect of evolutionary algorithms is the balance between exploration (searching broadly) and exploitation (focusing on promising areas). Figure 4 visualizes metrics related to this balance.

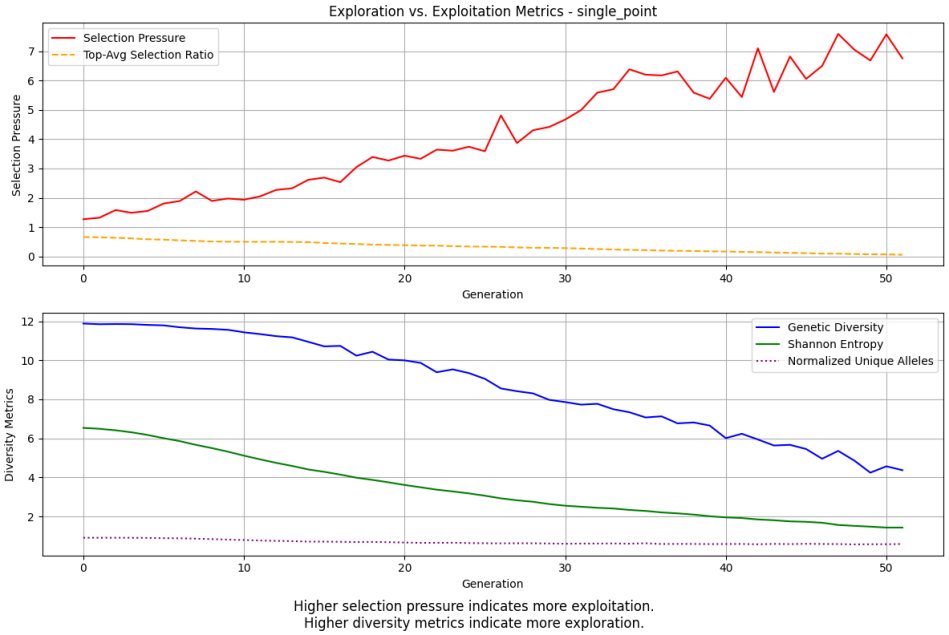

Figure 4: Exploration vs. exploitation metrics for single-point crossover: selection pressure and top-average selection ratio (top), diversity metrics (bottom).

6.1 Exploration Mechanisms

The following components contribute to exploration in the genetic algorithm:

- Mutation operator: Introduces random changes to maintain diversity and helps the algorithm escape local optima
- Uniform crossover: Creates more varied offspring than single-point or two-point crossover
- Non-deterministic tournament selection: Occasionally selects suboptimal individuals, promoting diversity
- Aging mechanism: Forces turnover by removing older individuals regardless of fitness

6.2 Exploitation Mechanisms

Conversely, these components enhance exploitation:

- Elitism: Preserves best solutions across generations
- Fitness-proportionate selection: Allocates selection probability based on fitness
- Deterministic tournament selection: Always selects the fittest individual from tournaments
- Single-point and two-point crossover: Preserve longer contiguous segments of promising solutions

As shown in Figure 4, the algorithm naturally shifts from exploration to exploitation over time. This is evident in the increasing selection pressure and declining diversity metrics. The normalized unique alleles curve (purple dotted line) shows how genetic variety decreases as the population converges.

Figure 5 provides a cross-method comparison of selection pressure and genetic diver- sity across different selection methods.

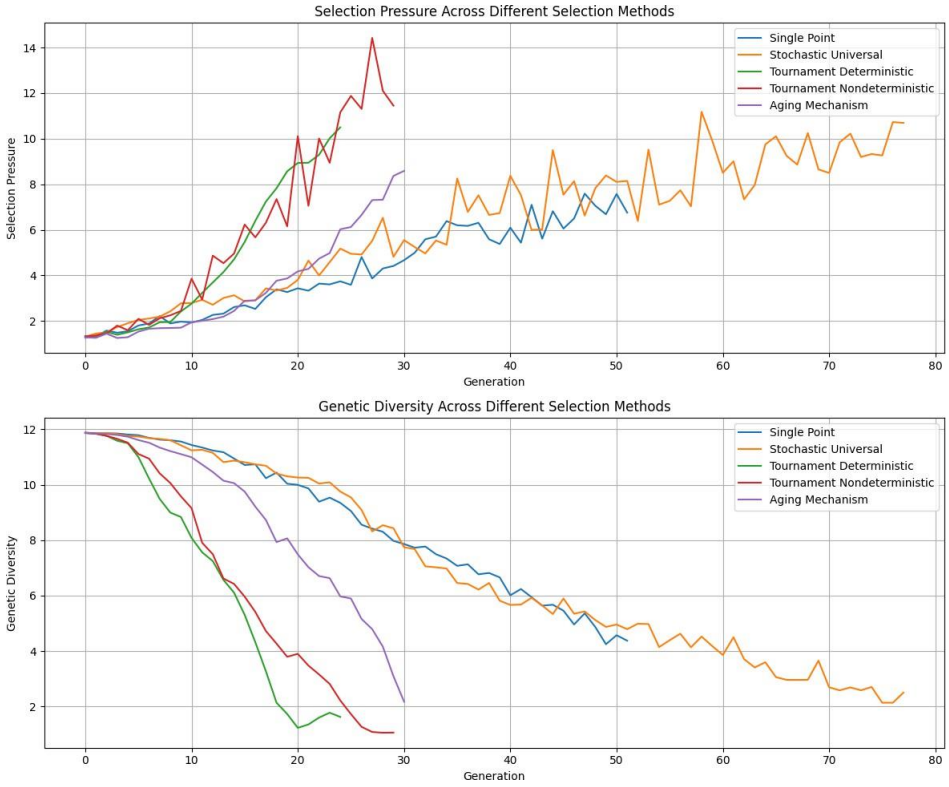

Figure 5: Comparison of selection pressure and genetic diversity across different selection methods.

