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, $G =< g_1, g_2, \dots \dots g_{2n} >$. 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, $\{g_1, g_2, \dots \dots g_n\}$ whereas j is the number of 1s in the second half of the genotype, $\{g_{n+1}, g_{n+2}, \dots \dots g_{2n}\}$. 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 u 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

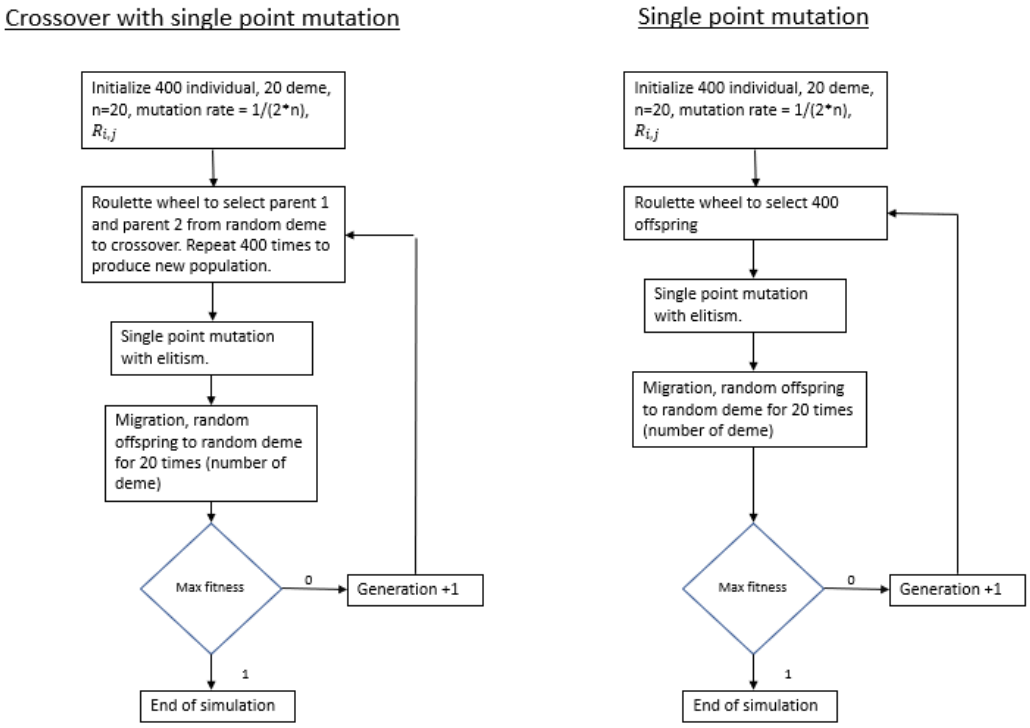

Figure 1: Both crossover and non-crossover algorithm with a single point mutation implemented.

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 and 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 site 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.

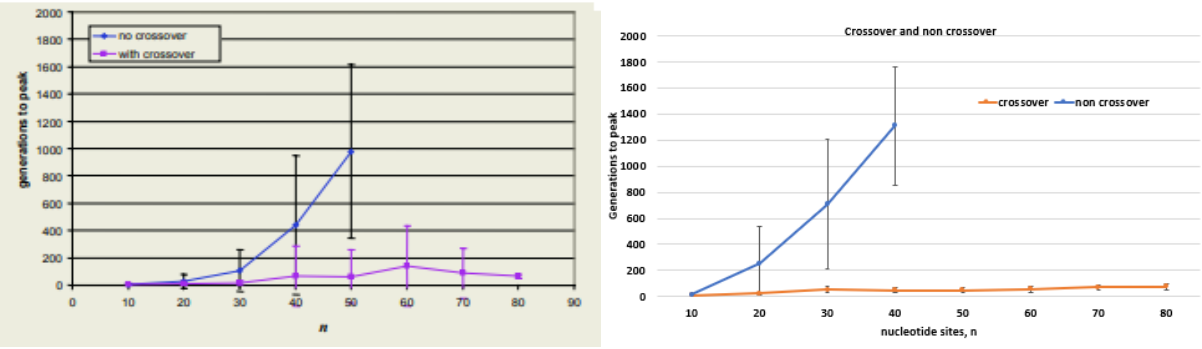

Figure 2: Simulated result of figure 3 in [1]. *left) Original result, right) Reimplemented result.*

