Symbolic Regression Report

Name: Connor Mcleod (cmj2275), Daming Xing (dx2222)

Course Number: MECS 4510

Course Name: Evolutionary Computation & Design Automation

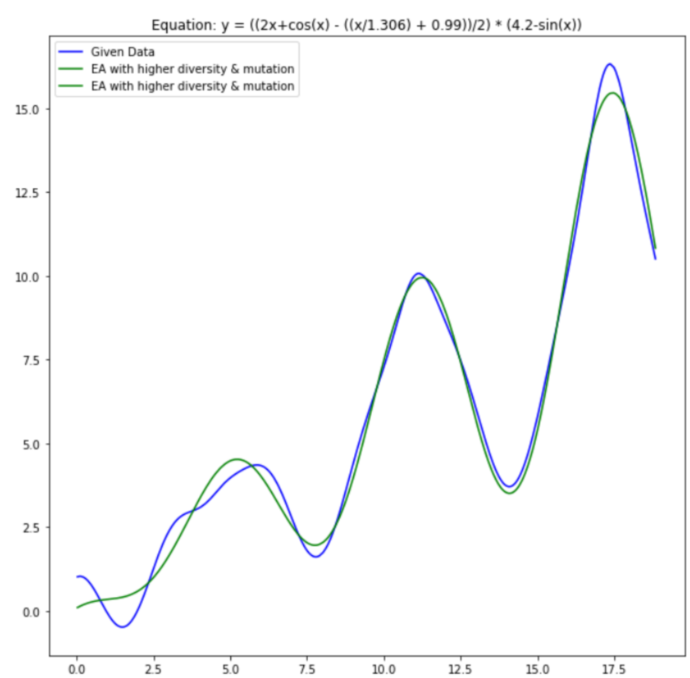
Date Submitted: 10/26/2021

Grade Hour Used: 0

Grade Hour Remaining: 198

Results:

Method	MSE achieved	Number of Iterations Max MSE achieved	Total Number of Iterations
Test Function (4X + 5)	0.0129	896	50,000
Random	2.4375	13,804	50,000
Random Mutation Hill Climber	18.285	18,514	50,000
Hill Climber Random Restart	3.8166	1,584	50,000
Genetic Algorithm With Random Diversity and Crossover	4.664	305	50,000
Genetic Algorithm With Random Diversity, Mutation, and Crossover	3.806	807	50,000
Genetic Algorithm with Incremental Length Addition, Mutation, Crossover	0.1714	9,525	50,000



Methods:

Representation Method Used:

The representation used in this assignment is a simple listing of X and Y coordinates, with the binary heap represented as a list up to size 255 without the initial ‘null’ value. The first element in the list is ‘nah’, which represents a null value. Methods use varying sized lists according to their depth size. Mean Squared Error (MSE) is utilized to determine error and fitness of the found equations. 80% of the data were used for training and 20% of the data were used for validation.

Random Search Method:

Random search method is a simple search method that consistently calculates a binary heap of random depth (up to 8, for a binary heap of length 255). Once the binary heap is generated, the equation is passed into an evaluating routine which compiles the equation and then calculates an array of Y values which are then used to calculate the MSE of the new equation to the standard equation points. The random search method then returns the final evaluated error, equation, and calculated “Y” values for the equation.

Hill Climber Method:

Several different variations of hill-climber were used to solve this problem, including a random mutation hill climber, a random restart hill climber.

The random mutation hill climber generates a first generation of an assigned depth, which it then immediately calculates the “Y” values for, and a corresponding MSE valuation. Then, for a set number of iterations, it passes that generation of the equation through a mutation function which mutates a number of values of the binary heap based on an assigned mutation rate in order to produce a new equation. If the MSE of this new binary heap (BH) decreases, the new BH is kept and the old is tossed. This process is iterated blindly until complete.

A random restart hill climber method was also used, in which a first random equation is generated and evaluated for a set number of trials. During each of these trials, a number of elements of the binary heap are mutated based on an assigned mutation rate. After a number of trials have elapsed, the random restart hill climber resets to a random first generation BH in order to find a separate local maximum. After a number of iterations, the hill climber ceases operations and returns the best final equation found, as well as the calculated “Y” values and error.

Evolutionary Algorithm Method:

The evolutionary algorithm built in this class consisted of a 50% mutation rate, starting initially from a randomly generated population. Three separate evolutionary algorithms were made.

First, a basic crossover function was built, which performs only crossover with randomly generated functions. When passed into the crossover function, a random node in the binary heap of both parents is selected, and all their children/grandchildren are identified. These two binary heaps are removed from both equations, and swapped into the respective positions of the other removed binary heap. Possible syntax errors are rectified, and returned as two new children. The crossover function results in two “children”, which are then compared and the child with the least MSE is kept, while the other is discarded. This continues for a specified number of iterations, utilizing crossover with the randomly generated

Secondly, an evolutionary algorithm with added diversity was created in which half of the binary heap is mutated upon each iteration, and crossover between the two functions happens similarly to the last function. However, random diversity is added in the new function which allows for random crossover to other local maximums with higher ease.

Thirdly, an evolutionary algorithm was created which mutates via crossover, adds random diversity in the crossover solution, and further mutates the pool of points. This method results in a much more rapid increase towards the total maximum solution, and does not halt around local maximums as much as previous iterations.

Finally, an evolutionary algorithm was created which randomly assigned a BH, and gradually lengthened the depth of the heap from 3 to 8. Each iteration tested mutations of the heap, with every 10 and 50 iterations testing completely random additions to the heap to jump local maximums. Once a complete binary heap is built, a second binary heap was built using the same method. These two BH, named the “Father” and “Mother”, were subjected to crossover using the same methods as before to produce two children. The best of all four is kept as the “champion”, and the total iterative process starts again. Once another “champion” is created, it is subject to crossover with the previous champion to produce two more children, and the best of those is kept on as champion.