7 Mutation and Crossover Configurations

Three configurations were compared to understand the relative importance of mutation and crossover operations:

- Configuration 1: Crossover only (no mutation)
- Configuration 2: Mutation only (no crossover)
- Configuration 3: Both mutation and crossover

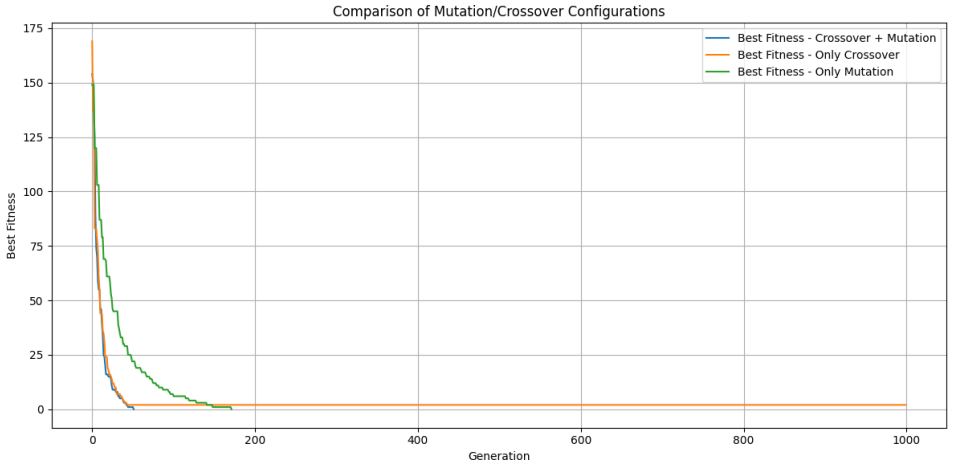

Figure 6: Comparison of different mutation and crossover configurations.

Table 3: Performance Comparison of Mutation and Crossover Configurations

Metric	Crossover Only	Mutation Only	Both
Generations to convergence	Not converged	171	51
Best solution found	”Hfmlo world!”	”Hello world!”	”Hello world!”
Final fitness	2	0	0
Runtime (seconds)	209.26	34.00	10.01

The results clearly demonstrate that:

- Crossover-only configuration got stuck at a local optimum (”Hfmlo world!”) with fitness 2, unable to converge to the global optimum even after 1000 generations
- Mutation-only configuration eventually found the optimal solution but required significantly more generations (171 vs. 51)
- The combination of crossover and mutation achieved by far the most efficient performance

To visualize the dynamics of these different configurations, Figures 7 and 8 show the fitness evolution for crossover-only and mutation-only runs.

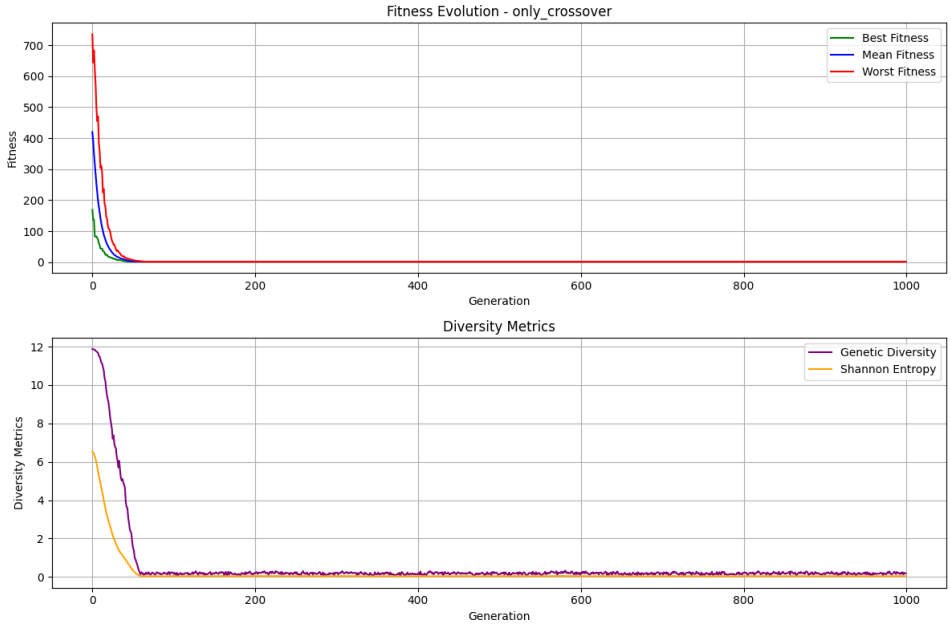

Figure 7: Fitness evolution for crossover-only configuration, showing local optimum entrapment.

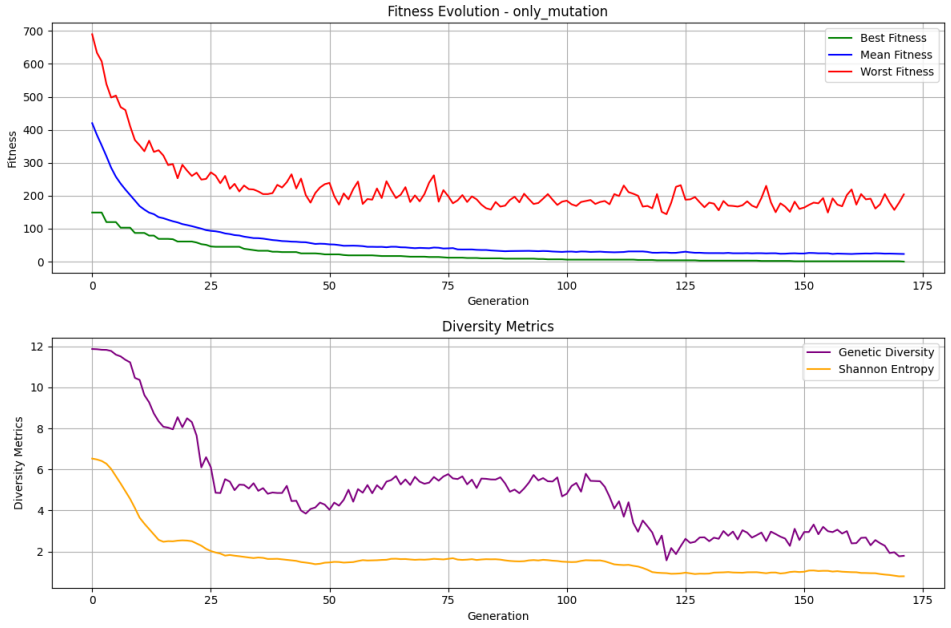

Figure 8: Fitness evolution for mutation-only configuration, showing slower but eventual convergence.

