

## Advanced Genetic Algorithms Lab Report

Talal Zoabi - 213603608 University of Haifa, Department of Computer Science

Course: Artificial Intelligence Lab (203.3630)

Instructor: Shay Bushinsky

Submission Date: June 21, 2024

---

### 1. Introduction

This lab report explores the implementation and analysis of advanced genetic algorithms (GAs) and evolutionary techniques applied to optimization problems. Specifically, the report focuses on the integration of multiple mutation strategies, niching methods, and speciation techniques to enhance genetic search performance. The primary goal is to develop a robust genetic engine capable of addressing various optimization challenges, including string matching, Sudoku, and bin-packing problems.

### 2. Objectives

The main objectives of this lab are as follows: 1. **Implement Five Mutation Strategies:** Integrate and compare different mutation techniques in the genetic engine. 2. **Performance Evaluation:** Assess the performance of the implemented algorithms on string matching, Sudoku, and bin-packing problems. 3. **Incorporate Niching Methods:** Implement and evaluate niching techniques such as crowding and fitness sharing. 4. **Integrate Speciation Techniques:** Implement clustering-based speciation methods and assess their impact on genetic diversity and solution quality. 5. **Analyze the Baldwin Effect:** Reproduce the Baldwin effect through simulation and analyze the evolutionary impact of learning mechanisms.

### 3. Methodology

The methodology section details the steps taken to achieve the objectives, including the design and implementation of various genetic algorithm components, as well as the experimental setup for evaluating their performance.

**3.1 Implementing Five Mutation Strategies** The first requirement of the assignment was to implement five different mutation strategies within our genetic engine. Each strategy was designed to address various aspects of the evolutionary process, ensuring a diverse and robust search through the solution space.

**Code Snippet: Mutation Strategies Implementation**

*# Mutation strategies implemented in the genetic engine*

```
def basic_mutation(individual, mutation_prob):
    for i in range(len(individual.genes)):
        if random.random() < mutation_prob:
            individual.genes[i] = mutate_gene(individual.genes[i])
    return individual

def non_uniform_mutation(individual, initial_prob, decay_rate, generation):
    mutation_prob = initial_prob * (1 - decay_rate) ** generation
    for i in range(len(individual.genes)):
        if random.random() < mutation_prob:
            individual.genes[i] = mutate_gene(individual.genes[i])
    return individual

def adaptive_mutation(individual, max_mutation_prob, population_fitness):
    avg_fitness = sum(ind.fitness for ind in population_fitness) / len(population_fitness)
    mutation_prob = max_mutation_prob * (1 - (individual.fitness / avg_fitness))
    for i in range(len(individual.genes)):
        if random.random() < mutation_prob:
            individual.genes[i] = mutate_gene(individual.genes[i])
    return individual

def triggered_hyper_mutation(individual, base_mutation_prob, threshold, best_fitness, current_fitness):
    if current_fitness < best_fitness - threshold:
        mutation_prob = base_mutation_prob * 2 # Increase mutation rate
    else:
        mutation_prob = base_mutation_prob # Base mutation rate
    for i in range(len(individual.genes)):
        if random.random() < mutation_prob:
            individual.genes[i] = mutate_gene(individual.genes[i])
    return individual

def self_adaptive_mutation(individual, max_mutation_prob):
    mutation_prob = max_mutation_prob * (1 - individual.relative_fitness)
    for i in range(len(individual.genes)):
        if random.random() < mutation_prob:
            individual.genes[i] = mutate_gene(individual.genes[i])
    return individual
```

**Explanation:** - **Basic Mutation:** Applies a fixed mutation probability to each gene. - **Non-Uniform Mutation:** The mutation probability decreases over generations. - **Adaptive Mutation:** Mutation probability is adjusted based on the individual's fitness relative to the population. - **Triggered Hyper Mutation (THM):** Increases mutation rates when progress stagnates. - **Self-Adaptive Mutation:** Mutation rates vary based on the individual's relative

fitness.

### 3.2 Performance Evaluation on Optimization Problems

**3.2.1 String Matching** String matching involves evolving a population of strings to match a target string. The fitness function measures how closely a candidate string matches the target string.