Evolutionary Algorithm Selection Methods:

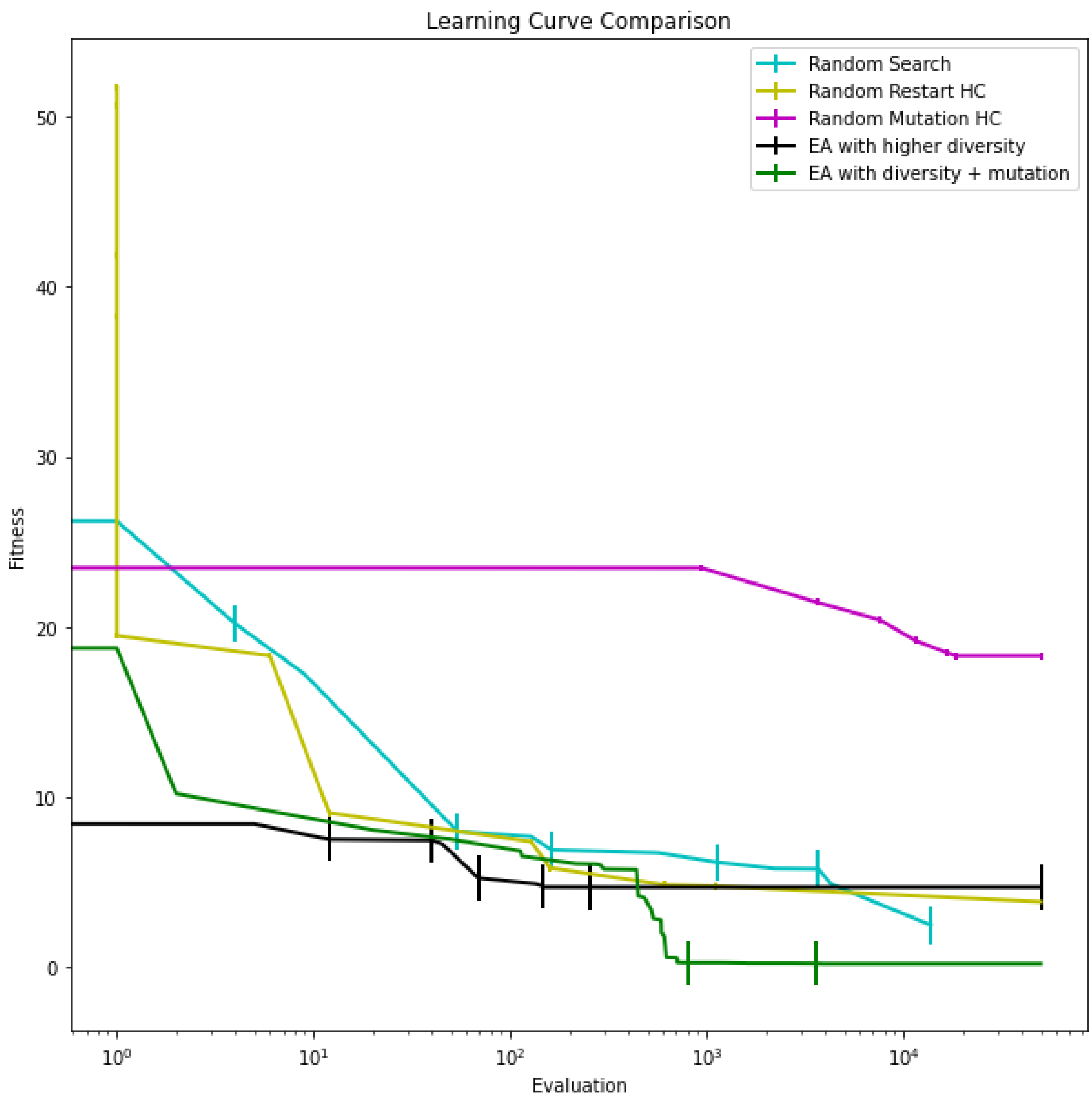
The selection process in all evolutionary algorithms consisted of selected random numbers in the binary heap. From there, every child and grandchild of the selected point was selected. This happened for two individual points in the two individual equations; the children and grandchildren of that point were deleted from their respective equations and then swapped. This involves crossover, with a truncation based selection method of choosing the top 50% of performing equations to continue forward.

Analysis of Performance:

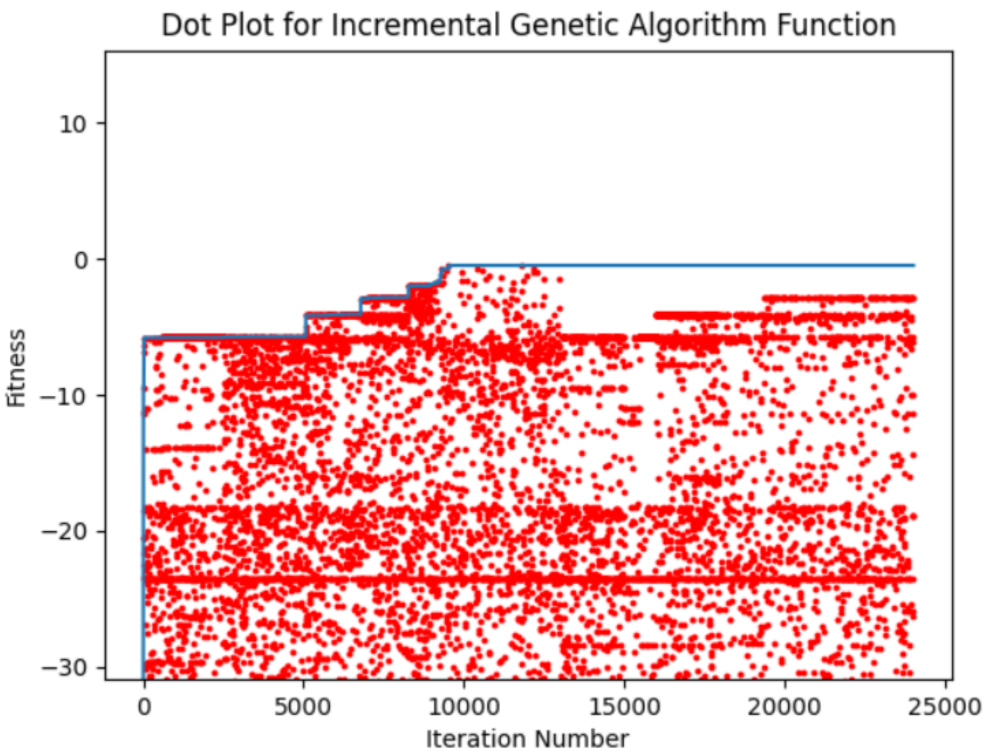
Overall, the performance of the EA was far better than the performance of both the Hill Climber and the Random Search. The random search clearly is limited by lack of improvement over time, whereas the hill climber is limited by local maximums and eventually gets to the point where it no longer can make meaningful improvements over time. The genetic algorithm/EA worked the best, with a maximum MSE of 3.806. This was a meaningful improvement over the other examples, and demonstrated the versatility of crossover with random mutation. It is possible that reducing the depth of mutation towards the tail-end of the iterations could result in a marginally better performance from the binary heap, but as it stands, crossover with random mutation does a very good job of approximating the given points. Gradually increasing depth while mutating also provided some great results (MSE below 0.5), but the complexity of the operation was lengthy and unreliable over small numbers of iterations. It also resulted in well-defined local maximums at certain depths (I.E. MSE at approx. 5.7 for a depth of 3). These local maximums are visible as horizontal clusters of points in the below Dot Plot. Overall,

