

1. Introduction

The paper chosen to be investigated in this coursework will be “A Simple Two-Module Problem to Exemplify Building-Block Assembly Under Crossover” by Richard A. Watson [1]. The motivation behind this paper is regarding the fact that a genetic algorithm with crossover will perform better than a genetic algorithm without crossover given some fitness landscapes according to [2], [3]. Aside from investigating the genetic algorithm with and without crossover, the paper also stresses upon the difficulties in formulating and defining a fitness landscape that could exemplify the idea of building-block assembly. In John H. Holland’s paper titled building blocks, Cohort Genetic Algorithms, and Hyperplane-Defined Functions [4], two main characteristics of building block were given and they are i) Must be easy to be differentiated once selected or discovered, ii) Must be able to recombine easily to generate a wide range of structures. In terms of genetic algorithm, building block plays a vital role in Darwin’s original formulation about natural selection and artificial selection.

In addition, the experiment aims to prove that crossover algorithm will take polynomial time to discover fit genotypes instead of exponential time for the case of a mutation hill-climber algorithm. This leads to a hypothesis where sequential discovery (mutation hill climber) of high fitness schemata is more difficult than a parallel discovery (crossover). As a control to the experiment, a simple multi-deme island model as per [5] has been implemented. The following sections will justify the aforementioned goals and ways to improve the existing model.

2. Reimplementation of the Experiment

2.1 Initialization of variables

Equation 1 is defined as the fitness of a genotype, $f(G)$ with the genotype, G defined as a $2n$ binary vector, $G = \langle g_1, g_2, \dots, g_{2n} \rangle$. n here is defined as the nucleotide sites.

$$f(G) = R_{(i,j)}(2^i + 2^j) \quad (1)$$

i in equation 1 is defined as the number of 1s in the first half of the genotype, $\{g_1, g_2, \dots, g_n\}$ whereas j is the number of 1s in the second half of the genotype, $\{g_{n+1}, g_{n+2}, \dots, g_{2n}\}$. The fixed array ‘noise’, $R_{(i,j)}$ here is a vector with the size of $i \times j$ and each individual in the vector returns a value drawn randomly between (0.5 – 1). The values in vector $R_{(i,j)}$ remains constant for each simulation run (disregarding the number of generations). The fittest genotype are all 1s configuration of the genotype with a length of $2n$. The fitness landscape across i, j can then be plotted with each individual fitness calculated using equation 1. 2^i and 2^j essentially describe a landscape in which the nucleotide sites, n within genes have a lot of synergy and sites in separate genes have additive fitness effects.

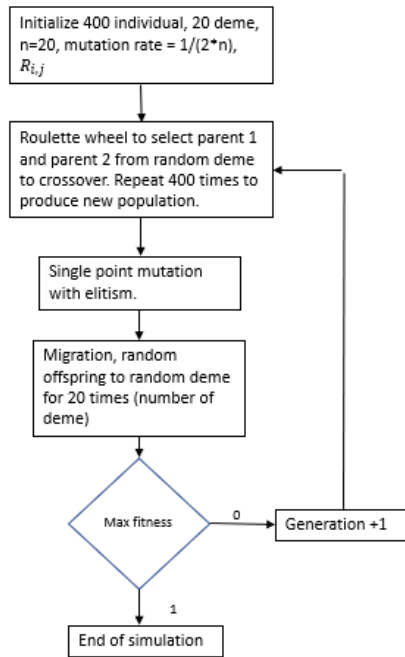
2.2 Motivations

At early stages of the genetic algorithm (first few generations), it is easy to move up the fitness gradient as 1-mutations are strongly rewarded within the gene and each fitness contribution is strong enough to overcome the random noise $R_{(i,j)}$ in the landscape. However, as the fitness in left side gene in the genotype progresses, it becomes difficult to find the good allele for the right-side gene due to the local optima (ridges) formed by noise. This is shown in the center of figure 2 in [1]. A mutation hill climbing algorithm cannot demote the fitness level in and this will converge to only a local optima

solution. Therefore, crossover genetic algorithm is better in this case as it creates an offspring at the intersection between 2 ridges and this essentially corresponds to the peak of the fitness because the fitness function is an additive sum between 2^i and 2^j . To ensure diversity, a multi deme model was used.

2.3 Algorithm

Crossover with single point mutation



Single point mutation

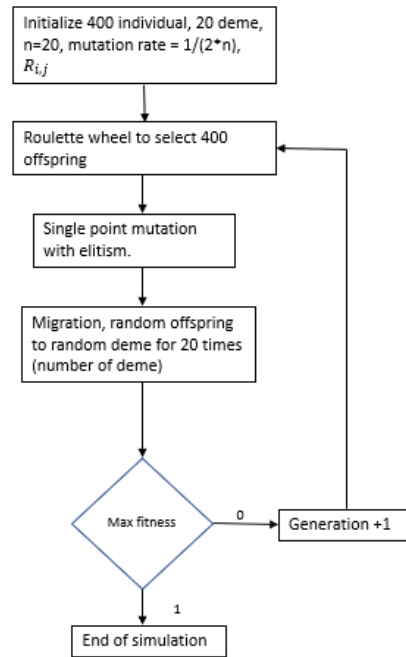


Figure 1: Both crossover and non-crossover algorithm with a single point mutation implemented.

