Homework #1

Kyle MacMillan

February 5, 2019

I'm aware that any "paper" wouldn't contain the word "I". "I" is used extensively throughout this paper because I wanted to focus on what *I* did and I was not interested in skirting around the language to find clever ways to say things without including "I".

Problem 1

1.1 Statement

Compare the effectiveness of the text's hill climbing (two) and simulated annealing algorithms to find the max of

$$f(x) = 2^{-2((x-0.1)/0.9)^2}(\sin(5\pi x))^6 \quad \text{with} \quad x \in [0,1].$$

Use a real valued representation. Include a plot of the function with the location of the max and a plot of the estimate as a function of the iteration number. How sensitive are the algorithms to initial values?

1.2 Method

I built the Hill Climb algorithm according to page 65 in the text. At first I had trouble getting anything to work because I wasn't familiar with parallel arrays in numpy. After fixing the bouncing fitness issues (due to parallel array sorting) I had to figure out how to perturb the data.

I decided on a simple mutation to each x value, recombination was skipped. I played around with perturbations a lot. I started with a random multiplier between 0 and 1 for the current value of x. This lead to *negative* values because `np.random.normal` can yield positive or negative values in the specified range. I was not getting many hits where the perturbation was better than the original x so I decided to add the modifier to the original value. This yielded much better results with around 60% of perturbations being better than the original x value on the first run as shown in Figure 5. The efficacy decreases because as each value approaches an optimal value it becomes increasingly unlikely that they can hop to a new local maxima. The reason for this is because I restricted the random number generation to between $-0.25 < x < 0.25$. With local maxima being periodic on 0.2 we can only move at best one local maxima per generation, and it is unlikely to make a full hop.

This restriction makes initial values very important! It is necessary to have a good *uniform distribution* so we are likely to find the global maximum. I set the maximum number of generations to 30. It is therefore quite likely to hop up to near the global maximum. I ran dozens of tests and in only one test did it not "find" the global maximum.

Part of the problem was to "plot of the function with the location of the max and a plot of the estimate as a function of the iteration number." I took this to mean plot the max at every iteration. I chose the run that is shown in Figure 3 because it was interesting in that there was clearly a max value that appeared on the "peak" that would lead to a global max but it hopped to a local maximum due to the stochastic algorithm. It eventually returned to the "peak" with the global maximum and eventually found it but it was an interesting run.

It was not necessary to sort by fitness for *this* probelem but I needed to know how to do it for the future. A simpler solution would have been to find the index of the highest fitness value and use that.

For simulated annealing we know we're only going to have the *possibility* of hitting the random number if $random(0,1) < e^x$. The book stated the equation was $\exp -\Delta E/T$. Since I allowed for perturbations to sometimes lean the other way I accept the perturbation if:

$$random[0,1) < \exp^{-abs(eval(x)-eval(x'))/T}$$

I want a lot of chance for bouncing at the start and for it to "cool" as iterations are increased, but any close value can be accepted. T's update was handled simply with the following `UpdateT()`:

$$T = T * 0.8$$

I played around with the numbers a lot and I found the best thing I could to to represent simulated annealing was to set T to 0.25 and update T by making it 80% of the current value. Initial T of 0.25 was chosen because that was just over the maximum difference possible between x and x'.

So a maximum random perturbation would yield e^{-1} and a close value of 0.01 would yield $e^{-0.4}$ for a very high chance of perturbation. This *ensures* all values of x are modified on the first run. By the 10th iteration a ΔE of 0.01 would require a random value less than 0.689. A ΔE of 0.249 would be near impossible. A ΔE of 0.125 would require a random value less than 0.009. To continue, on the 20th iteration it becomes near-impossible to force a simulated annealing jump. I tested a multitude of variables but T being set to $T = T * 0.9$ yielded the results in Figure 6 and $T = T * 0.8$ gave the results shown in Figure 7. 0.8 was chosen because it heated and cooled rapidly, which seemed to yield good results.

The two previously mentioned figures show perturbations > 10, when my pool size is only 10. This was done so if there was an update to x due to simulated annealing I could discern how my changes influenced the output. For example, Figure 7 iteration 1 led to 5 simulated annealing-specific changes, and 3 from the regular perturbations of x. Figure 10 shows the max at every iteration.

Simulated Annealing does not seem as sensitive to initial values because it is capabale of jumping out of local minima.

