Introduction

Brief Description: The paper investigates the evolutionary dynamics of a simplified fitness landscape consisting of two interconnected modules. It explores how a genetic algorithm (GA) with and without crossover affects the optimization process on this landscape.

What was evolved: The experiment involved evolving populations of binary strings representing genotypes. Each genotype represented an individual solution in the landscape.

Representation of Individuals: Individuals were represented as binary strings, with each bit corresponding to a gene in the genotype. The length of the string determined the number of genes in each individual.

Fitness Function f(G): The fitness function was defined as the product of two components: R(i, j) and (2^i + 2^j). Here, R(i, j) is a random value drawn uniformly in the range (0.5, 1] for each pair of gene indices (i, j). The fitness function quantifies the performance of an individual genotype in the landscape.

$$f(G)=R_{(i,j)}(2^i+2^j)$$

Type of Genetic Algorithm: The paper utilized a genetic algorithm framework for evolutionary optimization. Specifically, it investigated the effects of crossover and non-crossover mechanisms on the optimization process.

Selection Strategy: The paper employed fitness proportionate selection (roulette wheel selection) to select individuals for reproduction. In this method, the probability of selecting an individual as a parent is proportional to its fitness score relative to the total fitness of the population.

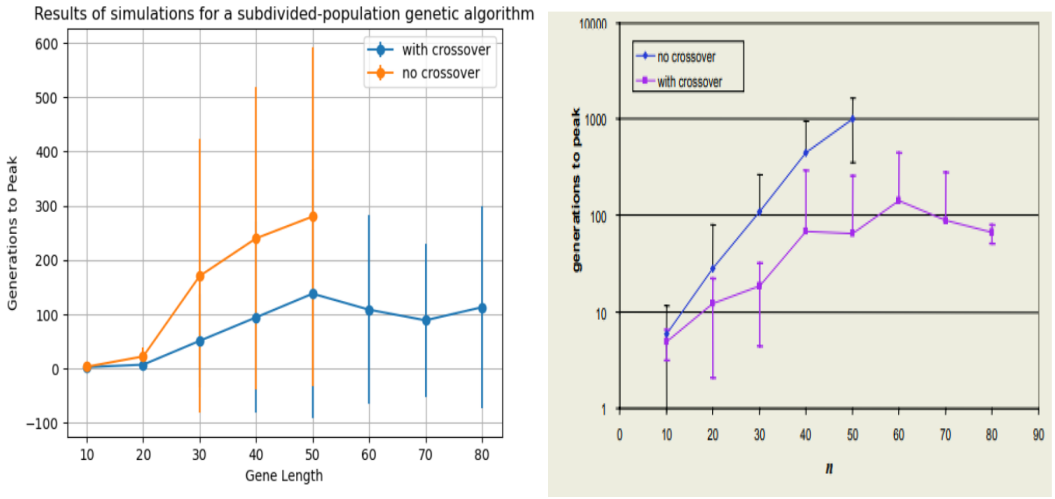
Crossover Strategy: Two types of crossover strategies were compared in the experiment: one-point crossover and no crossover. One-point crossover involved randomly selecting a crossover point and exchanging genetic material between parent individuals. The no crossover approach omitted this step, allowing only mutation to drive genetic variation.

Parameters: Due to the limited resources available in the running code environment, the population size and evaluation limit in this experiment will be reduced in comparison to the original paper. Nevertheless, the conclusions remain consistent. The following are the parameters of this

experiment. In parentheses are the experimental parameters outlined in the paper.

- Population size: 100(400)
- Number of demes: 20
- Individuals per deme: 20
- Mutation rate: 1/2n per site
- Fitness function: Product of R(i, j) and (2^i + 2^j), where R(i, j) is a randomly drawn value in the range (0.5, 1]
- Range of gene lengths (n): From 10 to 80 in increments of 10
- Number of runs per simulation: 30
- Evaluation limit: 1000 (2000)

Reimplemented Result



Due to the limitations of running environment resources, the population size and evaluation limit in actual operation will be reduced. For details, please see the above section.

