The use of coevolution vs crossover in optimisation gap jumping.

Christopher Culley

January 4, 2019

1 Abstract

Recent research has examined situations where genetic algorithms with crossover gives demonstrable advantage over genetic algorithms without crossover. This works aims to re-implement one such example and reformulate the problem to one of a coevolutionary flavour and demonstrate that through problem decomposition, coevolutionary solutions using mutation only can be competitive with genetic algorithms with crossover.

2 Introduction

Genetic algorithms with sex can solve particular problems that hill-climbers cannot, this has been shown mathematically [1] and empirically [2]. Broadly these problems are those with solvable blocks where epistasis is present, that is, once one high fitness module solution is found, additional modules become more difficult with mutation only approaches unable to find a 'good' mutation in polynomial time.

Schema theory [3], where evolution encourages an increase in frequency of schemata of above average fitness, followed by the building block hypothesis [4] where low-order blocks with high fitness combine with other low-order blocks with high fitness to produce individuals with give a higher overall fitness, highlight the benefit of modularity when using genetic algorithms to solve problems. This modularity can be represented as genes in an individual with crossover giving beneficial multi-modular combinations or could more abstractly be seen as the coevolution of modules defined in separate individuals, perhaps different species, where the combination of these distinct individuals is the solution to an multi-modular optimisation problem. It is this perspective shift from crossover to coevolution that we wish to explore.

Coevolution approaches are ideal for problems where no single evaluation is formulated and instead aggregated to provide overall solutions [5], in this paper we wish to explore whether even when a single evaluation is formulated and shown to be solvable with crossover, that a coevolution reformulation could decrease the number of generations before a solution is reach. Previous works have explored the functional approximation problem, where coevolutionary formulation has been shown to be beneficial [6] but we aim to focus specifically on problems where crossover has been shown to be a boon over mutation only to see if coevolution works as well.

We structure the paper as follows, we start by replicating the results in [2], a problem where crossover is shown to be demonstrably optimal over mutation only. The section that follows lays out our approach to reformulating the problem as a coevolutionary one and describes in detail the methodology for generating the results which we give in the section thereafter. Finally we close with discussion and conclusion.

3 Replicating results

The original fitness landscape conceived in [2] and shown in Figure 1 is one such landscape where crossover offers clear benefit to mutation only

approaches. In this example each individual is represented as a bit string of length $2n$ which is broken down into the number of 1's in first n bits constituting the i axis and the number of 1's in the latter n bits being the j axis of the fitness function, $f(i,j) = 2^i + 2^j$ with which random noise is added to create a more complex landscape to traverse. The difficulty for mutation only approaches lies in the choppy nature of the peaks, where, once an individual has optimised one axis any deviation towards increasing the other axis is likely to result in a decrease in fitness. The use of crossover can overcome this limitation. If two individuals optimised on opposite axes are recombined at the point on or very close to n (corresponding to the axes separation on the fitness landscape), then the offspring will 'jump' the local optimums and reach the overall peak where all bits are 1.

In our re-implementation we follow [2] in using a pool of 400 individuals broken into 20 demes, generational fitness proportional selection, elitism, the chance of mutation in both crossover and without being $\frac{1}{n}$ and with migration between demes occurring once per generation (where one individual is added to each deme from one of the other demes at random). We note that for the mutation only experiments that the use of demes and migration has no effect, since each individual is mutated in isolation and the other individuals have no effect, either at the reproduction stage or fitness evaluation. As such, and for the purposes of computational efficiency, we negate this population decomposition for the experiments without crossover.

We note at this point the computation efficiency of the code required to complete the experiments in reasonable time since our initial implementation struggled to complete experiments, particularly for large n. Further examination found that the time to complete the mutations in each round was the bottleneck. To overcome this, we use the numpy library to create a matrix of random floats between 0 and 1 where the columns are size n which are changed to true if the random float in a cell is smaller than the mutation rate, $\frac{1}{n}$. With this we use

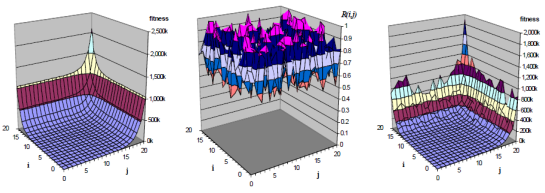

Figure 1: The original fitness landscape conceived in [2], $f(i,j) = 2^i + 2^j$ (left) which, with the produce of noise (center) creates the final fitness function (right). This fitness function contains choppy peaks making the optimal solution in the top right corner difficult to reach for mutation only algorithms.

the computationally efficient XOR bitwise operation to flip any bits in the individual where this is the case. Empirically we find this which is faster than the previous approach of drawing random numbers for each bit individually and using if statements for the flip.

Another programmatic improvement we find useful in reducing the time to compute the experiments is in the fitness proportional selection procedure which we reduce to linear time through the sorting of the floats selected as a selection criterion. This allows for a single loop through the selected floats and a count to increment the individuals lower (in cumulative probability) than the current selection float. Since the selection floats are sorted there is no need to reset the count. Of course the cost of the sort need be accounted for but this still offers an improvement on the initially $O(n^2)$ selection procedure attempted without sorting.

We follow [2] in running 30 experiments for each $n = [10, 20..70, 90]$ which we present with the original results in Figure 2. We see very similar results to the original paper noting we also observed the slight fall off of generations to optimal with $n > 70$ due to the same reasons mentioned in the paper, where some experiments failed to reach a solution which were not counted in the mean and standard deviation lines.