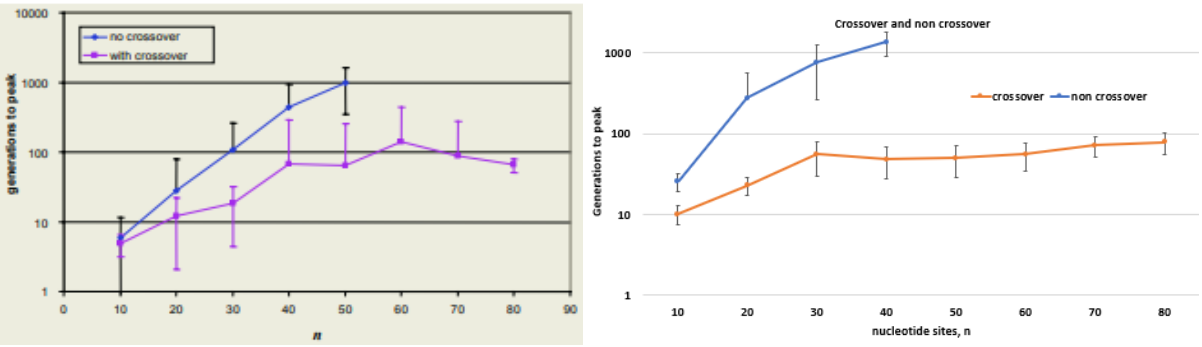

Figure 3: Simulated logscale result of figure 3 in [1]. *left) Original result, right) Reimplemented result.*

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 manged 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

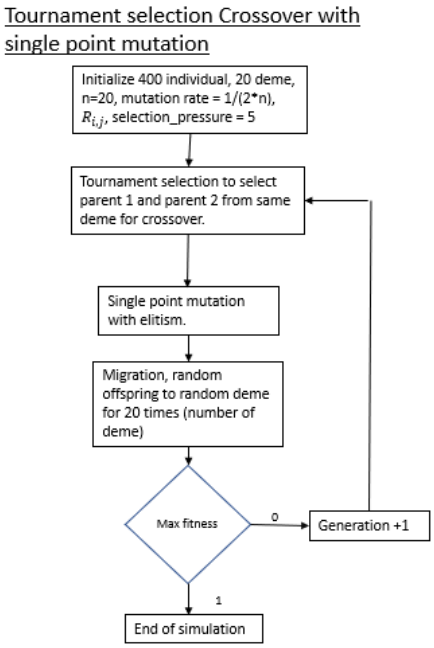

Figure 4: Tournament selection crossover with single point mutation with a selection pressure of 5.

