Electronics and Computer Science Faculty of Physical and Applied Sciences University of Southampton

Brian Formento

8 Jan 2018

COMP6026: Assignment 2

A Simple Two-Module Problem to Exemplify Building-Block Assembly Under Crossover

It is widely known that a genetic algorithm (GA) without crossover can outperform a GA with crossover in various landscapes and vice versa. However it is unclear in academic research why this is the case and what components of these algorithms contribute to such behaviour, some theories indicate the presence of building blocks, areas in the genome with a low defining length and short schema of above average fitness are responsible. It is widely accepted that crossover allows for an efficient reassembly of such blocks. However it not easy to build such intuition, hence it is often hard to teach such concepts. The aim of the paper is to explain using a simple landscape the benefits of crossover.

Individuals representation

this paper uses a two module fitness landscape where an individual is defined with a bitstring $G =< g1, g2, \ldots, g2n >$. Half (n) of such bitstring belonging to i and the later half belonging to j.

Fitness function

The fitness function is defined as:

$$f(G) = R(i,j)(2i + 2j)$$

where i and j are the sum of 1s in the left and right part of the bitstring respectively. R is a noise matrix used to add local maximas to model more realistic landscapes.

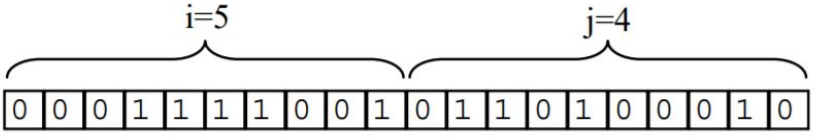

Fig1: Structure of one individual. All 1s are summed in the i and j section. They are used to calculate the fitness.

it is important to note that the global maxima won't necessarily be where G all 1s, due to the addition of R to the exponential nature of $(2^i + 2^j)$ there are cases where the highest function has a $0 \in G$ .This is prevalent in a 3D space a combination of low $R(-1,-1)$ with a high $R(-2,-1)$ will result in the maxima being at the latter position.

Used GA

To choose the individuals from their island the algorithm uses roulette wheel selection, this works by assigning every individual a probability of being selected, dependent on fitness. A cumulative distribution function (CDF) is then generated using these probabilities. A random value Rm is generated between 0 and $max(CDF)$. By definition, therefore, Rm has the highest probability of falling within the individual with the largest value.

Crossover methods

To solve the optimisation problem 3 algorithms have been used, mutation only, one point crossover (OPC) and uniform crossover. mutation only works by picking one parent, mutating each nucleotide by probability $\frac{1}{2n}$. The new individual is then added to the new population. With OPC a random location in the gene of two selected parents is selected, a 'cut' is carried out and a child generated from one of the two sections in both parents.

Results reimplementation

Both Fig3 and Fig6 have been implemented. They have all been run using the parameters discussed in the Parameters section.

Parameters

A mean of 30 tests has been carried out with 20 individuals, in 20 islands together with a migration rate of 1 To further test the relationship between OPC and the fitness function a ‘randomised genetic map’ has been introduced, where the position of the nucleotides is shuffled at the beginning of the problem, essentially disrupting the ability for OPC to maintain building blocks.