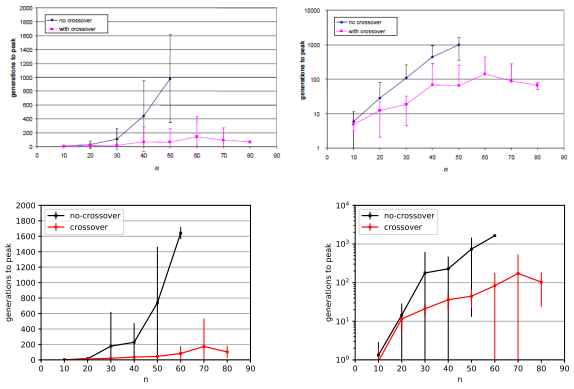

Figure 2: The re-implementation of the results from [2], with the original shown top and the re-implementation shown bottom. The plotted x-axis is the number of bits in each side of each individual and the y-axis the number of generation before an individual reaches the optimal state, averaged over 30 runs with the bars representing one standard deviation. Our results confer with the previous works, albeit with more variance of solutions.

4 Methods

We next present a coevolutionary approach to solving the same problem. To do so we introduce a prey-predator dynamic where ones fitness is determined both by its own bitstring, but also but the bitstring in its opponent. We maintain an implicit objective function which is the same as the previous works, which is unknown to population and is instead evaluated at each generation using an individual from both the prey and predator populations. We define the fitness functions as follows:

$$f_{prey}(p_y, p_d) = 2^{p_y} + 2^{(p_y - p_d)}, \quad (1)$$

$$f_{pred}(p_y, p_d) = 2^{p_d} - 2^{(p_y - p_d)}, \quad (2)$$

$$f_{obj}(p_y, p_d) = f_{pred} + f_{prey} = 2^{p_d} + 2^{p_y}, \quad (3)$$

with equations 1, 2, 3 being the prey, predator and objective fitness functions respectively, which we plot in Figure 3 where we define p_y to be the number of ones in a bit string for the prey and p_d being the number ones in the predator bit string. We note that our intent is not to mimic a particular prey-predator dynamic, but rather explore how subdividing a problem into coevolutionary parts can be a boon for functional optimisation where crossover has been shown improvement over mutation only. It should suffice therefore, to give a loose analogy that makes somewhat intuitive sense to pose the prey-predator dynamic.

Taking the fitness function of prey first we see that member of the prey group does best when its bitstring is maximimsed, which could mean its breeding opportunities are maximised and/or is ability to metabolise nutrients is increased, this is captured in the 2^{p_y} part of the fitness function. The second part of its fitness function, $2^{(p_y - p_d)}$, can be justified as the following: it is assumed that the individual has even more mating opportunities the longer it lives which is, at least in part, determined by its ability to outmaneuver its opponent predator. We leave the description of the predators function since this should follow logically from the preys and instead focus on the objective; which could be seen as the fitness of all individuals in an ecosystem.

We note that skewed scaling of fitness functions between the prey and predators is not a limitation. This is because the fitnesses are only evaluated relative to others in the species pool so the scaling factor, where the maximal prey is greater than the maximal predator, does not effect the selection process within individual pools.

We also note that there may be some ambiguity as to whether this is a cooperative or competitive coevolution set-up. We would lean to the former since, although individuals fitness is determined, in part, by the gain over an opponent, the overall objective is cooperative one.

The coevolutionary experiments follow the previous research using a population of 400 of the two types, prey and predator. Each individual is made up of n bits (making the $2n$ bits as before when com-

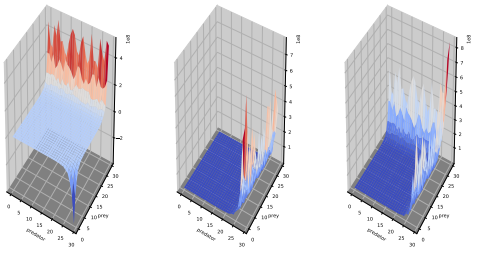

Figure 3: The coevolution game fitness landscapes each with added noise, with $f_{prey}(p_y, p_d) = 2^{p_y} + 2^{(p_y - p_d)}$ being the prey fitness function (shown left) $f_{pred}(p_y, p_d) = 2^{p_d} - 2^{(p_y - p_d)}$ the predator fitness function (center) and $f_{obj}(p_y, p_d) = f_{prey} + f_{pred} = 2^{p_d} + 2^{p_y}$, the implicit objective (right).

bined). We continue to use fitness proportional selection, elitism and a mutation rate of $\frac{1}{n}$. The only difference between this set-up and the last is the way in which fitness is calculated. For this, each individual plays a percentage of opponents from its opposite type, prey plays predator and predator plays prey using its corresponding fitness function. We extract the mean from each tournament the individual hosts as its fitness score to be used in the fitness proportional selection stage. The objective function is evaluated with each generation using a percentage of pairs from the predator-prey pool and the maximum objective fitness extracted. Once the maximum objective fitness is equal to the maximum possible, corresponding to, most probably, a set of all ones in both the prey and predator bitstrings, the objective is fulfilled and the game is stopped. The percentage of the population that each individual plays is a further parameter that we explore.

5 Results

We start by exploring the how the play percent (the percent of opponents each individual plays) effect the generations to peaks with varying n in Figure 5. From this we note that the increased number of opponents increases the selection pressure. This of course comes at a computational cost for large problems but we find in our case we were able to compute larger play percentages in good time.