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

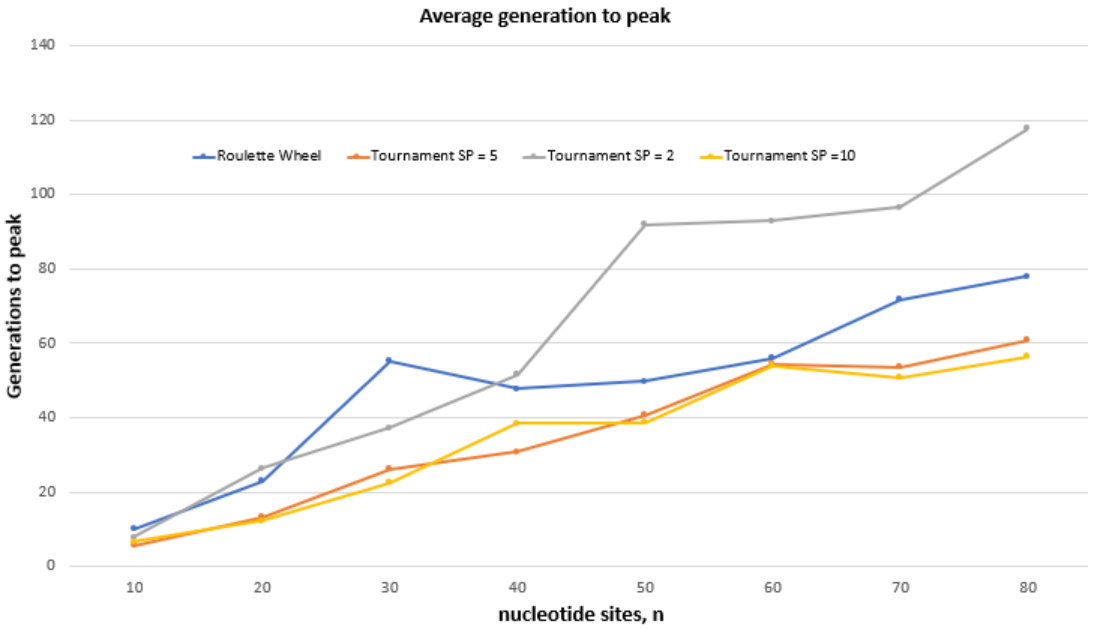

Figure 5: Graph of Generation to peak against the nucleotide sites, n. Roulette wheel and tournament selection algorithm were compared. The selection pressure of tournament selection was tuned to be 2, 5, and 10 accordingly.

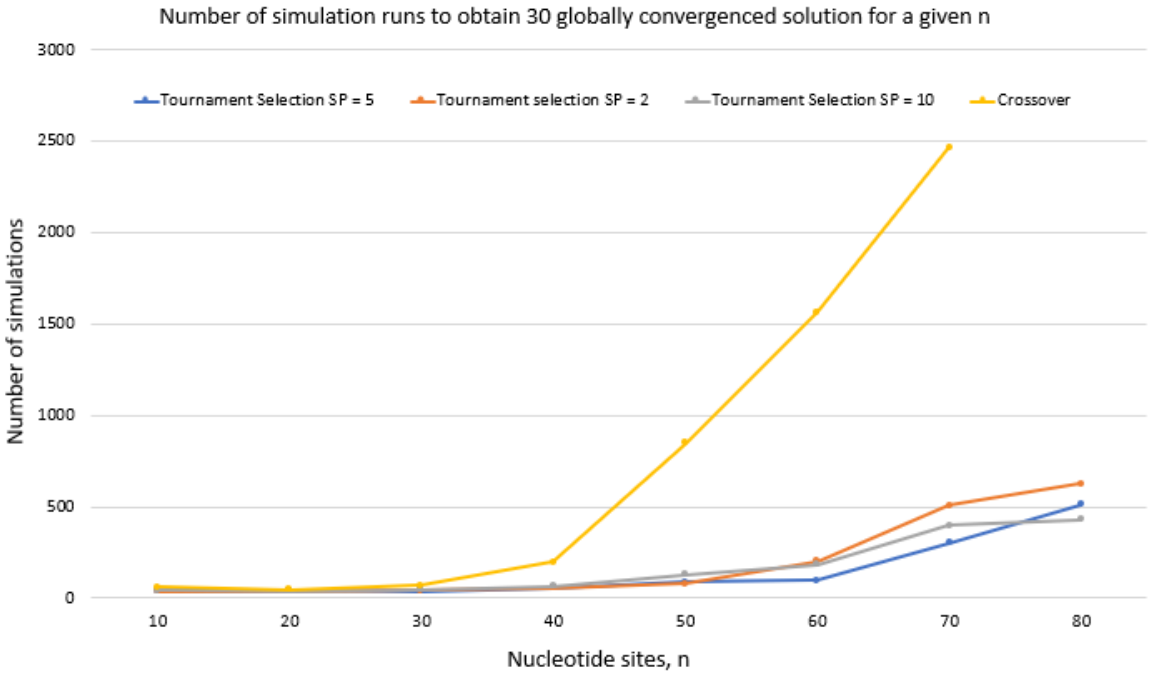

Figure 6: Total number of simulations runs required to obtain 30 points of globally converged solution for a given nucleotide site.