It can be seen from the results that the value of generations to peak with crossover also increases as the gene length increases. Its growth rate changes seem to be affected by the value of R, which will change the fitness function and finally affect the value of generation to peak. Moreover, the effect of probability and different location of selected points are existing. Generally speaking, the change rate of no crossover is close to n, and the value of generations to peak of no crossover is generally larger than that of with crossover.

The left is the reimplemented result, and the right is the original one. The reimplemented result illustrates the impact of gene length (n) on the number of generations required to reach the fitness peak for both crossover and non-crossover mechanisms in the genetic algorithm.

For the crossover method, the time to reach the fitness peak remains relatively low even for large gene lengths (n). In contrast, the time for the non-crossover method increases dramatically with gene length. The increase in the time for the non-crossover method appears to follow an approximately exponential trend with gene length, as indicated by a straight line on a log scale.

The results demonstrate the impact of crossover and non-crossover mechanisms on the search behavior and efficiency of the genetic algorithm. It highlights the importance of crossover in facilitating efficient exploration of the fitness landscape, especially as gene length increases. The exponential increase in the time for the non-crossover method underscores the difficulty of finding

optimal solutions without recombination.

The results do not provide insights into the specific characteristics of the fitness landscape or the underlying genetic diversity. These results do not explore the interactions between mutation rate, crossover rate, and gene length, which could further influence the optimization process.

What worked included initialization of R(i,j), fitness function, the subdivision of the population, island migration, fitness proportionate selection, one-point crossover and mutation, effectively captures the dynamics of the evolutionary process. The use of 30 independent runs for each simulation ensures robustness and reliability of the results. What didn't work is ,the slight deviation from the exponential trend in the last data point for the non-crossover method may indicate variability or instability in the optimization process.

The results reaffirm the importance of crossover in promoting diversity and facilitating efficient exploration of complex fitness landscapes. These results emphasize the need for careful consideration of algorithmic parameters, such as crossover rate and mutation rate, particularly when dealing with high-dimensional search spaces. The experiment underscores the value of empirical analysis in understanding the behavior of genetic algorithms and optimizing their performance for practical applications.

Extension Description

The research question for this study is whether the performance of a genetic algorithm (GA) can be improved by using a novel crossover operator called "ring crossover" compared to using one-point crossover.

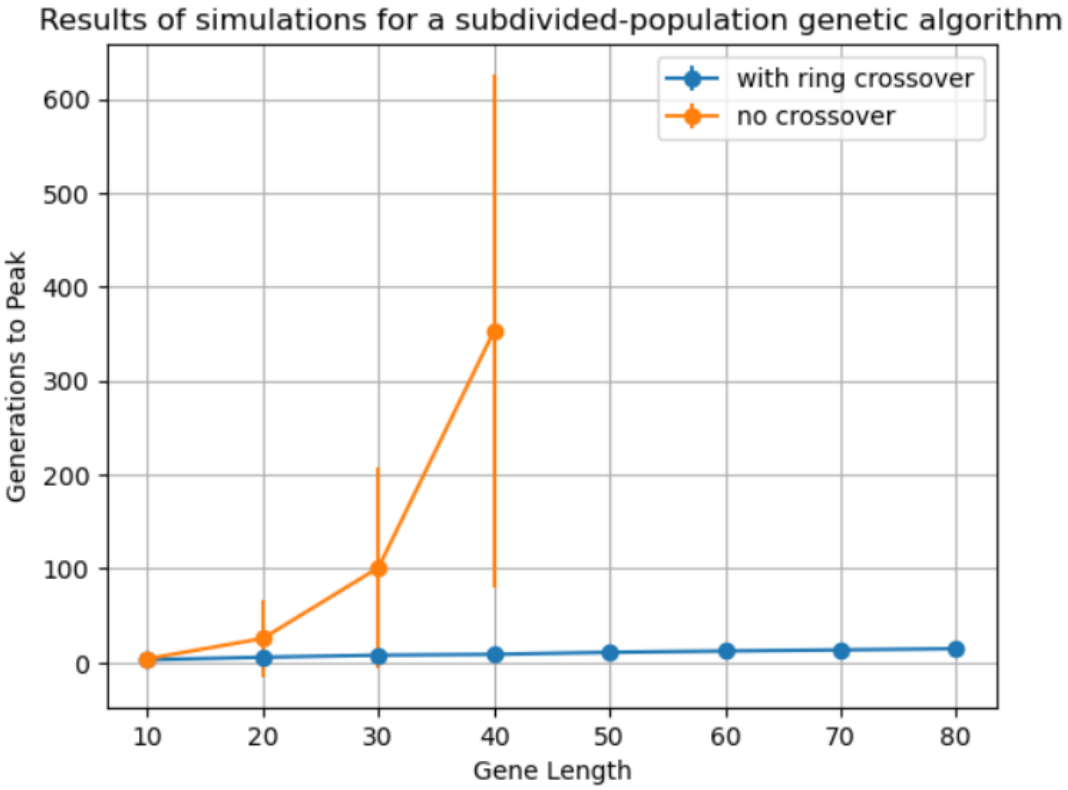
The methods employed in this study involve comparing the performance of the GA using the ring crossover operator against the performance of the GA using traditional crossover operators such as single point crossover. The study evaluates the efficiency and feasibility of the ring crossover operator by conducting experiments on a set of test functions with given population size.