Our results show relatively consistent generation to convergence rates, particularly for large n which are comparable to the previous crossover solution results as shown in Figure 6, where we fix the play percent to all of the opponents when evaluating fitness to maximise pressure. We see that the coevolutionary set-up is competitive with the crossover set-up, particularly large n, where, unlike the crossover solution where a solution isn't always found, our approach always finds the optimal solution in under 100 generations.

For completeness, we present in Figure 4 an example game to show how the objective and subjective finesses change with each generation. This picture highlights a disadvantage not present in the crossover set-up, where we see huge fluctuations between generations. This makes intuitive sense, since the fitness pressure on individual pools (prey or predators) is contingent on being ahead of opponents and at the time of objective maximisation both are even meaning there is large pressure for both sides to break away from the status-quo even though the objective function is maximised.

6 Discussion

We note caution with our results, although indeed we have shown that coevolution can do better in optimisation problems with block epistasis than single species mutation and crossover set-ups there are some caveats. Firstly, the lack of consistency with results between generations highlights a need to track the optimum which isn't needed in the single

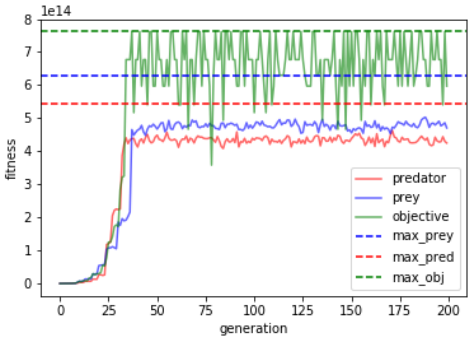

Figure 4: An example run through using the coevolutionary approach with $n = 50$ and individual finesses calculated against all other players in the opponent pool. With the maximum objective, prey and predator functions dotted, we can see clearly both the ability to reach the objective maximum and the cyclic nature of the results over generations.

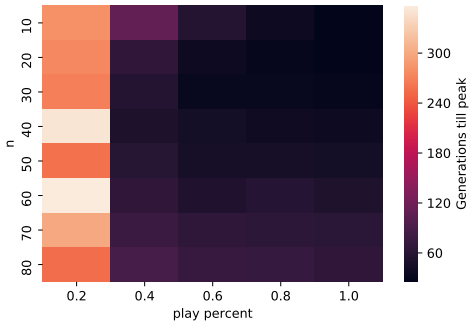

Figure 5: A heat map showing the number of generations to peak given different combinations of bit lengths n, and the percentage of opponents each individual is evaluated against when determining fitness (0.2 corresponds to playing 20% of the opponent population). Clearly higher play percentages lead to more selection pressure and a shorter time till fitness peak.

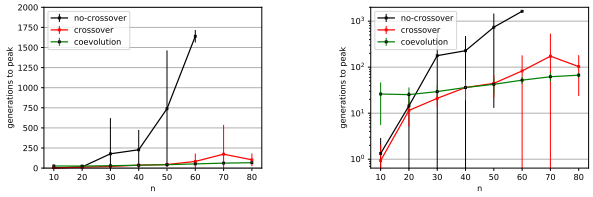

Figure 6: The full set of results of time to peak solution (left) with logarithmic scale (right) with the ftitness in the coevolutionary case being determined through taking the mean of results played against all opponents in the opponent pool. We see, although, the coevolutionary solution fares slightly worse initially, with larger n it can outperform the crossover solution.

species set-ups which might prove more difficult with more complex problems.

Secondly, this approach suffers from well studied coevolutionary problems most notability it relies on the ability to define sub-problems. This does not lend itself to the full spectrum of problems which genetic algorithms with crossover can be applied to.

We also highlight that in [2] the approach was not positioned in the hope of maximising a particular problem, moreover it was to show the benefits of crossover over mutation on particular fitness landscapes. In the same vain we do not claim to have found the optimal approach to solving the problem but instead believe we have shown that coevolutionary re-formulations, if possible, can do better than single species populations with crossover for 'gap function' problems, where module epistasis is present and where the solving of one block possibly hinders the solving of others.

Further to this, we also experimented with a more pure coevolutionary fitness function without the prey predator dynamic. Instead each individual contributes half of the bitstrings for the objective function and is evaluated with another individual providing the other n bits for the same objective. Surpris-

ingly, this approach has given worse results thus far and further investigation is needed to explore why the competitive prey predator dynamic gives better results than a pure coevolutionary one.

7 Conclusion

We have shown that a coevolution approach to gap jumping problems, ones where mutation alone struggle due to block epistasis, is both viable and competitive with crossover approaches.

References

[1] T. Jansen and I. Wegener, "Real royal road functions—where crossover provably is essential," *Discrete applied mathematics*, vol. 149, no. 1-3, pp. 111–125, 2005.

[2] R. A. Watson, "A simple two-module problem to exemplify building-block assembly under crossover," in *International Conference on Parallel Problem Solving from Nature*, pp. 161–171, Springer, 2004.

[3] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[4] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Machine learning*, vol. 3, no. 2, pp. 95–99, 1988.

[5] E. Popovici, A. Bucci, R. P. Wiegand, and E. D. De Jong, "Coevolutionary principles," in *Handbook of natural computing*, pp. 987–1033, Springer, 2012.

[6] M. A. Potter and K. A. De Jong, "A cooperative coevolutionary approach to function optimization," in *International Conference on Parallel Problem Solving from Nature*, pp. 249–257, Springer, 1994.