Plots:

Learning Curve

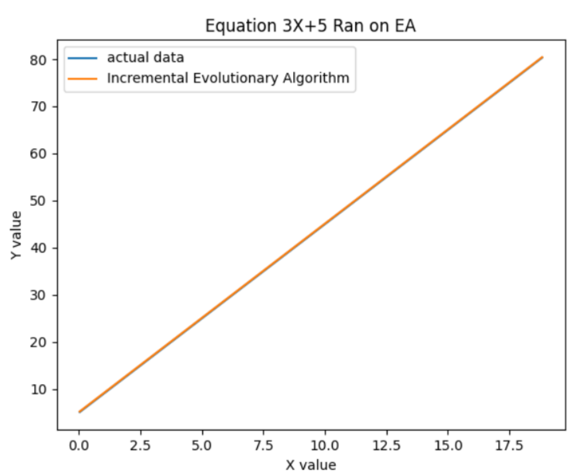


Dot Plot



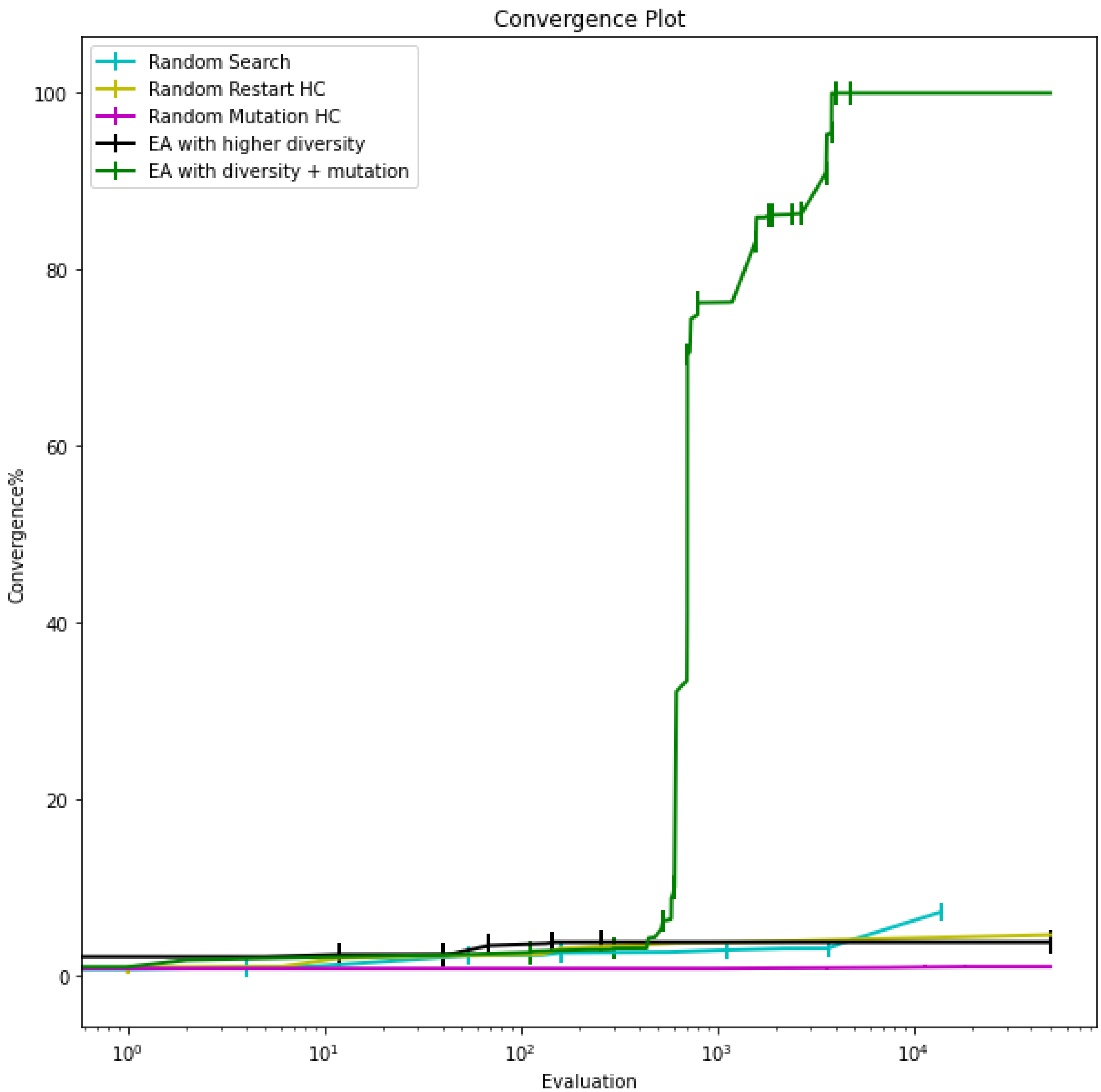
Simpler problem(s) tested for debugging

Test equation 3X + 5 ran on the evolutionary algorithm:
Returned MSE of 0.0129, with final BH:

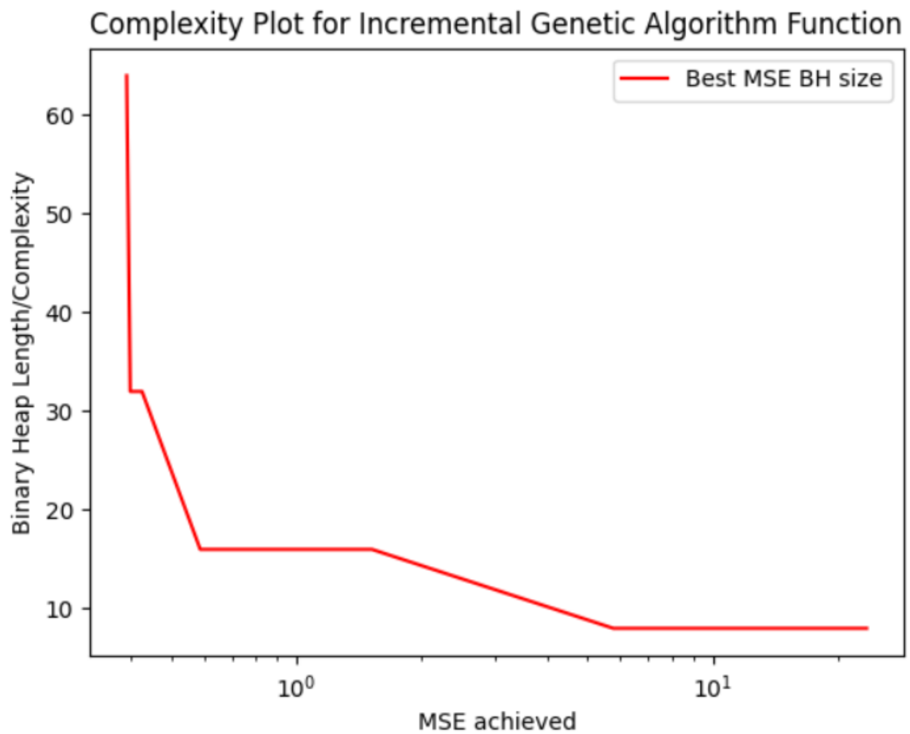


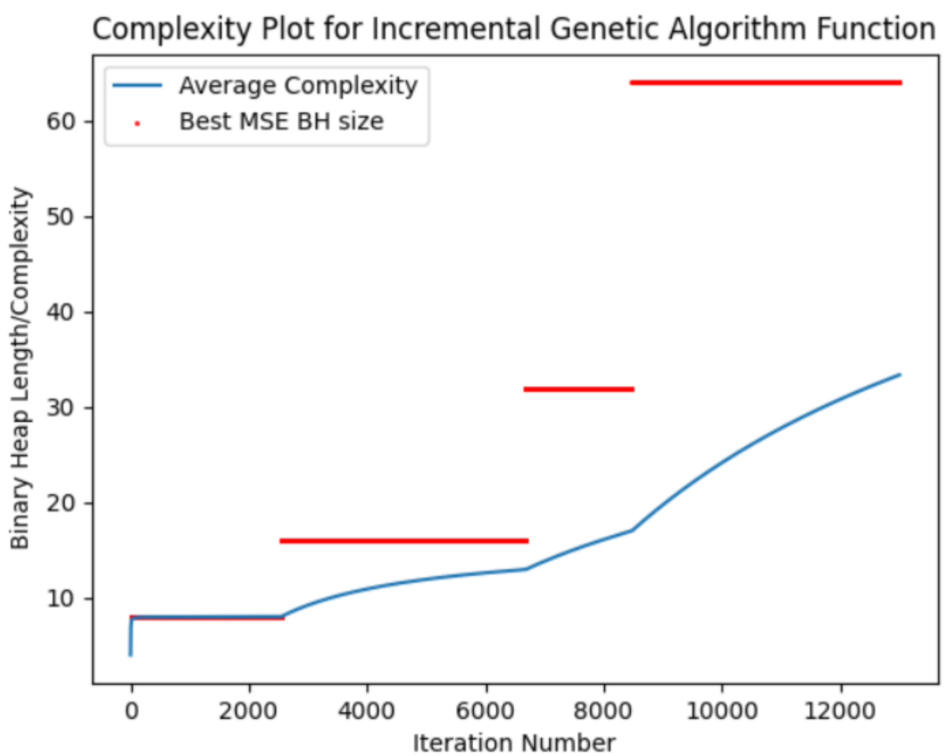
['nah', '+', '+', 3.717, '/', '*', '-', 'cos', 9.264, 6.573, 'x', 3.9986, 1.505, 9.509, 8.846, 7.923, 1.1243, 3.8865, 'x', 'x', 'x', 3.4494, 6.5028, 7.443, 7.3091, 5.6983, 7.5813, 3.574, 7.4341, 7.4046, 'x', 5.0694]