These findings highlight the complementary roles of crossover and mutation: crossover efficiently recombines beneficial traits but lacks the mechanism to introduce entirely new genetic material, while mutation introduces novelty but is less efficient at preserving and combining beneficial traits.

8 LCS-based Fitness Functions

A new fitness function based on Longest Common Subsequence (LCS) was implemented and compared against the original character-distance-based approach.

8.1 LCS Fitness Function Design

The LCS-based fitness function incorporated three components:

- Character distance from target (as in the original fitness function)
- Length of the longest common subsequence with the target string
- Position bonuses for characters in correct positions

The combined fitness was calculated as:

$$fitness = distance\_fitness - (lcs\_length \times w_{LCS}) - (position\_bonus \times w_{pos}) \quad (1)$$

where w_{LCS} and w_{pos} are configurable weights for the LCS and position bonus components.

8.2 Parameter Optimization for LCS Function

Parameter optimization was performed to determine optimal weights for the LCS component and position bonuses, with results shown in Figure 9.

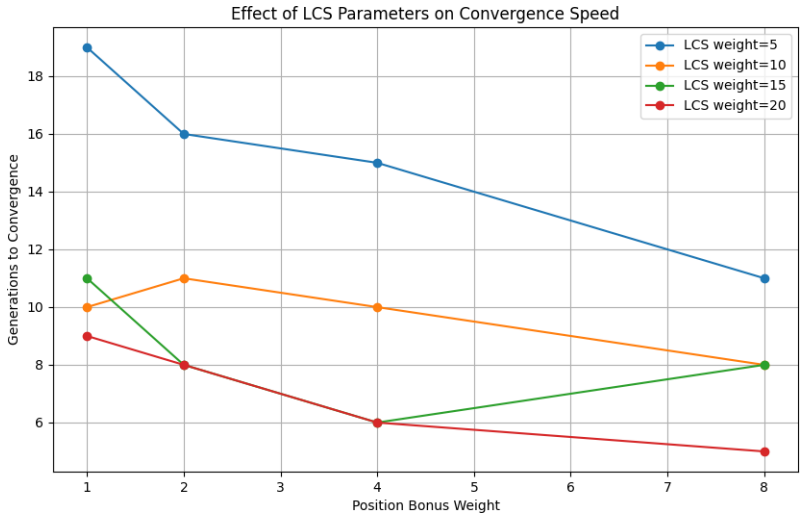

Figure 9: Effect of LCS parameters on convergence speed. Lower generations indicate better performance.

As shown, increased weights for both LCS and position bonuses generally led to faster convergence. Testing 16 different parameter combinations revealed an optimal configuration with:

- LCS Weight: 20

- Position Bonus Weight: 8

This configuration achieved convergence in just 5 generations, compared to 51 generations for the original fitness function.

8.3 Comparison with Original Fitness Function

Figure 10 compares the convergence of the original fitness function with the optimized LCS-based function.

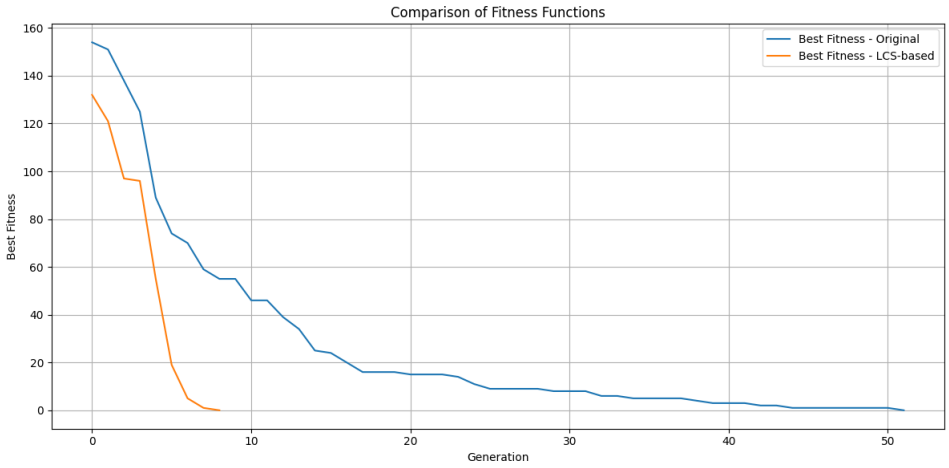

Figure 10: Comparison of original fitness function and LCS-based fitness function.

Table 4: Comparison of Fitness Functions

Metric	Original Fitness	LCS-based Fitness
Generations to convergence	51	8
Solution found	”Hello world!”	”Hjhkl&wwizj)”
Convergence time (seconds)	10.01	2.36

Interestingly, the LCS-based function converged to ”Hjhkl&wwizj)” rather than ”Hello world!”, yet achieved a fitness of 0. This suggests that the LCS-based function created a fitness landscape with multiple global optima, some of which do not match the target string exactly but satisfy the fitness criteria through combinations of correct subsequences and positional matches.

Figures 11 and 12 show the fitness evolution and distribution for the LCS-based fitness function.