Re-implementation code:

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import random
from matplotlib import cm
import pandas as pd
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np
get_ipython().run_line_magic('load_ext', 'line_profiler')


n_individuals = 400
n_demes = 20
bits_in_indiv = 40
n_individuals_per_deme = int(n_individuals / n_demes)
bits_per_side = int(bits_in_indiv / 2)


@np.vectorize
def pure_fitness(x,y):
    return 2**float(x) + 2**float(y)

def noise(x,y, bits_per_side):
    return np.random.uniform(0.5, 1, size = (bits_per_side, bits_per_side))

def create_fitness_landscape(bits_per_side):
    mesh_x, mesh_y = np.meshgrid(np.arange(0, bits_per_side),
    np.arange(0, bits_per_side))
    noise_land = noise(mesh_x, mesh_y, bits_per_side)
    pure_fitness_land = pure_fitness(mesh_x, mesh_y)
    return mesh_x, mesh_y, (pure_fitness_land * noise_land),
    noise_land, pure_fitness_land


x, y, fitness_land, noise_land, pure_fitness_land = create_fitness_landscape(50)

fig = plt.figure()
ax = fig.gca(projection = "3d")
surf = ax.plot_surface(x, y , fitness_land)


fig = plt.figure()
ax = fig.gca(projection = "3d")
surf = ax.plot_surface(x, y, pure_fitness_land)


fig = plt.figure()
ax = fig.gca(projection = "3d")
surf_noise = ax.plot_surface(x,y, noise_land)


def fitness_function(bit_string):
    x = sum(bit_string[0 :  bits_per_side])
    y = sum(bit_string[bits_per_side :])
    print(x, y)
    return 2**x + 2**y


def initialise_individuals(bits_in_indiv, n_individuals_per_deme):
    return np.random.randint(0,2, size=(bits_in_indiv, n_individuals_per_deme))


@profile
def individual_fitnesses(indivs, bits_per_side):
    x = [max(sum(t[0: bits_per_side]) -1, 0)  for t in indivs]
    y = [max(sum(t[bits_per_side :]) -1,0) for t in indivs]
    return fitness_land[x ,y ]
```

```
def get_person_fitness(person, bits_per_side):
    max_bit = (bits_per_side - 1)
    return fitness_land[max(sum(person[0: bits_per_side])-1, 0),
    max(sum(person[bits_per_side:])-1, 0)]

@profile
def get_individual_probabilities(fitnesses):
    return  fitnesses / sum(fitnesses)

@profile
def get_max_index(probs):
    return probs.argmax()


@profile
def selection(probs, max_fit, indivs, bits_per_side):
    sel = np.random.rand(len(probs)- 1)
    sel.sort()
    current_tot = 0
    index = 0
    selected = []
    for it in sel:
        while(current_tot < it):
            current_tot += probs[index]
            index += 1
        if index < len(probs) - 1:
            selected.append(index)
        else:
            selected.append(index - 1)
    elite = indivs[probs.argmax(), :]
    elite_score = get_person_fitness(elite, bits_per_side)
    return selected, elite, elite_score == max_fit

@profile
def fitness_prop_selection(indivs, max_fit, bits_per_side):
    fitnesses = individual_fitnesses(indivs, bits_per_side)
    probs = get_individual_probabilities(fitnesses)
    return selection(probs, max_fit, indivs, bits_per_side)


@profile
def mutation(indv, mutation_rate, bits_in_indiv):
    return np.bitwise_xor(indv, np.random.uniform(0,1, size = bits_in_indiv) <= mutation_rate)

@profile
def one_point_crossover(indiv_a,indiv_b):
    crossover_point = random.randint(0, len(indiv_a) - 1)
    return np.concatenate([indiv_a[: crossover_point], indiv_b[crossover_point:]])

@profile
def migration(demes, no_indiv, no_genes, no_demes):
    for i, deme in enumerate(demes):
        migrant = demes[random.randint(0, no_demes - 1)][random.randint(0, no_indiv - 2),:]
        migrant = migrant.reshape(1, no_genes)
        demes[i] = np.concatenate([deme, migrant])
    return demes

def apply_crossover(demes, no_of_demes, no_indiv):
    for i, deme in enumerate(demes):
        crossed_deme = []
        for t in range(no_indiv):
            if(t < no_indiv - 1):
                new_indiv = one_point_crossover(deme[random.randint(0, no_indiv),:],
                 deme[random.randint(0, no_indiv),:])
            else:
                new_indiv = deme[t, :]
            crossed_deme.append(new_indiv)
        demes[i] = np.asarray(crossed_deme)
```

```
    return demes


@profile
def round(demes, no_indiv, bits_in_indiv, max_xy, crossover, bits_per_side):
    no_of_demes = len(demes)
    mutation_rate = 1 / bits_in_indiv
    new_demes = []
    for i,deme in enumerate(demes):
        selected_indexs, max_pop, max_found = fitness_prop_selection(deme, max_xy, bits_per_side)
        if max_found:
            return "max_found"
        prop_selected = deme[selected_indexs,:]
        mutated = np.array([mutation(x, mutation_rate, bits_in_indiv) for x in prop_selected])
        demes[i] = np.concatenate([mutated, max_pop.reshape(1, bits_in_indiv )], axis = 0)
    demes = migration(demes, no_indiv, bits_in_indiv, no_of_demes)
    if crossover:
        demes = apply_crossover(demes, no_of_demes, no_indiv)
    return demes


n_individuals = 400
n_demes = 20
bits_in_indiv = 40
n_individuals_per_deme = int(n_individuals / n_demes)


import line_profiler

@profile
def apply_rounds(demes, n_individuals_per_deme, bits_in_indiv,
fittest_index, crossover_included, bits_per_side):
    for r in range(2000):
        if crossover_included:
            demes = round(demes, n_individuals_per_deme, bits_in_indiv,
             fittest_index, crossover_included, bits_per_side)
        else:
            demes = mutation_round(demes, n_individuals_per_deme,
             bits_in_indiv, fittest_index, crossover_included, bits_per_side)
        if demes == "max_found":
            return r
            break
    return -1