The first step is to initialize the variable according to section 2.1. Next, in the non-crossover algorithm, the roulette wheel selection has a higher tendency of selecting individuals with a higher fitness value. This ensures that the algorithm is not as biased as a mutational hill climber where only the fittest individual can be retained. In one point crossover, parents 1 and 2 were selected at random by using roulette wheel selection. An inter-local position was also chosen at random and the sites to the left will be filled with alleles from parent 1 and sites to the right will be copied from parent 2.

Both the crossover and non-crossover algorithm were subjected to single point mutation with a small probability. To ensure that genetic drift will not affect the performance of the algorithm, elitism was used where the fittest individual from the previous generation was retained. Island migration between sub-population was also implemented to increase the variation in the search process.

2.4 Comparison of simulated result

In **Error! Reference source not found.** and Figure 3 below, it can be observed that the algorithm with crossover managed to find the fittest gene in less than 200 generations with the nucleotide sites as the increasing variable from 10 to 80 in each simulation. The non-crossover algorithm on the other hand shows an exponential increase and the standard deviation increases as the number of sites increases, thus making it an unreliable algorithm in this fitness landscape. In terms of qualitative analysis, the crossover algorithm took polynomial time to converge while the non-crossover algorithm

took exponential time. This agrees with the results obtained in the investigated paper [1] and also the hypothesis formed earlier in the introduction section.

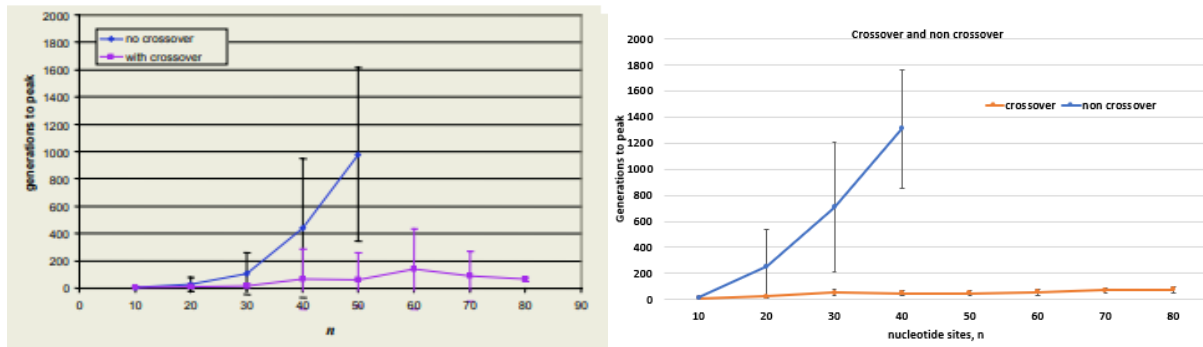


Figure 2: Simulated result of figure 3 in [1]. left) Original result, right) Reimplemented result.

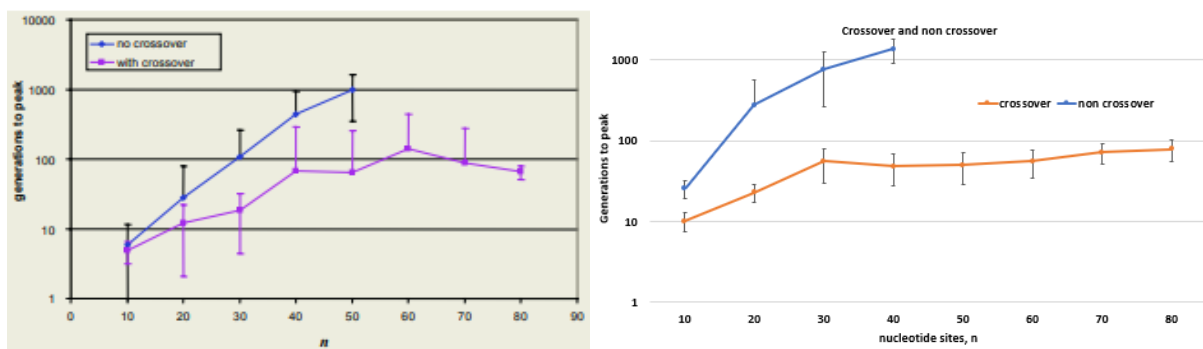


Figure 3: Simulated logscale result of figure 3 in [1]. left) Original result, right) Reimplemented result.

Quantitatively, the generations to peak in the crossover algorithm in both the original and reimplemented results showed similar values but the standard deviation differs. To explain this, one possibility would be because the mutation rate was too low. Once the solution converges to the local optima, there is no way to move down the hill and increase the search area which leads to the global optima. In addition, the reimplemented experiment was not able to find the fittest genotypes for n larger than 40 in the non-crossover algorithm. However, if the simulation time increases, there is certainly a chance that at some point of time a global solution can be achieved.

Based on the results obtained, it can also be concluded that for such an additive fitness landscape, algorithms such as crossover which supports parallel discovery can converge to the global solution. Mutation hill climber that does the search process sequentially tends to improve only one genotype in the gene and this pulls the search process away into the ridges which are caused by noise.

3. Extensions

3.1 Motivations, Problem statement, literature review and hypothesis

In the previous experiment in section 2, convergence at local optima is a major problem. As the size of the nucleotide sites increase, the problem became worse. It is practically impossible to obtain 30 readings (generations to obtain the fittest individual) in just 30 simulation runs. At $n=60$, only 1 out of 30 runs managed to converge at the global optima and the rest fail because the search process was pulled to the ridges.