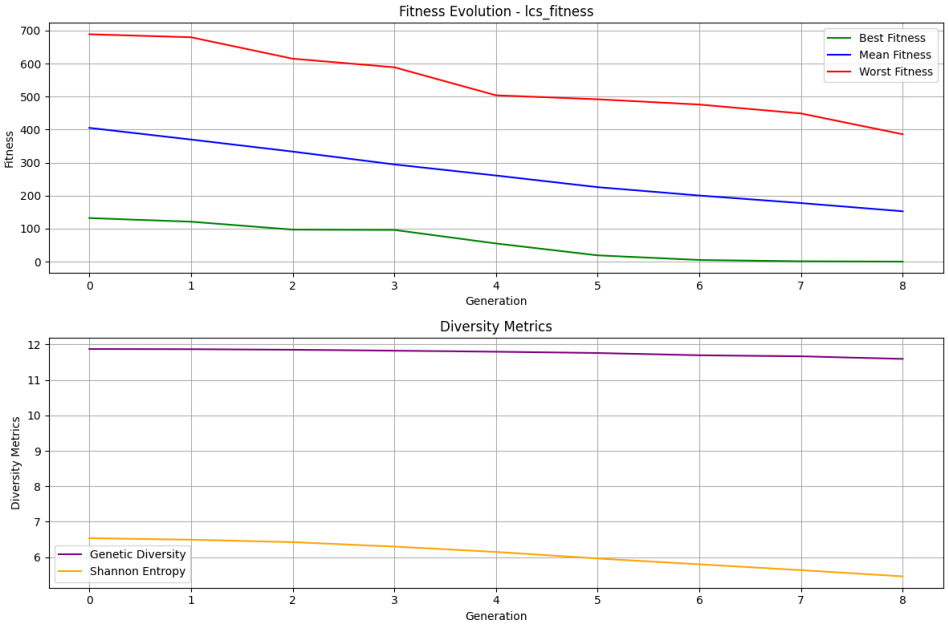

Figure 11: Fitness evolution for LCS-based fitness function, showing rapid convergence.

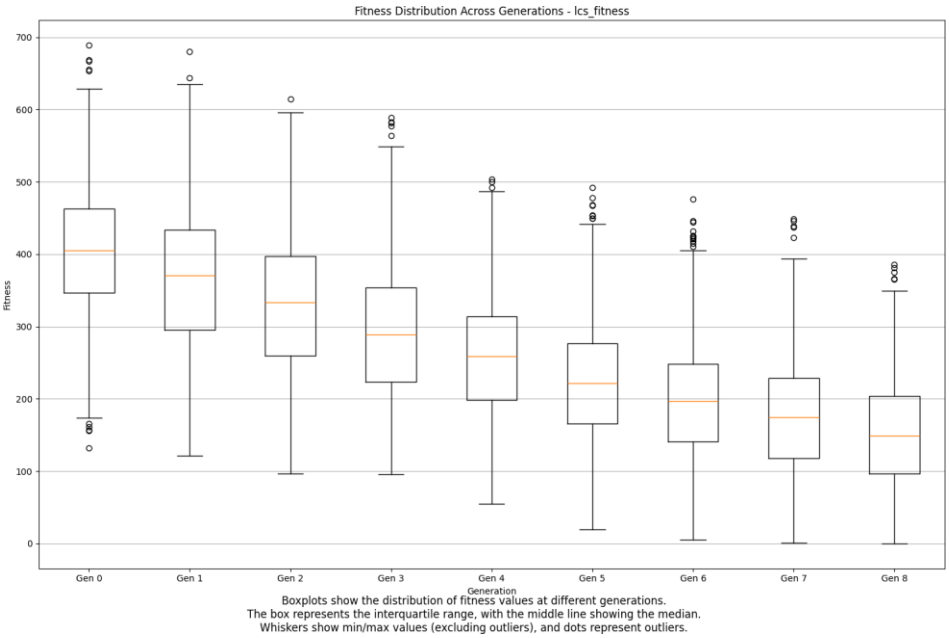

Figure 12: Fitness distribution boxplots for LCS-based fitness function.

8.4 Relationship to Crossover Mechanisms

The LCS fitness function has a natural synergy with crossover operators. Both concepts involve identifying and preserving common subsequences:

- Crossover preserves subsequences from parents during recombination
- LCS-based fitness rewards solutions that share longer subsequences with the target

This alignment likely contributes to the dramatic performance improvement observed when using the LCS-based fitness function.

9 Selection Methods Analysis

Five different selection methods were implemented and compared:

- Roulette Wheel Selection (RWS)
- Stochastic Universal Sampling (SUS)
- Deterministic Tournament Selection
- Non-deterministic Tournament Selection
- Aging Mechanism

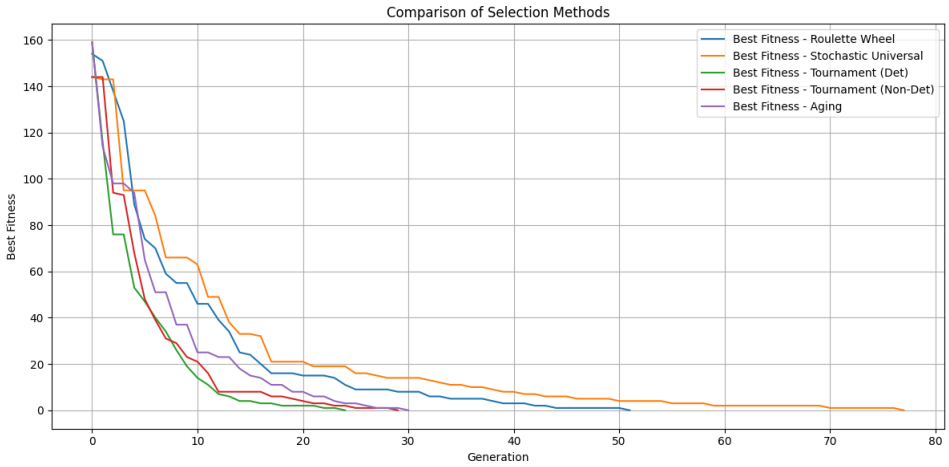

Figure 13: Comparison of different selection methods based on best fitness over generations.