1.3 Results

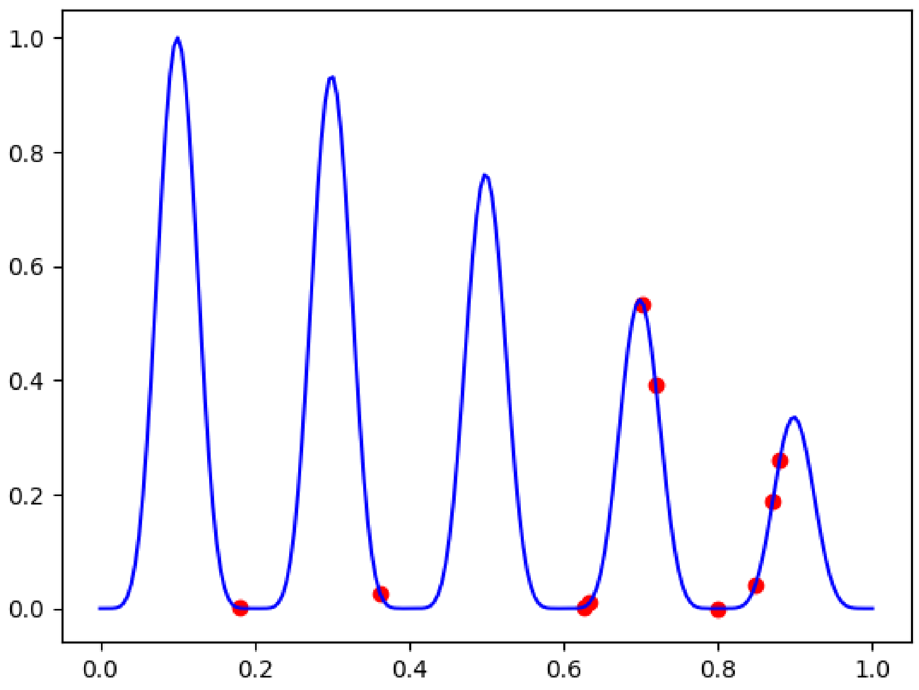


Figure 1: Initial point distribution

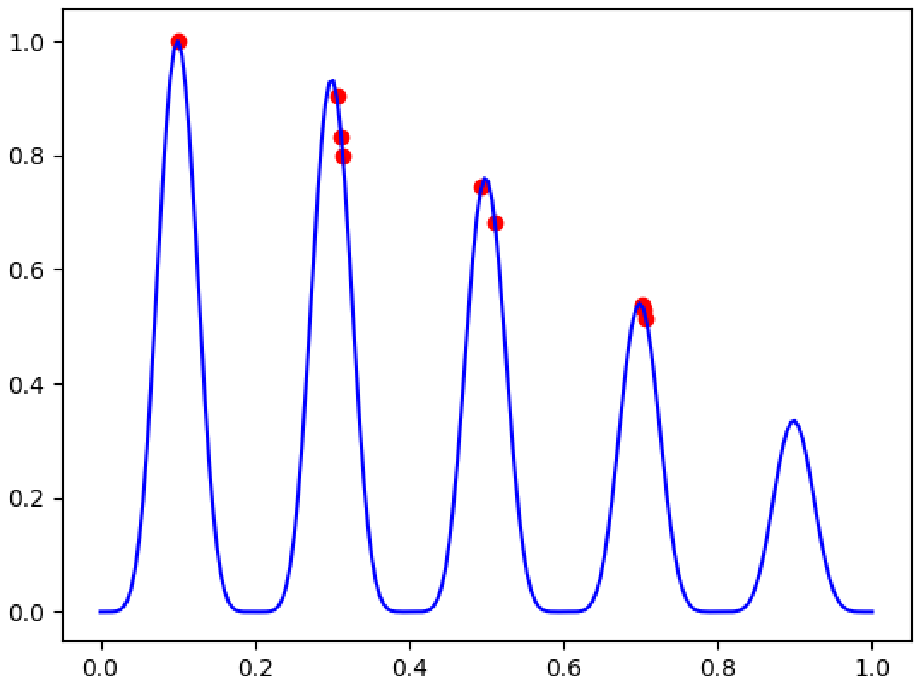


Figure 2: Points after Hill Climb algorithm

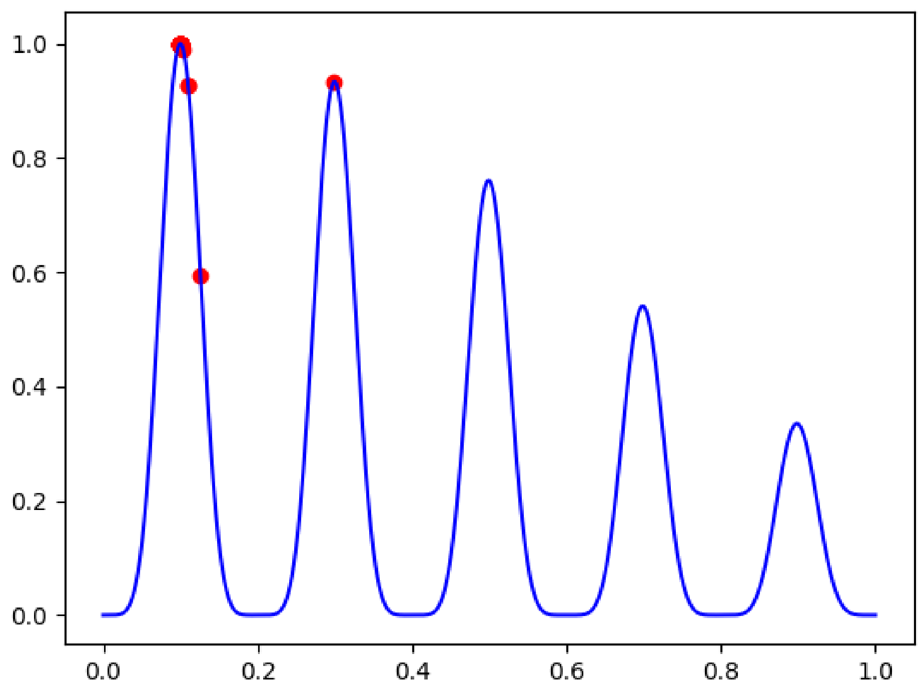
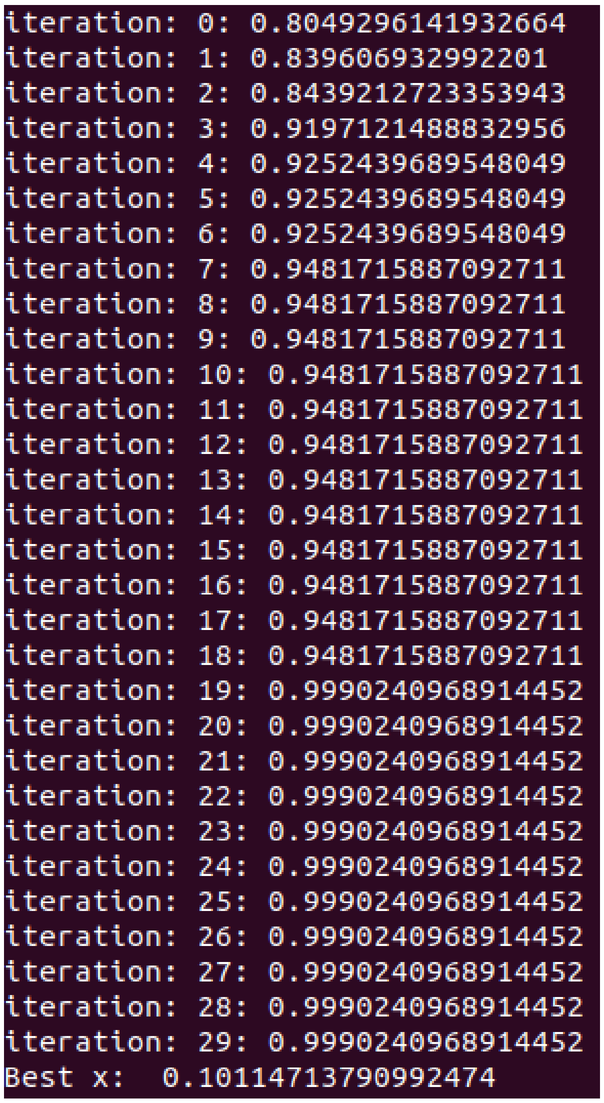
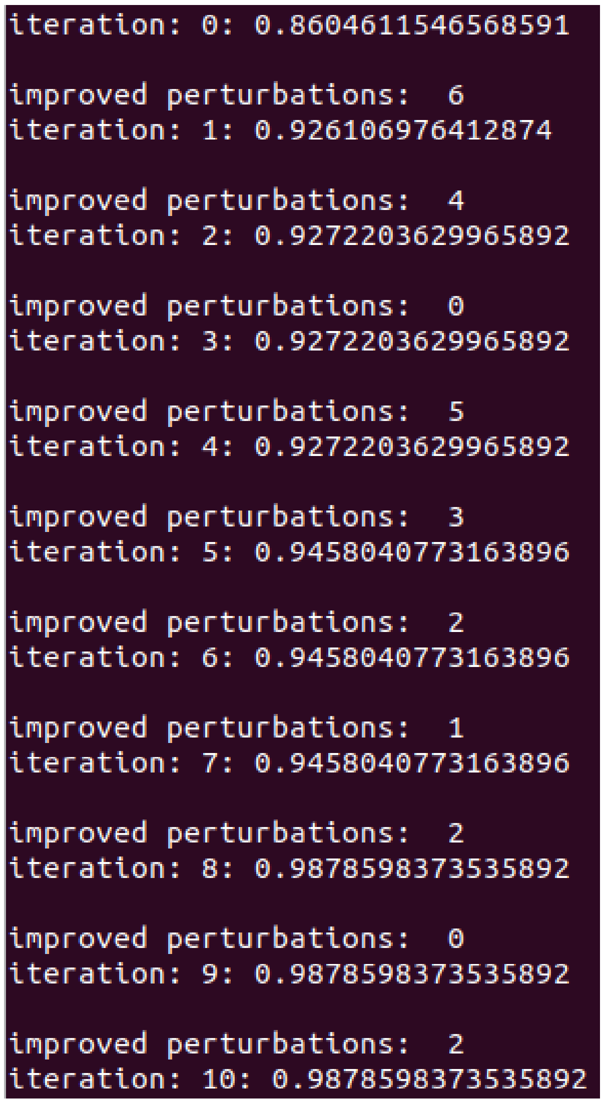