@profile
def init_pops(n_individuals_per_deme, bits_in_indiv, n_demes):
    demes_cross = []
    demes_mut = initialise_individuals(n_individuals_per_deme * n_demes, bits_in_indiv)
    for x in range(n_demes):
        initalised = initialise_individuals(n_individuals_per_deme, bits_in_indiv)
        demes_cross.append(initalised)
    return demes_cross, demes_mut

def mutation_round(demes_mut, n_individuals_per_deme, bits_in_indiv, fittest, cross,
        bits_per_side):
        mutation_rate = 1 /  bits_in_indiv
        selected_indexs, max_pop, max_found = fitness_prop_selection(demes_mut, fittest,
         bits_per_side)
        if max_found:
            return "max_found"

        prop_selected = demes_mut[selected_indexs,:]
        mutated = np.array([mutation(x, mutation_rate, bits_in_indiv) for x in prop_selected])
        demes_mut = np.concatenate([mutated, max_pop.reshape(1, bits_in_indiv )], axis = 0)
        return demes_mut

results = pd.DataFrame(columns=['bit_size', 'round', 'crossover', 'mutation'])
```

```
for t in range(10, 90, 10):
    for try_no in range(30):
        print('t = ', t, ' try : ', try_no)
        bits_in_indiv = t
        bits_per_side = int(t / 2)
        x, y, fitness_land, noise_land, pure_fitness_land =
        create_fitness_landscape(bits_per_side)
        fittest = np.max(fitness_land)
        demes_cross, demes_mut = init_pops(n_individuals_per_deme,
        bits_in_indiv, n_demes)
        if t <= 70:
            print('starting mut')
            mut_res = apply_rounds(demes_mut, n_individuals_per_deme,
             bits_in_indiv, fittest, False, bits_per_side)
        else:
            mut_res = -1
        print('starting cross')
        cross_res = apply_rounds(demes_cross, n_individuals_per_deme,
        bits_in_indiv, fittest, True, bits_per_side)
        results = results.append({'bit_size': bits_in_indiv,
                        'round' : try_no,
                        'crossover' : cross_res,
                        'mutation' : mut_res}, ignore_index=True)
        print(results)

print(results)

results.to_csv("results.csv")

results = pd.read_csv("results.csv")
results['crossover'] = pd.to_numeric(results['crossover'])
results['mutation'] = pd.to_numeric(results['mutation'])
results = results.replace(-1, np.nan)

desc= results.groupby(['bit_size'])['crossover', 'mutation'].describe()
desc.index

cross_mean = desc['crossover']['mean']
cross_std = desc['crossover']['std']
cross_bit = desc.index

mut_mean = desc['mutation']['mean']
mut_std = desc['mutation']['std']
mut_bit = desc.index

fig = plt.figure(figsize=(12, 8), dpi=80)
ax = fig.add_subplot(2,2, 1)
ax.errorbar(mut_bit, mut_mean, mut_std, marker='s', mfc='black',
         mec='black', ms=0.5, mew=3,elinewidth = 0.75,
         label = 'no-crossover', ecolor = "black", color = "black")

ax.errorbar(cross_bit, cross_mean, cross_std, marker='s', mfc='red',
         mec='red', ms=0.5, mew=3,elinewidth = 0.75, label = 'crossover',
         ecolor = "red", color = "red")

plt.gca().yaxis.grid(True)
plt.xlabel('n')

plt.gca().set_ylim([0,2000])
plt.ylabel('generations to peak')
plt.legend(loc = 'upper left')
plt.gca().set_xlim([0,90])
plt.xticks(np.arange(0, 100, 10))
plt.yticks(np.arange(0, 2200, 200))
ax2 = fig.add_subplot(2,2, 2)
ax2.errorbar(mut_bit, mut_mean, mut_std, marker='s', mfc='black',
         mec='black', ms=0.5, mew=3,elinewidth = 0.75, label = 'no-crossover',
         ecolor = "black", color = "black")
```

```
ax2.errorbar(cross_bit, cross_mean, cross_std, marker='s', mfc='red',
         mec='red', ms=0.5, mew=3,elinewidth = 0.75, label = 'crossover',
          ecolor = "red", color = "red")

plt.gca().yaxis.grid(True)
plt.xlabel('n')
plt.ylabel('generations_to_peak')
plt.legend(loc = 'upper_left')
plt.yscale('log')
plt.gca().set_ylim([1,10000])
plt.gca().set_xlim([0,90])
plt.subplots_adjust(wspace = 0.5)
plt.xticks(np.arange(0, 100, 10))
plt.savefig('results_reimp.pdf')
```

Reformulation code:

```
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np
get_ipython().run_line_magic('load_ext', 'line_profiler')



def fitness(x,y):
    return 2**x + 2**y
    return (10 - y + x)*2**y + (10 - x + y)*2**x + 10*2**np.abs(y - x)

def noise(x,y, bits_per_side):
    return np.random.uniform(0.5, 1, size = (bits_per_side, bits_per_side))

@np.vectorize
def fit_pred(x,y):
    return 2**float(x) + 2**(float(x) - float(y))

@np.vectorize
def fit_prey(x,y):
    return 2**float(y) - 2**(float(x) - float(y))

@np.vectorize
def fit_coop_pred(x,y):
    return 2**float(x) + 2**float(y)