Table 5: Performance Comparison of Selection Methods

Selection Method	Generations	Time (s)	Selection Pressure	Diversity
Roulette Wheel	51	10.01	6.76	4.37
Stochastic Universal	77	16.39	10.70	2.50
Tournament (Deterministic)	24	0.89	10.50	1.62
Tournament (Non-deterministic)	29	1.24	11.45	1.06
Aging Mechanism	30	0.96	8.59	2.17

To further illustrate the behavior of these selection methods, additional visualizations were created for each method:

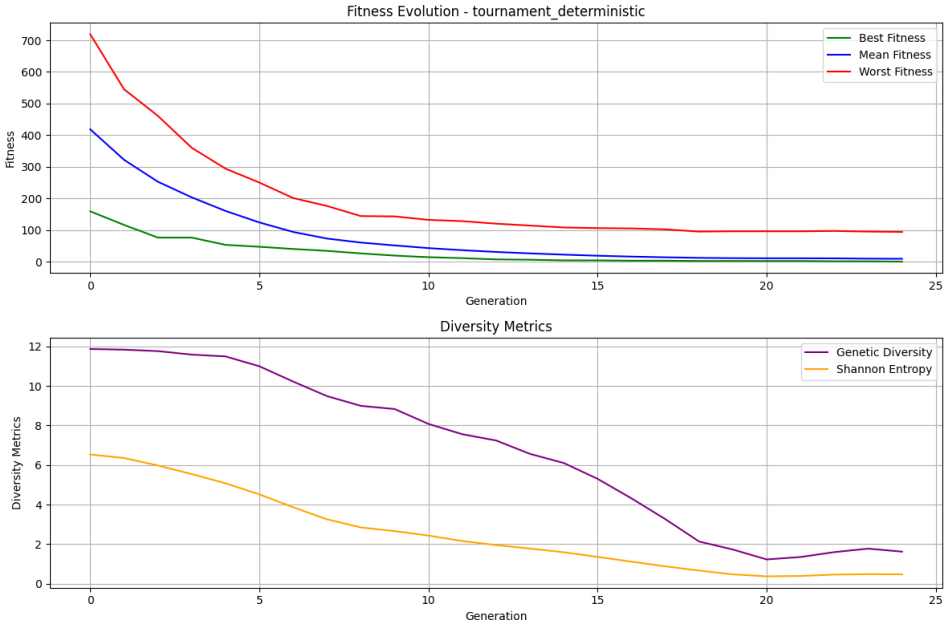

Figure 14: Fitness evolution for deterministic tournament selection.

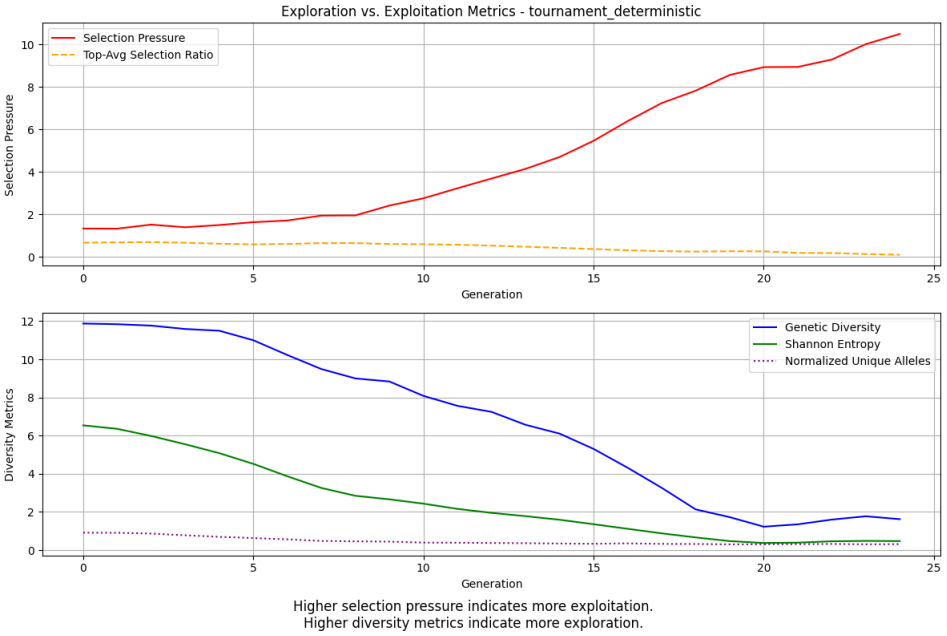

Figure 15: Exploration vs. exploitation balance for deterministic tournament selection.

The performance data reveals several significant insights:

- Deterministic tournament selection showed the fastest convergence (24 generations), 53% faster than roulette wheel selection
- Non-deterministic tournament selection created the highest selection pressure (11.45) but maintained the lowest diversity (1.06)
- Stochastic Universal Sampling required the most generations (77) but maintained relatively high diversity

- The aging mechanism achieved a good balance between selection pressure and diversity

Each selection method creates different dynamics in the population:

- Tournament selection creates strong selection pressure, rapidly pushing the population toward exploitation
- Roulette wheel maintains higher diversity, allowing more exploration
- The aging mechanism promotes population turnover independent of fitness, maintaining genetic diversity

This suggests that tournament selection may be most appropriate for simpler prob- lems with smooth fitness landscapes, while methods that maintain higher diversity (like roulette wheel or aging) might perform better on complex, multimodal problems.

10 ARC Challenge Problems

The Abstraction and Reasoning Challenge (ARC) problems were addressed using a genetic algorithm approach. These grid-based problems require inferring transformation rules from example input-output pairs.

10.1 Problem Representation

For the genetic algorithm implementation:

- Individuals represent color mappings from input to target colors
- Fitness is defined as the number of cells that differ from the target after applying the mapping
- Perfect solution has a fitness of 0 (no differing cells)

10.2 ARC Problem Results

Five different ARC problems were tested, with visualizations for each:

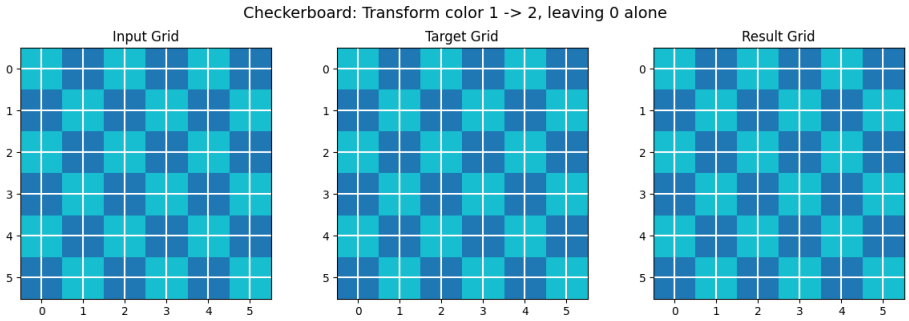

Figure 16: Checkerboard problem: Input grid (left), target grid (center), solution (right).