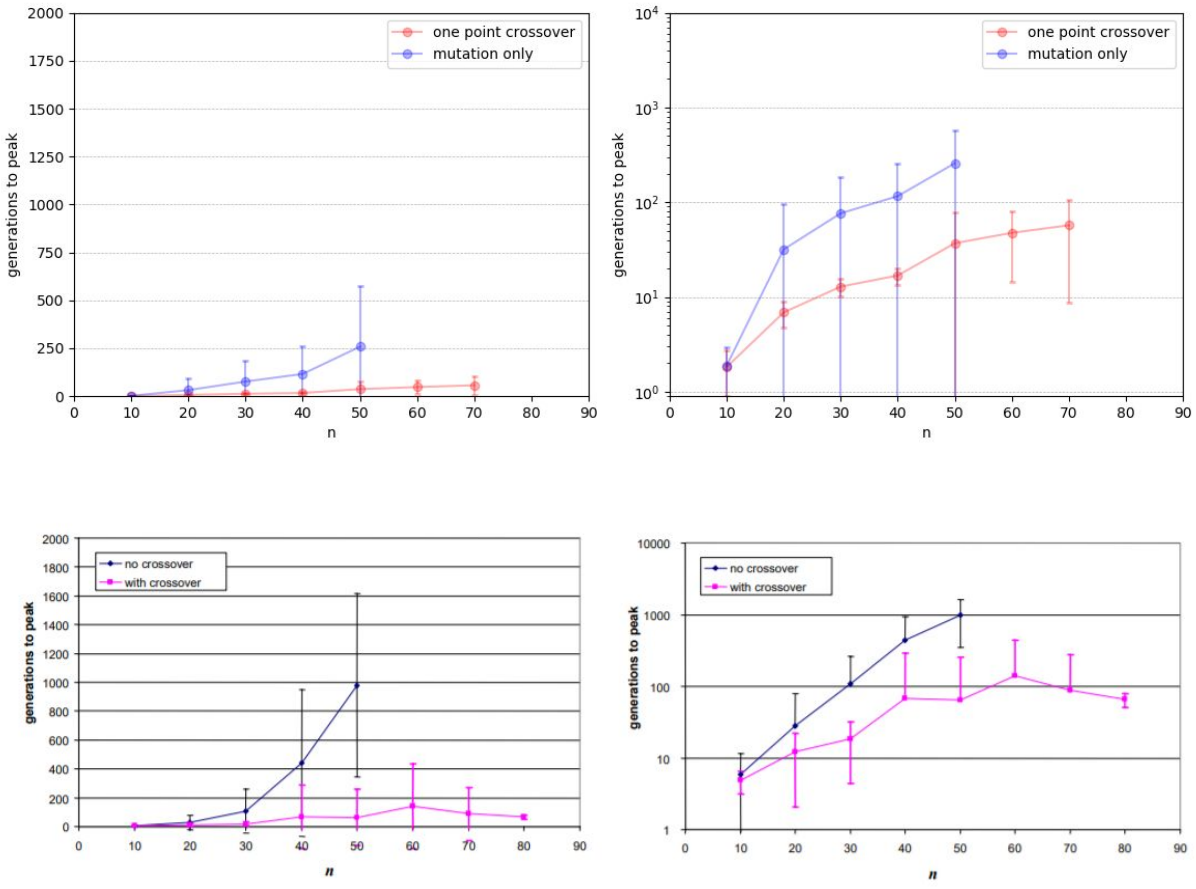

Fig2: Reimplementation of Fig3. Left) amount of generations taken to find the global optima for genome length. Right) Logarithmic behaviour, mutation only is almost exponential.

It can be seen that although OPC and mutation only run faster the logarithmic behaviour is similar.

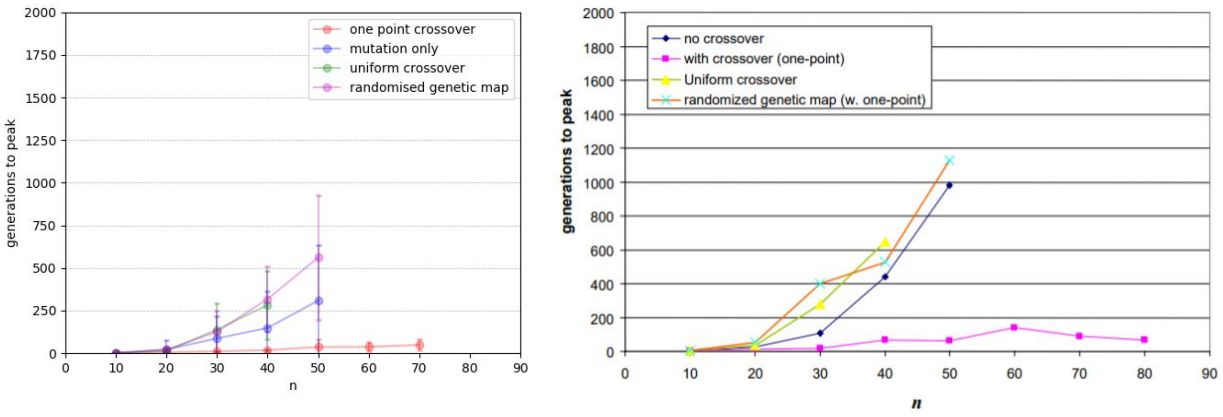

Fig3: Reimplementation of Fig6. Both Uniform crossover and OPC under a random genetic map, as expected show the same performance. This is because uniform crossover by definition is indifferent to building blocks when generating a child.

Increasing n also increases the number of runs that fail. this is more prominent in mutation only, uniform crossover and using a randomised genetic map. A failed run is defined as the optimisation algorithm taking more than 2000 generation to find the global optima. To mitigate this, and allow the algorithm to complete the tests in a reasonable time, the maximum generation at which the test is restarted has been lowered. It was clear from the tests that an iteration wouldn't complete if not finished within a certain bracket of generations. In principle, this lowered the simulation times but also removed outliers that would have pushed the mean higher, making the system over-reliant on lucky initial conditions for the individuals.

Extension: neural networks as a probabilistic method

Question

How do neural networks (NN) perform compared to crossover when employed on a simple two module problem, and what does this say about building blocks.

Hypothesis

Although a simple problem, the simple two module landscape allows for a simple intuition of why crossover can succeed on such a task. The accepted theory is that crossover allows for regions of above average fitness ‘building blocks’ to be carried out through generations [1]. This becomes intuitive from the two module problem, as when a randomised genetic map is introduced it disrupts the ability of crossover to keep these building blocks in a tight formation on a set genome.

In this report, an attempt will be made to compare the performance of crossover with a neural network on the same simple two module problem. To distinguish on what cases these methods should be applied in engineering.