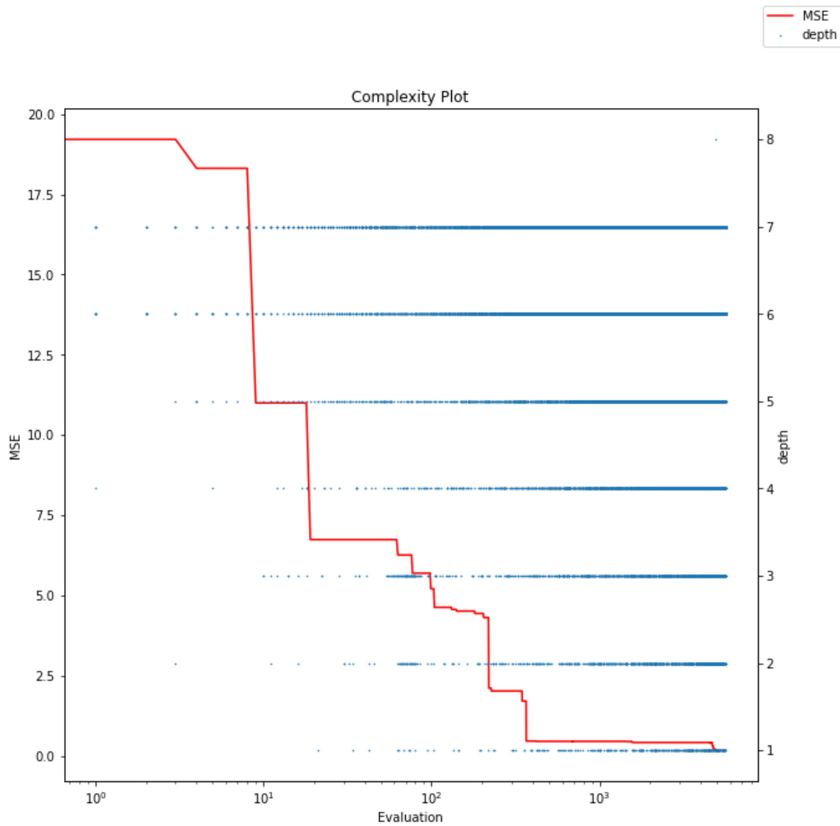
Convergence Plot



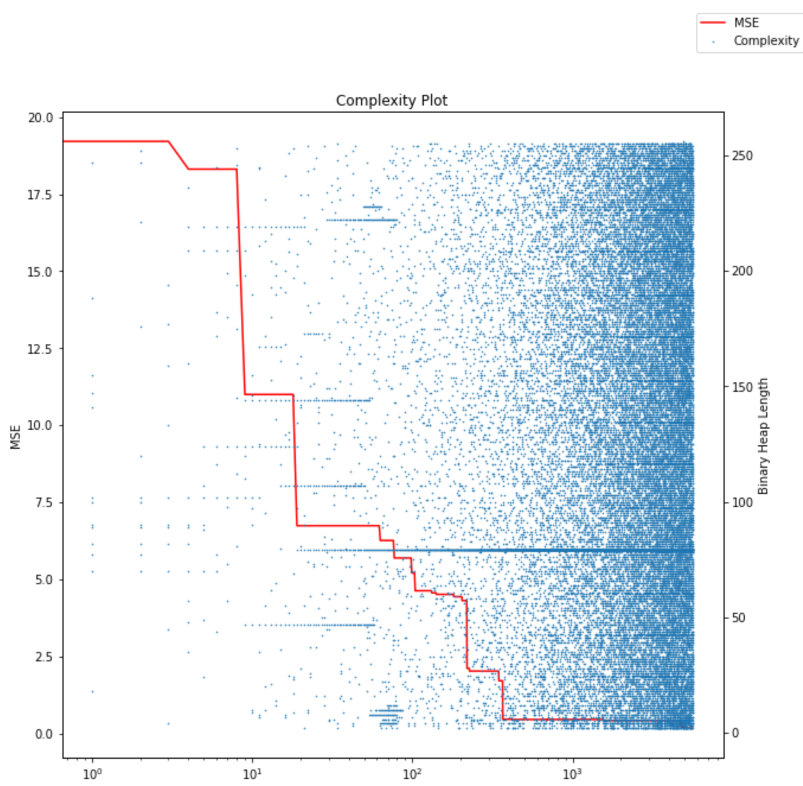
Complexity Plot





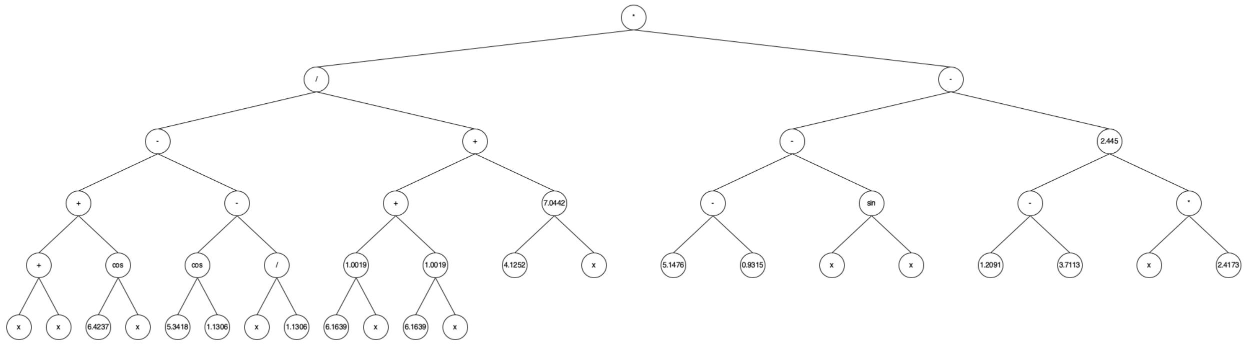


MSE
depth
Complexity Plot
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
MSE
8
7
6
5
4
3
2
1
depth
10^0
10^1
10^2
10^3
Evaluation



MSE
Complexity
Complexity Plot
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
MSE
250
200
150
100
50
0
Binary Heap Length
10^0
10^1
10^2
10^3
Evaluation