Traditionally, most published research paper aiming to solve the local optima issue has focused only on the recombination part (single point crossover, uniform crossover, non-crossover, building blocks in crossover) of the genetic algorithm [6], [7] and not the selection part. This then brings forth the idea of varying the selection process and keeping the genetic algorithm constant to investigate the effect this would have on the search process. In paper [8], the performance between different selection strategies on simple genetics algorithms was investigated and implemented on multiple fitness function. One of the functions was an additive function given by $x_i^2 + x_j^2$ in figure 3 of the paper. This confirms that selection process does have an impact in an additive landscape such as the one defined in equation 1. Hence, tournament selection was used to replace the roulette wheel selection method in the previous experiment. The hypothesis that was formed here is:

1. Tournament selection outperforms roulette wheel selection in a simple 2 module single point crossover genetic algorithm.

The aims are therefore:

1. To prove that the number of generations required to achieve maximum fitness will be lower than roulette wheel selection.
2. To prove quantitatively and qualitatively that to obtain 30 data points of globally converged solutions, the total number of simulations run needed will be lower in the case of tournament selection.
3. Investigate the effect of selection pressure on the search process in a genetic algorithm.

3.2 Method

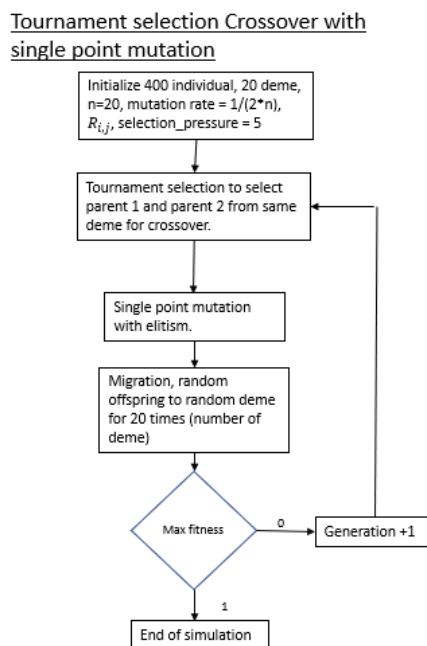


Figure 4: Tournament selection crossover with single point mutation with a selection pressure of 5.

To implement the tournament selection method, a deme was first selected. 5 individuals were then selected at random to compete and the fittest individual will be parent 1. Next, the step is repeated to select parent 2. Single point crossover will then happen with the sites to the left of the crossover point from parent 1 and the right copied from parent 2. Next, the selected offspring are subjected to

single point mutation with elitism to prevent genetic drift which could affect the simulated results. Random migration also takes place to ensure high level of diversity. Keeping in mind that the 5 individuals selected represents the selection pressure, which is a variable, it will be varied to also include a selection pressure of 2 and 10.

3.3 Results

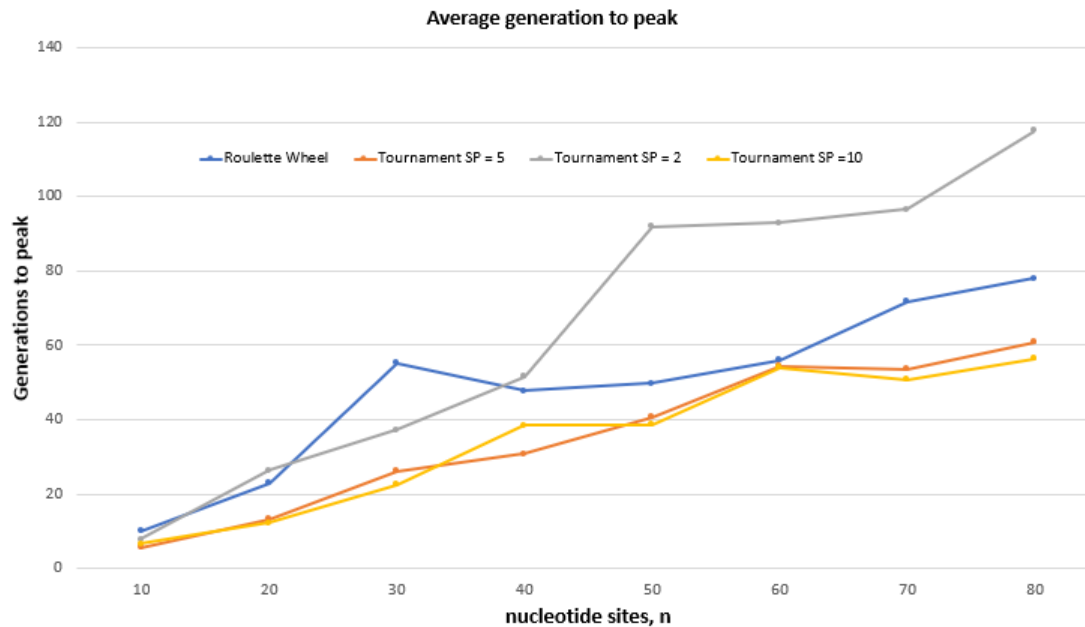


Figure 5: Graph of Generation to peak against the nucleotide sites, n. Roulette wheel and tournament selection algorithm were compared. The selection pressure of tournament selection was tuned to be 2, 5, and 10 accordingly.

