Homework #1

Micah Runner

January 29, 2021

Due: Jan. 29, 2021 6pm MST.

Problem 1

1.1 Statement

Compare the effectiveness of the deterministic hill climbing 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? How did you decide to terminate the iteration?

1.2 Method

Hill Climbing

The *Hill Climbing* algorithm was solved first by simply starting from the initial boundary condition and then moving at a set range of change on the equation until the final boundary condition has been reached. The rate of change is constantly added to the previous x-value every iteration, increasing the x-value. This method can only used if there is a set boundary condition meaning if the equation given did not have a boundary conditions, it would not work.

$$x_{int} = 0, \quad x_{fin} = 1, \quad rate_{\Delta} = 0.01, \quad i_{max} = iteration$$

$$x_{prime} = \sum_{i=0}^{i_{max}} x_i + rate_{\Delta}$$

$$x = x_{prime} \quad if \quad f(x_{prime}) > f(x)$$

To remedy this situation, we decided to add a little bit of randomness to the rate of change using a function from the *numpy* library called **random.random()**, which selects a float value between 0 and 1, this keeps the value within the boundary conditions. As an added level of precision, we still allow a user to give rate of change, which allows for smaller random steps to be taken when climbing. This change keeps occurring and the current x-value then evaluated within the defined f(x) equation given by the problem statement. Lastly, we compare the newly calculated x-value against the previous x-value and either set the current x-value to the newly calculated x-prime or we leave the current x-value alone. This ensures that we trap the largest x-value that has been found during the maximum amount of iterations. The equation below shows the changes from the first attempt of the algorithm.

$$x_{int} = 0, \quad x_{fin} = 1, \quad rate_{\Delta} = 0.01, \quad i_{max} = iteration$$
$$x_{prime} = \sum_{i=0}^{i_{max}} x_i + value_{random} \times rate_{\Delta}$$
$$x = x_{prime} \quad if \quad f(x_{prime}) > f(x)$$

To use the hill climbing class the user must supply the rate of change and the number of maximum iterations, when creating an instance of the class. The instance of the class can be done by using **HillClimbing(rate, niter)** to a variable name. The variables rate and niter have a default of 0.01 and 100, respectively, but the user can enter in their own parameters. Use the **fit()** function which will run the algorithm until the maximum amount of iterations have been reached. This will show an animation of the change of the current x-value placed on the equation line. Afterwards the found maximum value will be displayed along with the iteration the value was found within.

Simulated Annealing

The *Simulated Annealing* algorithm was solved through the power of exponentials and making their value either get very large or very small over time. This algorithm uses temperature to heat up increasing the likely hood of finding new local maximum on a given equation. We decided to increase the value of an initial temperature slowly over every iteration simulating a metal getting hotter within a kiln. The x-value can be thought of as electrons that make up the metal, as the temperature gets hotter the electrons become excited and jump around to a higher energy level. This is how the global maximum x-value works and stops from getting stuck on local maximum. This is an improvement over the *Hill Climbing* algorithm as it can get stuck on a local maximum if the global maximum was skipped over during a random starting point. To get a uniform random number we used **random.uniform()** from the *numpy*. The equation below shows the equation used to find the maximum.

$$x_{prime} = x + rate_{\Delta} \times value_{random}$$
$$if \quad e^{-(f(x)-f(x_{prime})/T} > rand_{uniform}(0,1) : x = x_{prime}$$

To use the simulated annealing class the user must supply the rate of change, number of iterations, temperature rate, initial temperature, and final temperature, respectively. The instance of the class can be done by using **SimulatedAnnealing(rate, niter, Trate, Tint, Tfin)** to a variable name. The default values for the varaibles are as follows: rate = 0.01, niter = 1000, Trate = 0.999 Tint = 3000, Tfin = 0.001. Use the **maximize()** function which will run the algorithm until the number of iterations using the equation listed above to find the maximum x-value. There is a possibility that the user did not use enough iterations to increase the temperature, or they did not set either of the miniature and maximum temperatures. An animation will play if the number of iterations is under 3,000 otherwise only the initial starting graph and the final graph would be displayed.

1.3 Results

Hill Climbing

We first began the hill climbing with starting at x = 0, rate = 0.01, niter = 10000, while using the slightly randomized version of the original deterministic hill climbing. Figure 1 shows the beginning of the climb, before the animation plots the movement of the x-value and the corresponding f(x)-value on the equation's line. While Figure 2 shows a forced starting point of x = 0.2, which will show the problem with the hill climbing algorithm, which is getting stuck at a local maximum based mostly off of the initial starting point.

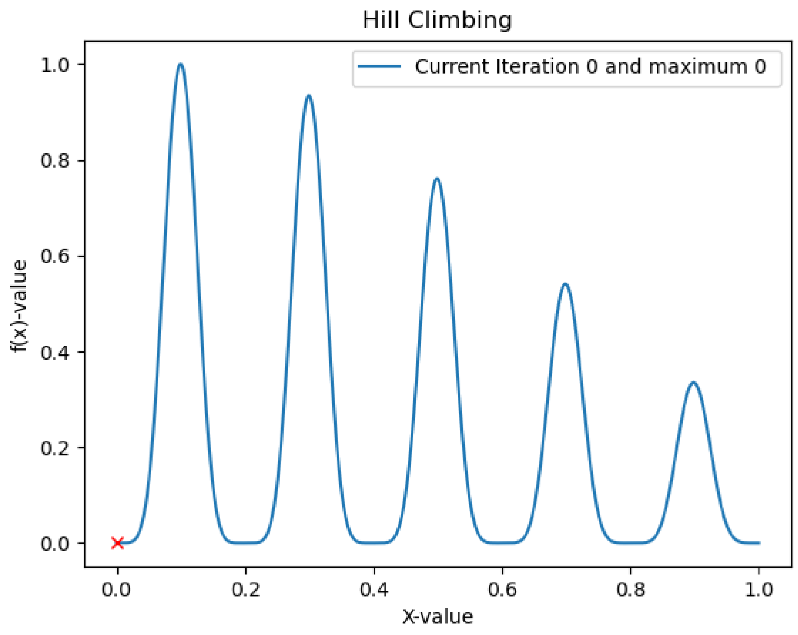

Figure 1: *Image of the initial starting point for the slightly randomized hill climb algorithm.*