Automatically draw tree representing best solution



Symbolic Regression Timelapse Video

https://youtu.be/Q6RwjZ3JXLk

Appendix

```
# coding: utf-8


# In[611]:




import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
import copy
import math
import tqdm
from datetime import datetime


today = datetime.now()
d = today.strftime("%Y-%m-%d-%H-%M")
path = "/home/dxing/Desktop/ea_symbolicRegression/"




filename = "data"
df = pd.read_csv(f'{filename}.txt', header = None, sep=", ", names=["x",
'y'], engine='python')
operators1 = ['+', '-', '*', '/', 'sin', 'cos', 'const', 'x']
```

```
operators2 = ['const', 'x']


num_evaluation = 50000
depth = 8


x_s = df["x"].to_numpy()
y_s = df["y"].to_numpy()
y_s = [float(y) for y in y_s]




# In[614]:




def find_all_children(arr):
   """
   random pick a parent node and return it in a list with ALL its children
(children, grand-children)
   """


   res = []
   ptr = 0
   n = random.randint(2*len(arr)//5, len(arr)//2) # len(arr)-1
   res.append(n)


   while res[ptr] <= len(arr)-1:
       if 2 * res[ptr] <= len(arr)-1:
           res.append(2*res[ptr])
       else: break
       if 2 * res[ptr] + 1 <= len(arr)-1:
           res.append(2*res[ptr]+1)
       else: break


       ptr += 1
   return res




def merge(dict1, dict2):
   """merge two dictionaries"""
   res = {**dict1, **dict2}
   return res
```

```
def random_generate(depth):
   """
   generate a sequence of binary heap
   INPUTS:
       depth: depth of the binary tree -> int

   OUTPUT:
       binary heap
   """

   max_length = 2 ** depth - 1

   length = random.randint(1, max_length)
   first_half = length//2
   second_half = length - first_half

   res = ['nah']

   for i in range(1, first_half):
       pick = random.choice(operators1)

       if (pick == res[i-1]) and (res[i//2] == '-'):
           tmp = copy.deepcopy(operators1)
           tmp.remove(res[i-1])
           pick = random.choice(tmp)

       if pick == "const":
           pick = round(random.random() * 10 + 0.1, 4)

       res.append(pick)

   for i in range(first_half, length):

       pick = random.choices(population=operators2, weights=(0.5, 0.5))[0]

       if (pick == res[i-1]) and (res[i//2] == '-'):
           tmp = copy.deepcopy(operators2)
           tmp.remove(res[i-1])
           pick = random.choice(tmp)

       if pick == "const":
           pick = round(random.random() * 10 + 0.1, 4)
```

```python
        res.append(pick)

    return res


# In[616]:


def mutate(binary_heap, mutate_rate):

    binaryheap = copy.deepcopy(binary_heap)

    ele = int(len(binaryheap) * mutate_rate)

    for i in range(ele):
        ptr = random.randint(1, len(binaryheap)-1)

        if ptr < len(binaryheap)//2:
            pick = random.choice(operators1)

            if (pick == binaryheap[ptr-1]) and (binaryheap[ptr//2] == '-'):
                tmp = copy.deepcopy(operators1)
                tmp.remove(binaryheap[ptr-1])
                pick = random.choice(tmp)

        else:
            pick = random.choice(operators2)

            if (pick == binaryheap[ptr-1]) and (binaryheap[ptr//2] == '-'):
                tmp = copy.deepcopy(operators2)
                tmp.remove(binaryheap[ptr-1])
                pick = random.choice(tmp)

        if pick == "const":
            pick = round(random.random() * 10 + 0.1, 4)

        binaryheap[ptr] = pick

    return binaryheap
```

```
def evaluate_binary_heap(binary_heap, x):

   """
   Evaluate binary from back to front

   """

   bh = copy.deepcopy(binary_heap)
   for i in range(len(bh)-1, 0, -1):
       try:
           if bh[i] == 'x':
               bh[i] = x

           elif bh[i] == '+':

               bh[i] = bh[2*i] + bh[2*i+1]

           elif bh[i] == '-':

               bh[i] = bh[2*i] - bh[2*i+1]

           elif bh[i] == '*':

               bh[i] = bh[2*i] * bh[2*i+1]

           elif bh[i] == '/':

               bh[i] = bh[2*i] / bh[2*i+1]

           elif bh[i] == 'sin':

               bh[i] = np.sin(bh[2*i])

           elif bh[i] == 'cos':

               bh[i] = np.cos(bh[2*i])
       except:
#              return random.random()
```

```
           return 10
#             pass
#             print(f"Math invalid")
#             print(binary_heap)

   return bh[1]



# In[623]:



def calculate_y(x_s, equation):
   calculated_y = []
   for ele in x_s:
       res = evaluate_binary_heap(equation, x=ele)
       calculated_y.append(res)
   return calculated_y





def calculate_mse(y, y_hat):
   """
   use np to calculate mse
   """
#     calculated_y = np.array(calculate_y(x_s, equation=equation))
#     calculated_y = [x.astype(float) for x in calculated_y] # cast to
np.float
#     y = [n.astype(float) for n in y]
#     y_hat = [n.astype(float) for n in y_hat]
#     print(len(y) == len(y_hat))
   y = np.array(y)
   y_hat = np.array(y_hat)
   mse = np.square(np.subtract(y, y_hat)).mean()
   return mse



# In[503]:



def calc_mse(y, y_hat):
   """
```

```
    home-made mse calculation
    """
    errors_sq = []
    tmp = 0
    for i in range(len(y)):
        try:
            e = (y[i] - y_hat[i]) ** 2
            tmp = e
        except TypeError:
            pass
        errors_sq.append(tmp)
    return round(sum(errors_sq)/len(errors_sq), 8)




def generate_population(population, depth):
    """
    generate some number of population
    """
    pool = {}

    for i in range(population):
        equation = random_generate(depth)
        y_calculated = calculate_y(x_s, equation=equation)
        mse = calc_mse(y_calculated, y_s)

        pool[mse] = equation

    return pool


# In[564]:


def crossover(parent1, parent2):
    """
    crossover operation
    1. random pick a point at parent1
    2. random pick a point at parent2
```

```
    3. completely swap the points and their children
    4. return two trees
    """

    p1 = copy.deepcopy(parent1)
    p2 = copy.deepcopy(parent2)

    rm_idxs1 = find_all_children(p1) # top node and its grand-grandchildren
- random
    rm_idxs2 = find_all_children(p2) # top node and its grand-grandchildren
- random

#     print(rm_idxs1)
#     print(rm_idxs2)

    removed1 = [p1[x] for x in rm_idxs1]
    removed2 = [p2[x] for x in rm_idxs2]

    for idx in rm_idxs1:
        p1[idx] = 'nah'

    for idx in rm_idxs2:
        p2[idx] = 'nah'


    # if they are in the same length
    if len(removed1) == len(removed2):
        for ele in removed1:
            for i in range(1, len(p2)):
                if p2[i] == "nah":
                    p2[i] = ele
                    break

        for ele in removed2:
            for i in range(1, len(p1)):
                if p1[i] == "nah":
                    p1[i] = ele
                    break

    # if not
    elif len(removed1) > len(removed2):


        ptr_rm = 0
        ptr_idx = 0
```

```
        while ptr_rm < len(removed1):
            while ptr_idx < len(rm_idxs2):
                p2[rm_idxs2[ptr_idx]] = removed1[ptr_rm]
                ptr_idx += 1
                ptr_rm += 1

            p2.append(removed1[ptr_rm])
            ptr_rm += 1

        for ele in removed2:
            for idx in rm_idxs1:
                p1[idx] = ele
                break
    else:
        ptr_rm = 0
        ptr_idx = 0
        while ptr_rm < len(removed2):
            while ptr_idx < len(rm_idxs1):
                p1[rm_idxs1[ptr_idx]] = removed2[ptr_rm]
                ptr_idx += 1
                ptr_rm += 1

            p1.append(removed2[ptr_rm])
            ptr_rm += 1

        for ele in removed1:
            for idx in rm_idxs2:
                p2[idx] = ele
                break



    for i in range(1, len(p1)):
        if p1[i] == 'nah':