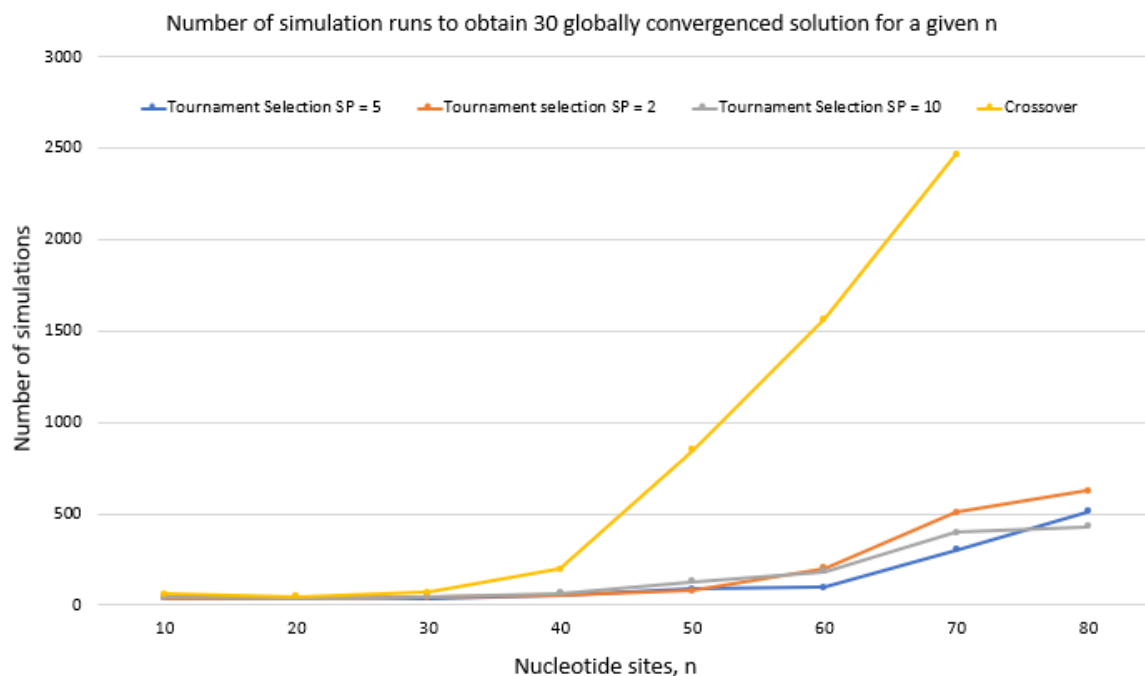


Figure 6: Total number of simulations runs required to obtain 30 points of globally converged solution for a given nucleotide site.

For a selection pressure of 5 and 10 In the tournament selection algorithm, it was observed that the performance was better because it took lesser generation to reach a peak fitness. However, when the selection pressure was tuned to 2, the performance dropped significantly especially at high n. Since only 2 random individuals were selected from a deme without any reference to their fitness level, there is a possibility that the lowest fitness individual was selected, and this reduces the chance of producing a high fitness offspring during crossover. Another observation in Figure 5 is that the generations required to achieve maximum fitness did not show much difference for a selection pressure of 5 and 10. Hence, it can be say that the optimum selection pressure is between 5 and 10 and anything lesser than 5 or 10 would increase the computational cost.

In Figure 6, it is evident that tournament selection performed better than roulette wheel selection. For n = 70 and a selection pressure of 5, the probability of reaching the maximum fitness is 9.9% while roulette wheel only gives 1.2%. The results in Figure 5 and Figure 6 confirmed that tournament selection outperforms roulette wheel selection in a simple 2 module single point crossover genetic algorithm. The results also agree with the findings in [8] where the simulation time of tournament selection is shorter than roulette wheel selection. Therefore, the hypothesis formed is valid and three major goals defined has been investigated thoroughly.

4. Conclusion

As a conclusion, the reimplementaion of ‘Simple Two-Module Problem to Exemplify Building-Block Assembly Under Crossover’ has been satisfactory in terms of the results obtained. The only downside is the simulation time and the issue of solutions converging at the local optima. This problem was solved in the extension where tournament selection method was used to replace roulette wheel selection. The experiment done could potentially provide a different perspective on the importance of selection method in genetic algorithms. Ranked selection, Boltzman selection, elitism selection and steady state selection are also a few of the many selections method available and is to be explored in future work.

5. Reference

- [1] R. A. Watson, “LNCS 3242 - A Simple Two-Module Problem to Exemplify Building-Block Assembly Under Crossover,” 2004.
- [2] D. H. Wolpert and W. G. Macready, “No Free Lunch Theorems for Optimization,” 1997.
- [3] John H. Holland, *Adaptation in Natural and Artificial Systems*. 1975.
- [4] J. H. Holland, “Building Blocks, Cohort Genetic Algorithms, and Hyperplane-Defined Functions.”
- [5] S. Wright, “EVOLUTION IN MENDELIAN POPULATIONS,” *Genetics*, vol. 16, no. 2, pp. 97–159, Mar. 1931, doi: 10.1093/genetics/16.2.97.
- [6] M. Mitchell, J. H. Houand, and S. Forrest, “When Will a Genetic Algorithm Outperform Hill Climbing?”
- [7] R. A. Watson and T. Jansen, “A building-block royal road where crossover is provably essential,” in *Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO '07*, 2007, p. 1452. doi: 10.1145/1276958.1277224.
- [8] Z. Jinghui, H. Xiaomin, G. Min, and Z. Jun, “Comparison of performance between different selection strategies on simple genetic algorithms,” in *Proceedings - International Conference on Computational Intelligence for Modelling, Control and Automation, CIMCA 2005 and International Conference on Intelligent Agents, Web Technologies and Internet*, 2005, vol. 2, pp. 1115–1120. doi: 10.1109/cimca.2005.1631619.