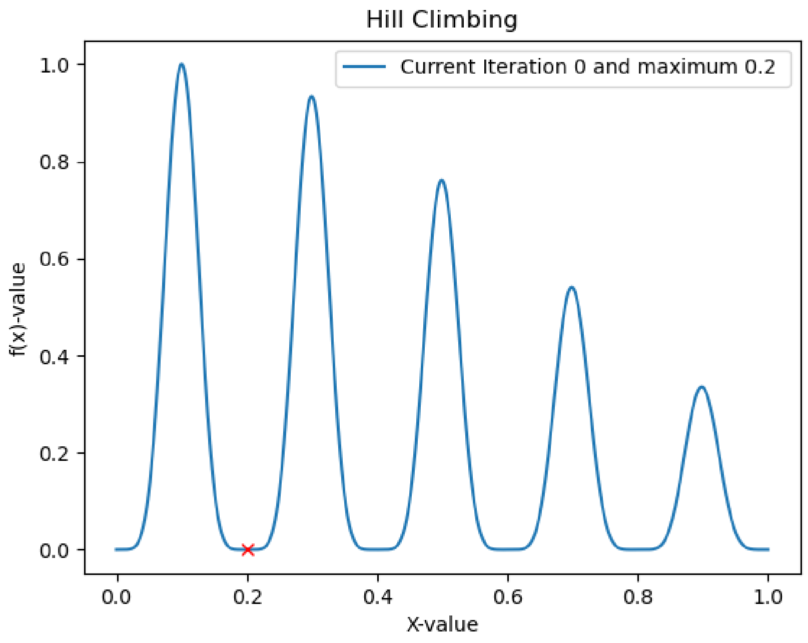

Figure 2: *Image of the initial starting point for the slightly randomized hill climb algorithm with a bad starting point.*

Figures 3 and 4 show the results of the hill climbing algorithm of having a good starting x-value and a bad starting x-value, respectively. Figure 3 clearly shows that it can be super efficient if the starting point is slightly below the x-value. Figure 4 shows that without a negative random value to subtract from the previous x-value, there is no way to go back from the starting point, which is why we allowed the user to set the learning rate, otherwise known as the rate of change. If the user uses rate = 1, then it could randomly bounce back and forth, which we found to be a very rare chance of happening. The same result in Figure 4 is most likely going to occur if the user sets rate = 1.

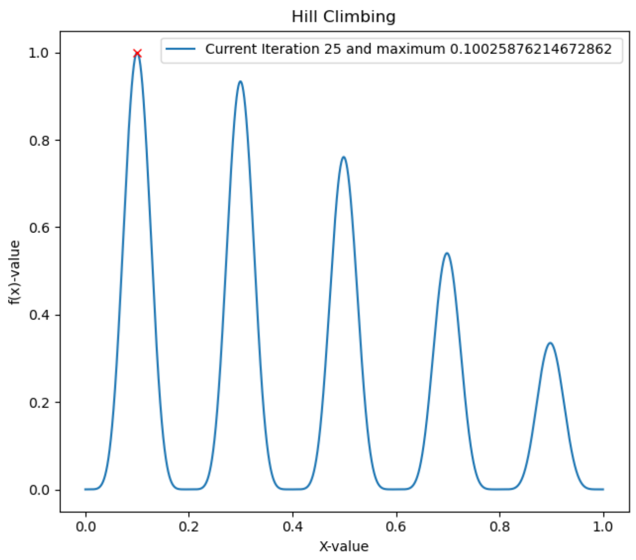

Figure 3: *Image of the final point for the slightly randomized hill climb algorithm.*

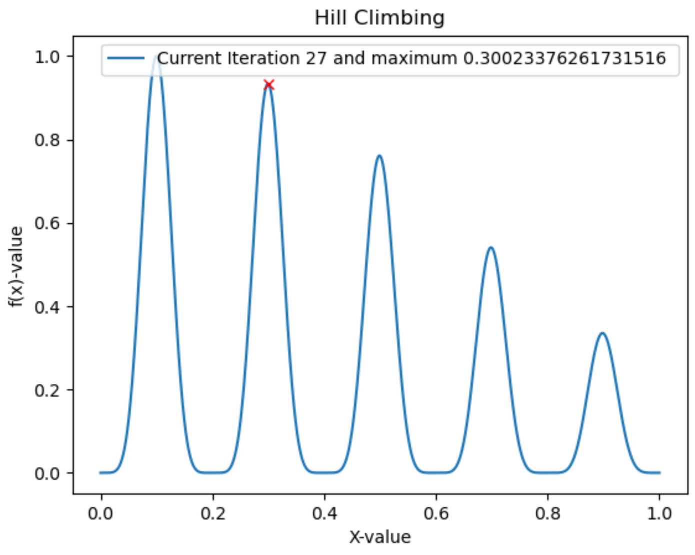

Figure 4: *Image of the final point for the slightly randomized hill climb algorithm with a bad starting point.*

Simulated Annealing

We first began the simulated annealing maximizing function by using rate = 0.1, niter = 2999, Trate = 0.999, Tint = 0.001, and Tfin = 3000. The choice of using 2,999 number of iterations was to cause the animation to appear as the value bonces around until it finds the maximum x-value. Figure 5 shows the random initial point of the function, which just like the hill climbing algorithm will make a difference but in a different manner. Figure 6 shows the final state of the simulated annealing with the maximum x-value of the equation. This shows the large improvement over the hill climbing algorithm as simulated annealing does not get stuck in a local maximum after a set amount of iterations.

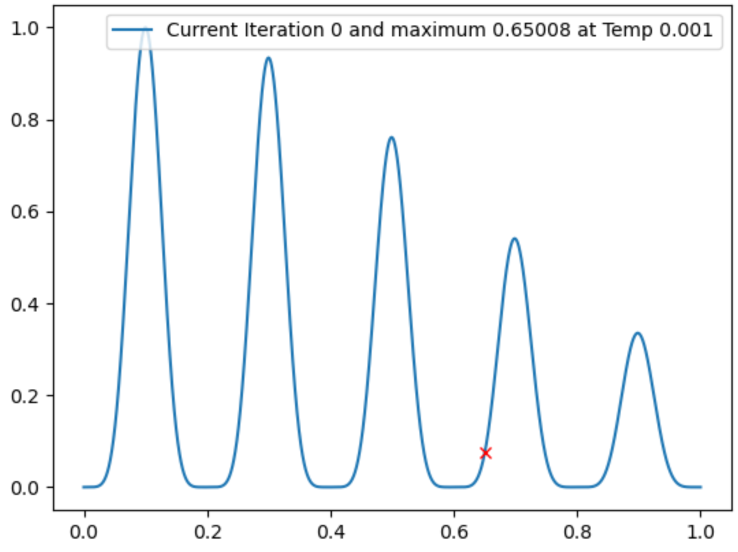

Figure 5: *Image of the random starting point of the maximizing simulated annealing algorithm.*