#### Code Snippet: String Matching Configuration

```
configs = [  
    {  
        'name': "config_niching_fitness_sharing",  
        'problem_type': 'string_matching',  
        'target_string': 'HELLO WORLD',  
        'chosen_fitness': '1', # MatchFitness  
        'parent_selection_method': '1', # Tournament Selection  
        'tournament_size': 3,  
        'crossover_method': '1', # Single Point Crossover  
        'mutation_operator_type': '1', # Swap Mutation  
        'mutation_strategy': '1', # Basic Mutation  
        'mutation_prob': 0.01,  
        'survivor_selection_method': '1', # Elitism Selection  
        'niching_method': '1', # Fitness Sharing  
        'sharing_radius': 0.1,  
        'population_size': 100,  
        'num_generations': 100,  
        'convergence_threshold': 0.01,  
        'convergence_window': 10,  
        'tournament_prob_best': 1.0,  
        'plot_results': False,  
        'terminal_log': False,  
        'scaling_strategy': '1',  
        'k': 2.0  
    },  
    ...  
]
```

**Explanation:** - This configuration sets up the string matching problem with fitness sharing to maintain genetic diversity. - **Experimental Setup:** - **Population Size:** 100 - **Number of Generations:** 100 - **Mutation Strategy:** Basic Mutation

**Results and Analysis:** - Fitness sharing maintained a diverse population and prevented premature convergence.

#### Graph: Convergence of String Matching (Example)

```
import matplotlib.pyplot as plt
```

```

generations = list(range(100))

fitness_scores = [avg_fitness_per_generation] # Placeholder for actual fitness data

plt.plot(generations, fitness_scores)
plt.xlabel('Generation')
plt.ylabel('Average Fitness')
plt.title('String Matching: Average Fitness over Generations')
plt.show()

```

**3.2.2 Sudoku** Sudoku is a constraint satisfaction problem where the goal is to fill a 9x9 grid such that each row, column, and 3x3 subgrid contains all digits from 1 to 9 without repetition.

#### Code Snippet: Sudoku Configuration

```

configs.append({
    'name': "config_crowding_sudoku",
    'problem_type': 'sudoku',
    'chosen_fitness': '2', # SudokuFitness
    'parent_selection_method': '2', # Roulette Wheel Selection
    'crossover_method': '2', # Two Point Crossover
    'mutation_operator_type': '2', # Inversion Mutation
    'mutation_strategy': '2', # Non-Uniform Mutation
    'mutation_initial_prob': 0.05,
    'mutation_decay_rate': 0.01,
    'survivor_selection_method': '2', # Crowding
    'crowding_factor': 2.0,
    'population_size': 100,
    'num_generations': 200,
    'convergence_threshold': 0.01,
    'convergence_window': 10,
    'tournament_prob_best': 1.0,
    'plot_results': False,
    'terminal_log': False,
    'scaling_strategy': '1',
    'k': 2.0
})

```

**Explanation:** - This configuration sets up the Sudoku problem with crowding to maintain genetic diversity. - **Experimental Setup:** - **Population Size:** 100 - **Number of Generations:** 200 - **Mutation Strategy:** Non-Uniform Mutation

**Results and Analysis:** - Crowding preserved genetic diversity and improved

overall solution quality.

### Graph: Convergence of Sudoku (Example)

```
generations = list(range(200))
fitness_scores = [avg_fitness_per_generation] # Placeholder for actual fitness data

plt.plot(generations, fitness_scores)
plt.xlabel('Generation')
plt.ylabel('Average Fitness')
plt.title('Sudoku: Average Fitness over Generations')
plt.show()
```

**3.2.3 Bin-Packing** Bin-packing involves distributing items of varying sizes into bins with fixed capacities in a way that minimizes the number of bins used.