6. Appendix A – Code

Non-crossover.py

```

1  from operator import index
2  import random
3  from telnetlib import NEW_ENVIRON
4  from cv2 import split
5  from numpy import number
6  from random import choice
7
8  # constant parameter
9  pop_size = 400
10 demes = 20
11 number_of_sites = 40
12 mutation_rate = 1/ (2 * number_of_sites)
13 ind_per_island = int(400 / 20)
14 # =====
15
16 # initialization
17 pop = []
18 R_noise = []
19 fitness = []
20 new_pop = []
21 probability = []
22 mutated = []
23 max_prob_index = []
24 max_fitness = 0
25 generation = 0
26 sim = []
27 simulation = 0
28 # =====
29
30
31 def Rij():
32     #function to generate the different constant R for i,j
33     R_noise = [[random.uniform(0.5, 1.0) for i in range(number_of_sites)]
34               for j in range(number_of_sites)]
35     return R_noise
36
37
38 def rand_key(number_of_sites):
39     # function to generate individual with 2n binary
40     length = 2 * number_of_sites
41     individual = ""
42     for i in range(length):
43         temp = str(random.randint(0, 1))
44         # temp = '1'
45         individual += temp
46     return (individual)
47
48
49
50 def pop_generator(pop_size):
51     pop = []
52     # function to generate 400 population of 2n binary
53     for i in range(pop_size):
54         random_str = rand_key(number_of_sites)
55         pop.append(random_str)
56         split_lists = [
57             pop[x:x + ind_per_island]
58             for x in range(0, len(pop), ind_per_island)
59         ]
60     return split_lists
61
62
63 def zip_fitness_eval(population, R_noise):
64     maxi_fitness = 0
65     fitness = []
66     for i in range(demes):
67         for j in range(ind_per_island):
68             gene_i = population[i][j][:number_of_sites].count('1')
69             gene_j = population[i][j][number_of_sites:].count('1')
70             # for k in range(demes):
71             #     for a in range(int(pop_size / demes)):
72             #         for z in range(2 * number_of_sites):
73             #             if pop[k][a][z] == '1' and z < number_of_sites:
74             #                 i += 1
75             #             if pop[k][a][z] == '1' and z >= number_of_sites:
76             #                 j += 1
77
78             fitness_equation = R_noise[gene_i - 1][gene_j - 1] * (2**gene_i +
79             2**gene_j)
80             fitness.append(fitness_equation)
81
82     maxi_fitness = max(fitness)
83
84     split_lists = [
85         fitness[x:x + ind_per_island]
86         for x in range(0, len(fitness), ind_per_island)
87     ]
88     return split_lists, maxi_fitness
89
90
91 def get_probability_list(deme_dict):
92     fitness = deme_dict.values()
93     total_fit = float(sum(fitness))
94     relative_fitness = [f / total_fit for f in fitness]
95     probabilities = [
96         sum(relative_fitness[:i + 1]) for i in range(len(relative_fitness))

```

```

96         sum(relative_fitness[:i + 1]) for i in range(len(relative_fitness))
97     ]
98     return probabilities
99
100
101 def roulette_wheel_pop(population, probabilities, number):
102     for n in range(number):
103         r = random.random()
104         for (i, individual) in enumerate(population):
105             if r <= probabilities[i]:
106                 individual = individual
107                 break
108     return individual
109
110
111 def fitness_proportionate_selection(pop, fitness, max_fitness_index):
112     new_pop = [[None for i in range(ind_per_island)] for j in range(demes)]
113     # print(new_pop)
114     for i in range(demes):
115         for j in range(ind_per_island):
116             if j == 0:
117                 new_pop[i][j] = pop[i][max_fitness_index[i]]
118             else:
119                 deme_dict = dict(zip(pop[i], fitness[i]))
120                 new_pop[i][j] = roulette_wheel_pop(
121                     pop[i], get_probability_list(deme_dict), 1)
122     return new_pop
123
124
125
126 def mutation(chosen_one):
127     child_list = []
128     single_deme = []
129     offsprings = []
130     for i in range(demes):
131         single_deme = []
132
133         for j in range(ind_per_island):
134
135             child_list = []
136             if j == 0:
137                 child_list.append(chosen_one[i][j])
138                 child = "".join(child_list)
139             else:
140                 for k in range(2 * number_of_sites):
141                     if (random.random() < mutation_rate):
142                         if chosen_one[i][j][k] == '1':
143                             new_random_character = '0'
144
145                     elif chosen_one[i][j][k] == '0':
146                         new_random_character = '1'
147
148                 child_list.append(new_random_character)
149                 child = "".join(child_list)
150             else:
151                 child_list.append(chosen_one[i][j][k])
152                 child = "".join(child_list)
153
154             single_deme.append(child)
155             offsprings.append(single_deme)
156     return offsprings
157
158
159 def max_fitness_index(fitness):
160     max_index = []
161     zzz = []
162     for i in range(demes):
163         max_value = max(fitness[i])
164         zzz.append(max_value)
165         max_index.append(fitness[i].index(max_value))
166
167     return max_index
168
169
170 # def replacement(pop, fitness_index, mutated):
171 #     #function to retain the highest fitness and also to replace one of the individual in the
172 #     for i in range(demes):
173 #         pop[i] = choice([
174 #             i for i in range(0, ind_per_island) if i not in [fitness_index[i]]
175 #             ]) = mutated[i]
176 #     return pop
177
178
179 def migration(new_generation):
180     # randomly selected individual and randomly selected deme
181     for z in range(demes):
182         # random_selected_individual_index = random.randint(0, ind_per_island-1)
183         # random_selected_deme_index = random.randint(0, demes-1)
184         # random_individual_location = random.randint(0, ind_per_island-1)
185         # random_selected_deme_location = random.randint(0, demes - 1)
186
187         random_selected_individual_index = random.randint(0, demes - 1)
188         random_selected_deme_index = random.randint(0, demes - 1)
189         random_individual_location = random.randint(1, demes - 1)
190         new_generation[z][random_individual_location] = new_generation[