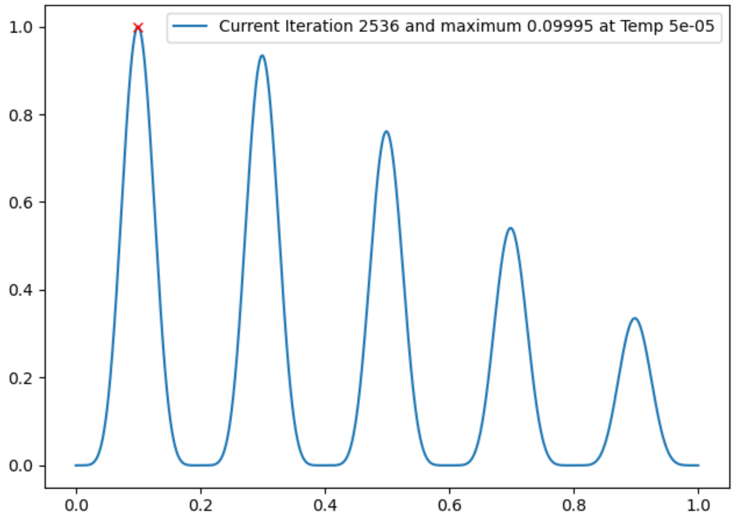

Figure 6: *Image of the final resting point of the maximizing simulated annealing algorithm.*

For this algorithm, we decided to record the x-values per every iteration, which allows the user to see the difference of each run and the starting position. Figure 7 is from the same starting point as in Figure 5 and the final point as in Figure 6. This runs shows how only the x-prime values that subtracted a set amount from the current x-value. This is the largest advantage over the hill climbing algorithm that does not have random negative variance. Figure 8 shows the a different starting point that started before the maximum x-value. Both of these figures show that as the temperature hits steady-state as it levels out to the maximum x-value. This of course is only done if enough iterations are allowed for the temperature can reach the steady-state.

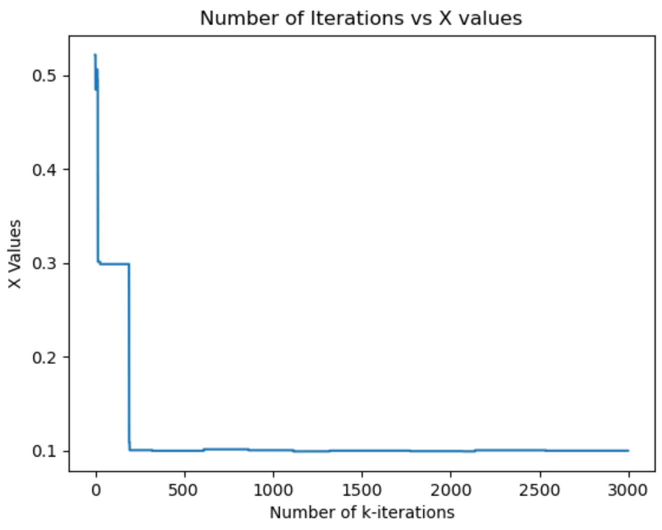

Figure 7: *Image of number of iterations and the corresponding x-values as the Simulated Annealing reaches steady-state, even from a starting point ahead of the global maximum x-value.*

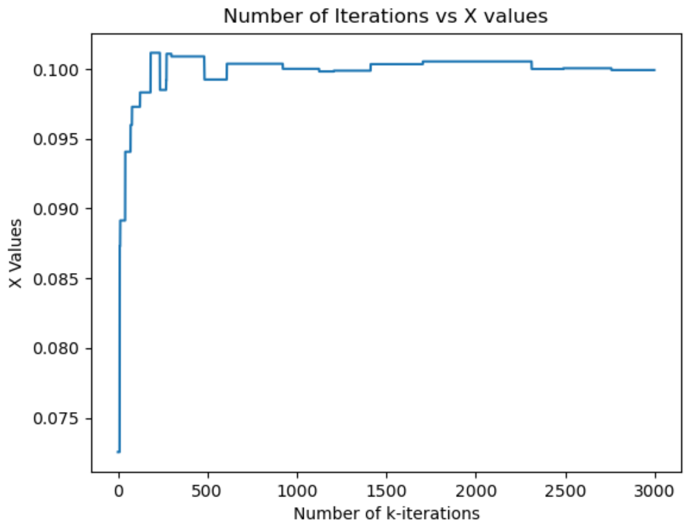

Figure 8: *Image of number of iterations and the corresponding x-values as the Simulated Annealing reaches steady-state from a point before the maximum x-value.*

1.4 Code

Please see the main file attached at the end of Problem 2 code section, which was used during testing.

Hill Climbing Class

```
# Code by Micah Runner or a monkey with a keyboard
import numpy as np
import matplotlib.pyplot as plt

class HillClimbing (object):

    # Initialize Class
    def __init__(self, rate=0.01, niter=100):
        self.niter = niter # number of iterations
        self.rate = rate # used to hold the learning rate

    # Climbing for problem 1
    def fit(self):
        # create the loop variable k and x setting both to zero
        k = 0
        x = 0

        # graph initial point
        self.k = k
        self.maximum = x
        self.graph_max(x, 0)

        # use animation if short enough
        if self.niter < 30000:
            plt.show()
            plt.ion()
            plt.figure()

        # work our way through the climb until we hit max amount of user iterations
        for k in range(self.niter):
            self.cur_iter = k

            x_prime = x + np.random.random() * self.rate

            # if f(x_prime) > f(x)
            if self.evaulate(x_prime) > self.evaulate(x):
                x = x_prime
                # get the iteration that it took to get the value
                self.k = k
                if self.niter < 30000:
                    self.graph_max(x, 1)
        # end animation
        if self.niter < 30000:
            plt.ioff()
            plt.close()

        # get the maximum that was found
        self.maximum = x

        # Final graph
        self.graph_max(x, 0)

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


    def graph_max(self, max, skip):
        x = np.arange(0, 1, 0.0001)
        y = np.power(2, -2 * np.power((x-0.1)/0.9, 2)) * np.power(np.sin(5*np.pi*x),
            ↪ 6)

        # used ot plot animation
        if skip == 1:
            plt.clf()

        plt.plot(x, y, label=f'Current Iteration {self.k} and maximum {max} ')
        plt.plot(max, self.evaulate(max), 'rx')
        plt.xlabel('X-value')
        plt.ylabel('f(x)-value')
        plt.title('Hill Climbing')
        plt.legend(loc='best')

        if skip == 1:
            plt.pause(0.01)
        else:
            plt.show()
```