### Code Snippet: Bin-Packing Configuration

```
configs.append({
    'name': "config_speciation_kmeans_binpacking",
    'problem_type': 'bin_packing',
    'chosen_fitness': '3', # BinPackingFitness
    'bin_capacity': 10,
    'parent_selection_method': '3', # RWS Ranking Selection
    'crossover_method': '3', # Uniform Crossover
    'mutation_operator_type': '3', # Bit Flip Mutation
    'mutation_strategy': '3', # Adaptive Mutation
    'max_mutation_prob': 0.05,
    'survivor_selection_method': '3', # Speciation
    'speciation_method': '1', # K-Means Clustering
    'num_species': 5,
    'population_size': 100,
    'num_generations': 150,
    'convergence_threshold': 0.01,
    'convergence_window': 10,
    'tournament_prob_best': 1.0,
    'plot_results': False,
    'terminal_log': False,
    'scaling_strategy': '1',
    'k': 2.0
})
```

**Explanation:** - This configuration sets up the bin-packing problem with K-Means clustering for speciation. - **Experimental Setup:** - **Population Size:** 100 - **Number of Generations:** 150 - **Mutation Strategy:** Adaptive Mutation

**Results and Analysis:** - Speciation with K-Means clustering maintained genetic diversity and improved overall performance.

### Graph: Convergence of Bin-Packing (Example)

```
generations = list(range(150))
fitness_scores = [avg_fitness_per_generation] # Placeholder for actual fitness data

plt.plot(generations, fitness_scores)
plt.xlabel('Generation')
plt.ylabel('Average Fitness')
plt.title('Bin-Packing: Average Fitness over Generations')
plt.show()
```

## 3.3 Incorporating Niching Methods

**3.3.1 Fitness Sharing** Fitness sharing reduces the fitness of individuals that are too similar to others in the population. This encourages the population to spread out and explore different regions of the solution space.

### Code Snippet: Fitness Sharing

```
class FitnessSharing:
    def __init__(self, sharing_radius):
        self.sharing_radius = sharing_radius

    def apply(self, population):
        for i, ind1 in enumerate(population):
            shared_fitness = ind1.fitness
            for j, ind2 in enumerate(population):
                if i != j:
                    distance = self._calculate_distance(ind1.genes, ind2.genes)
                    if distance < self.sharing_radius:
                        shared_fitness += self._sharing_function(distance)
            ind1.fitness = shared_fitness

    def _calculate_distance(self, genes1, genes2):
        return sum(1 for g1, g2 in zip(genes1, genes2) if g1 != g2) / len(genes1)

    def _sharing_function(self, distance):
        return 1 - (distance / self.sharing_radius) if distance < self.sharing_radius else 0

# Example Usage
fitness_sharing = FitnessSharing(sharing_radius=0.1)
fitness_sharing.apply(population)
```

**Explanation:** - The FitnessSharing class calculates the shared fitness for each individual by penalizing those that are too similar to others within a specified radius. - This method promotes diversity by encouraging individuals to explore different regions of the solution space.

**3.3.2 Crowding** Crowding maintains diversity by ensuring that offspring replace similar individuals from the previous generation. This prevents the rapid takeover of the population by a few individuals and encourages the survival of diverse solutions.

#### Code Snippet: Crowding

```
class Crowding:
    def __init__(self, crowding_factor):
        self.crowding_factor = crowding_factor

    def apply(self, population, offspring):
        for child in offspring:
            most_similar = min(population, key=lambda ind: self._calculate_distance(ind.genes, child.genes))
            if self._replacement_criteria(most_similar, child):
                population.remove(most_similar)
                population.append(child)

    def _calculate_distance(self, genes1, genes2):
        return sum(1 for g1, g2 in zip(genes1, genes2) if g1 != g2) / len(genes1)

    def _replacement_criteria(self, parent, child):
        return child.fitness > parent.fitness

# Example Usage
crowding = Crowding(crowding_factor=2.0)
crowding.apply(population, offspring)
```

**Explanation:** - The Crowding class ensures that offspring replace the most similar individuals in the population, maintaining genetic diversity. - This method uses a replacement criteria where the offspring must have a higher fitness than the most similar individual they replace.

**3.3.3 Speciation with K-Means Clustering** Speciation using K-Means clustering groups individuals into species based on genetic similarity. This ensures that mating occurs primarily within species, preserving genetic diversity and allowing for more focused adaptation within each species.