Methods and advancements

The test setup, including variables, used up to this point have not been changed. What has been changed is the probabilistic method. Every island has its own mini NN, that is trained iteratively. The neural network is a denoising autoencoder. Where the input is an individual and the output is another individual.

The training setup is simple. The output is compared to a potential optimal solution. In this case, it is likely to be an individual with all 1s. Using a mean squared error loss function.

- Population in each island is initiated randomly, together with a NN for each island.
- Iterate through each island
- While training the network for just 1 epoch over every individual
    - For every individual used to train the network first generate a child and add it to the next generation's population.
    - Select a parent using roulette wheel (keep a strong fitness selection bias)
    - Generate a global optimum (the individual we are aiming for) defined by the size max(i) and max(j) on the landscape.

Value of such questions

Many attempts have been made to replace crossover with other more powerful probabilistic methods, such as the linkage tree GA [2] and [3] a modified hill climber. More recently [4] a deep learning method has also been proposed as an optimisation method. Although [4] has been accepted as the better solution to solve hard optimisation problems, if having to use it on a simple landscape to replace crossover [2] would still prefer due to its lower problem-solving time. Therefore it is hard to see where to use such a powerful algorithm in an engineering environment, as we will show in this report that NNs are still generally slower than crossover.

As an alternative to [2] and [3] crossover and small denoising autoencoders will be used respectively to show the benefits/drawbacks of each on a simple landscape.

Prediction

What it is expected is that the NN will perform much better than crossover, to the point that the problem is too simple and only one generation is needed to find an optimal solution. However the aim is not to see if it can generalise, but it's in how many generations and how long it takes to do so and compare it to crossover. A time comparison will be done to compare how NNs perform with larger ns.

Results

Two tests have been carried out to compare NNs to crossover outputting 4 graphs. The first test compared the number of generations needed to find the optimal solution, this was graphed with a linear and logarithmic scale.

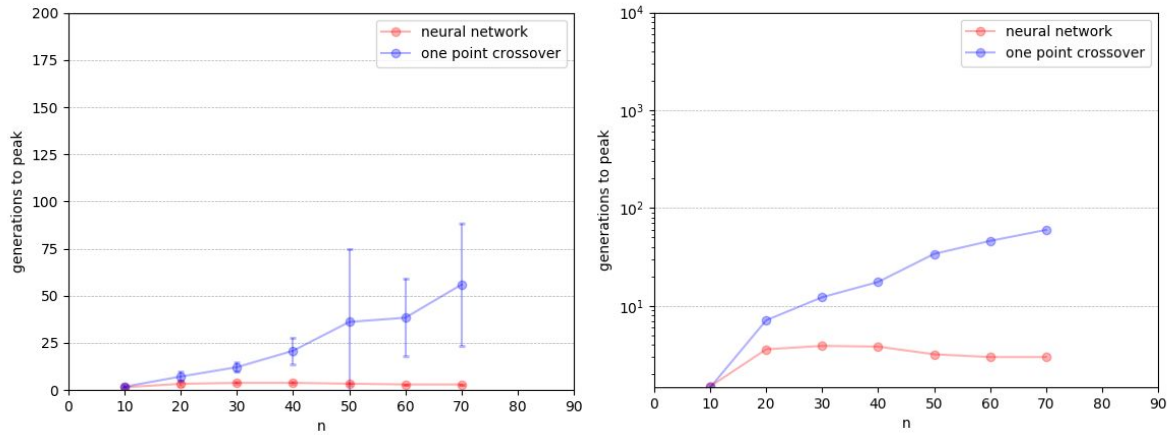

Fig1: Generations performance of NNs and crossover for different length n. Left) linear. Right) logarithmic.

The second test involved tracking the amount of time it takes to find an individual optimal solution and how long it takes to find 30 individual solutions. Again this was graphed on a linear and logarithmic scale.

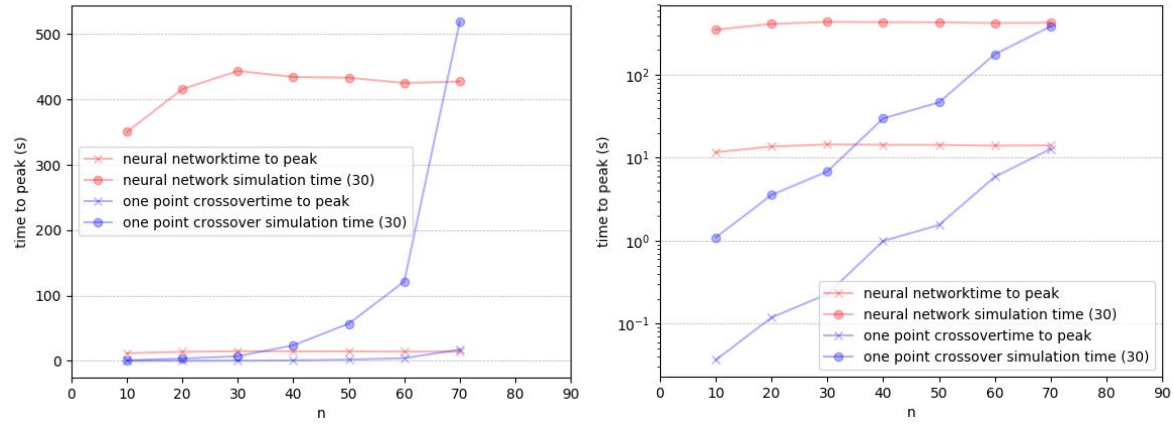

Fig2: Time performance of NNs and crossover for 1 and 30 iterations. Left) linear. Right) logarithmic.