Simulated Annealing

```
# Code by Micah Runner or a monkey with a keyboard
import numpy as np
import matplotlib.pyplot as plt

class SimulatedAnnealing (object):

    # Initialize Class
    def __init__(self, rate=0.01, niter=1000, Trate=0.999, Tint=3000, Tfin=0.001):
        self.rate = rate # learning rate
        self.niter = niter # number of iterations
        self.Trate = Trate # the are at which T is augmented
        self.Tint = Tint # used to hold the initial temperature
        self.Tfin = Tfin # used to hold the final temperature value
        self.k = 0 # used to hold the current iteration

    def maximize(self):
        # pick random starting location
        x = np.random.uniform(0, 1)
        # used to show animation
        graphing = False
        if self.niter < 3000:
            graphing = True
        # need to change x if it is equal to 1 to stay within bounds
        if x > 0.9:
            x = x / 2 # just decided to cut it in half

        # set k = 0
        k = 0

        # graph initial point
        self.k = k
        self.maximum = x
        self.Tcurr = self.Tint
        self.graph_max(x, 0)

        # create empty arrays that will hold the values k-iterations and the
           ↪ corresponding x-values
        x_list = np.empty(0)
        k_list = np.empty(0)


        if graphing == True:
            plt.show()
            plt.ion()
            plt.figure()

        while (k < self.niter) and (self.Tcurr < self.Tfin):
            # use normal distribution to "randomly" move around
            x_prime = x + np.random.normal() * self.rate

            # Could not get rid of run time error without this -_-
            xp = ((self.evaluate(x) - self.evaluate(x_prime))/self.Tcurr)
            xp = np.round(xp, 6)
            uni = np.round(np.random.uniform(0, 1), 6)
            if np.exp(-xp) > uni:
                if 1 > x_prime > 0:
                    # x = x_prime
                    x = x_prime

                    # put graph here
                    self.k = k
                    if graphing == True:
                        self.graph_max(x, 1)

            if self.niter < 3000:
                x_list = np.append(x_list, x)
                k_list = np.append(k_list, k)

            # k++
            k += 1

            # change the rate of the T slowly
            self.Tcurr *= self.Trate

        if graphing == True:
            plt.ioff()
            plt.close()

        self.maximum = x

        self.graph_max(x, 0)

        if self.niter < 3000:
            self.graph_iterations(x_list, k_list)

    def minimize(self):
        # set k to 0
        self.k = 0
        graphing = False
        # set T to initial Temperature
        self.Tcurr = self.Tint

        # create array of random points
        x = np.random.randint(50, size=(50, 2))
        #x = np.vstack((x,x[0]))
        self.graph_TPS(x, 0)

        # Used to get show animation
        if graphing == True:
            plt.show()
            plt.ion()
            plt.figure()

        while self.k < self.niter and not(self.Tcurr < self.Tfin):
            # select the best of the different random tours
            x_prime = self.rand_select(x, int(x.size/4))

            # This is the minimizing function used correctly :)
            if np.random.uniform(0, 1) < np.exp((self.distance(x)-self.distance(
                ↪ x_prime))/self.Tcurr):
                if self.distance(x_prime) < self.distance(x):
                    x = x_prime
                    min = self.distance(x)
                    min_k = self.k
                    if graphing == True:
                        self.graph_TPS(x, 1)

            #print(self.k)
            # k++
            self.k += 1

            # T *= rate
            self.Tcurr *= self.Trate
        if graphing == True:
            plt.ioff()
            plt.close()

        print("Current Temp: ", self.Tcurr)
        print("Number of max iterations: ", self.k)
        print("Best distance: ", self.distance(x))
        self.graph_TPS(x, 0)
        self.k = min_k
        self.graph_TPS(x, 0)

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

    def rand_select(self, x, half):
        # shuffle x randomly :)
        x_a = x[np.random.choice(x.shape[0], half*2, replace=False), :]
        # swap two columns
        x_b = x.copy()
        indone = np.random.randint(0, half)
        indtwo = np.random.randint(0, half)
        temp = x_b[indone]
        x_b[indone] = x[indtwo]
        x_b[indtwo] = x[indone]
        # invert back half of list
        x_c = x.copy()
        x_c[half:] = x_c[half:][::-1]
        # invert random sub lists
        x_d = x.copy()
        start = np.random.randint(0, half)
        end = half + start
        if end >= x_d.size:
            end -= 1
        x_d[start:end] = x_d[start:end][::-1]

        distances = np.array([self.distance(x_a), self.distance(x_b), self.distance(
            ↪ x_c), self.distance(x_d)])

        min_distance = np.where(distances == np.amin(distances))
        min_distance = min_distance[0][0]
        #min_distance = 1
        #print(min_distance)
        if min_distance == 0:
            x_prime = x_a
            #print("Picked A")
        elif min_distance == 1:
            x_prime = x_b
            #print("Picked B")
        elif min_distance == 2:
            x_prime = x_c
            #print("Picked C")
        else:
            x_prime = x_d
            #print("Picked D")
        return x_prime

    def distance(self, city_list):
        # matrix of matrices. Numpy is great :)
        return np.sum(np.sqrt(np.square(city_list[1:, 0] - city_list[:int(city_list.
            ↪ size/2)-1, 0]) + np.square(city_list[1:, 1] - city_list[:int(city_list.
            ↪ size/2)-1, 1])))

    def graph_TPS(self, city_list, skip):
        # Fix matrix mess
        x_array, y_array = np.hsplit(city_list, 2)
        x_array = np.concatenate(x_array, axis=0)
        x_array = np.append(x_array, x_array[0]).tolist()
        y_array = np.concatenate(y_array, axis=0)
        y_array = np.append(y_array, y_array[0]).tolist()

        # turn on animation
        if skip == 1:
            plt.clf()

        # plot cities
        plt.plot(x_array, y_array, color='red', label=f'Current Iteration {self.k}
            ↪ and min {self.distance(city_list)}')
        plt.legend(loc='best')
        plt.scatter(x_array, y_array, marker='o')

        # plot numbers for cities
        for i in range(len(x_array)-1):
            plt.text(x_array[i] * (1+0.02), y_array[i] * (1+0.02), i, fontsize=12)
        plt.xlabel("City Location in x-direction")
        plt.ylabel("City Location in y-direction")
        plt.title('TSP using SA')
        if skip == 1:
            plt.pause(0.01)
        else:
            plt.show()

    def graph_max(self, max, skip):
        # arrays used to show results
        x = np.arange(0, 1, 0.0001)
        y = np.power(2, -2 * np.power((x - 0.1) / 0.9, 2)) * np.power(np.sin(5 * np.
            ↪ pi * x), 6)

        # used ot plot animation
        if skip == 1:
            plt.clf()

        # plotting stuff
        plt.plot(x, y, label=f'Current Iteration {self.k} and maximum {round(max, 5)}
            ↪  at Temp {round(self.Tcurr, 5)}')
        plt.plot(max, self.evaluate(max), 'rx')
        plt.legend(loc='best')

        if skip == 1:
            plt.pause(0.01)
        else:
            plt.show()

    def graph_iterations(self, x_list, k_list):
        plt.plot(k_list, x_list)
        plt.title('Number of Iterations vs X values')
        plt.ylabel('X Values')
        plt.xlabel("Number of k-iterations")
        plt.show()
        plt.close()
```

Problem 2

2.1 Statement

For this problem, you will duplicate the results presented in class on the Traveling Salesman Problem (TSP): use the simulated annealing algorithm to find an approximation of the optimal route. Demonstrate your code with 50 cities (randomly selected in a square). Use the normal distance function. Show the initial tour, final tour, iteration count and any details on restarts.