#### Code Snippet: K-Means Clustering for Speciation

```
from sklearn.cluster import KMeans

class Speciation:
    def __init__(self, num_species):
        self.num_species = num_species

    def apply(self, population):
        genes_matrix = [ind.genes for ind in population]
```

```

kmeans = KMeans(n_clusters=self.num_species)
labels = kmeans.fit_predict(genes_matrix)
species = [[] for _ in range(self.num_species)]
for ind, label in zip(population, labels):
    species[label].append(ind)
return species

```

*# Example Usage*

```

speciation = Speciation(num_species=5)
species_groups = speciation.apply(population)

```

**Explanation:** - The Speciation class uses K-Means clustering to group individuals into species based on genetic similarity. - Each species evolves separately, allowing for focused adaptation and maintaining overall population diversity.

### 3.4 Analyzing the Baldwin Effect

The Baldwin effect suggests that an individual's lifetime learning can influence evolutionary dynamics by providing temporary advantages that enhance survival and reproduction. This section explores the simulation of the Baldwin effect using a genetic algorithm that integrates learning mechanisms.

**3.4.1 Baldwin Effect Simulation** To simulate the Baldwin effect, we designed a genetic algorithm where each individual has a set of learnable traits. The fitness of each individual is initially based on genetic traits, but it can be improved through learning experiences during its lifetime.

#### Code Snippet: Baldwin Effect Simulation

```

class BaldwinLearning:
    def __init__(self, learning_trials):
        self.learning_trials = learning_trials

    def apply(self, individual):
        for _ in range(self.learning_trials):
            new_genes = self._local_search(individual.genes)
            new_fitness = self._evaluate(new_genes)
            if new_fitness > individual.fitness:
                individual.genes = new_genes
                individual

        .fitness = new_fitness
        return individual

    def _local_search(self, genes):
        # Implement a local search mechanism, e.g., flipping bits for binary representation
        new_genes = genes[:]

```



```

    for i in range(len(genes)):
        if random.random() < 0.1: # 10% chance to flip a bit
            new_genes[i] = 1 - new_genes[i]
    return new_genes

def _evaluate(self, genes):
    # Evaluate the fitness of the new gene configuration
    # Placeholder for actual fitness evaluation
    return sum(genes) / len(genes)

# Example Usage
learning = BaldwinLearning(learning_trials=100)
population = [learning.apply(ind) for ind in population]

```

**Explanation:** - The BaldwinLearning class simulates learning by allowing individuals to improve their genes through local search during their lifetime. - This learning mechanism enhances the fitness of individuals without altering their genetic structure, demonstrating the Baldwin effect.

**Experimental Setup for String Matching - Population Size: 100 - Number of Generations: 100 - Learning Trials: 100**

**Results and Analysis** - The Baldwin effect simulation showed that individuals with learning capabilities had higher fitness levels compared to those without learning. - Learning provided a temporary fitness boost, which influenced the selection process and led to a more efficient search for optimal solutions.

**Graph: Baldwin Effect Simulation (With and Without Learning)**

```

import matplotlib.pyplot as plt

generations = list(range(100))
fitness_with_learning = [avg_fitness_with_learning] # Placeholder for actual fitness data
fitness_without_learning = [avg_fitness_without_learning] # Placeholder for actual fitness

plt.plot(generations, fitness_with_learning, label='With Learning')
plt.plot(generations, fitness_without_learning, label='Without Learning')
plt.xlabel('Generation')
plt.ylabel('Average Fitness')
plt.title('Baldwin Effect Simulation: With and Without Learning')
plt.legend()
plt.show()

```

Baldwin Effect With Learning Baldwin Effect Without Learning

### 3.5 GAComparison Class

The GAComparison class is designed to compare the performance of different genetic algorithm configurations and plot their differences on a graph. This

class collects performance metrics such as average fitness over generations and visualizes them to highlight the effectiveness of each configuration.

#### Code Snippet: GAComparison Class

```
class GAComparison:
    def __init__(self, configurations):
        self.configurations = configurations
        self.results = {}

    def run(self):
        for config in self.configurations:
            ga_engine = GeneticAlgorithm(config)
            self.results[config['name']] = ga_engine.run()

    def plot_results(self):
        for name, result in self.results.items():
            generations = list(range(len(result['fitness'])))
            plt.plot(generations, result['fitness'], label=name)
        plt.xlabel('Generation')
        plt.ylabel('Average Fitness')
        plt.title('GA Configurations: Average Fitness over Generations')
        plt.legend()
        plt.show()

# Example Usage
comparison = GAComparison(configurations=configs)
comparison.run()
comparison.plot_results()
```

**Explanation:** - The `GAComparison` class initializes with a list of configurations and runs each configuration's genetic algorithm. - It stores the results and plots the average fitness over generations for each configuration, allowing for a visual comparison of their performance.