From the results on Fig1, it is possible to see how crossover and the neural network both perform in a linear manner and as predicted the NN outperforms crossover. NNs are able to find a global optimum in approximately 3 generations with large n. However, more interestingly as shown in fig2, crossover to find the global optima in 30 different landscapes has an exponential time increase with the increase of n. This can be attributed to crossover failing to find an optimal solution within the 2000 generation limit, and as discussed in the implementation of the simple two module problem if an algorithm does not find the optima within this limit it will never find it, hence, it is stuck in a local optimum.

Fig2 clearly shows how although the individual under crossover finds a solution in better linear time than when using NNs, when the fail cases are also included, overall it performs worse than NNs. The constant poor performance of NNs' simulations can be attributed to the creation of the models, training and predictions. Components that are much more computationally expensive than simple crossover, that is however more powerful, highlighted by their stability when n is increased.

Conclusion

As mentioned in the results section crossover is good for simple solutions, however, it is unstable for larger ns as it tends to get stuck on local optimums, therefore, it is not appropriate for engineering solutions. While on the other hand, NNs are slow, not suitable for simple problems and other algorithms are faster. They are, however, suitable for larger more complex landscapes and problems where stability is essential. This results, therefore, allows the building of an intuition on where crossover should be used compared to deep learning and vice versa in engineering.

Further work can be carried out, to strengthen this case and build further intuition on why building blocks play an essential component in transferring fitness through generations. First of all, a comparison between NNs and linkage trees will detect if a lower fail rate allows this probabilistic method to outperform NNs on all fronts. This can be extended to testing such algorithms on harder landscapes. The prediction is that, as the landscape grows in complexity NNs will start outperforming them. Such initial landscapes could be the royal road function [5] and well/basin barrier [6], they are both harder problems but still allow for simple intuition when building blocks are thought to be involved. Only when a landscape where NNs struggle in, due to the lack of modularity support in the problem structure, more advanced deep learning methods can be tested. Ideally testing crossover, a NN and a 1D convolutional NN on the same problem to see which one performs best. Given the nature of building blocks and the ability for these 3 algorithms to each increasingly exploit modularity it is to be expected that 1D convolutional NN will perform best, hence giving more supporting evidence that exploiting building blocks is indeed an evolutionary advantage.

References

[1] Mitchell, M, Holland, JH, & Forrest, S, 1995 “When will a Genetic Algorithm Outperform Hillclimbing?” Advances in Neural Information Processing Systems, 6:51--58, Morgan Kaufmann, CA.

[2] Dirk Thierens, 2010, The linkage tree genetic algorithm, Institute of Information and Computing Sciences, Universiteit Utrecht, The Netherlands

[3] David Iclanzan, Dan Dumitrescu 16 Feb 2007 Overcoming Hierarchical Difficulty by Hill-Climbing the Building Block Structure

[4] J.R. Caldwell, R.A Watson and C. Thies, 2018, Deep Optimisation: Solving Combinatorial Optimisation Problems using Deep Neural Networks Agents, Interaction and Complexity University of Southampton

[5]Shapiro, Jonathan L. and Prügel-Bennett, Adam (1997) Genetic algorithm dynamics in two-well potentials with basins and barrier. Belew, R. K. and Vose, M. D. (eds.) At *Foundations of Genetic Algorithms - 4 Foundations of Genetic Algorithms - 4.* pp. 101-116.

[6] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and GA performance.