```

```

192         random_selected_deme_index][random_selected_individual_index]
193     # print(new_generation)
194     return new_generation
195
196
197 #main program
198 # for n in range (1,8):
199     # number_of_sites = n*10
200
201 while simulation != 30:
202     R_noise = Rij()
203     generation = 0
204     total_population = pop_generator(pop_size)
205     # total_population = [['11111111111111111111' for i in range(ind_per_island)] for j in range(demes)]
206     zzz = R_noise[number_of_sites - 1][number_of_sites - 1] * (
207         2**(number_of_sites) + 2**(number_of_sites))
208     while max_fitness != zzz and generation<2000:
209         fitness, max_fitness = zip_fitness_eval(total_population, R_noise)
210         maximum_fitness_index = max_fitness_index(fitness)
211         chosen_ones = fitness_proportionate_selection(total_population, fitness,
212             maximum_fitness_index)
213
214         mutated = mutation(chosen_ones)
215
216         new_migrated_generation = migration(mutated)
217
218         generation += 1
219         print(generation)
220         total_population = new_migrated_generation
221         mutated = []
222
223     # if max_fitness == R_noise[number_of_sites - 1][number_of_sites - 1] * (
224     #     2**(number_of_sites) + 2**(number_of_sites)):
225     #     print('hamkachan diuleilou mou waste of time')
226
227 if generation<=2000:
228     sim.append(generation)
229     simulation +=1
230
231 print(sim)
232
233 print(sim)
234

```


crossover.py

```

1 from audiopop import cross
2 from operator import index
3 import random
4 from re import X
5 from cv2 import split
6 from numpy import number
7 from random import choice, randrange
8
9 # constant parameter
10 pop_size = 400
11 demes = 20
12 number_of_sites = 60
13 mutation_rate = 1 / (2 * number_of_sites)
14 ind_per_island = int(pop_size / demes)
15 # =====
16
17 # initialization
18 pop = []
19 R_noise = []
20 fitness = []
21 new_pop = [[None for i in range(ind_per_island)] for j in range(demes)]
22 probability = []
23 mutated = []
24 max_prob_index = []
25 sim = []
26 max_fitness = 0
27 generation = 0
28 counter = 0
29 simulation = 0
30 # =====
31
32 def Rij():
33     #function to generate the different constant R for i,j
34     R_noise = [[random.uniform(0.5, 1.0) for i in range(number_of_sites)]
35                for j in range(number_of_sites)]
36     return R_noise
37
38 def rand_key(number_of_sites):
39     # function to generate individual with 2n binary
40     length = 2 * number_of_sites
41     individual = ""
42     for i in range(length):
43         temp = str(random.randint(0, 1))
44         individual += temp
45     return (individual)
46
47
48
49
50 def pop_generator(pop_size):
51     pop = []
52     # function to generate 400 population of 2n binary
53     for i in range(pop_size):
54         random_str = rand_key(number_of_sites)
55         pop.append(random_str)
56     split_lists = [
57         pop[x:x + ind_per_island]
58         for x in range(0, len(pop), ind_per_island)
59     ]
60     return split_lists
61
62
63 def zip_fitness_eval(population, R_noise):
64     maxi_fitness = 0
65     fitness = []
66     for i in range(demes):
67         for j in range(ind_per_island):
68             gene_i = population[i][j][:number_of_sites].count('1')
69             gene_j = population[i][j][number_of_sites:].count('1')
70             # for k in range(demes):
71             #     for a in range(int(pop_size / demes)):
72             #         for z in range(2 * number_of_sites):
73             #             if pop[k][a][z] == '1' and z < number_of_sites:
74             #                 i += 1
75             #             if pop[k][a][z] == '1' and z >= number_of_sites:
76             #                 j += 1
77             fitness_equation = R_noise[gene_i - 1][gene_j - 1] * (2**gene_i +
78                                                                    2**gene_j)
79             fitness.append(fitness_equation)
80
81     maxi_fitness = max(fitness)
82
83     split_lists = [
84         fitness[x:x + ind_per_island]
85         for x in range(0, len(fitness), ind_per_island)
86     ]
87     return split_lists, maxi_fitness
88
89
90
91 def get_probability_list(deme_dict):
92     fitness = deme_dict.values()
93     total_fit = float(sum(fitness))
94     relative_fitness = [f / total_fit for f in fitness]
95     probabilities = [
96
97         sum(relative_fitness[:i + 1]) for i in range(len(relative_fitness))
98     ]
99     return probabilities
100
101
102 def roulette_wheel_pop(population, probabilities, number):
103     parents = []
104     for n in range(number):
105         r = random.random()
106         for (i, individual) in enumerate(population):
107             if r <= probabilities[i]:
108                 parents.append(individual)
109                 break
110     return parents
111
112
113 def crossover(father, mother):
114     child = []
115     crossover_point = random.randint(0, (2*number_of_sites) - 1)
116     for i in range(2 * number_of_sites):
117         # if random.randint(0, 1) == 0:
118         if i < crossover_point:
119             child.append(father[i])
120         else:
121             child.append(mother[i])
122         # else:
123         #     if i < crossover_point:
124         #         child.append(mother[i])
125         #     else:
126         #         child.append(father[i])
127     xover = "".join(child)
128
129     return xover
130
131
132 def mutation(chosen_one):
133     child_list = []
134     single_deme = []
135     offsprings = []
136     for i in range(demes):
137         single_deme = []
138         for j in range(ind_per_island):
139             child_list = []
140             if j == 0:
141                 child_list.append(chosen_one[i][j])
142
143                 child = "".join(child_list)
144                 else:
145                     for k in range(2 * number_of_sites):
146                         if (random.random() < mutation_rate):
147                             if chosen_one[i][j][k] == '1':
148                                 new_random_character = '0'
149                             elif chosen_one[i][j][k] == '0':
150                                 new_random_character = '1'
151                             child_list.append(new_random_character)
152                             child = "".join(child_list)
153                         else:
154                             child_list.append(chosen_one[i][j][k])
155                             child = "".join(child_list)
156                     single_deme.append(child)
157                     offsprings.append(single_deme)
158                 return offsprings
159
160
161 def max_fitness_index(fitness):
162     max_index = []
163     zzz = []
164     for i in range(demes):
165         max_value = max(fitness[i])
166         zzz.append(max_value)
167         max_index.append(fitness[i].index(max_value))
168     # print(zzz)
169     return max_index
170
171
172 def replacement(pop, fitness_index, mutated):
173     # function to retain the highest fitness and also to replace one of the individual in the deme
174     for i in range(demes):
175         pop[i][choice(
176             [i for i in range(0, ind_per_island) if i not in [fitness_index[i]]]
177             )] = mutated[i]
178     return pop
179
180
181 def migration(new_generation):
182     # randomly selected individual and randomly selected deme
183     for z in range(demes):
184         random_selected_individual_index = random.randint(0, demes - 1)
185         random_selected_deme_index = random.randint(0, demes - 1)