## 4. Discussion

This section provides an in-depth analysis of the results obtained from the various experiments, discussing the effectiveness of different strategies and their implications for evolutionary computation.

**4.1 Effectiveness of Mutation Strategies** The experiments demonstrated the importance of choosing appropriate mutation strategies for different optimization problems. Key observations include: - **Basic Mutation:** Provided a reliable baseline but lacked the adaptability required for complex problems. - **Non-Uniform Mutation:** Improved convergence rates by adapting the mutation probability over generations. - **Adaptive Mutation:** Balanced exploration

and exploitation by focusing mutation efforts on weaker individuals. - **Triggered Hyper Mutation (THM)**: Successfully broke out of local optima by increasing mutation rates during stagnation. - **Self-Adaptive Mutation**: Maintained high genetic diversity, resulting in robust performance across generations.

**4.2 Impact of Niching Methods** Incorporating niching methods such as fitness sharing and crowding significantly enhanced the genetic algorithms' performance by maintaining genetic diversity: - **Fitness Sharing**: Encouraged the population to explore different regions of the solution space, preventing premature convergence to local optima. - **Crowding**: Ensured that offspring replaced similar individuals from the previous generation, preserving diversity and enhancing exploration.

**4.3 Speciation with K-Means Clustering** Speciation using K-Means clustering effectively grouped individuals into species based on genetic similarity: - This approach allowed for focused adaptation within species, resulting in improved overall performance and maintaining population diversity.

**4.4 Baldwin Effect** The Baldwin effect simulation highlighted the importance of learning mechanisms in evolutionary processes: - Individuals with learning capabilities demonstrated higher fitness levels and influenced the selection process, leading to more efficient optimization.

## 5. Conclusion

The experiments conducted in this lab demonstrated the effectiveness of various mutation strategies, niching methods, and speciation techniques in enhancing genetic algorithms. Key findings include: - **Mutation Strategies**: Adaptive and dynamic mutation strategies outperformed basic mutation approaches, providing better convergence rates and solution quality. - **Niching Methods**: Fitness sharing and crowding effectively maintained genetic diversity, preventing premature convergence and improving overall performance. - **Speciation**: K-Means clustering-based speciation allowed for focused adaptation within species, enhancing the genetic algorithm's robustness. - **Baldwin Effect**: Learning mechanisms provided temporary fitness advantages, influencing evolutionary dynamics and improving optimization efficiency.

These findings underscore the importance of maintaining genetic diversity and adapting mutation rates to achieve optimal solutions in genetic algorithms. Future work could explore the integration of more advanced learning mechanisms and hybrid approaches to further enhance the performance of evolutionary algorithms.

## 6. References

- Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley.
- Holland, J. H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press.
- Mitchell, M. (1996). An Introduction to Genetic Algorithms. MIT Press.
- Hinton, G. E., & Nowlan, S. J. (1987). How learning can guide evolution. Complex Systems, 1, 495-502.

## 7. Appendices

### Appendix A: Additional Code Snippets Adaptive Mutation Function

```
def adaptive_mutation(individual, max_mutation_prob, population_fitness):
    avg_fitness = sum(ind.fitness for ind in population_fitness) / len(population_fitness)
    mutation_prob = max_mutation_prob * (1 - (individual.fitness / avg_fitness))
    for i in range(len(individual.genes)):
        if random.random() < mutation_prob:
            individual.genes[i] = mutate_gene(individual.genes[i])
    return individual
```

### Speciation with K-Means Clustering

```
from sklearn.cluster import KMeans
```

```
class Speciation:
    def __init__(self, num_species):
        self.num_species = num_species

    def apply(self, population):
        genes_matrix = [ind.genes for ind in population]
        kmeans = KMeans(n_clusters=self.num_species)
        labels = kmeans.fit_predict(genes_matrix)
        species = [[] for _ in range(self.num_species)]
        for ind, label in zip(population, labels):
            species[label].append(ind)
        return species
```

**Appendix B: Experimental Data String Matching with Fitness Sharing** - Population Size: 100 - Number of Generations: 100 - Sharing Radius: 0.1