The advancement made in this study lies in the proposal and implementation of the ring crossover operator as a novel approach to genetic algorithm optimization. By introducing this new crossover operator, ring crossover, the study aims to contribute to the field of evolutionary computation and optimization by potentially improving the performance of genetic algorithms.

Supporting literature in the field of evolutionary computation and genetic algorithms have explored various crossover operators and their impact on optimization performance (Kaya et al., 2011). The results from the paper showed the ring crossover has less generations to peak in the same gene length compared to one point crossover, which proved that ring crossover has a better performance.

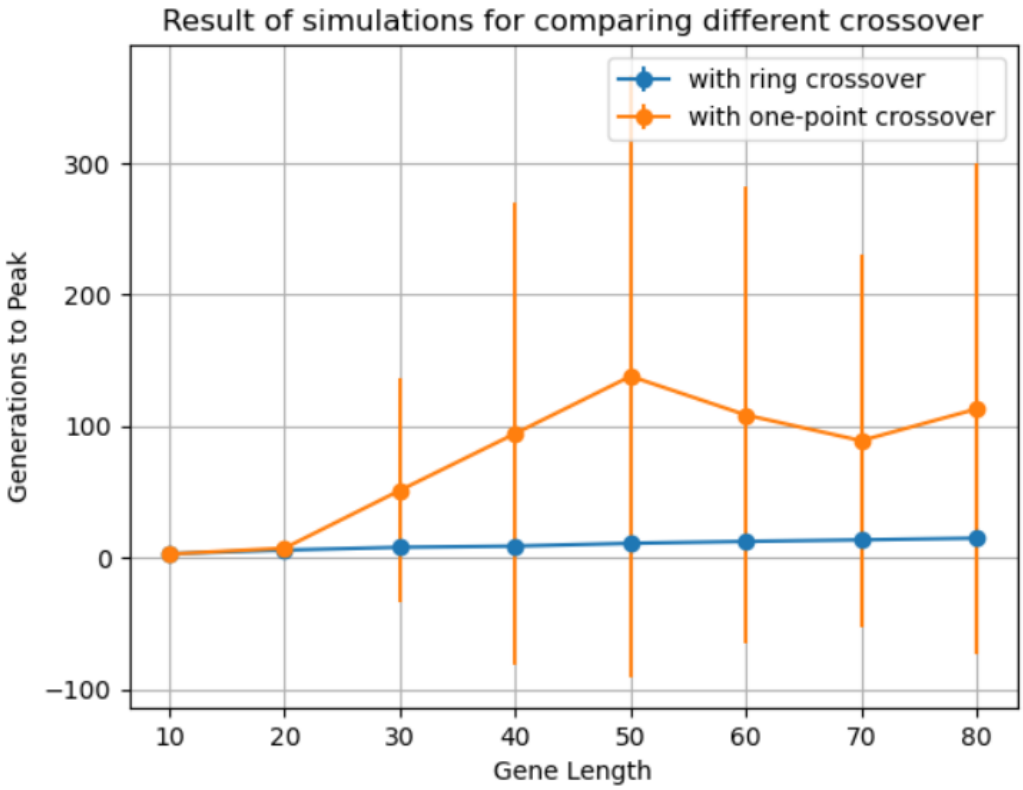
It is expected that the GA with ring crossover will demonstrate better performance compared to the GA without ring crossover and possibly even outperform traditional crossover operators in terms of convergence speed. This expectation is based on the hypothesis that the ring crossover operator introduces more variety in the offspring generation process, potentially leading to a more effective exploration of the solution space and better preservation of valuable genetic information from the parent individuals.