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 reimplementation 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

```python
from operator import index
import random
from telnetlib import NEW_ENVIRON
from cv2 import split
from numpy import number
from random import choice

# constant parameter
pop_size = 400
demes = 20
number_of_sites = 40
mutation_rate = 1/ (2 * number_of_sites)
ind_per_island = int(400 / 20)
# ################################

# initialization
pop = []
R_noise = []
fitness = []
new_pop = []
probability = []
mutated = []
max_prob_index = []
max_fitness = 0
generation = 0
sim = []
simulation = 0
# ################################


def Rij():
    #function to generate the different constant R for i,j
    R_noise = [[random.uniform(0.5, 1.0) for i in range(number_of_sites)]
               for j in range(number_of_sites)]
    return R_noise


def rand_key(number_of_sites):
    # function to generate individual with 2n binary
    length = 2 * number_of_sites
    individual = ""
    for i in range(length):
        temp = str(random.randint(0, 1))
        # temp = '1'
        individual += temp

    return (individual)


def pop_generator(pop_size):
    pop = []
    # function to generate 400 population of 2n binary
    for i in range(pop_size):
        random_str = rand_key(number_of_sites)
        pop.append(random_str)
        split_lists = [
            pop[x:x + ind_per_island]
            for x in range(0, len(pop), ind_per_island)
        ]
    return split_lists


def zip_fitness_eval(population, R_noise):
    maxi_fitness = 0
    fitness = []
    for i in range(demes):
        for j in range(ind_per_island):
            gene_i = population[i][j][:number_of_sites].count('1')
            gene_j = population[i][j][number_of_sites:].count('1')
            # for k in range(demes):
            #     for a in range(int(pop_size / demes)):
            #         for z in range(2 * number_of_sites):
            #             if pop[k][a][z] == '1' and z < number_of_sites:
            #                 i += 1
            #             if pop[k][a][z] == '1' and z >= number_of_sites:
            #                 j += 1

            fitness_equation = R_noise[gene_i - 1][gene_j - 1] * (2**gene_i +
                                                                  2**gene_j)
            fitness.append(fitness_equation)

    maxi_fitness = max(fitness)

    split_lists = [
        fitness[x:x + ind_per_island]
        for x in range(0, len(fitness), ind_per_island)
    ]
    return split_lists, maxi_fitness


def get_probability_list(deme_dict):
    fitness = deme_dict.values()
    total_fit = float(sum(fitness))
    relative_fitness = [f / total_fit for f in fitness]
    probabilities = [
        sum(relative_fitness[:i + 1]) for i in range(len(relative_fitness))
    ]
    return probabilities


def roulette_wheel_pop(population, probabilities, number):
    for n in range(number):
        r = random.random()
        for (i, individual) in enumerate(population):
            if r <= probabilities[i]:
                individual = individual
                break
    return individual


def fitness_proportionate_selection(pop, fitness, max_fitness_index):
    new_pop = [[None for i in range(ind_per_island)] for j in range(demes)]
    # print(new_pop)
    for i in range(demes):
        for j in range(ind_per_island):
            if j == 0:
                new_pop[i][j] = pop[i][max_fitness_index[i]]

            else:
                deme_dict = dict(zip(pop[i], fitness[i]))
                new_pop[i][j] = roulette_wheel_pop(
                    pop[i], get_probability_list(deme_dict), 1)
    return new_pop