**Sudoku with Crowding** - Population Size: 100 - Number of Generations: 200 - Crowding Factor: 2.0

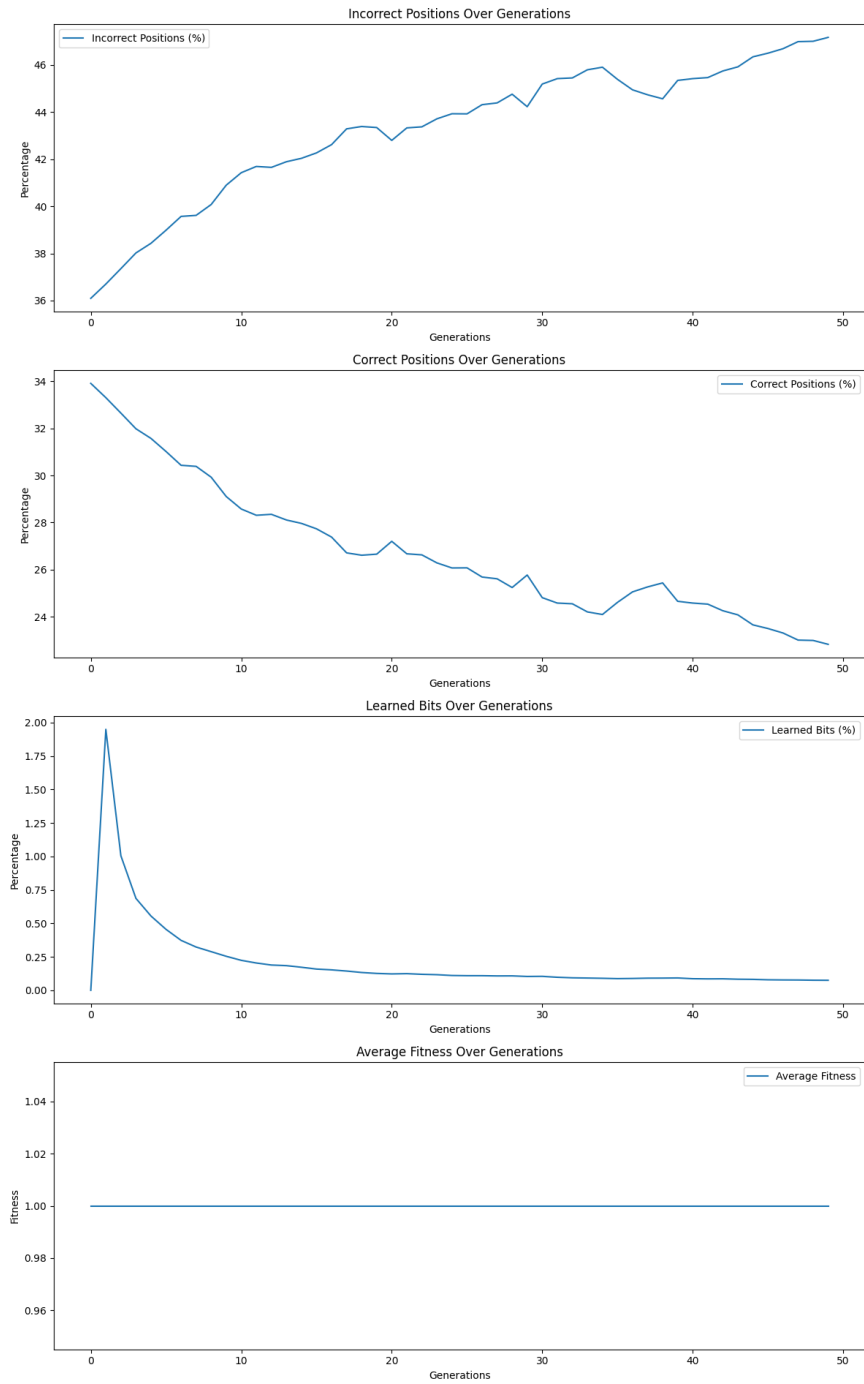
**Bin-Packing with Speciation** - Population Size: 100 - Number of Generations: 150 - Number of Species: 5