2.2 Method

The minimizing *Simulated Annealing* was used to solve the shortest path for a random list of cities that represent a *traveling salesman problem.* Unlike the maximizing function, the minimizing function requires to the inequality to be solved be reversed. This is shown in the equation below, which shows how the exponential will reach a low steady-state. This means that only an exponentially decreasing value will be accepted.

$$x_{prime} = random \quad arrangement \quad selection$$
$$if \quad rand_{uniform}(0,1) < e^{(f(x)-f(x_{prime})/T} : x = x_{prime}$$

The **rand_select()** function takes in the current city tour list as a *ndarray* from the *numpy* library, then creates four copies of the list. The first potential choice is a complete shuffle of all the cities. The second choice is a random row swap between two elements. The third choice is an inversion of the back half of the city tour list. The four choice is a random amount of elements within the city tour list. We then use the euclidean distance equation to find the minimum distance between the four types of random city tour lists. This is the value that is then returned as the x-prime value that will be used within the if-statement.

This class is the same as the maximum simulated annealing but the user must realize that the initial temperature should be a large-value and the final temperature value should be a low-value. The **minimize()** function must have a large amount of iterations, a large initial temperature, a very small temperature rate, and an extremely small final temperature. The user will be greeted with a random list of fifty cities having a random integer value in both the x & y values between the values of 0 to 50.

2.3 Results

We first began our testing with a small list of just 5-cities, which would require 120 iterations to find the lowest path between the cities if a brute force method was used instead of simulated annealing. We used the following values to create the instance of the class: rate = 0.01, niter = 30000, Trate = 0.9999, Tint = 3000, Tfin = 0.00001. Figures 9 and 10 show the initial tour of the cities and the final tour of the cities, respectively. Figure 11 shows at what iteration the minimized tour was found, which beat the amount of iterations needed for the brute-force method. We got extremely lucky and the minimized distance was found at just iteration 2, which *clearly proves that* bogo-sort is the best sort that one can use for any sorting needs.

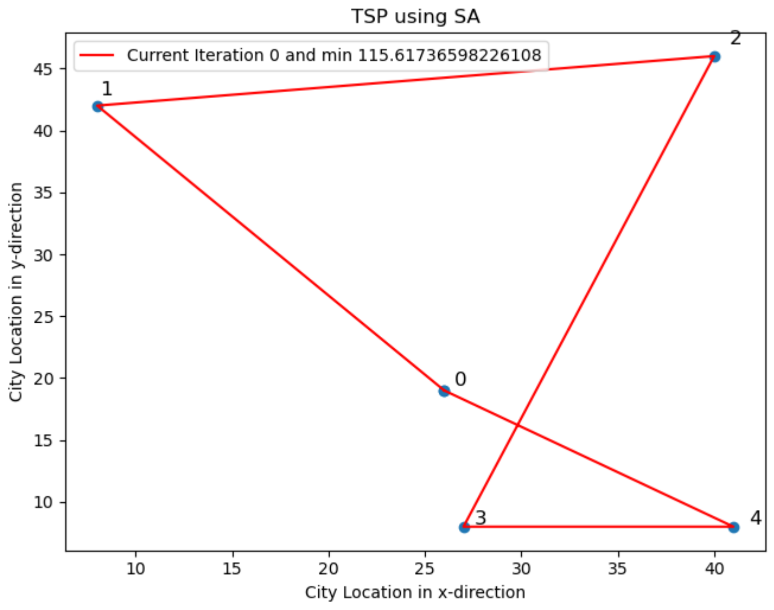

Figure 9: *Image of the initial 5-city tour that was randomly generated.*

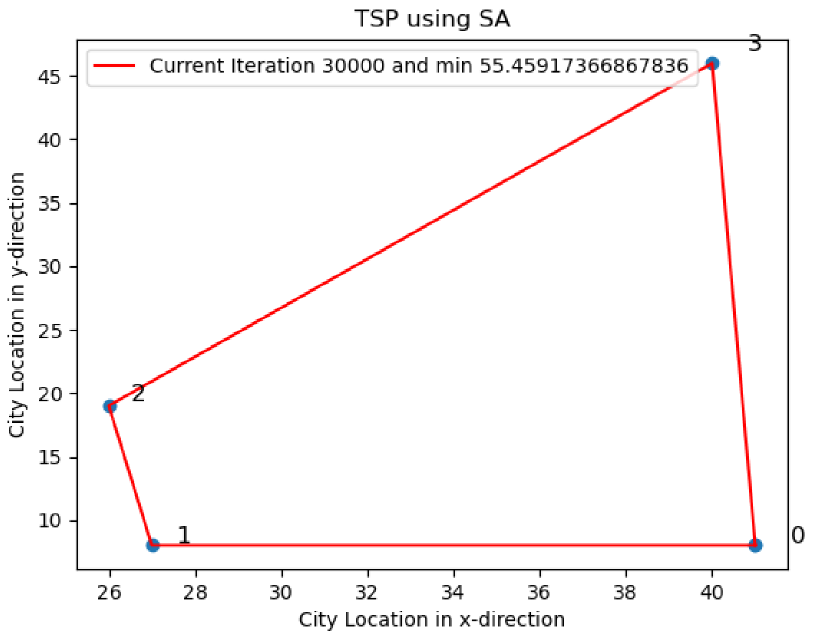

Figure 10: *Image of the last iteration of the 5-city tour that was randomly generated.*

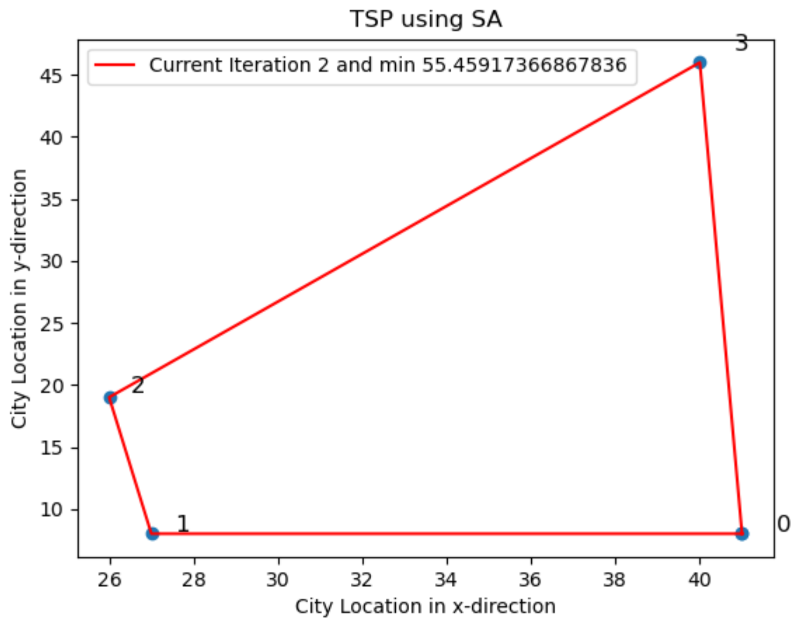

Figure 11: *Image of the minimized 5-city tour and what iteration it was found within.*

Next we tested the 50-city tour, which required the following initial class variables: rate = 0.01, niter = 3000000, Trate = 0.9999, Tint = 300000, and Tfin = 0.00001. This does take a very long time work on and so far, we have not been able to solve any city-tour list over 15-cities. This maybe due to the fact that we setup the minimizing function to solve the minimize path instead of circuit, but we graph the circuit. Figures 12 and 13 show the initial and final tour of the cities, while Figure 14 shows at what iteration the minimized function was found. We do however get a path reduction from the initial tour length of 1429 to the minimized tour of 459, which was found at iteration 1570.