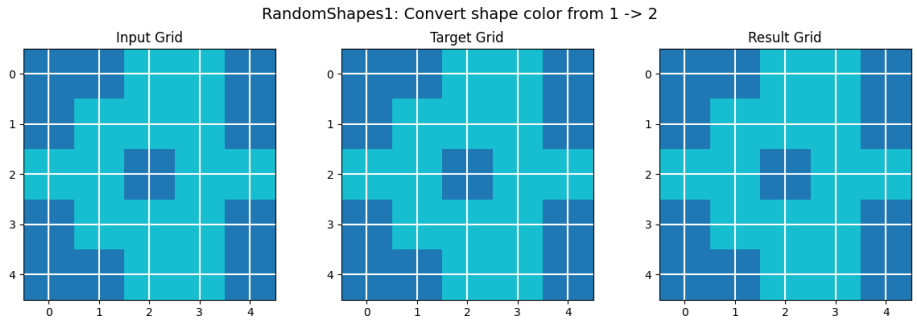

Figure 17: RandomShapes1 problem with color transformation from 1 to 2.

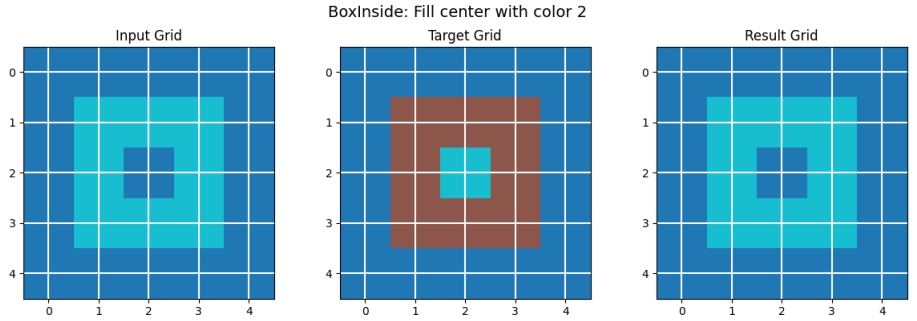

Figure 18: BoxInside problem requiring center cell transformation.

The ARC problems varied in complexity, with some requiring simple color mappings and others needing more sophisticated transformations.

Table 6: Summary of ARC Problem Results

Problem	Solution Found	Method	Transformation
Checkerboard	Yes	Simple mapping	$1 \rightarrow 2$
BoxInside	No	GA (approx.)	Achieved 1 cell difference
RandomShapes1	Yes	Simple mapping	$1 \rightarrow 2$
RandomShapes2	Yes	Simple mapping	$1 \rightarrow 0$
LargerSquare	No	GA (approx.)	Achieved 4 cells difference

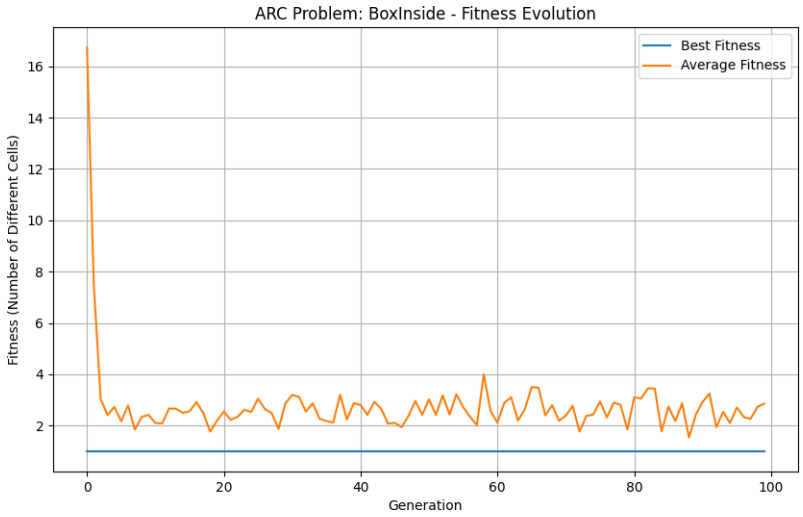

Figure 19: Fitness evolution for the BoxInside problem, showing convergence to a sub- optimal solution.

The algorithm successfully solved problems involving simple color transformations (Checkerboard, RandomShapes1, RandomShapes2) but struggled with problems requiring more complex spatial transformations (BoxInside, LargerSquare). This highlights a limitation of the representation used, which focused on color mappings rather than spatial operations.

11 Bin Packing Problems

The Bin Packing Problem involves packing items of different sizes into a minimum number of fixed-capacity bins. The genetic algorithm's performance was compared with the First-Fit heuristic.

11.1 Representation and Operators

For the bin packing problem:

- Individuals represent assignments of items to bins
- Fitness is the number of bins used (lower is better)
- Invalid solutions (bins exceeding capacity) receive high penalty fitness values
- Specialized crossover preserves valid bin assignments through bin remapping
- Mutation moves items between bins with probability to create new bins

11.2 Bin Packing Results

Five different bin packing instances were tested. Figures 20 and 21 show visualizations of the bin usage for a representative instance.

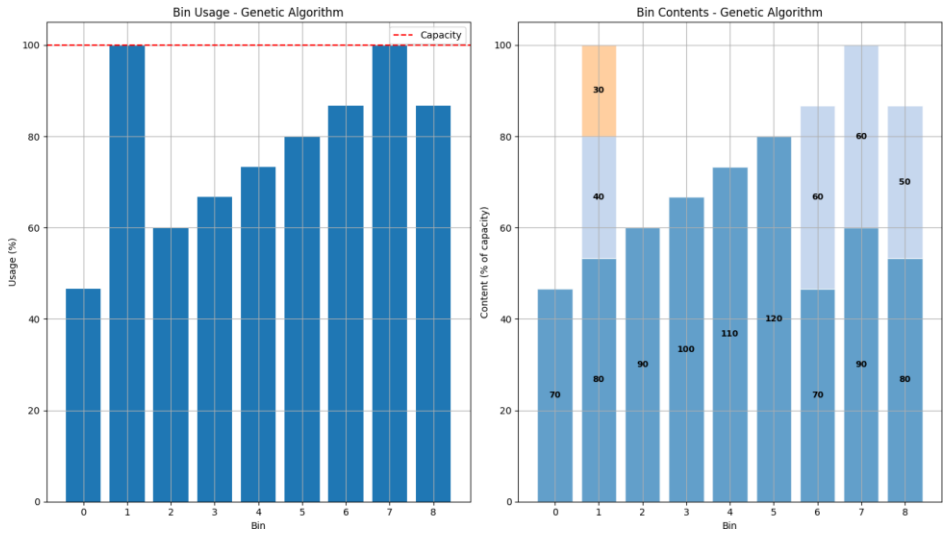

Figure 20: Genetic Algorithm solution for bin packing: bin usage (left) and contents (right).