Figure 3: Max points during Hill Climb algorithm



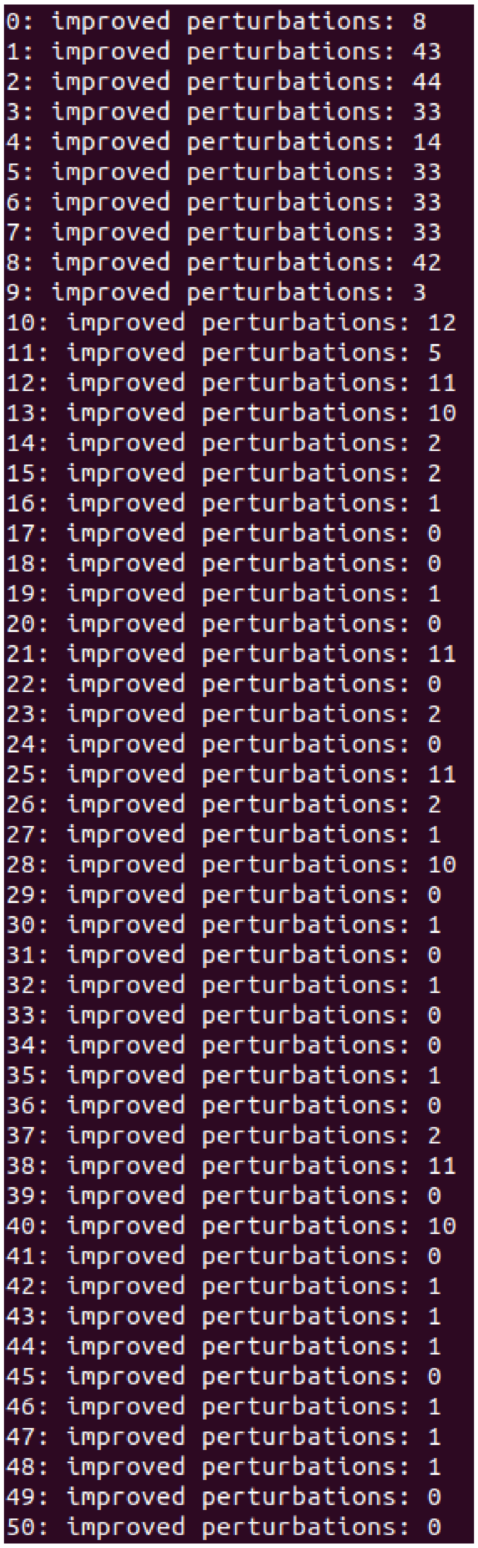
```
iteration: 0: 0.8049296141932664
iteration: 1: 0.839606932992201
iteration: 2: 0.8439212723353943
iteration: 3: 0.9197121488832956
iteration: 4: 0.9252439689548049
iteration: 5: 0.9252439689548049
iteration: 6: 0.9252439689548049
iteration: 7: 0.9481715887092711
iteration: 8: 0.9481715887092711
iteration: 9: 0.9481715887092711
iteration: 10: 0.9481715887092711
iteration: 11: 0.9481715887092711
iteration: 12: 0.9481715887092711
iteration: 13: 0.9481715887092711
iteration: 14: 0.9481715887092711
iteration: 15: 0.9481715887092711
iteration: 16: 0.9481715887092711
iteration: 17: 0.9481715887092711
iteration: 18: 0.9481715887092711
iteration: 19: 0.9990240968914452
iteration: 20: 0.9990240968914452
iteration: 21: 0.9990240968914452
iteration: 22: 0.9990240968914452
iteration: 23: 0.9990240968914452
iteration: 24: 0.9990240968914452
iteration: 25: 0.9990240968914452
iteration: 26: 0.9990240968914452
iteration: 27: 0.9990240968914452
iteration: 28: 0.9990240968914452
iteration: 29: 0.9990240968914452
Best x:  0.10114713790992474
```

Figure 4: Maximum values at each iteration/-generation



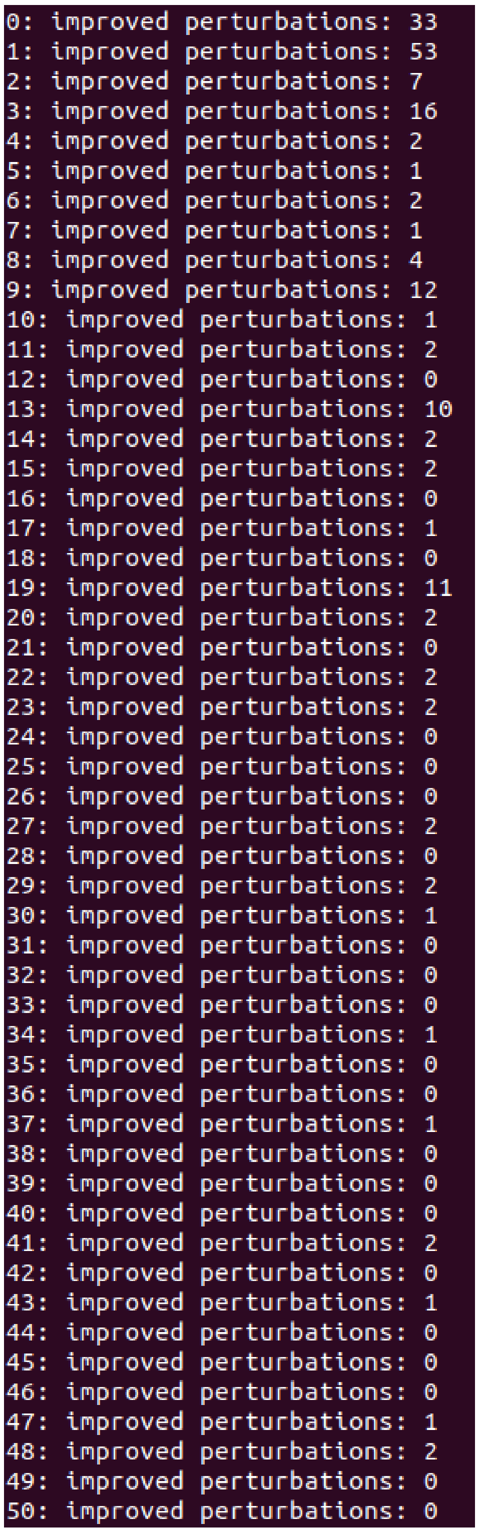
```
iteration: 0: 0.8604611546568591

improved perturbations:  6
iteration: 1: 0.926106976412874

improved perturbations:  4
iteration: 2: 0.9272203629965892

improved perturbations:  0
iteration: 3: 0.9272203629965892

improved perturbations:  5
iteration: 4: 0.9272203629965892

improved perturbations:  3
iteration: 5: 0.9458040773163896

improved perturbations:  2
iteration: 6: 0.9458040773163896

improved perturbations:  1
iteration: 7: 0.9458040773163896

improved perturbations:  2
iteration: 8: 0.9878598373535892

improved perturbations:  0
iteration: 9: 0.9878598373535892

improved perturbations:  2
iteration: 10: 0.9878598373535892
```