def mutation(chosen_one):
    child_list = []
    single_deme = []
    offsprings = []
    for i in range(demes):
        single_deme = []

        for j in range(ind_per_island):

            child_list = []
            if j == 0:
                child_list.append(chosen_one[i][j])
                child = "".join(child_list)
            else:
                for k in range(2 * number_of_sites):
                    if (random.random() < mutation_rate ):
                        if chosen_one[i][j][k] == '1':
                            new_random_character = '0'

                    elif chosen_one[i][j][k] == '0':
                        new_random_character = '1'

                    child_list.append(new_random_character)
                    child = "".join(child_list)
                else:
                    child_list.append(chosen_one[i][j][k])
                    child = "".join(child_list)

            single_deme.append(child)
        offsprings.append(single_deme)
    return offsprings


def max_fitness_index(fitness):
    max_index = []
    zzz = []
    for i in range(demes):
        max_value = max(fitness[i])
        zzz.append(max_value)
        max_index.append(fitness[i].index(max_value))

    return max_index


# def replacement(pop, fitness_index, mutated):
#     #function to retain the highest fitness and also to replace one of the individual in the d
#     for i in range(demes):
#         pop[i][choice([
#             i for i in range(0, ind_per_island) if i not in [fitness_index[i]]
#         ])] = mutated[i]
#     return pop


def migration(new_generation):

    # randomly selected individual and randomly selected deme
    for z in range(demes):
        # random_selected_individual_index = random.randint(0,ind_per_island-1)
        # random_selected_deme_index = random.randint(0, demes-1)
        # random_individual_location = random.randint(0, ind_per_island-1)
        # random_selected_deme_location = random.randint(0, demes - 1)

        random_selected_individual_index = random.randint(0, demes - 1)
        random_selected_deme_index = random.randint(0, demes - 1)
        random_individual_location = random.randint(1, demes - 1)
        new_generation[z][random_individual_location] = new_generation[
```

```
            random_selected_deme_index][random_selected_individual_index]
    # print(new_generation)
    return new_generation


#main program
# for n in range (1,8):
    # number_of_sites = n*10

while simulation != 30:
    R_noise = Rij()
    generation = 0
    total_population = pop_generator(pop_size)
    # total_population = [['1111111111111111111' for i in range(ind_per_island)] for j in range(demes)]
    zzz = R_noise[number_of_sites - 1][number_of_sites - 1] * (
        2**(number_of_sites) + 2**(number_of_sites))
    while max_fitness != zzz and generation<=2000:
        fitness, max_fitness = zip_fitness_eval(total_population, R_noise)
        maximum_fitness_index = max_fitness_index(fitness)
        chosen_ones = fitness_proportionate_selection(total_population, fitness,
                                                      maximum_fitness_index)

        mutated = mutation(chosen_ones)

        new_migrated_generation = migration(mutated)

        generation += 1
        print(generation)
        total_population = new_migrated_generation
        mutated = []

        # if max_fitness == R_noise[number_of_sites - 1][number_of_sites - 1] * (
        #         2**(number_of_sites) + 2**(number_of_sites)):
        #     print('hamkachan diuleilou mou waste of time')

    if generation<=2000:
        sim.append(generation)
        simulation +=1

    print(sim)

print(sim)