Appendix 1.1 genetic_algo.py

```
import numpy as np

import matplotlib.pyplot as plt

plt.switch_backend('agg')

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter,
FuncFormatter, StrMethodFormatter, FixedFormatter
import math
import Plot_fitness_function as PFF
import sys
from keras.layers import Input, Dense
from keras.models import Model
from keras.backend import clear_session
import time


# Create the test, each island is held in a 'list' called graph. Each
island is a sub_popoation object. Where in each sub_population object a
list holds all the individuals as objects
# the individuals have I = list of nucleotides, fitness
class Popolation:

    PLX = False
    N_inidividuals = 0
    genome_lenght = 0
    found = False
    PLOTS = 0
    R = 0
    global_i = 0
    global_j = 0
    random_genetic_map_Bstring = []

    def __init__(self, N_islands, N_inidividuals, genome_lenght):
        self.set_genome_lenght(genome_lenght)
        self.set_N_individuals(N_inidividuals)
        self.set_random_genetic_map()
        self.set_R()
        self.pop = self.create_graph(N_islands)
        self.models = []
```

```
        if Popolation.PLX == True:
            Popolation.PLOTS.plot(Popolation.R, format_axis = False)
            Popolation.PLOTS.show()

    def __del__(self):
        pass

    def create_graph(self, N_islands ):
        self.graph = []
        for i in range(N_islands):
            self.graph.append(Sub_Popolation(i))


    def calculate_every_individuals_fitness(self):
        for island in self.graph:
            for island_sub_pop in range(len(island.individuals)):
                indiv = island.individuals[island_sub_pop]
                fit = indiv.calc_fittness(indiv.I)
                indiv.fitness = fit


                if indiv.fitness ==
Popolation.R[(Popolation.global_i),(Popolation.global_j)]:
                    Popolation.found = True




    def migration(self, n_migrations):

        # go throw every island, pick and individual at random, remove
it and place it in the migration island
        for i in range(n_migrations):
            migration_island = []
            for island in self.graph:
                np.random.shuffle(island.individuals)
                Random_individual_from_island = island.individuals.pop()
                migration_island.append(Random_individual_from_island)

            np.random.shuffle(migration_island)

            for island in self.graph:
                island.individuals.append(migration_island.pop())
```

```
    def mutation_only(self):
        for island in self.graph:
            coral_reef_volcano = []
            for i in range(len(island.individuals)):
                pearent = self.wheel_selection(island.individuals)
                pearent.mutate()
                coral_reef_volcano.append(pearent)
            island.individuals = coral_reef_volcano

    def cross_over_only(self):
        for island in self.graph:
            coral_reef_volcano = []
            for i in range(len(island.individuals)):
                pearent1 = self.wheel_selection(island.individuals)
                pearent2 = self.wheel_selection(island.individuals)
                cut_at = np.random.randint(1, len(pearent1.I))
                first_half = pearent1.I[:cut_at]
                second_half = pearent2.I[cut_at:]
                child = Individual()
                child.I = np.concatenate((first_half , second_half),
axis=0)
                coral_reef_volcano.append(child)
            island.individuals = coral_reef_volcano

    def uniform_cross_over(self):
        for island in self.graph:
            coral_reef_volcano = []
            for i in range(len(island.individuals)):
                pearent1 = self.wheel_selection(island.individuals)
                pearent2 = self.wheel_selection(island.individuals)
                child = []
                for k in range(len(pearent1.I)):
                    From_who = np.random.randint(0,1)
                    if From_who == 0:
                        child.append(pearent1.I[k])
                    else:
                        child.append(pearent2.I[k])
                C = Individual()
                C.I = np.array(child)
                C.mutate()
                coral_reef_volcano.append(C)

            island.individuals = coral_reef_volcano
```

```
    def dense_generation(self, n):

        for island in range(len(self.graph)):

            self.new_popolation = []

self.models[island].fit_generator(self.dense_generator(self.graph[island
],island, n), epochs=1,
steps_per_epoch=len(self.graph[island].individuals), workers = 0,
verbose=0)
            self.graph[island].individuals = self.new_popolation




    def dense_generator(self, island,island_index, n):

        while True:

            pearent1 = self.wheel_selection(island.individuals)
            child = Individual()
            P1 = np.reshape(pearent1.I,(1,n))


            child_I = self.models[island_index].predict(P1)
            child_I = np.reshape(child_I, (n,))
            C_temp = np.zeros(len(child_I))
            for i in range(len(child_I)):
                if child_I[i] >= 0.5:
                    C_temp[i] = 1
                elif child_I[i] < 0.5:
                    C_temp[i] = 0
                else:
                    print('Error the output has to be > 0 and < 1')
                    sys.exit()
            child_I = C_temp


            child.I = np.array(child_I)
            print (child.I)
            self.new_popolation.append(child)

            what_we_want = np.zeros(n)
```

```
            the_wanted_i = np.zeros(int(n/2))
            the_wanted_j = np.zeros(int(n/2))

            for i in range(Popolation.global_i):
                the_wanted_i[i] = 1
            for j in range(Popolation.global_j):
                the_wanted_j[j] = 1

            np.random.shuffle(the_wanted_i)
            np.random.shuffle(the_wanted_j)


            what_we_want[:int(n/2)] = the_wanted_i
            what_we_want[int(n/2):] = the_wanted_j

            www_child = np.reshape(what_we_want, (1,n))
            yield P1, www_child




    def build_models(self, n):

        for island in self.graph:

            input_img = Input(shape=(n,))
            # "encoded" is the encoded representation of the input
            print ('model input shape', input_img.shape)