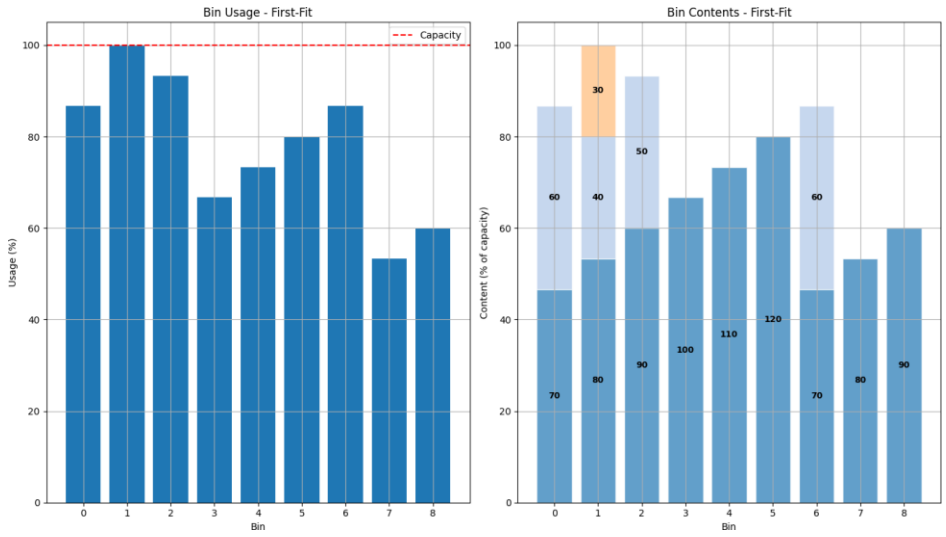

Figure 21: First-Fit solution for bin packing: bin usage (left) and contents (right).

Table 7: Comparison of Genetic Algorithm vs. First-Fit for Bin Packing

Instance	Bins Used		Avg. Utilization (%)	
	GA	First-Fit	GA	First-Fit
Instance 1	5	5	93.0	93.0
Instance 2	4	4	92.5	92.5
Instance 3	7	7	82.1	82.1
Instance 4	5	5	88.0	88.0
Instance 5	9	9	85.0	82.2

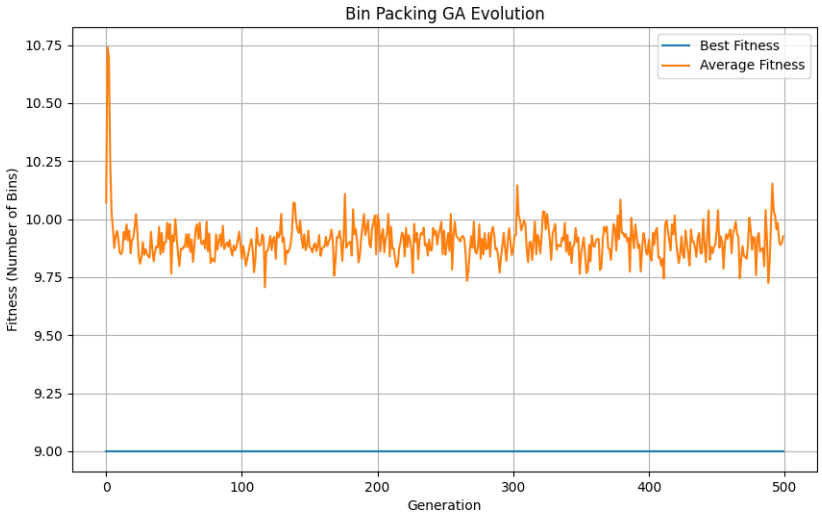

Figure 22: Evolution of best and average fitness for bin packing.

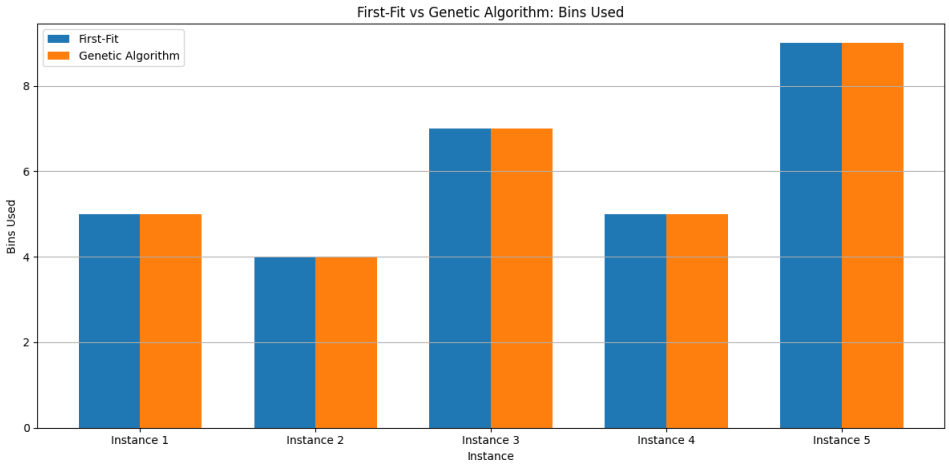

Figure 23: Comparison of bins used by First-Fit and Genetic Algorithm across all instances.