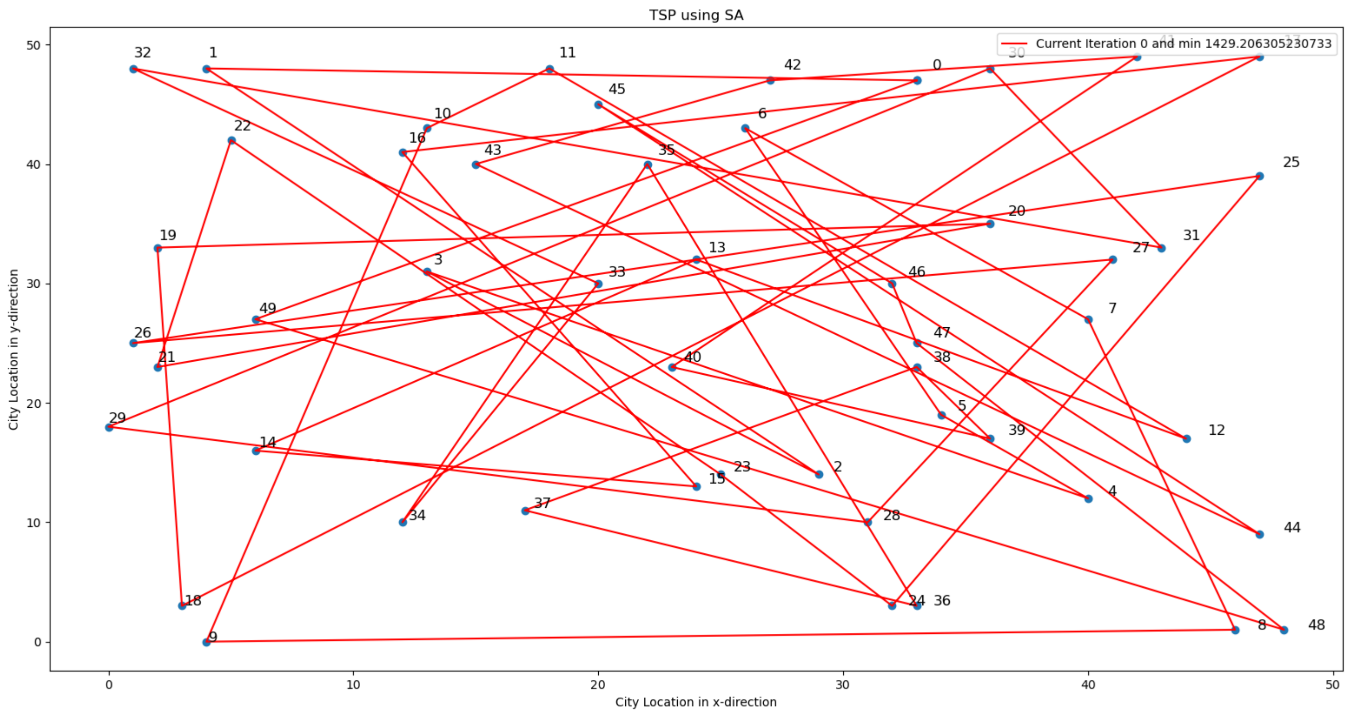

Figure 12: *Image of the initial 5-city tour that was randomly generated.*

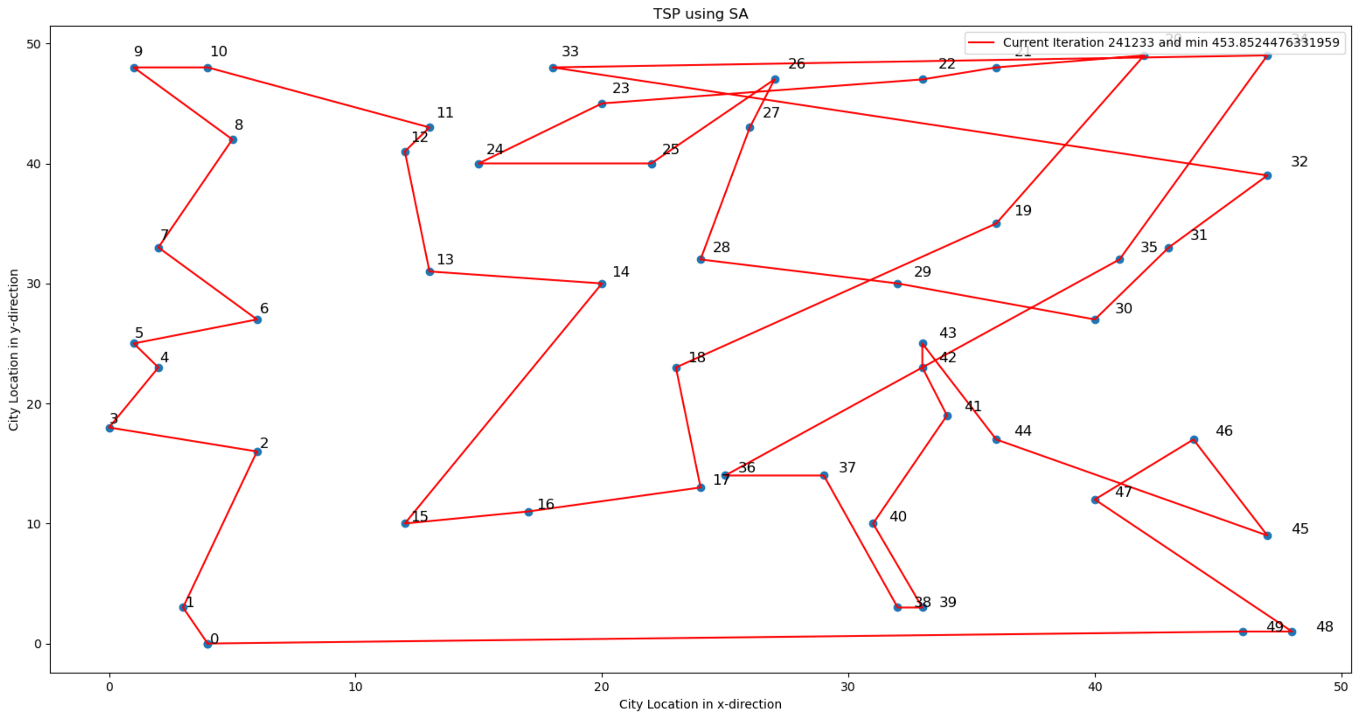

Figure 13: *Image of the last iteration of the 5-city tour that was randomly generated.*