            encoded = Dense(int(n/2), activation='relu')(input_img)
            # "decoded" is the lossy reconstruction of the input
            decoded = Dense(n, activation='sigmoid')(encoded)

            # this model maps an input to its reconstruction
            autoencoder = Model(input_img, decoded)
            autoencoder.compile(optimizer='adadelta',
loss='mean_squared_error')
            self.models.append(autoencoder)



    def wheel_selection(self, individuals):
        individuals.sort(key=lambda x: x.fitness, reverse=True)
        S= sum(I.fitness for I in individuals )
        Win = 0
        R = np.random.randint(0,S, dtype=np.int64)
```

```
        sum_p = 0
        for i in range(len(individuals)):
            sum_p += individuals[i].fitness
            if sum_p > R:
                Win = individuals[i]
                break

        return Win



    def set_N_individuals(self, N):
        Popolation.N_inidividuals = N

    def set_genome_lenght(self,N):
        Popolation.genome_lenght = N

    def set_R(self):
        Popolation.PLOTS = PFF.Plots(int(Popolation.genome_lenght/2 +
1))
        Popolation.R, Popolation.global_i, Popolation.global_j =
Popolation.PLOTS.generate_matrix('Rugged')

    def set_random_genetic_map(self):
        Popolation.random_genetic_map_Bstring = [i for i in
range(Popolation.genome_lenght)]
        np.random.shuffle(Popolation.random_genetic_map_Bstring)

class Sub_Popolation:
    def __init__(self, Sub_pop_Id):
        self.Id = Sub_pop_Id
        self.individuals = []
        self.initialise_pop()

    def initialise_pop(self):

        for i in range(Popolation.N_inidividuals):
            I = Individual()
            I.initialise_create_I()
            self.individuals.append(I)


class Individual:

    Random_genetic_map = False
```

```
def __init__(self):
    self.fitness = 0
    self.I = []


def initialise_create_I(self):
    self.I = np.zeros(Popolation.genome_lenght)
    for i in range(Popolation.genome_lenght):
        self.I[i] = np.random.randint(0,2)

def initialise_create_I_zeros(self):
    self.I = np.zeros(Popolation.genome_lenght)
    print (self.I)
    return (self.I)


def mutate(self):
    temp_I = np.zeros(len(self.I))
    for i in range(len(self.I)):
        if np.random.uniform(0, 1) < (1/len(self.I)):
            temp_I[i] = int(np.random.randint(0,2))
        else:
            temp_I[i] = int(self.I[i])
    temp_fittness = self.calc_fittness(temp_I)
    if temp_fittness > self.fitness:
        self.I = temp_I



def calc_fittness(self, genome):
    if Individual.Random_genetic_map == True:
        temp_genome = np.zeros(len(genome))
        for i in range(len(genome)):
            index = Popolation.random_genetic_map_Bstring[i]
            temp_genome[index] = genome[i]
        genome = temp_genome


    try:
        I_genome = genome[:int(Popolation.genome_lenght/2)]
        J_genome = genome[int(Popolation.genome_lenght/2):]
    except Exception as e:
        print (e, 'YOU CANT HAVE AN ODD GENOME')
```

```
            sys.exit()
        sum_I = int(sum(I_genome))
        sum_J = int(sum(J_genome))
        fitness = Popolation.R[sum_I,sum_J]
        return fitness



class Simulation():
    play = ['neural network', 'one point crossover']
    n = 0
    x_axis = []
    avaiable_colours = ['r', 'b', 'g', 'm']

    def __init__(self):
        pass



    def plot_graph(self):

        plot = PLOT()
        plot.generate_graph()
        plot.draw_stencil()
        print ('generating neural network only ')
        plot.graph_line(is_log = False)
        Simulation.fig.savefig('neural_network.png')
        print ('First done')


        plot = PLOT()
        plot.generate_graph()
        plot.draw_stencil_time()
        print ('generating neural network times ')
        plot.graph_line_time(is_time=True)
        Simulation.fig.savefig('neural_network_times.png')
        print ('Second done')

        plot = PLOT()
        plot.generate_graph()
        plot.draw_stencil()
        print ('generating neural network logonly ')
        plot.graph_line(is_log = True)
        Simulation.fig.savefig('neural_network_log.png')
        print ('third done')
```

```
        plot = PLOT()
        plot.generate_graph()
        plot.draw_stencil_time()
        print ('generating neural network times log ')
        plot.graph_line_time(is_log=True, is_time = True)
        Simulation.fig.savefig('neural_network_times_log.png')
        print ('forth done')

    def show(self):
        plt.show()


# To show the results create a plot, for each plot it is possible to
create a line. The generate graph function is the function that
generates the popolation and dose the wanted crossover
# when a Line() object is created and initialised with the right
parameters.

class PLOT():
    def __init__(self):
        self.lines_to_plot = []

    def generate_graph(self):
        for number_of_lines in range(len(Simulation.play)):
            use = Simulation.play[number_of_lines]
            L = Line()
            if use == 'one point crossover':
                L.n = 80

            elif use == 'mutation only':
                L.n = 50
            elif use == 'uniform crossover':
                L.n = 40
            elif use == 'randomised genetic map':
                L.n = 50
            elif use == 'neural network':
                L.n = 80

            L.x_axis = [x for x in range(10,L.n,10)]

            L.generate_line_values(use)
            self.lines_to_plot.append(L)

    def draw_stencil(self):
```

```
        Simulation.fig = plt.figure()
        self.ax = Simulation.fig.add_subplot(111)
        self.ax.grid(True, 'major', 'y', ls='--', lw=.5, c='k',
alpha=.3)
        self.ax.set_xlim(0, 90)
        self.ax.yaxis.set_major_formatter(FormatStrFormatter('%d'))
        self.ax.xaxis.set_major_formatter(FormatStrFormatter('%d'))

        self.ax.set_xlabel('n')
        self.ax.set_ylabel('generations to peak')


    def graph_line(self,is_log = False):
        for i in range(len(self.lines_to_plot)):
            if is_log == True:
                self.ax.semilogy(self.lines_to_plot[i].x_axis,
self.lines_to_plot[i].means, '-o', label=Simulation.play[i],
c=Simulation.avaiable_colours[i], alpha=0.3) # main_point
                self.ax.set_ylim(0,10000)
            else:
                self.ax.plot(self.lines_to_plot[i].x_axis,
self.lines_to_plot[i].means, '-o', label= Simulation.play[i],
c=Simulation.avaiable_colours[i], alpha=0.3) # main_point
                self.ax.set_ylim(0,200)
                self.ax.errorbar(self.lines_to_plot[i].x_axis,
self.lines_to_plot[i].means, yerr = self.lines_to_plot[i].stds, fmt =
'none' , ecolor= Simulation.avaiable_colours[i], alpha=0.3, capsize=2,
capthick= 2)

            self.ax.legend()

    def graph_line_time(self,is_log = False, is_time = False):
        for i in range(len(self.lines_to_plot)):
            if is_time == True:
                if is_log == True:
                    lable = Simulation.play[i] + 'time to peak'
                    lable2 = Simulation.play[i] + ' simulation time
(30)'
                    self.ax.semilogy(self.lines_to_plot[i].x_axis,
self.lines_to_plot[i].mean_abs_pop_creation_keeper, '-x', label=lable,
c=Simulation.avaiable_colours[i], alpha=0.3) # main_point
                    self.ax.semilogy(self.lines_to_plot[i].x_axis,
self.lines_to_plot[i].abs_simulation_keeper, '-o', label= lable2,
c=Simulation.avaiable_colours[i], alpha=0.3) # main_point
                else:
```

```
                    lable = Simulation.play[i] + 'time to peak'
                    lable2 = Simulation.play[i] + ' simulation time
(30)'
                    self.ax.plot(self.lines_to_plot[i].x_axis,
self.lines_to_plot[i].mean_abs_pop_creation_keeper, '-x', label= lable,
c=Simulation.avaiable_colours[i], alpha=0.3) # main_point
                    self.ax.plot(self.lines_to_plot[i].x_axis,
self.lines_to_plot[i].abs_simulation_keeper, '-o', label= lable2,
c=Simulation.avaiable_colours[i], alpha=0.3) # main_point


            self.ax.legend()

    def draw_stencil_time(self):
        Simulation.fig = plt.figure()
        self.ax = Simulation.fig.add_subplot(111)
        self.ax.grid(True, 'major', 'y', ls='--', lw=.5, c='k',
alpha=.3)
        self.ax.set_xlim(0, 90)
        self.ax.yaxis.set_major_formatter(FormatStrFormatter('%d'))
        self.ax.xaxis.set_major_formatter(FormatStrFormatter('%d'))

        self.ax.set_xlabel('n')
        self.ax.set_ylabel('time to peak (s)')




class Line():
    def __init__(self):
        self.means = []
        self.stds = []
        self.n = 0
        self.x_axis = []
        self.mean_abs_pop_creation_keeper = []
        self.abs_simulation_keeper = []

    def generate_line_values(self, use):

        for n in range(10,self.n,10):
                iterations_taken = []
                number_of_time = 0
                time_start_for_simulation_to_finish = time.time()
                time_each_popolation = []
                while number_of_time < 30:
```

```
                    time_start_popolation_creation = time.time()
                    Pacific_island = Popolation(20,20,n)
                    Popolation.found = False
                    generation = 0
                    temp_number_of_time = number_of_time
                    if use == 'neural network':
                        clear_session()
                        Pacific_island.build_models(n)

                    while Popolation.found == False:

Pacific_island.calculate_every_individuals_fitness()

                        Pacific_island.migration(1)

                        if use == 'uniform crossover':
                            Pacific_island.uniform_cross_over()
                        elif use == 'one point crossover':
                            Pacific_island.cross_over_only()
                        elif use == 'mutation only':
                            Pacific_island.mutation_only()
                        elif use == 'randomised genetic map':
                            Individual.Random_genetic_map = True
                            Pacific_island.uniform_cross_over()
                        elif use == 'neural network':
                            Pacific_island.dense_generation(n)


                        print ('genome_lenght = ',n, 'test n = ',
number_of_time, 'generation =', generation)
                        generation +=1

                        if use == 'one point crossover':
                            if generation == 400:
                                number_of_time -=1 # repeat the test
                                Popolation.found = True
                        else:
                            if n == 30:
                                if generation == 600:
                                    number_of_time -=1
                                    Popolation.found = True
                            elif n ==40:
                                if generation == 800:
                                    number_of_time -=1
                                    Popolation.found = True
```

```
                            elif n ==50:
                                if generation == 1300:
                                    number_of_time -=1
                                    Popolation.found = True
                            else:
                                if generation == 1400:
                                        number_of_time -=1
                                        Popolation.found = True

                    del Pacific_island
                    if temp_number_of_time == number_of_time:
                        iterations_taken.append(generation)
                    else:
                        pass
                    number_of_time += 1
                    time_finish_popolation_creation = time.time()
                    abs_pop_creation = time_finish_popolation_creation -
time_start_popolation_creation
                    time_each_popolation.append(abs_pop_creation)
                print (iterations_taken)
                mean = sum(iterations_taken)/30
                std = np.std(iterations_taken)
                self.means.append(mean)
                self.stds.append(std)
                time_finish_for_simulation_to_finish = time.time()
                abs_simulation = time_finish_for_simulation_to_finish -
time_start_for_simulation_to_finish
                mean_abs_pop_creation = sum(time_each_popolation)/30

self.mean_abs_pop_creation_keeper.append(mean_abs_pop_creation)
                self.abs_simulation_keeper.append(abs_simulation)


test = 0

S = Simulation()

S.plot_graph()

S.show()
```