```

crossover.py

```python
from audioop import cross
from operator import index
import random
from re import X
from cv2 import split
from numpy import number
from random import choice, randrange

# constant parameter
pop_size = 400
demes = 20
number_of_sites = 60
mutation_rate = 1 / (2 * number_of_sites)
ind_per_island = int(pop_size / demes)
# ##############################

# initialization
pop = []
R_noise = []
fitness = []
new_pop = [[None for i in range(ind_per_island)] for j in range(demes)]
probability = []
mutated = []
max_prob_index = []
sim = []
max_fitness = 0
generation = 0
counter = 0
simulation = 0
# ##############################


def Rij():
    #function to generate the different constant R for i,j
    R_noise = [[random.uniform(0.5, 1.0) for i in range(number_of_sites)]
               for j in range(number_of_sites)]
    return R_noise


def rand_key(number_of_sites):
    # function to generate individual with 2n binary
    length = 2 * number_of_sites
    individual = ""
    for i in range(length):
        temp = str(random.randint(0, 1))
        individual += temp

    return (individual)


def pop_generator(pop_size):
    pop = []
    # function to generate 400 population of 2n binary
    for i in range(pop_size):
        random_str = rand_key(number_of_sites)
        pop.append(random_str)
        split_lists = [
            pop[x:x + ind_per_island]
            for x in range(0, len(pop), ind_per_island)
        ]
    return split_lists


def zip_fitness_eval(population, R_noise):
    maxi_fitness = 0
    fitness = []
    for i in range(demes):
        for j in range(ind_per_island):
            gene_i = population[i][j][:number_of_sites].count('1')
            gene_j = population[i][j][number_of_sites:].count('1')
            # for k in range(demes):
            #     for a in range(int(pop_size / demes)):
            #         for z in range(2 * number_of_sites):
            #             if pop[k][a][z] == '1' and z < number_of_sites:
            #                 i += 1
            #             if pop[k][a][z] == '1' and z >= number_of_sites:
            #                 j += 1

            fitness_equation = R_noise[gene_i - 1][gene_j - 1] * (2**gene_i +
                                                                  2**gene_j)
            fitness.append(fitness_equation)

    maxi_fitness = max(fitness)

    split_lists = [
        fitness[x:x + ind_per_island]
        for x in range(0, len(fitness), ind_per_island)
    ]
    return split_lists, maxi_fitness


def get_probability_list(deme_dict):
    fitness = deme_dict.values()
    total_fit = float(sum(fitness))
    relative_fitness = [f / total_fit for f in fitness]
    probabilities = [
        sum(relative_fitness[:i + 1]) for i in range(len(relative_fitness))
    ]
    return probabilities


def roulette_wheel_pop(population, probabilities, number):
    parents = []
    for n in range(number):
        r = random.random()
        for (i, individual) in enumerate(population):
            if r <= probabilities[i]:
                parents.append(individual)
                break
    return parents


def crossover(father, mother):
    child = []
    crossover_point = random.randint(0, (2*number_of_sites) - 1)
    for i in range(2 * number_of_sites):
        # if random.randint(0, 1) == 0:
        if i < crossover_point:

            child.append(father[0][i])
        else:
            child.append(mother[0][i])
        # else:
        #     if i < crossover_point:
        #         child.append(mother[0][i])
        #     else:
        #         child.append(father[0][i])
    xover = "".join(child)

    return xover


def mutation(chosen_one):
    child_list = []
    single_deme = []
    offsprings = []
    for i in range(demes):
        single_deme = []

        for j in range(ind_per_island):

            child_list = []
            if j == 0:
                child_list.append(chosen_one[i][j])
                child = "".join(child_list)
            else:
                for k in range(2 * number_of_sites):
                    if (random.random() < mutation_rate):