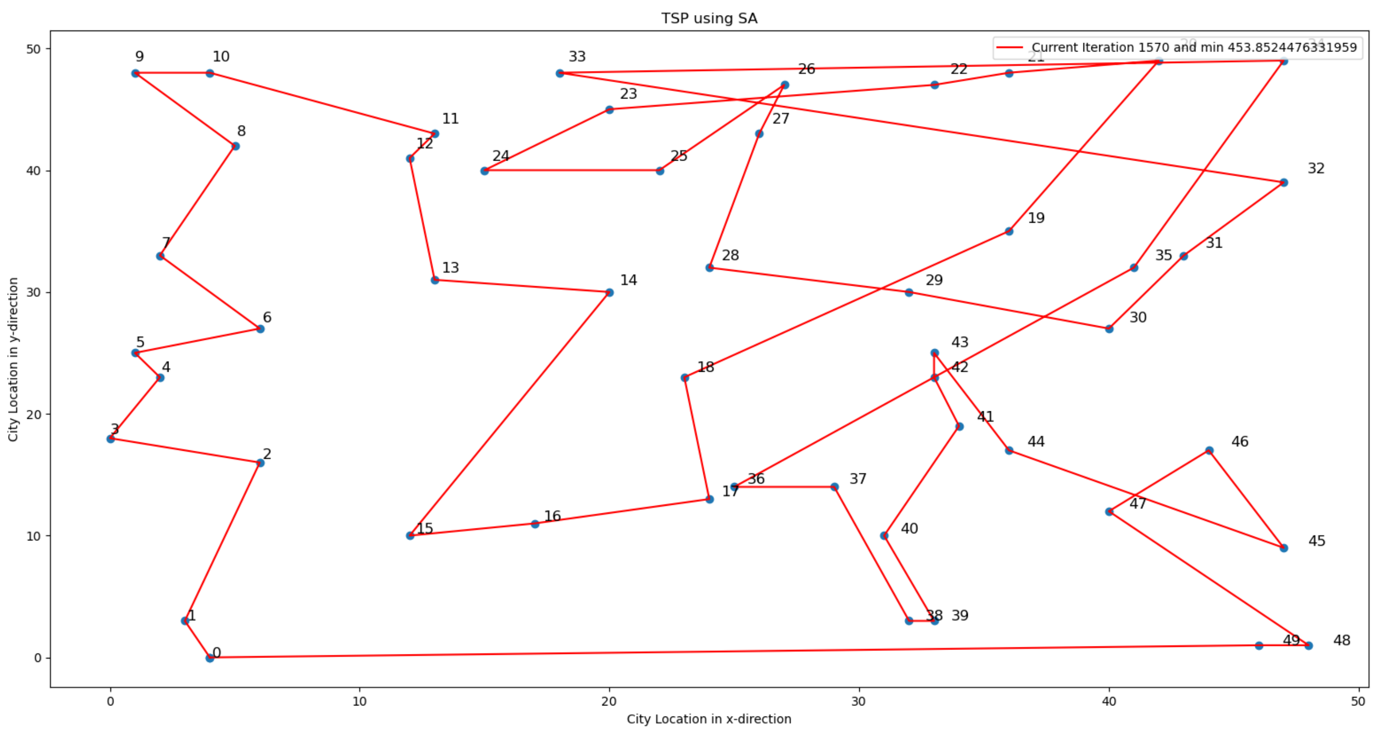

Figure 14: *Image of the minimized 5-city tour and what iteration it was found within.*

2.4 Code

Main File for Testing

```
import numpy as np, matplotlib.pyplot as plt
from Homework1 import HillClimbing, SimulatedAnnealing


def main():
    #hill()
    #maxim()
    min()


def hill():
    hill = HillClimbing(0.01, 10000)
    hill.fit()
    print("Max of : ", hill.evaulate(hill.maximum))
    print("Given the X value of: ", hill.maximum)
    print("Found at iteration: ", hill.k)


def maxim():
    akneel = SimulatedAnnealing(0.1, 2999, 0.999, 0.001, 3000)
    akneel.maximize()
    print("Max of : ", akneel.evaluate(akneel.maximum))
    print("Given the X value of: ", akneel.maximum)
    print("Found at iteration: ", akneel.k)
    print("tempature max is: ", akneel.Tcurr)

def min():
    akneel = SimulatedAnnealing(0.01, 3000000, 0.9999, 300000, 0.00001)
    akneel.minimize()
    print("Done")

if __name__ == "__main__":
    main()
```