```

```

193     random_individual_location = random.randint(1, demes - 1)
194     new_generation[z][random_individual_location] = new_generation[
195         random_selected_deme_index][random_selected_individual_index]
196     # print(new_generation)
197     return new_generation
198
199 while simulation != 10:
200     generation = 0
201     R_noise = Rij()
202     total_population = pop_generator(pop_size)
203     # total_population = [['11111111111111111111' for i in range(ind_per_island)] for j in range(demes)]
204     zzz = R_noise[number_of_sites - 1][number_of_sites - 1] * (
205         2**(number_of_sites) + 2**(number_of_sites))
206
207     while max_fitness != zzz and generation <= 100:
208         # print('fitness calculation landscape')
209         fitness, max_fitness = zip_fitness_eval(total_population, R_noise)
210         # print(len(fitness))
211         # print(max_fitness)
212
213         # print("maximum fitness index")
214         index_fit = max_fitness_index(fitness)
215         # print(len(maximum_fitness_index))
216
217         for i in range(demes):
218             for j in range(ind_per_island):
219                 if j == 0:
220                     new_pop[i][j] = total_population[i][index_fit[i]]
221                 else:
222                     deme_selected = random.randint(0, demes - 1)
223                     deme_dict = dict(zip(total_population[deme_selected], fitness[deme_selected]))
224                     father = roulette_wheel_pop(total_population[deme_selected], get_probability_list(deme_dict), 1)
225
226                     deme_selected = random.randint(0, demes - 1)
227                     deme_dict = dict(zip(total_population[deme_selected], fitness[deme_selected]))
228                     mother = roulette_wheel_pop(total_population[deme_selected], get_probability_list(deme_dict), 1)
229
230                     new_pop[i][j] = crossover(father, mother)
231
232         # print("mutation")
233         # mutated size is 20x20
234         mutated = mutation(new_pop)
235         # print(mutated[0][0])
236         # print(mutated)

```

```

241     # print('migrated')
242     new_migrated_generation = migration(mutated)
243     # print(len(new_migrated_generation))
244
245     generation += 1
246     # print(generation)
247
248     total_population = new_migrated_generation
249     # total_population = new_pop
250     mutated = []
251
252     # for i in range(demes):
253     #     print(total_population[i][0])
254     #     if max_fitness == R_noise[number_of_sites - 1][number_of_sites - 1] * (
255     #         2**(number_of_sites) + 2**(number_of_sites)):
256     #         print('hamkachan diuleilou mou waste of time')
257
258     if generation <= 100:
259         sim.append(generation)
260         simulation += 1
261
262     print(sim)
263
264     counter += 1
265
266     print(sim)
267     print(counter)