Figure 5: Perturbations better than their original values. Float value represents fitness.



```
0: improved perturbations: 8
1: improved perturbations: 43
2: improved perturbations: 44
3: improved perturbations: 33
4: improved perturbations: 14
5: improved perturbations: 33
6: improved perturbations: 33
7: improved perturbations: 33
8: improved perturbations: 42
9: improved perturbations: 3
10: improved perturbations: 12
11: improved perturbations: 5
12: improved perturbations: 11
13: improved perturbations: 10
14: improved perturbations: 2
15: improved perturbations: 2
16: improved perturbations: 1
17: improved perturbations: 0
18: improved perturbations: 0
19: improved perturbations: 1
20: improved perturbations: 0
21: improved perturbations: 11
22: improved perturbations: 0
23: improved perturbations: 2
24: improved perturbations: 0
25: improved perturbations: 11
26: improved perturbations: 2
27: improved perturbations: 1
28: improved perturbations: 10
29: improved perturbations: 0
30: improved perturbations: 1
31: improved perturbations: 0
32: improved perturbations: 1
33: improved perturbations: 0
34: improved perturbations: 0
35: improved perturbations: 1
36: improved perturbations: 0
37: improved perturbations: 2
38: improved perturbations: 11
39: improved perturbations: 0
40: improved perturbations: 10
41: improved perturbations: 0
42: improved perturbations: 1
43: improved perturbations: 1
44: improved perturbations: 1
45: improved perturbations: 0
46: improved perturbations: 1
47: improved perturbations: 1
48: improved perturbations: 1
49: improved perturbations: 0
50: improved perturbations: 0
```

Figure 6: T updated with T = T * 0.9



```
0: improved perturbations: 33
1: improved perturbations: 53
2: improved perturbations: 7
3: improved perturbations: 16
4: improved perturbations: 2
5: improved perturbations: 1
6: improved perturbations: 2
7: improved perturbations: 1
8: improved perturbations: 4
9: improved perturbations: 12
10: improved perturbations: 1
11: improved perturbations: 2
12: improved perturbations: 0
13: improved perturbations: 10
14: improved perturbations: 2
15: improved perturbations: 2
16: improved perturbations: 0
17: improved perturbations: 1
18: improved perturbations: 0
19: improved perturbations: 11
20: improved perturbations: 2
21: improved perturbations: 0
22: improved perturbations: 2
23: improved perturbations: 2
24: improved perturbations: 0
25: improved perturbations: 0
26: improved perturbations: 0
27: improved perturbations: 2
28: improved perturbations: 0
29: improved perturbations: 2
30: improved perturbations: 1
31: improved perturbations: 0
32: improved perturbations: 0
33: improved perturbations: 0
34: improved perturbations: 1
35: improved perturbations: 0
36: improved perturbations: 0
37: improved perturbations: 1
38: improved perturbations: 0
39: improved perturbations: 0
40: improved perturbations: 0
41: improved perturbations: 2
42: improved perturbations: 0
43: improved perturbations: 1
44: improved perturbations: 0
45: improved perturbations: 0
46: improved perturbations: 0
47: improved perturbations: 1
48: improved perturbations: 2
49: improved perturbations: 0
50: improved perturbations: 0
```