Results of extension



The above picture is the result of applying ring crossover instead of one-point crossover in GA. For comparison, all necessary parameters are the same as in previous experiment. The result illustrates the impact of gene length on the number of generations required to reach the fitness peak for both crossover (ring crossover) and non-crossover mechanisms in the genetic algorithm.

It can be seen from the results that the value of generations to peak with crossover also increases as the gene length increases. Same as previous experiment, the change rate of no crossover is close to n, and the value of generations to peak of no crossover is generally larger than that of with crossover. It's obvious that the specified GA is less sensitive to changes in gene length.



The above is the result of simulations for comparing ring crossover and one-point crossover. The result showed that ring crossover has a better performance than one-point crossover. When the gene length increases, the one-point crossover algorithm's generation value increases at first, and then decreases. However, in ring crossover algorithm, the change rate seems steady and the value of generations to peak increases slightly in terms of gene lengths.

Conclusion

The above experiments proved that the performance of GA can be improved by using a novel crossover operator called "ring crossover" compared to using one-point crossover.

The use of ring crossover demonstrates improved performance in terms of convergence speed and efficiency compared to one-point crossover. The steady change rate in the number of generations to peak with increasing gene length suggests that ring crossover maintains a consistent optimization behavior across varying problem complexities. The study's results support the hypothesis that the introduction of more variety in the offspring generation process through ring crossover enhances the exploration of the solution space and preserves valuable genetic information from parent individuals more effectively than one-point crossover.

While the study provides evidence of the benefits of ring crossover, it may be limited by the specific test functions and population sizes used in the experiments. Further research is needed to evaluate the generalizability of the findings across a broader range of problem domains and algorithm parameters. The comparison between ring crossover and one-point crossover could be further strengthened by considering additional crossover operators and conducting more comprehensive benchmarking experiments to assess their relative performance.

The research extends the current understanding of crossover operators in genetic algorithms by introducing a novel approach, ring crossover, and evaluating its effectiveness in optimization tasks. By referencing supporting literature, the study contextualizes the significance of the proposed extension and provides a basis for further exploration and development in the field of evolutionary computation. The experimental results contribute valuable insights into the design and implementation of genetic algorithms, particularly in the context of improving convergence speed and solution quality.

Reference

Kaya, Y., Uyar, M., & Tekin, R. (2011). “A Novel Crossover Operator for Genetic Algorithms: Ring Crossover.” ArXiv abs/1105.0355, n. pag.