@np.vectorize
def fit_coop_prey(x,y):
    return 2**float(y) + 2**float(x)


def create_fitness_landscape(bits_per_side, fit_func):
    mesh_x, mesh_y = np.meshgrid(np.arange(0, bits_per_side), np.arange(0, bits_per_side))
    noise_land = noise(mesh_x, mesh_y, bits_per_side)
    pure_fitness_land = fit_func(mesh_x, mesh_y)
    return mesh_x, mesh_y, (pure_fitness_land * noise_land), noise_land, pure_fitness_land


def create_set_of_fit_landscapes(n, fit_prey, fit_pred):
    x_pred, y_prey, full_fit_land_prey , noise_prey, fit_prey = create_fitness_landscape(n, fit_prey)
    x_pred, y_prey, full_fit_land_pred , noise_pred, fit_pred = create_fitness_landscape(n, fit_pred)
    full_fit_land_sum = np.add(full_fit_land_pred, full_fit_land_prey)
    return full_fit_land_prey, full_fit_land_pred, full_fit_land_sum, x_pred, y_prey


fig = plt.figure()
```

```
ax = fig.gca(projection='3d')

surf = ax.plot_surface(x_pred, y_prey,full_fit_land_prey , cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
```

 # In[81]:

```
fig = plt.figure()
ax = fig.gca(projection='3d')

surf = ax.plot_surface(x_pred, y_prey, full_fit_land_pred , cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
```

 # In[82]:

```
fig = plt.figure()
ax = fig.gca(projection='3d')

surf = ax.plot_surface(x_pred, y_prey, full_fit_land_sum , cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
```

 # In[311]:

```
def objective_fitness(pred, prey):
    return full_fit_land_sum[prey , pred]

def get_corpus_of_ones(agents):
    return [max(sum(t) -1, 0)  for t in agents]

def play_a_selection_of_opposition_pred(preds, prey, no_of_ops):
    opp = np.random.randint(0, len(preds), no_of_ops)
    return np.mean([full_fit_land_prey[preds[pred], prey] for pred in opp])

def play_a_selection_of_opposition_prey(pred, preys, no_of_ops):
    opp = np.random.randint(0, len(preys), no_of_ops)
    return np.mean([full_fit_land_pred[pred, preys[prey]] for prey in opp])

def tournements(preds_in, preys_in, no_of_ops, no_of_bits):
    preds = get_corpus_of_ones(preds_in)
    preys = get_corpus_of_ones(preys_in)
    prey_res = [play_a_selection_of_opposition_pred(preds, p, no_of_ops) for p in preys]
    pred_res = [play_a_selection_of_opposition_prey(p, preys, no_of_ops) for p in preds]
    pred_ex = [preds[i] for i in np.random.randint(0,len(preds), no_of_ops)]
    prey_ex = [preys[i] for i in np.random.randint(0,len(preys), no_of_ops)]
    total = np.max(objective_fitness(pred_ex, prey_ex))
    return pred_res, prey_res, total



full_fit_land_prey, full_fit_land_pred, full_fit_land_sum, x_pred, y_prey =
    create_set_of_fit_landscapes(30, fit_pred, fit_prey)



fig = plt.figure(figsize=(18, 10))
ax = fig.add_subplot(1, 3, 1 , projection='3d')
ax2 = fig.add_subplot(1, 3, 2, projection='3d')
ax3 = fig.add_subplot(1, 3, 3, projection='3d')

surfa = ax.plot_surface(x_pred, y_prey, full_fit_land_pred , cmap=cm.coolwarm,
```

```
                        linewidth=0, antialiased=False)

ax.set_xlabel('predator')
ax.set_ylabel('prey')

plt.gca().patch.set_facecolor('white')
ax.w_xaxis.set_pane_color((0.8, 0.8, 0.8, 1.0))
ax.w_yaxis.set_pane_color((0.8, 0.8, 0.8, 1.0))

ax.w_zaxis.set_pane_color((0.5, 0.5, 0.5, 1.0))
surfb = ax2.plot_surface(x_pred, y_prey, full_fit_land_prey, cmap=cm.coolwarm,
                        linewidth=0, antialiased=False)

ax2.set_xlabel('predator')
ax2.set_ylabel('prey')
ax2.w_xaxis.set_pane_color((0.8, 0.8, 0.8, 1.0))
ax2.w_yaxis.set_pane_color((0.8, 0.8, 0.8, 1.0))

ax2.w_zaxis.set_pane_color((0.5, 0.5, 0.5, 1.0))

surfc = ax3.plot_surface(x_pred, y_prey, full_fit_land_sum, cmap=cm.coolwarm,
                        linewidth=0, antialiased=False)


ax3.set_xlabel('predator')
ax3.set_ylabel('prey')
ax3.w_xaxis.set_pane_color((0.8, 0.8, 0.8, 1.0))
ax3.w_yaxis.set_pane_color((0.8, 0.8, 0.8, 1.0))
ax3.w_zaxis.set_pane_color((0.5, 0.5, 0.5, 1.0))
plt.savefig('example_fitness_landscape_comp.pdf')


def get_individual_probabilities(fitnesses):
    return  fitnesses / sum(fitnesses)


def get_max_index(probs):
    return probs.argmax()



def selection(probs,indivs, bits_per_side):
    sel = np.random.rand(len(probs)- 1)
    sel.sort()
    current_tot = 0
    index = 0
    selected = []
    for it in sel:
        while(current_tot < it):
            current_tot += probs[index]
            index += 1
        if index < len(probs) - 1:
            selected.append(index)
        else:
            selected.append(index - 1)
    elite = indivs[probs.argmax(), :]
    return selected, elite