Figure 7: T updated with T = T * 0.8

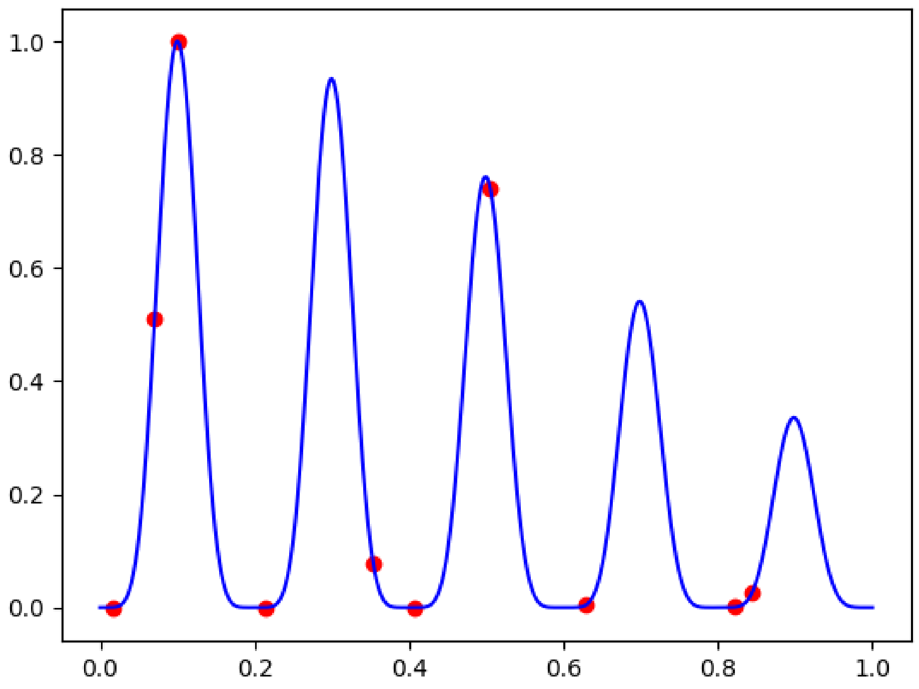


Figure 8: Initial point distribution

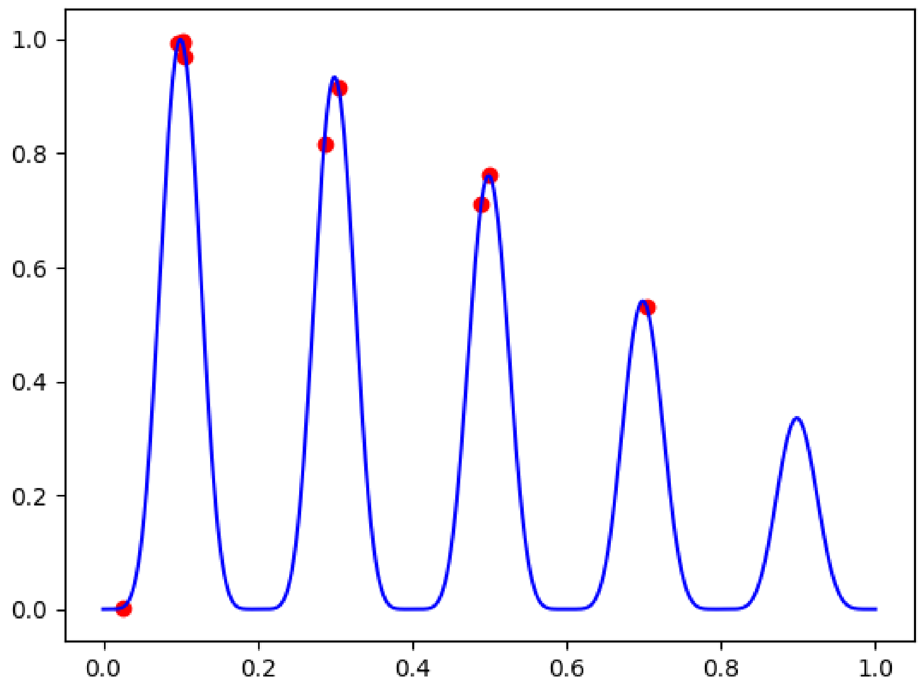


Figure 9: Points after Simulated Annealing algorithm

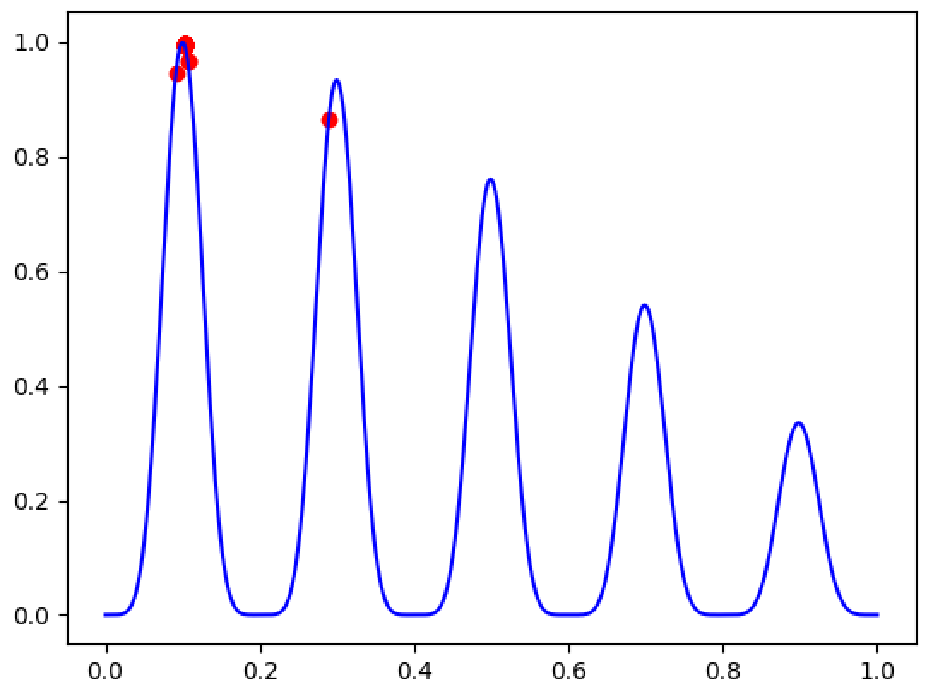


Figure 10: Max points during Simulated Annealing algorithm

1.4 Code

1.4.1 Hill Climb

```
import numpy as np
import matplotlib.pyplot as plt


class GA():
    def __init__(self, pool=10, generations=10, debug=False):
        self.pool = pool
        self.generations = generations
        self.debug = debug
        self.t = 0
        self.max_iter = 30
        self.InitializeX()

    def reset(self):
        self.x = np.copy(self.og)
        self.t = 0

    def InitializeX(self):
        self.og = np.random.uniform(0, 1, size=(1, self.pool))
        self.x = np.copy(self.og)
        self.function = np.linspace(0, 1, 2000)
        self.function = self.EvalX(self.function)
        self.max = np.zeros(self.max_iter)
        self.max_fitness = np.zeros(self.max_iter)

    def Eval(self):
        self.fitness = np.power(2, -2 * ((self.x - 0.1) / 0.9)**2) *
                       (np.sin(5 * np.pi * self.x)**6)
        # Make self.X a 2D array [x, fitness]
        self.X = np.concatenate((self.x, self.fitness), axis=0)
        # Sort and flip so the fitness is in order
        self.X = np.flip(np.argsort(self.X, axis=1))
        if self.debug:
            print('iteration: {}: {}\n'.
                format(self.t, self.fitness[0][self.X[0][0]]))
        # Note:
        # self.X is now in [fitness, x] order

    def EvalX(self, x):
        return np.power(2, -2 * ((x - 0.1) / 0.9)**2) *
               (np.sin(5 * np.pi * x)**6)

    def Draw(self):
        plt.scatter(self.x, self.fitness, color='r', marker='o')
        plt.plot(np.linspace(0, 1, 2000), self.function, color='b')
        plt.show()

    def DrawMax(self):
        plt.scatter(self.max, self.max_fitness, color='r', marker='o')
        plt.plot(np.linspace(0, 1, 2000), self.function, color='b')
        plt.show()

    def SetMax(self):
        self.max[self.t] = self.x[0][self.X[0][0]]
        self.max_fitness[self.t] = self.fitness[0][self.X[0][0]]

    def MainLoopHC(self):
        self.Eval()
        self.SetMax()
        self.Draw()
        self.t = 1
        while (self.t < self.max_iter):
            self.PerturbX()
            self.Eval()
            self.SetMax()
            self.t += 1
        self.X = np.flip(self.X)
        self.Draw()
        self.DrawMax()
        print('Best x: ', self.x[0][self.X[0][0]])

    def PerturbX(self, sa=False):
        count = 0
        for i in range(self.pool):
            modifier = np.abs(np.random.normal(0, 0.25, 1) *
                              self.x[0][i] + self.x[0][i])
            # If the fitness of the modified value > x, x = modified
            if self.fitness[0][i] < self.EvalX(modifier):
                self.x[0][i] = modifier
                count += 1
        if self.debug:
            print('improved perturbations: ', count)

    def Run(self):
        self.MainLoopHC()
        self.reset()

if __name__ == '__main__':
    ga = GA()
    ga.Run()
```