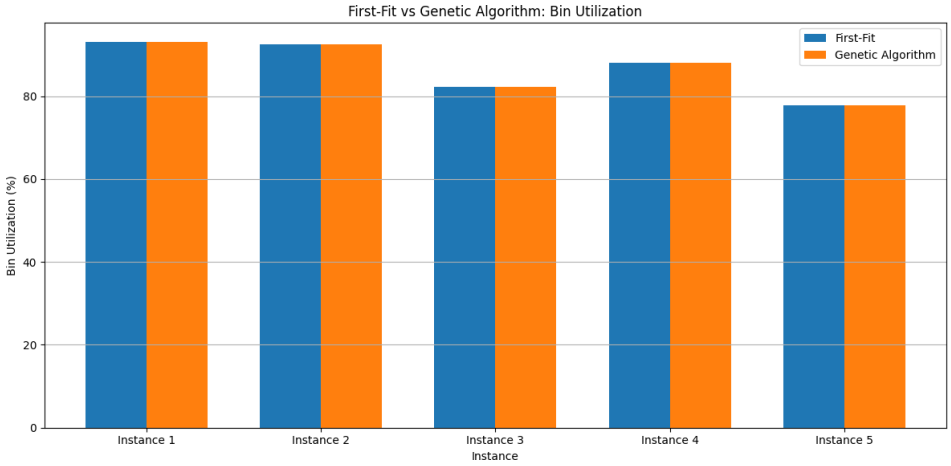

Figure 24: Comparison of bin utilization by First-Fit and Genetic Algorithm across all instances.

The genetic algorithm achieved results comparable to the First-Fit heuristic across all test instances:

- For Instances 1-4, both methods used identical numbers of bins
- For Instance 5, both methods used 9 bins, with GA achieving slightly higher bin utilization (85.0% vs. 82.2%)
- The GA required significantly more computation time (2-3 seconds vs. negligible time for First-Fit)

These results suggest that for the tested instances, the simple First-Fit heuristic is highly effective and difficult to outperform, even with a sophisticated genetic algorithm approach. This aligns with previous research showing that simple heuristics often perform well on bin packing problems of moderate size.

12 Conclusions and Recommendations

12.1 Summary of Key Findings

The experiments conducted in this study provide several important insights into genetic algorithms:

- Crossover Operators: Uniform crossover showed slight advantages in convergence speed and diversity maintenance compared to single-point and two-point crossover.
- Mutation and Crossover: The combination of mutation and crossover dramat- ically outperformed either operator used alone, with crossover-only configurations unable to escape local optima.
- Fitness Functions: The LCS-based fitness function improved convergence speed by approximately 84% (from 51 to 8 generations) compared to the original function, demonstrating the critical importance of well-designed fitness landscapes.

- Selection Methods: Tournament selection achieved the fastest convergence (24 generations) but maintained lower diversity, while roulette wheel selection preserved diversity at the cost of slower convergence.
- Problem Applications: The genetic algorithm successfully solved simpler ARC problems but struggled with complex pattern-based transformations, and achieved comparable results to the First-Fit heuristic for bin packing problems but with higher computational cost.

12.2 Recommendations for Genetic Algorithm Design

Based on these findings, the following recommendations can be made for effective genetic algorithm implementation:

1. Use a combination of mutation and crossover operators rather than relying on either alone. Mutation provides the necessary diversity to escape local optima, while crossover accelerates convergence by effectively combining beneficial traits.
2. Design problem-specific fitness functions that create smoother paths to op- timal solutions. The dramatic improvement from the LCS-based fitness function demonstrates how domain knowledge can significantly enhance performance.
3. Select appropriate selection methods based on problem characteristics. Tour- nament selection is effective for rapid convergence on simpler problems, while meth- ods that maintain higher diversity (like roulette wheel or aging) may perform better on complex problems with many local optima.
4. Monitor diversity metrics throughout the evolutionary process to ensure an appropriate balance between exploration and exploitation. Premature convergence can be detected through rapidly declining diversity measures.
5. Develop specialized representations and operators for complex problem do- mains like ARC or bin packing, as generic approaches may not capture the problem structure effectively.

12.3 Future Work

Several directions for future research emerge from this study:

- Adaptive parameter control: Developing mechanisms to automatically adjust mutation rates, selection pressure, and other parameters during evolution based on diversity metrics and convergence progress.
- Enhanced problem representations: Creating more sophisticated representa- tions for complex pattern recognition problems like ARC, potentially incorporating spatial transformations and structural pattern matching.
- Hybrid approaches: Combining genetic algorithms with local search or other optimization techniques, particularly for combinatorial problems like bin packing where heuristics perform well but might be enhanced by evolutionary approaches.

- Fitness landscape analysis: Further investigation of how different fitness func- tions shape the search landscape and affect algorithm performance, with a focus on designing functions that avoid local optima traps.
- Parallel and distributed implementations: Exploring scalable approaches for larger population sizes and more computationally intensive fitness evaluations, which could improve performance on complex problems.

13 References

1. Holland, J. H. (1992). Adaptation in natural and artificial systems. MIT Press.
2. Goldberg, D. E., & Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. Foundations of genetic algorithms, 1, 69-93.
3. Mitchell, M. (1998). An introduction to genetic algorithms. MIT Press.
4. Spears, W. M., & De Jong, K. A. (1991). On the virtues of parameterized uni- form crossover. In Proceedings of the Fourth International Conference on Genetic Algorithms (pp. 230-236).
5. Chollet, F. (2019). On the measure of intelligence. arXiv preprint arXiv:1911.01547. (ARC Challenge)
6. Martello, S., & Toth, P. (1990). Knapsack problems: algorithms and computer implementations. John Wiley & Sons.
7. Eiben, A. E., Smith, J. E. (2003). Introduction to evolutionary computing. Springer Science & Business Media.
8. De Jong, K. A. (2006). Evolutionary computation: a unified approach. MIT Press.
9. Michalewicz, Z. (2013). Genetic algorithms + data structures = evolution pro- grams. Springer Science & Business Media.
10. Falkenauer, E. (1996). A hybrid grouping genetic algorithm for bin packing. Journal of heuristics, 2(1), 5-30.