```

Tournament_selection.py

```

1  from audioop import cross
2  from calendar import monthrange
3  from operator import index
4  import random
5  from re import X
6  from cv2 import split
7  from numpy import number
8  from random import choice, randrange
9
10 # constant parameter
11 pop_size = 400
12 demes = 20
13 number_of_sites = 10
14 mutation_rate = 1 / (2 * number_of_sites)
15 ind_per_island = int(pop_size / demes)
16 # *****
17
18 # initialization
19 pop = []
20 R_noise = []
21 fitness = []
22 new_pop = [[None for i in range(ind_per_island)] for j in range(demes)]
23 probability = []
24 mutated = []
25 max_prob_index = []
26 sim = []
27 total = []
28 counter = 0
29 max_fitness = 0
30 generation = 0
31 simulation = 0
32 # *****
33
34 def Rij():
35     #function to generate the different constant R for i,j
36     R_noise = [[random.uniform(0.5, 1.0) for i in range(number_of_sites)]
37                for j in range(number_of_sites)]
38     return R_noise
39
40
41 def rand_key(number_of_sites):
42     # function to generate individual with 2n binary
43     length = 2 * number_of_sites
44     individual = ""
45     for i in range(length):
46         temp = str(random.randint(0, 1))
47         individual += temp
48
49     return (individual)
50
51
52 def pop_generator(pop_size):
53     pop = []
54     # function to generate 400 population of 2n binary
55     for i in range(pop_size):
56         random_str = rand_key(number_of_sites)
57         pop.append(random_str)
58         split_lists = [
59             pop[x:x + ind_per_island]
60             for x in range(0, len(pop), ind_per_island)
61         ]
62     return split_lists
63
64
65 def zip_fitness_eval(population, R_noise):
66     maxi_fitness = 0
67     fitness = []
68     for i in range(demes):
69         for j in range(ind_per_island):
70             gene_i = population[i][j][:number_of_sites].count('1')
71             gene_j = population[i][j][number_of_sites:].count('1')
72
73             fitness_equation = R_noise[gene_i - 1][gene_j - 1] * (2**gene_i +
74             2**gene_j)
75             fitness.append(fitness_equation)
76
77     maxi_fitness = max(fitness)
78
79     split_lists = [
80         fitness[x:x + ind_per_island]
81         for x in range(0, len(fitness), ind_per_island)
82     ]
83     return split_lists, maxi_fitness
84
85
86 def tournament_selection(population, fitness, selection_pressure, z):
87     # both in the size of 20x20
88     fittest = 0
89     temp = 0
90     for i in range(selection_pressure):
91         niama = random.randint(0, ind_per_island-1)
92         fittest = fitness[z][niama]
93         if fittest > temp:
94             index = [z, niama]
95             temp = fittest
96
97     return population[index[0]][index[1]]
98
99 def crossover(father, mother):
100     child = []
101     crossover_point = random.randint(0, (2 * number_of_sites) - 1)
102     for i in range(2 * number_of_sites):
103         # if random.randint(0, 1) == 0:
104         if i < crossover_point:
105             child.append(father[i])
106         else:
107             child.append(mother[i])
108     # else:
109     #     if i < crossover_point:
110     #         child.append(mother[0][i])
111     #     else:
112     #         child.append(father[0][i])
113     xover = "".join(child)
114
115     return xover
116
117
118 def mutation(chosen_one):
119     child_list = []
120     single_deme = []
121     offsprings = []
122     for i in range(demes):
123         single_deme = []
124
125         for j in range(ind_per_island):
126
127             child_list = []
128             if j == 0:
129                 child_list.append(chosen_one[i][j])
130                 child = "".join(child_list)
131             else:
132                 for k in range(2 * number_of_sites):
133                     if (random.random() < mutation_rate):
134                         if chosen_one[i][j][k] == '1':
135                             new_random_character = '0'
136
137                         elif chosen_one[i][j][k] == '0':
138                             new_random_character = '1'
139
140                 child_list.append(new_random_character)
141                 child = "".join(child_list)
142             else:
143                 child_list.append(chosen_one[i][j][k])
144
145         child = "".join(child_list)
146
147         single_deme.append(child)
148         offsprings.append(single_deme)
149     return offsprings
150
151
152 def max_fitness_index(fitness):
153     max_index = []
154     zzz = []
155     for i in range(demes):
156         max_value = max(fitness[i])
157         zzz.append(max_value)
158         max_index.append(fitness[i].index(max_value))
159
160     return max_index
161
162
163 def migration(new_generation):
164
165     # randomly selected individual and randomly selected deme
166     for z in range(demes):
167         random_selected_individual_index = random.randint(
168             0, ind_per_island - 1)
169         random_selected_deme_index = random.randint(0, demes - 1)
170         random_individual_location = random.randint(1, ind_per_island - 1)
171         new_generation[z][random_individual_location] = new_generation[
172             random_selected_deme_index][random_selected_individual_index]
173     # print(new_generation)
174     return new_generation
175
176
177 while simulation != 30:
178     generation = 0
179     R_noise = Rij()
180     selection_pressure = 10
181     total_population = pop_generator(pop_size)
182     zzz = R_noise[number_of_sites - 1][number_of_sites - 1] * (
183         2**(number_of_sites) + 2**(number_of_sites))
184
185     while max_fitness != zzz and generation <= 500:
186         fitness, max_fitness = zip_fitness_eval(total_population, R_noise,)
187         index_fit = max_fitness_index(fitness)
188
189         for i in range(demes):
190             for j in range(ind_per_island):
191                 if j == 0:

```

```

193         new_pop[i][j] = total_population[i][index_fit[i]]
194
195     else:
196         father = tournament_selection(total_population, fitness, selection_pressure, i)
197         mother = tournament_selection(total_population, fitness, selection_pressure, i)
198         new_pop[i][j] = crossover(father, mother)
199
200     mutated = mutation(new_pop)
201     new_migrated_generation = migration(mutated)
202     generation += 1
203     total_population = new_migrated_generation
204     # total_population = new_pop
205     mutated = []
206
207     if generation == 500 or max_fitness == zzz:
208         counter += 1
209
210
211     if generation <= 500:
212         sim.append(generation)
213         simulation += 1
214
215     print(sim)
216
217     total.append(sim)
218
219
220     print(total)
221     print(counter)

```