1.4.2 Simulated Annealing

```
import numpy as np
import matplotlib.pyplot as plt


class GA():
    def __init__(self, pool=10, generations=10, debug=False):
        self.pool = pool
        self.generations = generations
        self.debug = debug
        self.state = 'hc'
        self.max_iter = 30
        self.InitializeX()

    def reset(self):
        self.x = np.copy(self.og)
        self.InitializeT()

    def InitializeX(self):
        self.og = np.random.uniform(0, 1, size=(1, self.pool))
        self.reset()
        self.function = np.linspace(0, 1, 2000)
        self.function = self.EvalX(self.function)
        self.max = np.zeros(self.max_iter)
        self.max_fitness = np.zeros(self.max_iter)

    def InitializeT(self):
        if self.state == 'hc':
            self.t = 0
        else:
            self.t = 0.25

    def UpdateT(self):
        if self.state == 'hc':
            self.t += 1
        else:
            self.t *= 0.8

    def Eval(self):
        self.fitness = np.power(2, -2 * ((self.x - 0.1) / 0.9)**2) *
                                (np.sin(5 * np.pi * self.x)**6)
        # Make self.X a 2D array [x, fitness]
        self.X = np.concatenate((self.x, self.fitness), axis=0)
        # Sort and flip so the fitness is in order
        self.X = np.flip(np.argsort(self.X, axis=1))
        if self.debug:
            print('iteration: {}: {}\n'.
                format(self.t, self.fitness[0][self.X[0][0]]))
        # Note:
        # self.X is now in [fitness, x] order

    def EvalX(self, x):
        return np.power(2, -2 * ((x - 0.1) / 0.9)**2) *
               (np.sin(5 * np.pi * x)**6)

    def EvalXSA(self, x):
        ret_val = np.power(2, -2 * ((x - 0.1) / 0.9)**2) *
                  (np.sin(5 * np.pi * x)**6)
        if ret_val > 1 or ret_val < 0:
            return -1000
        else:
            return ret_val

    def EvalSA(self):
        self.fitness = np.power(2, -2 * ((self.x - 0.1) / 0.9)**2) *
                       (np.sin(5 * np.pi * self.x)**6)
        self.max_index = np.argsort(-self.fitness)[0][0]

    def Draw(self):
        plt.scatter(self.x, self.fitness, color='r', marker='o')
        plt.plot(np.linspace(0, 1, 2000), self.function, color='b')
        plt.show()

    def DrawMax(self):
        plt.scatter(self.max, self.max_fitness, color='r', marker='o')
        plt.plot(np.linspace(0, 1, 2000), self.function, color='b')
        plt.show()

    def SetMaxSA(self):
        self.max[self.iteration] = self.x[0][self.max_index]
        self.max_fitness[self.iteration] = self.fitness[0][self.max_index]

    def MainLoopSA(self):
        self.state = 'sa'
        self.reset()
        self.iteration = 0
        self.EvalSA()
        self.SetMaxSA()
        self.Draw()
        while (self.iteration < self.max_iter):
            self.PerturbX()
            self.EvalSA()
            self.SetMaxSA()
            self.UpdateT()
            self.iteration += 1
        print('Best x: ', self.x[0][self.max_index])
        self.Draw()
        self.DrawMax()

    def PerturbX(self):
        count = 0
        for i in range(self.pool):
            modifier = np.abs(np.random.normal(0, 0.25, 1) *
                              self.x[0][i] + self.x[0][i])
            fitness = self.EvalXSA(modifier)
            # If the fitness of the modified value > x, x = modified
            if self.fitness[0][i] < fitness:
                self.x[0][i] = modifier
                count += 1
            elif self.state == 'sa' and
                               (np.random.random_sample() <
                               np.exp(-np.abs(
                               self.fitness[0][i] - fitness) / self.t)):
                self.x[0][i] = modifier
                # 10 to differentiate
                count += 10
        if self.debug:
            print('{}: improved perturbations: {}'.format(self.iteration,
                                                          count))

    def Run(self):
        self.MainLoopSA()


if __name__ == '__main__':
    ga = GA()
    ga.Run()
```

Problem 2

2.1 Statement

In lecture we addressed the Traveling Salesman Problem using Simulated Annealing. To speed up convergence and increase the odds of finding the global extremal, it makes sense to try an evolutionary algorithm. The mutation operator can be adapted from the SA algorithm. Skip recombination in this problem. Write an evolutionary algorithm to solve the TSP as generated in the sample program. Compare deterministic and stochastic selection operators.

2.2 Experiments

A deterministic approach will use strictly rank to determine the next generation, whereas a stochastic approach will use some form of randomness. I assume the best approach is going to be stochastic where rank determines probability of being accepted.

Deterministic TSP:

```
$gen = 0$
$epoch = 0$
Initialize($temperature$)
Initialize($tour$)
Eval($tour$)
Rank($tour$)
**while** $temperature > 0.001$ **do**
    $cities' =$ SwapCity($(gen * 2)\%\ rollover$, $(gen * 2 + 1 + epoch)\%\ rollover$)
    Eval($tour$)
    Rank($tour$)
    **if** Accept($tour$) **then**
        $cities = cities'$
    **end if**
    $gen + +$
    **if** $gen\%\ rollover - 2 = 0$ **then**
        $epoch + +$
    **end if**
    UpdateTemp($temperature$)
**end while**
```

I would have preferred to take a much different approach: Set 50 tours as the pool. The top 5 cities will be kept through Elitism. The bottom 5 cities will be kept through “Derpism”. The middle 40 will go through 25 generations of city swaps. So cities 1, 2 swap on all 40 cities and are then evaluated and ranked. Top/Bottom 5 are kept and the next generation swaps cities 3, 4. This happens until 25 generations have been completed. I decided “Derpism” is something might be good because it is possible that as those cities get pushed out of the bottom tier they may generate better solutions. *I* *thought of it as a way to inject "random" into a deterministic approach.*

That approach would have taken a lot more time than I had so I am keeping it in my back pocket for later. To address the problem statement I utilized the existing code for TSP_SA and made the swap function either deterministic or stochastic. For the deterministic solution I settled on:

$$cities' = SwapCity((gen * 2)\%\ rollover, (gen * 2 + 1 + epoch)\%\ rollover)$$

The stochastic solution simply performed a random swap on cities.

2.3 Results

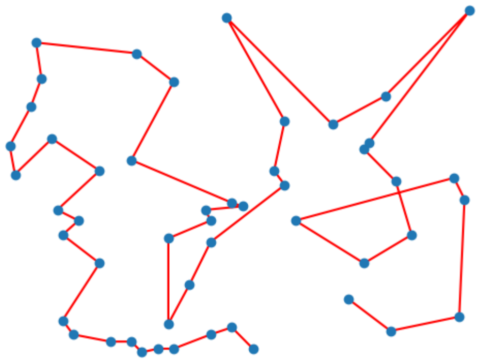


Figure 11: Deterministic tour

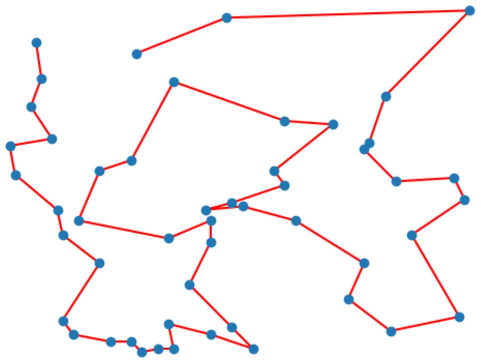


Figure 12: Stochastic tour

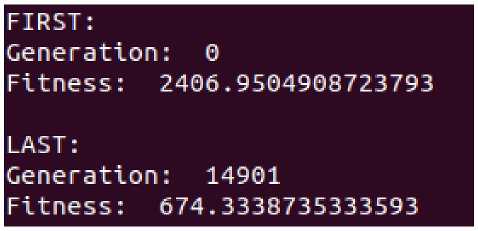

```
FIRST:
Generation:  0
Fitness:  2406.9504908723793

LAST:
Generation:  14901
Fitness:  674.3338735333593
```