Appendix 1.2 plot_fitness_function.py

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter,
FuncFormatter, StrMethodFormatter, FixedFormatter
import math


n = 21


# If told so it can plot the landscapes used for fig1 in the simple 2
module problem report.
# In the main program, the generate matrix function is called to create
a landscape
class Plots:

    def __init__(self, bits):
        self.bits = bits
        self.bits_expanded = [i for i in range(bits)]
        self.Noise = self.return_noise()

    def initaliaze(self):
        I, i, j = self.generate_matrix('Ideal')
        R, i, j = self.generate_matrix('Rugged')

        self.plot_noise(self.Noise)
        self.plot(I, 'ideal.png')
        self.plot(R, 'rugged.png')

        self.show()

    def return_noise(self):
        R = np.random.uniform(0.5,1,size=(self.bits,self.bits))
        return R


    def generate_matrix(self, function):
        M = np.zeros((self.bits,self.bits))
        maximum_found_fitness = 0
        global_i = 0
        global_j = 0
        for i in range(self.bits):
```

```
            Row = np.zeros(self.bits)
            for j in range(self.bits):
                if function == 'Ideal':
                    L = 2**(i)+2**(j)
                elif function == 'Rugged':
                    L = self.Noise[i,j]*(2**(i)+2**(j))
                else:
                    print ('Parse eather Ideal, or rugged as a
parameter')
                if maximum_found_fitness < L:
                    maximum_found_fitness = L
                    global_i = i
                    global_j = j
                Row[j] = L
            M[i:] = Row
        return (M, global_i, global_j)

    def plot(self, Z, which ,format_axis = True):
        fig = plt.figure()
        X, Y = np.meshgrid(self.bits_expanded, self.bits_expanded)
        ax = fig.add_subplot(111, projection='3d')
        if format_axis == True:
            ax.set_zlim(0, 2500000)
            ax.set_xlim(0, 20)
            ax.set_ylim(0, 20)
            ax.zaxis.set_major_formatter(FuncFormatter(lambda x, pos:
'%dk' % int( x/1000 )))
        ax.yaxis.set_major_formatter(FormatStrFormatter('%d'))
        ax.xaxis.set_major_formatter(FormatStrFormatter('%d'))

        ax.view_init(35, 230)
        ax.set_xlabel('i')
        ax.set_ylabel('j')
        ax.set_zlabel('Fitness')
        ax.zaxis.labelpad=15
        ax.zaxis.set_rotate_label(False)
        Axes3D.plot_surface(ax,X,Y,Z, cmap=cm.coolwarm)
        fig.savefig(which)

    def plot_noise(self,Z):
        fig = plt.figure()
        X, Y = np.meshgrid(self.bits_expanded, self.bits_expanded)
        ax = fig.add_subplot(111, projection='3d')
        ax.set_zlim(0, 1.1)
        ax.set_xlim(0, 20)
```

```
        ax.set_ylim(0, 20)
        ax.yaxis.set_major_formatter(FormatStrFormatter('%d'))
        ax.xaxis.set_major_formatter(FormatStrFormatter('%d'))
        ax.view_init(35, 230)
        ax.set_xlabel('i')
        ax.set_ylabel('j')
        ax.set_zlabel('R(i,j)')
        ax.zaxis.set_rotate_label(False)
        Axes3D.plot_surface(ax,X,Y,Z, cmap=cm.coolwarm)
        fig.savefig('noise.png')
    def show(self):
        plt.show()

Plots(n).initaliaze()
```