                        if chosen_one[i][j][k] == '1':
                            new_random_character = '0'

                        elif chosen_one[i][j][k] == '0':
                            new_random_character = '1'

                        child_list.append(new_random_character)
                        child = "".join(child_list)
                    else:
                        child_list.append(chosen_one[i][j][k])
                        child = "".join(child_list)

            single_deme.append(child)
        offsprings.append(single_deme)
    return offsprings


def max_fitness_index(fitness):
    max_index = []
    zzz = []
    for i in range(demes):
        max_value = max(fitness[i])
        zzz.append(max_value)
        max_index.append(fitness[i].index(max_value))

    # print(zzz)
    return max_index


# def replacement(pop, fitness_index, mutated):
#     #function to retain the highest fitness and also to replace one of the individual in the deme
#     for i in range(demes):
#         pop[i][choice([
#             i for i in range(0, ind_per_island) if i not in [fitness_index[i]]
#         ])] = mutated[i]
#     return pop


def migration(new_generation):

    # randomly selected individual and randomly selected deme
    for z in range(demes):
        random_selected_individual_index = random.randint(0, demes - 1)
        random_selected_deme_index = random.randint(0, demes - 1)
```

```
        random_individual_location = random.randint(1, demes - 1)
        new_generation[z][random_individual_location] = new_generation[
            random_selected_deme_index][random_selected_individual_index]
    # print(new_generation)
    return new_generation

while simulation != 10:
    generation = 0
    R_noise = Rij()
    total_population = pop_generator(pop_size)
    # total_population = [['1111111111111111111' for i in range(ind_per_island)] for j in range(demes)]
    zzz = R_noise[number_of_sites - 1][number_of_sites - 1] * (
        2**(number_of_sites) + 2**(number_of_sites))

    while max_fitness != zzz and generation<=100:
        # print('fitness calculation landscape')
        fitness, max_fitness = zip_fitness_eval(total_population, R_noise)
        #print(len(fitness))``
        # print(max_fitness)

        #print("maximum fitness index")
        index_fit = max_fitness_index(fitness)
        #print(len(maximum_fitness_index))


        for i in range(demes):
            for j in range(ind_per_island):
                if j == 0:
                    new_pop[i][j] = total_population[i][index_fit[i]]

                else:
                    deme_selected = random.randint(0, demes - 1)
                    deme_dict = dict(zip(total_population[deme_selected], fitness[deme_selected]))
                    father = roulette_wheel_pop(total_population[deme_selected],get_probability_list(deme_dict), 1)


                    deme_selected = random.randint(0, demes - 1)
                    deme_dict = dict(zip(total_population[deme_selected], fitness[deme_selected]))
                    mother = roulette_wheel_pop(total_population[deme_selected],get_probability_list(deme_dict), 1)


                    new_pop[i][j] = crossover(father, mother)

        #print("mutation")
        #mutated size is 20x20
        mutated = mutation(new_pop)
        # print(mutated[0][0])
        # print(mutated)
```

```
        #print('migrated')
        new_migrated_generation = migration(mutated)
        #print(len(new_migrated_generation))

        generation += 1
        # print(generation)


        total_population = new_migrated_generation
        # total_population = new_pop
        mutated = []

        # for i in range(demes):
        #     print(total_population[i][0])
        # if max_fitness == R_noise[number_of_sites - 1][number_of_sites - 1] * (
        #         2**(number_of_sites) + 2**(number_of_sites)):
        #     print('hamkachan diuleilou mou waste of time')

    if generation<=100:
        sim.append(generation)
        simulation +=1

    print(sim)

    counter +=1

print(sim)
print(counter)
```