#             p1[i] = 'x'
            p1[i] = round(random.random() * 10 + 0.1, 4) # not the best
    for i in range(1, len(p2)):
        if p2[i] == 'nah':
#             p2[i] = 'x'
            p2[i] = round(random.random() * 10 + 0.1, 4) # not the best

    child1 = p1
    child2 = p2
```

```
    return [child1, child2]


def mutate_pool(pool: dict, mutate_rate: float)-> dict:
    """
    mutate a pool of population by given mutation rate
    """
    copied_pool = copy.deepcopy(pool)

    new_pool = {}

    number = int(len(copied_pool) * mutate_rate)

    for i in range(number):
        keys = list(copied_pool.keys())
        random.shuffle(keys)
        mutant = mutate(copied_pool[keys[0]], mutate_rate=0.05)
        y_hat = calculate_y(x_s, equation=mutant)
        mse = calc_mse(y_s, y_hat)
        new_pool[mse] = mutant
    pool = merge(new_pool, copied_pool)
    return pool


def random_search(num, x_s, y_s, depth):

    """
    random search
    """
    evaluation = []
    error = []
    final_equation = []
    y_calculated = []

    for i in range(num):

        if i % 100 == 0:
            print(f"{i/num_evaluation * 100} % complete")

        calculated_y = []
```

```
        equation = random_generate(depth=depth)

        for ele in x_s:
            res = evaluate_binary_heap(equation, x=ele)
            calculated_y.append(res)

        calculated_y = np.array(calculated_y)
        try:
            mse = (np.square(y_s - calculated_y)).mean(axis=0)
        except:
            print("raise MSE error")

        if len(error) == 0:
            error.append(mse)
            evaluation.append(i)
            final_equation = equation
            y_calculated = calculated_y

        elif mse < error[-1]:
            error.append(mse)
            evaluation.append(i)
            final_equation = equation
            y_calculated = calculated_y

    return [evaluation, error, final_equation, y_calculated]



# In[589]:



def random_mutation_hill_climber(num, x_s, y_s, depth):
    """
    random hill climber
    """

    evaluation = []
    error = []
    final_equation = []
    y_calculated = []


    first_gen = random_generate(depth=depth)
```

```
    y_cal = calculate_y(x_s=x_s, equation=first_gen)
    mse = calculate_mse(y_s, y_cal)

    for i in range(0, num):

        mutant = mutate(first_gen, mutate_rate=0.01)
        calculated_y = calculate_y(x_s, equation=mutant)
        mse = calculate_mse(calculated_y, y_s)

        if len(error) == 0:
            error.append(mse)
            evaluation.append(i)
            final_equation = mutant
            y_calculated = calculated_y

        elif mse < error[-1]:
            error.append(mse)
            evaluation.append(i)
            final_equation = mutant
            y_calculated = calculated_y

        if i % 100 == 0:
            print(f"{i/num_evaluation * 100} % complete")

    return [evaluation, error, final_equation, y_calculated]



# In[590]:



def random_restart_hill_climber(num_eval, x_s, y_s, depth, num_tries):
    evaluations = []
    errors = []
    final_equation = []
    y_calculated = []
    counter = num_eval

    while counter > 0:

        first_gen = random_generate(depth=depth)
        y_cal = calculate_y(x_s=x_s, equation=first_gen)
        mse = calculate_mse(y_s, y_cal)
```

```
        if (num_eval - counter) % 100 == 0:
            print(f"{(num_eval - counter)/num_evaluation*100} % complete")

        counter -= 1

        if len(errors) == 0:
            errors.append(mse)
            evaluations.append(num_eval - counter)
            final_equation = first_gen
            y_calculated = calculate_y(x_s, equation=first_gen)

        while num_tries > 0 and counter > 0:

            mutant = mutate(first_gen, mutate_rate=0.01)
            calculated_y = calculate_y(x_s, equation=mutant)
            new_mse = calculate_mse(calculated_y, y_s)

            if (num_eval - counter) % 100 == 0:
                print(num_eval - counter)

            if new_mse < errors[-1]:
                errors.append(new_mse)
                evaluations.append(num_eval - counter)
                final_equation = mutant
                y_calculated = calculate_y(x_s, mutant)
            else:
                num_tries -= 1

        if mse < errors[-1]:
            errors.append(mse)
            evaluations.append(num_eval - counter)
            final_equation = first_gen
            y_calculated = calculate_y(x_s, equation=first_gen)

    return [evaluations, errors, final_equation, y_calculated]


# In[548]:


def evol_algo(num_eval, x_s, y_s, depth, init_pop):
```

```
    """
    selection 50%
    mutation are built-in to crossover
    """

    evaluations = []
    errors = []
    final_equation = []
    y_calculated = []

    counter = num_eval

    pool = generate_population(depth=depth, population=init_pop)
    # print(f"pool = {len(pool)}")
    num_keys = len(pool)
    try:
        while counter > 0 and num_keys > 0:
            children_pool = {}

            keys = sorted(pool.keys(), reverse=False)
#               keys = list(pool.keys())
            for i in range(0, len(keys)-1, 2):
                kid1, kid2 = crossover(pool[keys[i]], pool[keys[i+1]])

                y_hat_kid1 = calculate_y(x_s=x_s, equation=kid1)
                y_hat_kid2 = calculate_y(x_s=x_s, equation=kid2)
    #             children_pool[calculate_mse(y_s, y_hat_kid1)] = kid1
    #             children_pool[calculate_mse(y_s, y_hat_kid2)] = kid2

                children_pool[calc_mse(y=y_s, y_hat=y_hat_kid1)] = kid1
                children_pool[calc_mse(y=y_s, y_hat=y_hat_kid2)] = kid2

            # print(f"children pool = {len(children_pool)}")

            merge_pool = merge(children_pool, pool)
            # print(f"merged pool = {len(merge_pool)}")

            pool = merge_pool

            temp_pool = {}
```

```
        keys = sorted(pool.keys(), reverse=False)
        half_keys = keys[0:9*(len(keys)//10)] # selection X%
        num_keys = len(half_keys)
        print(f"# of keys = {num_keys}")

        for i in range(num_keys):
            temp_pool[keys[i]] = pool[keys[i]]
        # print(f"top pool = {len(temp_pool)}")

        lowest_error = min(half_keys)
        print(f"lowest error = {lowest_error}")
        if (len(errors) == 0) or (lowest_error < errors[-1]):
            evaluations.append(num_eval - counter)
            final_equation = temp_pool[lowest_error]
            y_calculated = calculate_y(x_s, final_equation)
            errors.append(lowest_error)

            plt.figure(figsize=(10,10))
            plt.gca().set_aspect('equal')
            plt.plot(x_s, y_s, 'r--',label="Given Data")
            plt.plot(x_s,y_calculated, label= "Learning Data")
            plt.legend()
            plt.xlim(-1,20)
            plt.ylim(-2,20)
            plt.title(f"MSE = {round(lowest_error, 3)}")
            plt.savefig(f'{path}screenshots/foo_{num_eval -
counter}.png',
                        bbox_inches='tight')
            plt.close()


            with open(f"{path}tmp/ea_curve.txt",'a') as e:
                e.write(str(num_eval-counter))
                e.write(', ')
                e.write(str(lowest_error))
                e.write('\n')

            with open(f"{path}tmp/ea_yhat.txt",'w') as e:
                for ele in y_calculated:
                    e.write(str(ele))
                    e.write('\n')
```

```
                with open(f"{path}tmp/ea_equation.txt",'w') as e:
                    for ele in final_equation:
                        e.write(str(ele))
                        e.write('\n')



            pool = temp_pool
            counter -= 1