def fitness_prop_selection(preds, preys, bits_per_side, comps):
    pred_fit, prey_fit, obj_score = tournements(preds,preys, comps, bits_per_side)
    pred_probs = get_individual_probabilities(pred_fit)
    prey_probs = get_individual_probabilities(prey_fit)
    pred_selected, pred_elite = selection(pred_probs,  preds, bits_per_side)
    prey_selected, prey_elite = selection(prey_probs,  preys, bits_per_side)
    return pred_selected, pred_elite, np.max(pred_fit), prey_selected, prey_elite, np.max(prey_fit), obj_score

def initialise_individuals(bits_in_indiv, n_individuals):
```

```
    return np.random.randint(0,2, size=(n_individuals, bits_in_indiv))

def init_pops(n_individuals, bits_in_indiv):
    pred = initialise_individuals(bits_in_indiv, n_individuals)
    prey = initialise_individuals(bits_in_indiv, n_individuals)
    return pred, prey

def mutation(indv, mutation_rate, bits_in_indiv):
    return np.bitwise_xor(indv, np.random.uniform(0,1, size = bits_in_indiv) <= mutation_rate)


def round(preds, preys, bits_in_indiv, max_fit, comps):
    mutation_rate = 1 / bits_in_indiv
    pred_selected, pred_elite, max_pred_score, prey_selected, prey_elite,
     max_prey_score, obj_score =  fitness_prop_selection(preds, preys, bits_in_indiv, comps)
    if obj_score >= max_fit:
            return preds, preys,"max_found", max_pred_score, max_prey_score
    preds_selected = preds[pred_selected,:]
    preys_selected = preys[prey_selected,:]
    mutated_preds = np.array([mutation(x, mutation_rate, bits_in_indiv) for x in preds_selected])
    mutated_preys = np.array([mutation(x, mutation_rate, bits_in_indiv) for x in preys_selected])
    preds = np.concatenate([mutated_preds, pred_elite.reshape(1, bits_in_indiv)])
    preys = np.concatenate([mutated_preys, prey_elite.reshape(1, bits_in_indiv)])
    return preds, preys, obj_score, max_pred_score, max_prey_score

def play_rounds(preds, preys, bits_in_indiv, max_fit, comps):
    for x in range(2000):
        preds, preys, obj_score, max_pred_score, max_prey_score = round(preds,
         preys, bits_in_indiv, max_fit, comps)
        if obj_score == "max_found":
            return x
    return -1

def play_n_capture_rounds(preds, preys, bits_in_indiv, max_fit, comps):
    pred_scores = []
    prey_scores = []
    obj_scores = []
    for x in range(200):
        preds, preys, obj_score, max_pred_score, max_prey_score = round(preds,
         preys, bits_in_indiv, max_fit, comps)
        pred_scores.append(max_pred_score)
        prey_scores.append(max_prey_score)
        if obj_score == "max_found":
            obj_scores.append(max_fit)
        else:
            obj_scores.append(obj_score)

    print(obj_scores)
    return pred_scores, prey_scores, obj_scores


no_indivs = 200
comps = 1
pred, prey = init_pops(5, 5)
results = pd.DataFrame(columns=['bit_size', 'round', 'comp_evo', 'comps'])

for n in range(10, 90, 10):
    for i,comp in enumerate(range(1, n, int(n / 5))):
        print(n)
        for r in range(30):
            full_fit_land_prey, full_fit_land_pred, full_fit_land_sum, x_pred,
             y_prey = create_set_of_fit_landscapes(n, fit_pred, fit_prey)
            max_score = np.max(full_fit_land_sum)
            pred, prey = init_pops(no_indivs, n)
            score = play_rounds(pred, prey, n, max_score, comp)
            results = results.append({'bit_size' : n, 'round' : r,
             'comp_evo' : score, 'comps' : i/5}, ignore_index=True)
            print(results)
```

```
results.to_csv('comp_results.csv')

results['comp_evo'] = pd.to_numeric(results['comp_evo'])
agg_res = results.groupby(['bit_size', 'comps'])['comp_evo'].describe()


new_res = agg_res[['mean', 'std']].reset_index()
new_res['comps'] = [1/5, 2/5, 3/5, 4/5, 1] * len(np.unique(new_res['bit_size']))
new_res = new_res[new_res['comps'] == 1]

piv = pd.pivot_table(new_res, values="mean",index=["bit_size"], columns=["comps"], fill_value=0)
sns.heatmap(piv, cbar_kws={'label': 'Generations till peak'})
plt.xlabel('play percent')
plt.ylabel('n')
plt.savefig('heat_map_comp.pdf')

full_fit_land_prey, full_fit_land_pred, full_fit_land_sum, x_pred, y_prey =
    create_set_of_fit_landscapes(50, fit_pred, fit_prey)
max_score = np.max(full_fit_land_sum)
pred, prey = init_pops(200, 50)
pred_cap, prey_cap, obj_cap = play_n_capture_rounds(pred, prey, 50, max_score ,50)


max_prey = np.max(full_fit_land_prey)
max_pred = np.max(full_fit_land_pred)
max_obj = np.max(full_fit_land_sum)

plt.plot(pred_cap, label = 'predator', alpha=0.6, c = 'red')
plt.plot(prey_cap, label = 'prey', alpha=0.6, c = 'blue')
plt.plot(obj_cap, label= 'objective', alpha=0.6, c= 'green')
plt.axhline(max_prey, label = 'max_prey', c = 'blue', linestyle= '--')
plt.axhline(max_pred, label = 'max_pred', c= 'red', linestyle= '--')
plt.axhline(max_obj, label = 'max_obj', c = 'green', linestyle= '--')
plt.xlabel('generation')
plt.ylabel('fitness')
plt.legend()
plt.savefig('n50maxCompPlottingPreyVsPred')