Tournament_selection.py

```python
from audioop import cross
from calendar import monthrange
from operator import index
import random
from re import X
from cv2 import split
from numpy import number
from random import choice, randrange

# constant parameter
pop_size = 400
demes = 20
number_of_sites = 10
mutation_rate = 1 / (2 * number_of_sites)
ind_per_island = int(pop_size / demes)
# ################################

# initialization
pop = []
R_noise = []
fitness = []
new_pop = [[None for i in range(ind_per_island)] for j in range(demes)]
probability = []
mutated = []
max_prob_index = []
sim = []
total = []
counter =0
max_fitness = 0
generation = 0
simulation = 0
# ################################


def Rij():
    #function to generate the different constant R for i,j
    R_noise = [[random.uniform(0.5, 1.0) for i in range(number_of_sites)]
               for j in range(number_of_sites)]
    return R_noise


def rand_key(number_of_sites):
    # function to generate individual with 2n binary
    length = 2 * number_of_sites
    individual = ""
    for i in range(length):
        temp = str(random.randint(0, 1))
        individual += temp

    return (individual)


def pop_generator(pop_size):
    pop = []
    # function to generate 400 population of 2n binary
    for i in range(pop_size):
        random_str = rand_key(number_of_sites)
        pop.append(random_str)
        split_lists = [
            pop[x:x + ind_per_island]
            for x in range(0, len(pop), ind_per_island)
        ]
    return split_lists


def zip_fitness_eval(population, R_noise):
    maxi_fitness = 0
    fitness = []
    for i in range(demes):
        for j in range(ind_per_island):
            gene_i = population[i][j][:number_of_sites].count('1')
            gene_j = population[i][j][number_of_sites:].count('1')

            fitness_equation = R_noise[gene_i - 1][gene_j - 1] * (2**gene_i +
                                                                  2**gene_j)
            fitness.append(fitness_equation)

    maxi_fitness = max(fitness)

    split_lists = [
        fitness[x:x + ind_per_island]
        for x in range(0, len(fitness), ind_per_island)
    ]
    return split_lists, maxi_fitness


def tournament_selection(population, fitness, selection_pressure,z):
    # both in the size of 20x20
    fittest = 0
    temp = 0
    for i in range(selection_pressure):
        niama = random.randint(0,ind_per_island-1)
        fittest = fitness[z][niama]
        if fittest>temp:
            index = [z,niama]
            temp = fittest
    return population[index[0]][index[1]]

def crossover(father, mother):
    child = []
    crossover_point = random.randint(0, (2 * number_of_sites) - 1)
    for i in range(2 * number_of_sites):
        # if random.randint(0, 1) == 0:
        if i < crossover_point:

            child.append(father[i])
        else:
            child.append(mother[i])
        # else:
        #     if i < crossover_point:
        #         child.append(mother[0][i])
        #     else:
        #         child.append(father[0][i])
    xover = "".join(child)

    return xover


def mutation(chosen_one):
    child_list = []
    single_deme = []
    offsprings = []
    for i in range(demes):
        single_deme = []

        for j in range(ind_per_island):

            child_list = []
            if j == 0:
                child_list.append(chosen_one[i][j])
                child = "".join(child_list)
            else:
                for k in range(2 * number_of_sites):
                    if (random.random() < mutation_rate):
                        if chosen_one[i][j][k] == '1':
                            new_random_character = '0'

                        elif chosen_one[i][j][k] == '0':
                            new_random_character = '1'

                        child_list.append(new_random_character)
                        child = "".join(child_list)
                    else:
                        child_list.append(chosen_one[i][j][k])
                child = "".join(child_list)

            single_deme.append(child)
        offsprings.append(single_deme)
    return offsprings


def max_fitness_index(fitness):
    max_index = []
    zzz = []
    for i in range(demes):
        max_value = max(fitness[i])
        zzz.append(max_value)
        max_index.append(fitness[i].index(max_value))

    return max_index


def migration(new_generation):

    # randomly selected individual and randomly selected deme
    for z in range(demes):
        random_selected_individual_index = random.randint(
            0, ind_per_island - 1)
        random_selected_deme_index = random.randint(0, demes - 1)
        random_individual_location = random.randint(1, ind_per_island - 1)
        new_generation[z][random_individual_location] = new_generation[
            random_selected_deme_index][random_selected_individual_index]
    # print(new_generation)
    return new_generation


while simulation != 30:

    generation = 0
    R_noise = Rij()
    selection_pressure= 10
    total_population = pop_generator(pop_size)
    zzz = R_noise[number_of_sites - 1][number_of_sites - 1] * (
        2**(number_of_sites) + 2**(number_of_sites))

    while max_fitness != zzz and generation <= 500:
        fitness, max_fitness = zip_fitness_eval(total_population, R_noise,)
        index_fit = max_fitness_index(fitness)

        for i in range(demes):
            for j in range(ind_per_island):
                if j == 0:
```

```
                    new_pop[i][j] = total_population[i][index_fit[i]]

                else:
                    father = tournament_selection(total_population, fitness, selection_pressure, i)
                    mother = tournament_selection(total_population, fitness, selection_pressure, i)
                    new_pop[i][j] = crossover(father, mother)

        mutated = mutation(new_pop)
        new_migrated_generation = migration(mutated)
        generation += 1
        total_population = new_migrated_generation
        # total_population = new_pop
        mutated = []

        if generation ==500 or max_fitness ==zzz:
            counter +=1


    if generation <= 500:
        sim.append(generation)
        simulation += 1

    print(sim)

total.append(sim)


print(total)
print(counter)
```