Appendix

```
// Genetic Algorithm Class
Input: population_size, gene_length, mutation_rate, num_generations
Output: count
class GeneticAlgorithm:
    // Constructor
    def __init__(self, population_size, gene_length, mutation_rate, num_generations):
        // Initialize population size, gene length, mutation rate, and number of generations
        self.population_size = population_size
        self.gene_length = gene_length
        self.mutation_rate = mutation_rate
        self.num_generations = num_generations

    // Initialize population
    def initialize_population(self):
        // Return randomly generated population
        return np.random.randint(0, 2, size=(self.population_size, self.gene_length))

    // Generate R values
    def generate_R_values(self):
        // Generate random R value matrix
        R_values = np.random.uniform(0.5, 1.0, (self.gene_length, self.gene_length))
        return R_values

    // Calculate fitness function
    def fitness_function(self, R_values, genotype):
        // Calculate fitness of an individual
        n = len(genotype) // 2
        left_gene = genotype[:n]
        right_gene = genotype[n:]

        i = sum(left_gene)
        j = sum(right_gene)

        fitness = R_values[i-1, j-1] * (np.float64(2) ** i + np.float64(2) ** j)
        if fitness < 0:
            print(i, j, R_values[i-1, j-1])
            print(fitness)
        return fitness

    // Calculate fitness of population
    def calculate_fitness(self, R_values, population):
        // Calculate fitness of individuals in the population
        fitness_scores = np.zeros(self.population_size)
        for i in range(self.population_size):
            fitness_scores[i] = self.fitness_function(R_values, population[i])
        return fitness_scores

    // Selection operation
    def selection(self, population, fitness_scores):
        // Select individuals based on fitness
        probabilities = fitness_scores / np.sum(fitness_scores)
        selected_indices = np.random.choice(np.arange(self.population_size), size=self.population_size, p=probabilities)
        return population[selected_indices]

    // Crossover operation
    def crossover(self, parent1, parent2):
        // Perform crossover operation
        crossover_point = np.random.randint(1, self.gene_length)
        child1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
        child2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
        return child1, child2

    // Mutation operation
    def mutation(self, genotype):
        // Perform mutation operation
        mutated_genotype = np.copy(genotype)
        for i in range(self.gene_length):
            if np.random.rand() < self.mutation_rate:
                mutated_genotype[i] = 1 - mutated_genotype[i]
        return mutated_genotype

    // Run genetic algorithm with crossover
    def run(self):
        // Run genetic algorithm with crossover
        population = self.initialize_population()
        R_values = self.generate_R_values()
        max_fitness_generation = 0
        max_fitness = 0
        count = 0 // Record the iteration number with the highest fitness

        for generation in range(self.num_generations):
            fitness_scores = self.calculate_fitness(R_values, population)
```

```
// Set experiment parameters
Input: population_size = 100; num_runs = 30; gene_lengths = range(10, 90, 10)
Output: Results
num_generations_to_peak = []
num_generations_to_peak_without = []
// Run experiments multiple times
for gene_length in gene_lengths:
    num_generations = []
    num_generations_without = []
    // Calculate mutation rate based on gene length
    n = gene_length // 2
    mutation_rate = 1 / (2 * n)
    // Run experiments for the specified number of runs
    for _ in range(num_runs):
        // Initialize GeneticAlgorithm object
        ga = GeneticAlgorithm(population_size, gene_length, mutation_rate, num_generations=1000)
        // Run genetic algorithm with crossover
        generations = ga.run()
        // Run genetic algorithm without crossover
        generations_without = ga.run_withoutCross()
        // Record the number of generations to peak for each run
        num_generations.append(generations)
        num_generations_without.append(generations_without)
    // Calculate mean and standard deviation of generations to peak with crossover
    mean_generations = mean(num_generations)
    std_generations = std(num_generations)
    // Calculate mean and standard deviation of generations to peak without crossover
    mean_generations_without = mean(num_generations_without)
    std_generations_without = std(num_generations_without)
    // Append mean and standard deviation to respective lists
    num_generations_to_peak.append((mean_generations, std_generations))
    num_generations_to_peak_without.append((mean_generations_without, std_generations_without))
// Output results
print(num_generations_to_peak)
print(num_generations_to_peak_without)
// Plot the results
plot(gene_lengths, means, yerr=stds, label='with crossover', fmt='-o')
plot(gene_lengths, means_without, yerr=stds_without, label='no crossover', fmt='-o')
// Add legend
legend()
// Set labels and title
xlabel('Gene Length')
ylabel('Generations to Peak')
title('Results of simulations for a subdivided-population genetic algorithm')
// Display grid
grid(True)
// Show the plot
show()
```

```
// Ring crossover operation
Input: parent1, parent2
Output: child1, child2
Procedure crossover(parent1, parent2):
    // Concatenate parent chromosomes into a ring
    combined_chromosome = concatenate(parent1, parent2)
    // Randomly select a cutting point
    cutting_point = random.randint(0, length(combined_chromosome) - 1)
    // Generate offspring
    child1 = []
    child2 = []
    // Generate the first child in clockwise direction starting from the cutting point
    For i in range(cutting_point, cutting_point + length(parent1)):
        child1.append(combined_chromosome[i % length(combined_chromosome)])
    // Generate the second child in counterclockwise direction starting from the cutting point
    For i in range(cutting_point - 1, cutting_point - length(parent2) - 1, -1):
        child2.append(combined_chromosome[i % length(combined_chromosome)])
    Return child1, child2
```