Full Code used for both Classes

```
import numpy as np
import matplotlib.pyplot as plt
import copy
from matplotlib.animation import FuncAnimation


class HillClimbing (object):

    # Initialize Class
    def __init__(self, rate=0.01, niter=100):
        self.niter = niter # number of iterations
        self.rate = rate # used to hold the learning rate

    # Climbing for problem 1
    def fit(self):
        # create the loop variable k and x setting both to zero
        k = 0
        x = 0

        # graph initial point
        self.k = k
        self.maximum = x
        self.graph_max(x, 0)

        # use animation if short enough
        if self.niter < 30000:
            plt.show()
            plt.ion()
            plt.figure()

        # work our way through the climb until we hit max amount of user iterations
        for k in range(self.niter):
            self.cur_iter = k

            x_prime = x + np.random.random() * self.rate

            # if f(x_prime) > f(x)
            if self.evaulate(x_prime) > self.evaulate(x):
                x = x_prime
                # get the iteration that it took to get the value
                self.k = k
                if self.niter < 30000:
                    self.graph_max(x, 1)

        # end animation
        if self.niter < 30000:
            plt.ioff()
            plt.close()

        # get the maximum that was found
        self.maximum = x

        # Final graph
        self.graph_max(x, 0)

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

    def graph_max(self, max, skip):
        x = np.arange(0, 1, 0.0001)
        y = np.power(2, -2 * np.power((x-0.1)/0.9, 2)) * np.power(np.sin(5*np.pi*x),
            ↪ 6)

        # used ot plot animation
        if skip == 1:
            plt.clf()

        plt.plot(x, y, label=f'Current Iteration {self.k} and maximum {max} ')
        plt.plot(max, self.evaulate(max), 'rx')
        plt.xlabel('X-value')
        plt.ylabel('f(x)-value')
        plt.title('Hill Climbing')
        plt.legend(loc='best')

        if skip == 1:
            plt.pause(0.01)
        else:
            plt.show()
```

```
class SimulatedAnnealing (object):

    # Initialize Class
    def __init__(self, rate=0.01, niter=1000, Trate=0.999, Tint=3000, Tfin=0.001):
        self.rate = rate # learning rate
        self.niter = niter # number of iterations
        self.Trate = Trate # the are at which T is augmented
        self.Tint = Tint # used to hold the initial temperature
        self.Tfin = Tfin # used to hold the final temperature value
        self.k = 0 # used to hold the current iteration

    def maximize(self):
        # pick random starting location
        x = np.random.uniform(0, 1)
        # used to show animation
        graphing = False
        if self.niter < 3000:
            graphing = True
        # need to change x if it is equal to 1 to stay within bounds
        if x > 0.9:
            x = x / 2 # just decided to cut it in half

        # set k = 0
        k = 0

        # graph initial point
        self.k = k
        self.maximum = x
        self.Tcurr = self.Tint
        self.graph_max(x, 0)

        # create empty arrays that will hold the values k-iterations and the
           ↪ corresponding x-values
        x_list = np.empty(0)
        k_list = np.empty(0)


        if graphing == True:
            plt.show()
            plt.ion()
            plt.figure()

        while (k < self.niter) and (self.Tcurr < self.Tfin):
            # use normal distribution to "randomly" move around
            x_prime = x + np.random.normal() * self.rate

            # Could not get rid of run time error without this -_-
            xp = ((self.evaluate(x) - self.evaluate(x_prime))/self.Tcurr)
            xp = np.round(xp, 6)
            uni = np.round(np.random.uniform(0, 1), 6)

            if np.exp(-xp) > uni:
                if 1 > x_prime > 0:
                    # x = x_prime
                    x = x_prime

                    # put graph here
                    self.k = k
                    if graphing == True:
                        self.graph_max(x, 1)

            if self.niter < 3000:
                x_list = np.append(x_list, x)
                k_list = np.append(k_list, k)

            # k++
            k += 1

            # change the rate of the T slowly
            self.Tcurr *= self.Trate

        if graphing == True:
            plt.ioff()
            plt.close()

        self.maximum = x

        self.graph_max(x, 0)

        if self.niter < 3000:
            self.graph_iterations(x_list, k_list)

    def minimize(self):
        # set k to 0
        self.k = 0
        graphing = False
        # set T to initial Temperature
        self.Tcurr = self.Tint

        # create array of random points
        x = np.random.randint(50, size=(50, 2))
        #x = np.vstack((x,x[0]))
        self.graph_TPS(x, 0)

        # Used to get show animation
        if graphing == True:
            plt.show()
            plt.ion()
            plt.figure()

        while self.k < self.niter and not(self.Tcurr < self.Tfin):
            # select the best of the different random tours
            x_prime = self.rand_select(x, int(x.size/4))

            # This is the minimizing function used correctly :)
            if np.random.uniform(0, 1) < np.exp((self.distance(x)-self.distance(
                ↪ x_prime))/self.Tcurr):
                if self.distance(x_prime) < self.distance(x):
                    x = x_prime
                    min = self.distance(x)
                    min_k = self.k
                    if graphing == True:
                        self.graph_TPS(x, 1)

            #print(self.k)
            # k++
            self.k += 1

            # T *= rate
            self.Tcurr *= self.Trate
        if graphing == True:
            plt.ioff()
            plt.close()

        print("Current Temp: ", self.Tcurr)
        print("Number of max iterations: ", self.k)
        print("Best distance: ", self.distance(x))
        self.graph_TPS(x, 0)
        self.k = min_k
        self.graph_TPS(x, 0)

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

    def rand_select(self, x, half):
        # shuffle x randomly :)
        x_a = x[np.random.choice(x.shape[0], half*2, replace=False), :]
        # swap two columns
        x_b = x.copy()
        indone = np.random.randint(0, half)
        indtwo = np.random.randint(0, half)
        temp = x_b[indone]
        x_b[indone] = x[indtwo]
        x_b[indtwo] = x[indone]
        # invert back half of list
        x_c = x.copy()
        x_c[half:] = x_c[half:][::-1]
        # invert random sub lists
        x_d = x.copy()
        start = np.random.randint(0, half)
        end = half + start
        if end >= x_d.size:
            end -= 1
        x_d[start:end] = x_d[start:end][::-1]

        distances = np.array([self.distance(x_a), self.distance(x_b), self.distance(
            ↪ x_c), self.distance(x_d)])

        min_distance = np.where(distances == np.amin(distances))
        min_distance = min_distance[0][0]
        #min_distance = 1
        #print(min_distance)
        if min_distance == 0:
            x_prime = x_a
            #print("Picked A")
        elif min_distance == 1:
            x_prime = x_b
            #print("Picked B")
        elif min_distance == 2:
            x_prime = x_c
            #print("Picked C")
        else:
            x_prime = x_d
            #print("Picked D")
        return x_prime

    def distance(self, city_list):
        # matrix of matrices. Numpy is great :)
        return np.sum(np.sqrt(np.square(city_list[1:, 0] - city_list[:int(city_list.
            ↪ size/2)-1, 0]) + np.square(city_list[1:, 1] - city_list[:int(city_list.
            ↪ size/2)-1, 1])))

    def graph_TPS(self, city_list, skip):
        # Fix matrix mess
        x_array, y_array = np.hsplit(city_list, 2)
        x_array = np.concatenate(x_array, axis=0)
        x_array = np.append(x_array, x_array[0]).tolist()
        y_array = np.concatenate(y_array, axis=0)
        y_array = np.append(y_array, y_array[0]).tolist()

        # turn on animation
        if skip == 1:
            plt.clf()

        # plot cities
        plt.plot(x_array, y_array, color='red', label=f'Current Iteration {self.k}
            ↪ and min {self.distance(city_list)}')
        plt.legend(loc='best')
        plt.scatter(x_array, y_array, marker='o')

        # plot numbers for cities
        for i in range(len(x_array)-1):
            plt.text(x_array[i] * (1+0.02), y_array[i] * (1+0.02), i, fontsize=12)
        plt.xlabel("City Location in x-direction")
        plt.ylabel("City Location in y-direction")
        plt.title('TSP using SA')
        if skip == 1:
            plt.pause(0.01)
        else:
            plt.show()

    def graph_max(self, max, skip):
        # arrays used to show results
        x = np.arange(0, 1, 0.0001)
        y = np.power(2, -2 * np.power((x - 0.1) / 0.9, 2)) * np.power(np.sin(5 * np.
            ↪ pi * x), 6)

        # used ot plot animation
        if skip == 1:
            plt.clf()

        # plotting stuff
        plt.plot(x, y, label=f'Current Iteration {self.k} and maximum {round(max, 5)}
            ↪  at Temp {round(self.Tcurr, 5)}')
        plt.plot(max, self.evaluate(max), 'rx')
        plt.legend(loc='best')

        if skip == 1:
            plt.pause(0.01)
        else:
            plt.show()
    def graph_iterations(self, x_list, k_list):
        plt.plot(k_list, x_list)
        plt.title('Number of Iterations vs X values')
        plt.ylabel('X Values')
        plt.xlabel("Number of k-iterations")
        plt.show()
        plt.close()
```