Figure 13: Deterministic tour distance

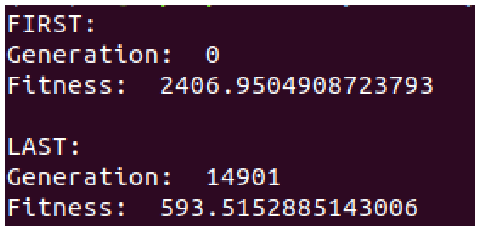

```
FIRST:
Generation:  0
Fitness:  2406.9504908723793

LAST:
Generation:  14901
Fitness:  593.5152885143006
```

Figure 14: Stochastic tour distance

As can be seen in Figure 14, a stochastic algorithm outperformed deterministic (as shown in Figure 13). Figure 11 and Figure 12 show the final tours of the algorithms. I locked the random and numpy random seeds to set values and then tested both a dozen times with a different seed for each test. Stochastic outperformed deterministic almost every time, which was expected. There are deterministic solutions that will be better than random depending on the layout of cities and the deterministic solution. In something like TSP some measure of a stochastic approach seems to fare better. Playing around with the order of swaps didn't seem to yield any one "better" solution. I would have needed to test a statistically significant amount of runs to determine if one ordering was better than another on this given city size. Would have been a good graduate bonus I guess.

"Does an array invert before or after the swap make for a better TSP solution?"

2.4 Code

```
def swap_rand(x):
    i = random.randint(0, len(x) - 2)
    j = random.randint(i, len(x) - 1)

    y = np.copy(x)
    y[i: j] = y[i: j][::-1]

    return y

def swap(x, gen, epoch):
    i = (gen * 2) % 50
    j = (gen * 2 + 1 + epoch) % 50
    y = np.copy(x)
    y[i: j] = y[i: j][::-1]
    temp = np.copy(y[i])
    y[i] = np.copy(y[j])
    y[j] = np.copy(temp)
    return y

def run(cities, cities_number, temperature = 800, cooling_factor = .001):
    current = evaluate(cities)
    i = 0
    gen = 0
    epoch = 0
    while temperature > 0.001:
        if gen == 0:
            print('FIRST:')
            print('Generation: ', gen)
            print('Fitness: ', current)
            print()
        new_solution = swap(cities, gen, epoch)
        # new_solution = swap_rand(cities)
        energy = evaluate(new_solution)
        if accept_solution(current, energy, temperature):
            cities = new_solution
            current = energy



        if (i%50==0):
            plot(cities,path = 1, wait = 0)
        if gen == 14901:
            print('LAST:')
            print('Generation: ', gen)
            print('Fitness: ', current)
```

```
                print()

            temperature *= 1 - cooling_factor
            i = i+1
            gen += 1
            if gen % 50 - 2 == 0:
                epoch += 1
        return cities
```

Problem 3

3.1 Statement

In problem 2, we implemented EA code to solve the Traveling Salesman Problem. In this problem, implement recombination (crossover) in your EA. For this problem you will need to use an encoding that prevents crossover that creates an invalid candidate. As before, compare deterministic and stochastic selection operators.

3.2 Method

I would like to try something I am calling the Starfish Approach. This is a two-part approach that involves recombination and mutation. I'll attempt both deterministic and stochastic approaches.

Chop off a starfish's leg and a new starfish will grow from it if part of the central disk is attached to that leg. The removed leg will regrow as well. In this case I intend reproduction through chopping up the best solution into ten parts. One "leg" (10 cities) will be removed from each copy of the best solution. Each amputated leg will be regrown with a mutation. Each body will be regrown with a mutation. All ten of the new starfish will be evaluated and the best will be kept for the next generation. So there are five bodies and five legs that will undergo mutation for a pool of 10 starfish to evaluate.

A "standard" crossover could be obtained by simply swapping the arms that were cut off but that is boring. I used an asexual reproduction approach. Also, I like the idea of the arm growing back with a mutation. When DNA is replicated there is opportunity for mutation.

3.3 Results

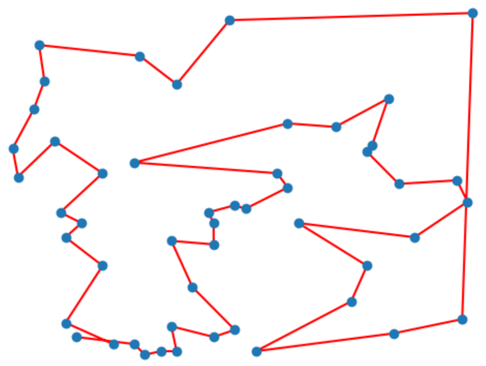


Figure 15: Deterministic Starfish tour distance

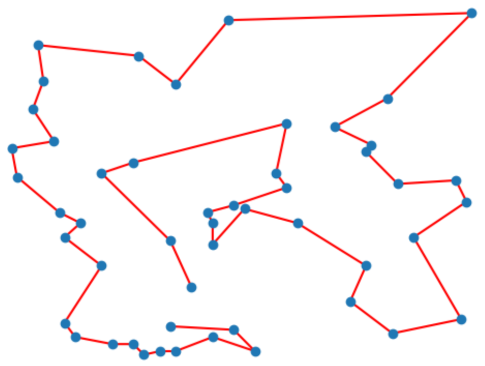
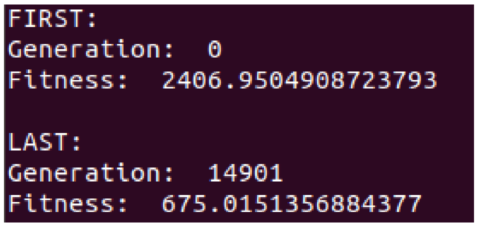
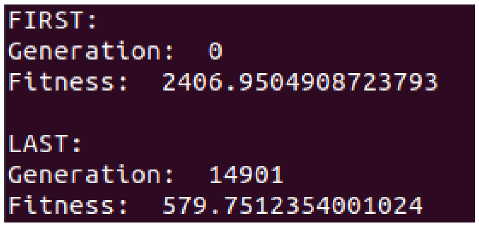


Figure 16: Stochastic Starfish tour distance



```
FIRST:
Generation:  0
Fitness:  2406.9504908723793

LAST:
Generation:  14901
Fitness:  675.0151356884377
```

Figure 17: Deterministic Starfish path



```
FIRST:
Generation:  0
Fitness:  2406.9504908723793

LAST:
Generation:  14901
Fitness:  579.7512354001024
```

Figure 18: Stochastic Starfish path

This method behaved about as expected and was incredibly slow. The slowness was to be expected because of all the array splicing and creation. The results of seeds 10-40 can be seen in the PDF in Section 3.3. 17 out of 40 times Starfish surpassed no recombination, 23 times out of 40 it failed. Perhaps more interesting would be that Starfish averaged a fitness of 629 and no recombination averaged 611.

Watching the plot update I found Starfish was **much** quicker to start picking out a coherent path compared to previous methods. I believe this method has a lot of potential and that it would outperform all other methods by a good margin if more time were put into tweaking it. For one, the splicing always happening at regular points is a huge area for improvement. It would benefit from a pool of random splice points and sizes. Figure 15 and Figure 16 show the comparison between deterministic and stochastic selection of mutations.