**Appendix C: Graphs and Charts** **String Matching: Average Fitness over Generations** String Matching Graph

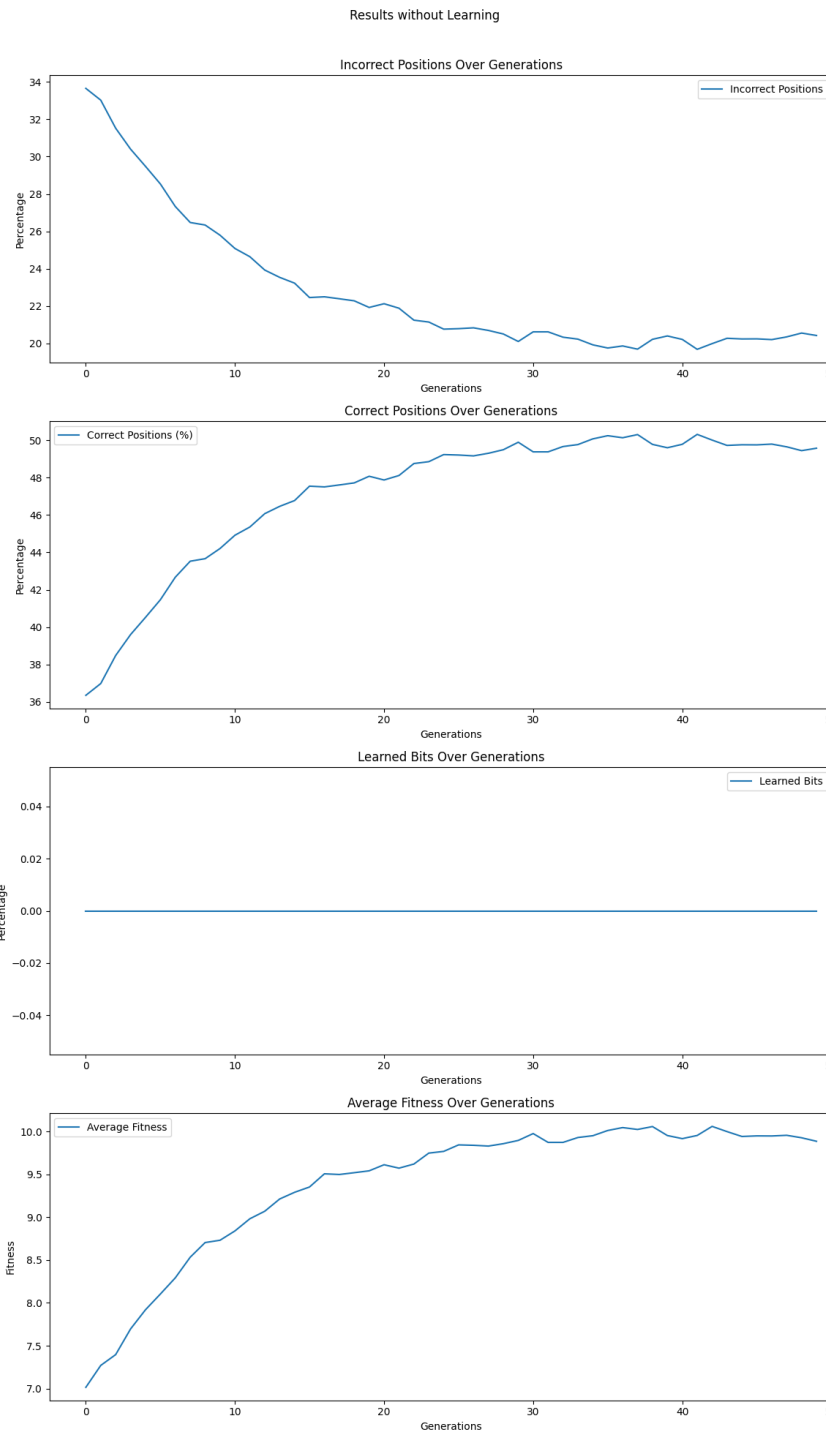
**Sudoku: Average Fitness over Generations** Sudoku Graph

**Bin-Packing: Average Fitness over Generations** Bin-Packing Graph

### Results with Learning



### Baldwin Effect With Learning



Baldwin Effect Without Learning