results_cross = pd.read_csv("results.csv")
results_cross['crossover'] = pd.to_numeric(results_cross['crossover'])
results_cross['mutation'] = pd.to_numeric(results_cross['mutation'])
results_cross = results_cross.replace(-1, np.nan)

desc= results_cross.groupby(['bit_size'])['crossover', 'mutation'].describe()

cross_mean = desc['crossover']['mean']
cross_std = desc['crossover']['std']
cross_bit = desc.index

mut_mean = desc['mutation']['mean']
mut_std = desc['mutation']['std']
mut_bit = desc.index

comp_mean = new_res['mean']
comp_std = new_res['std']
comp_bit = new_res['bit_size']

fig = plt.figure(figsize=(12, 8), dpi=80)
ax = fig.add_subplot(2,2, 1)
ax.errorbar(mut_bit, mut_mean, mut_std, marker='s', mfc='black',
         mec='black', ms=0.5, mew=3,elinewidth = 0.75, label = 'no-crossover',
         ecolor = "black", color = "black")

ax.errorbar(cross_bit, cross_mean, cross_std, marker='s', mfc='red',
         mec='red', ms=0.5, mew=3,elinewidth = 0.75, label = 'crossover',
          ecolor = "red", color = "red")
```

```
ax.errorbar(comp_bit, comp_mean, comp_std, marker='s', mfc='green',
         mec='black', ms=0.5, mew=3,elinewidth = 0.75, label = 'coevolution',
          ecolor = "green", color = "green")
plt.gca().yaxis.grid(True)
plt.xlabel('n')

plt.gca().set_ylim([0,2000])
plt.ylabel('generations_to_peak')
plt.legend(loc = 'upper_left')

ax2 = fig.add_subplot(2,2, 2)
ax2.errorbar(mut_bit, mut_mean, mut_std, marker='s', mfc='black',
         mec='black', ms=0.5, mew=3,elinewidth = 0.75, label = 'no-crossover',
          ecolor = "black", color = "black")

ax2.errorbar(cross_bit, cross_mean, cross_std, marker='s', mfc='red',
         mec='red', ms=0.5, mew=3,elinewidth = 0.75, label = 'crossover',
         ecolor = "red", color = "red")

ax2.errorbar(comp_bit, comp_mean, comp_std, marker='s', mfc='green',
         mec='black', ms=0.5, mew=3,elinewidth = 0.75, label = 'coevolution',
         ecolor = "green", color = "green")

plt.gca().yaxis.grid(True)
plt.xlabel('n')
plt.ylabel('generations_to_peak')
plt.legend(loc = 'upper_left')
plt.yscale('log')
plt.gca().set_ylim([0,2000])
plt.subplots_adjust(wspace = 0.5)

plt.savefig('results_additonal(max_comp).pdf')


full_fit_land_prey, full_fit_land_pred, full_fit_land_sum, x_pred, y_prey =
 create_set_of_fit_landscapes(30, fit_coop_pred, fit_coop_prey)
fig = plt.figure(figsize=(14, 8))
ax = fig.add_subplot(1, 3, 1 , projection='3d')
ax2 = fig.add_subplot(1, 3, 2, projection='3d')
ax3 = fig.add_subplot(1, 3, 3, projection='3d')

surfa = ax.plot_surface(x_pred, y_prey, full_fit_land_pred, cmap=cm.coolwarm,
                        linewidth=0, antialiased=False)

ax.set_xlabel('agent_b')
ax.set_ylabel('agent_a')

surfb = ax2.plot_surface(x_pred, y_prey, full_fit_land_prey, cmap=cm.coolwarm,
                        linewidth=0, antialiased=False)

ax2.set_xlabel('agent_b')
ax2.set_ylabel('agent_a')

surfc = ax3.plot_surface(x_pred, y_prey, full_fit_land_sum, cmap=cm.coolwarm,
                        linewidth=0, antialiased=False)

ax.text2D(0.05, 0.95, "Agent_a_fitness_landscape", transform=ax.transAxes)
ax2.text2D(1.2, 0.95, "Agent_b_fitness_landscape", transform=ax.transAxes)
ax3.text2D(2.4, 0.95, "Objective_fitness_landscape", transform=ax.transAxes)
ax3.set_xlabel('agent_b')
ax3.set_ylabel('agent_a')
plt.savefig('example_fitness_landscape_coop.pdf')


no_indivs = 200
comps = 1
pred, prey = init_pops(5, 5)
```

```
results_coop = pd.DataFrame(columns=['bit_size', 'round', 'comp_evo', 'comps'])

for n in range(10, 90, 10):
    for i,comp in enumerate(range(1, n, int(n / 5))):
        print(comp)
        print(n)
        for r in range(30):
            full_fit_land_prey, full_fit_land_pred, full_fit_land_sum,
             x_pred, y_prey = create_set_of_fit_landscapes(n, fit_coop_prey, fit_coop_pred)
            max_score = np.max(full_fit_land_sum)
            pred, prey = init_pops(no_indivs, n)
            score = play_rounds(pred, prey, n, max_score, comp)
            results_coop = results_coop.append({'bit_size' : n, 'round' : r,
             'comp_evo' : score, 'comps' : i}, ignore_index=True)
            print(results_coop)

results_coop.to_csv('results_coop.csv')
```