No Recombination		Starfish		Starfish Better
Fitness		Fitness		
2408.029305	597.4025601	2408.029305	619.1503406	FALSE
2515.18972	585.0442146	2515.18972	656.0134149	FALSE
2527.108804	643.8319489	2527.108804	691.0080334	FALSE
3107.026143	652.8420214	3107.026143	604.6570354	TRUE
2685.491537	586.9736729	2685.491537	579.9045463	TRUE
2417.12588	579.0966566	2417.12588	573.1654867	TRUE
2600.671587	605.9231637	2600.671587	608.0047612	FALSE
2808.905905	621.6627201	2808.905905	600.856986	TRUE
2759.87103	626.8011502	2759.87103	638.8766189	FALSE
2444.427334	698.460065	2444.427334	682.6324385	TRUE
2398.432797	560.7650449	2398.432797	547.1377711	TRUE
2236.183767	639.3411952	2236.183767	622.7471018	TRUE
2812.381047	604.3935495	2812.381047	676.4529452	FALSE
3166.545941	646.6331302	3166.545941	577.5206978	TRUE
2845.15388	669.8796643	2845.15388	623.707314	TRUE
2264.22559	537.7324574	2264.22559	563.8860343	FALSE
2898.337547	648.6486031	2898.337547	628.1715169	TRUE
2562.987136	580.8302489	2562.987136	620.7393711	FALSE
2744.410887	619.2987911	2744.410887	676.3585445	FALSE
2688.81926	575.4067257	2688.81926	646.4843754	FALSE
2546.195397	646.9733294	2546.195397	690.2169124	FALSE
2693.385722	583.5130639	2693.385722	653.148726	FALSE
2375.38861	633.0778105	2375.38861	632.7758468	TRUE
2609.110147	649.7540387	2609.110147	671.2444613	FALSE
2761.376813	562.0966608	2761.376813	639.0442462	FALSE
2474.727727	605.6682965	2474.727727	661.7480958	FALSE
2507.502105	587.7585378	2507.502105	645.1379074	FALSE
2612.94516	654.86682	2612.94516	647.0296937	TRUE
2408.215898	611.1556358	2408.215898	575.509988	TRUE
2536.755523	576.3171403	2536.755523	636.8781174	FALSE
2497.27343	594.8948042	2497.27343	663.7046556	FALSE
2394.568183	583.8877137	2394.568183	588.1223984	FALSE
2665.948621	608.0124748	2665.948621	620.0926935	FALSE
2536.941269	586.7396356	2536.941269	586.700176	TRUE
2507.826409	607.7658128	2507.826409	680.4204959	FALSE
2647.232832	640.2915134	2647.232832	656.083982	FALSE
2541.396933	585.5919175	2541.396933	580.7104546	TRUE
2425.974584	567.7697406	2425.974584	548.6451809	TRUE
2400.381847	642.0479731	2400.381847	698.6624685	FALSE
2444.233434	654.4950366	2444.233434	647.7756413	TRUE

3.4 Code

```
import numpy as np
import matplotlib.pyplot as plt
import random
import math

#  Computer the tour length
def evaluate(cities):
    distance = 0
    for index in range(len(cities)):
        a = cities[index]
        if index == len(cities) - 1:
            b = cities[0]
        else:
            b = cities[index + 1]

        distance += np.linalg.norm(a - b)
        index += 1

    return distance

# A perturbation is a city swap
def swap_rand(x):
    i = random.randint(0, x.size - 2)
    j = random.randint(i, x.size - 1)

    y = np.copy(x)
    # swap cities and invert sublist
    y[i: j] = y[i: j][::-1]

    return y

def swap(x, gen, epoch):
    i = (gen * 2) % (x.size // 2)
    j = (gen * 2 + 1 + epoch) % (x.size // 2)
    y = np.copy(x)

    temp = np.copy(y[i])
    y[i] = np.copy(y[j])
    y[j] = np.copy(temp)
    y[i: j] = y[i: j][::-1]
    return y


def accept_solution(energy1, energy2, temperature):
    if energy1 > energy2:
            return True
        else:
            a = math.exp((energy1 - energy2) / temperature)
            b = random.random()
            if a > b:
                return True
            else:
                return False

def Reproduce(cities, det, gen, epoch):
    childbody = np.zeros((5, 40, 2), dtype=int)
    childleg = np.zeros((5, 10, 2), dtype=int)

    for i in range(5):
        bod_min = i * 10
        bod_max = bod_min + 10
        childbody[i] = np.concatenate([np.copy(cities[:bod_min, :]),
                                       np.copy(cities[bod_max:, :])], axis=0)
        # Take only the leg
        childleg[i] = np.copy(cities[bod_min:bod_max])

    cb, cl = Mutate(childbody, childleg, det, gen, epoch)
    y = Regrow(cb, cl, cities)
    return y

def Mutate(cb, cl, det, gen, epoch):
    if det:
        for i in range(5):
            cb[i] = swap(cb[i], gen, epoch)
            cl[i] = swap(cl[i], gen, epoch)
    else:
        for i in range(5):
            cb[i] = swap_rand(cb[i])
            cl[i] = swap_rand(cl[i])

    return cb, cl

def Regrow(cb, cl, x):
    y = np.zeros((10, 50, 2), dtype=np.int)
    for i in range(5):
        bod_min = i * 10
        bod_max = bod_min + 10
        y[i * 2] = np.copy(x)
        y[i * 2+ 1] = np.copy(x)
        # Body
        y[i, :bod_min, :] = cb[i, :bod_min, :]
        y[i, bod_max:, :] = cb[i, bod_min:, :]
        # Leg
        y[i + 1, bod_min:bod_max, :] = cl[i]

    return y


def run(cities, cities_number, temperature = 800, cooling_factor = .001):
    det = False
    current = evaluate(cities)
    i = 0
    gen = 0
    epoch = 0
    while temperature > 0.001:
        y = Reproduce(cities, det, gen, epoch)
        max_energy = current
        min_index = -1
        for j in range(10):
            energy = evaluate(y[j])
            if accept_solution(current, energy, temperature):
                cities = np.copy(y[j])
                current = energy
                if max_energy > current:
                    max_energy = current
                    min_index = j

        # Overcomes energy mixups
        if min_index != -1:
            cities = np.copy(y[min_index])
            current = max_energy

        if (i%50==0):
            plot(cities,path = 1, wait = 0)

        temperature *= 1 - cooling_factor
        i = i+1
        gen += 1
        if gen % 50 - 2 == 0:
            epoch += 1
    return cities


def plot(cities, path, wait):
    plt.clf()
    if (path == 1):
        plt.plot(cities[:, 0], cities[:, 1], color='red', zorder=0)
    plt.scatter(cities[:, 0], cities[:, 1], marker='o')
    plt.axis('off')
    if (wait == 0):  plt.ion()
    plt.show()
    plt.pause(.001)

print()
cities_number = 50
for i in range(10):
    seed = i
    random.seed(seed)
    np.random.seed(seed)
    cities = (np.random.rand(cities_number, 2) * 100).astype(int)
    plot(cities,path = 0, wait = 1)
    cities = run(cities, cities_number, temperature = 3000)
    plt.ioff()
    plot(cities,path = 1, wait = 1)

print()
```