#         if counter % 10 == 0:
            print(f"counter = {num_eval - counter}")
    except ValueError:
        pass
    return [evaluations, errors, final_equation, y_calculated]



# In[569]:



def evol_algo_div(num_eval, x_s, y_s, depth, init_pop):
    """
    selection 50%
    mutation are built-in to crossover
    adding some random diversity
    """

    evaluations = []
    errors = []
    final_equation = []
    y_calculated = []

    counter = num_eval

    pool = generate_population(depth=depth, population=init_pop)
    # print(f"pool = {len(pool)}")
    num_keys = len(pool)
    try:
        while counter > 0 and num_keys > 0:
            children_pool = {}

            keys = sorted(pool.keys(), reverse=False)
```

```
#             keys = list(pool.keys())
            for i in range(0, len(keys)-1, 2):
                kid1, kid2 = crossover(pool[keys[i]], pool[keys[i+1]])

                y_hat_kid1 = calculate_y(x_s=x_s, equation=kid1)
                y_hat_kid2 = calculate_y(x_s=x_s, equation=kid2)

                children_pool[calc_mse(y=y_s, y_hat=y_hat_kid1)] = kid1
                children_pool[calc_mse(y=y_s, y_hat=y_hat_kid2)] = kid2

            # print(f"children pool = {len(children_pool)}")

            merge_pool = merge(children_pool, pool)
            # print(f"merged pool = {len(merge_pool)}")

            pool = merge_pool

            temp_pool = {}

            keys = sorted(pool.keys(), reverse=False)
            half_keys = keys[0:5*(len(keys)//10)] # selection X%
            num_keys = len(half_keys)
            if num_keys < init_pop:
                new_pool = generate_population(depth=depth,
population=(init_pop - num_keys)//2)
                temp_pool = merge(temp_pool, new_pool)
            print(f"# of keys = {num_keys}")

            for i in range(num_keys):
                temp_pool[keys[i]] = pool[keys[i]]
            print(f"top pool = {len(temp_pool)}")

            lowest_error = min(half_keys)
            print(f"lowest error = {lowest_error}")
            if (len(errors) == 0) or (lowest_error < errors[-1]):
                evaluations.append(num_eval - counter)
                final_equation = temp_pool[lowest_error]
                y_calculated = calculate_y(x_s, final_equation)
                errors.append(lowest_error)

                plt.figure(figsize=(10,10))
                plt.gca().set_aspect('equal')
```

```
                plt.plot(x_s, y_s, 'r--',label="Given Data")
                plt.plot(x_s,y_calculated, label= "Learning Data")
                plt.legend()
                plt.xlim(-1,20)
                plt.ylim(-2,20)
                plt.title(f"MSE = {round(lowest_error, 3)}")
                plt.savefig(f'{path}screenshots/foo_{num_eval - 
counter}.png',
                        bbox_inches='tight')
                plt.close()



                with open(f"{path}tmp/ea_d_curve.txt",'a') as e:
                    e.write(str(num_eval-counter))
                    e.write(', ')
                    e.write(str(lowest_error))
                    e.write('\n')

                with open(f"{path}tmp/ea_d_yhat.txt",'w') as e:
                    for ele in y_calculated:
                        e.write(str(ele))
                        e.write('\n')

                with open(f"{path}tmp/ea_d_equation.txt",'w') as e:
                    for ele in final_equation:
                        e.write(str(ele))
                        e.write('\n')

            pool = temp_pool
            counter -= 1

   #         if counter % 10 == 0:
            print(f"counter = {num_eval - counter}")
   except ValueError:
       pass
   return [evaluations, errors, final_equation, y_calculated]



# In[709]:



def evol_algo_div_mut(num_eval, x_s, y_s, depth, init_pop, mutate_rate):
```

```
    """
    selection 50%
    mutation are built-in to crossover
    adding some random diversity
    adding mutation in the pool
    """

    evaluations = []
    errors = []
    final_equation = []
    y_calculated = []

    counter = num_eval


    pool = generate_population(depth=depth, population=init_pop)
    # print(f"pool = {len(pool)}")
    num_keys = len(pool)
    try:
        while counter > 0 and num_keys > 0:
            children_pool = {}


            keys = sorted(pool.keys(), reverse=False)
#             keys = list(pool.keys())
            for i in range(0, len(keys)-1, 2):
                kid1, kid2 = crossover(pool[keys[i]], pool[keys[i+1]])


                y_hat_kid1 = calculate_y(x_s=x_s, equation=kid1)
                y_hat_kid2 = calculate_y(x_s=x_s, equation=kid2)


                children_pool[calc_mse(y=y_s, y_hat=y_hat_kid1)] = kid1
                children_pool[calc_mse(y=y_s, y_hat=y_hat_kid2)] = kid2


            # print(f"children pool = {len(children_pool)}")


            merge_pool = merge(children_pool, pool)


            mutated_pool = mutate_pool(pool=merge_pool,
mutate_rate=mutate_rate)
#             print(f"merged pool = {len(merge_pool)}")


            pool = mutated_pool
```

```
            temp_pool = {}

            keys = sorted(pool.keys(), reverse=False)
            half_keys = keys[0:5*(len(keys)//10)] # selection X%
            num_keys = len(half_keys)

            if num_keys < init_pop:
                new_pool = generate_population(depth=depth,
population=(init_pop - num_keys)//2)
                temp_pool = merge(temp_pool, new_pool)
            print(f"# of keys = {num_keys}")

            for i in range(num_keys):
                temp_pool[keys[i]] = pool[keys[i]]
            # print(f"top pool = {len(temp_pool)}")

            lowest_error = min(half_keys)
            print(f"lowest error = {lowest_error}")
            if (len(errors) == 0) or (lowest_error < errors[-1]):

                evaluations.append(num_eval - counter)
                final_equation = temp_pool[lowest_error]
                y_calculated = calculate_y(x_s, final_equation)

                plt.figure(figsize=(10,10))
                plt.gca().set_aspect('equal')
                plt.plot(x_s, y_s, 'r--',label="Given Data")
                plt.plot(x_s,y_calculated, label= "Learning Data")
                plt.legend()
                plt.xlim(-1,20)
                plt.ylim(-2,20)
                plt.title(f"MSE = {round(lowest_error, 3)}")
                plt.savefig(f'{path}screenshots/foo_{num_eval -
counter}.png',
                        bbox_inches='tight')
                plt.close()
                # path = "/home/dxing/Desktop/ea_symbolicRegression/"

                with open(f"{path}tmp/ea_dm_curve.txt",'a') as e:
                    e.write(str(num_eval-counter))
                    e.write(', ')
                    e.write(str(lowest_error))
                    e.write('\n')
```

```
            with open(f"{path}tmp/ea_dm_yhat.txt",'w') as e:
                for ele in y_calculated:
                    e.write(str(ele))
                    e.write('\n')

            with open(f"{path}tmp/ea_dm_equation.txt",'w') as e:
                for ele in final_equation:
                    e.write(str(ele))
                    e.write('\n')



            errors.append(lowest_error)

        pool = temp_pool
        counter -= 1

#         if counter % 10 == 0:
        print(f"counter = {num_eval - counter}")
    except ValueError:
        pass
    return [evaluations, errors, final_equation, y_calculated]



# In[710]:




res_ea_div_mut = evol_algo_div_mut(num_eval=num_evaluation, x_s=x_s,
y_s=y_s, depth=depth, init_pop=20, mutate_rate=0.5)
# res_ea_div = evol_algo_div(num_eval=num_evaluation, x_s=x_s, y_s=y_s,
depth=depth, init_pop=20)
# res_ea = evol_algo(num_eval=num_evaluation, x_s=x_s, y_s=y_s,
depth=depth, init_pop=100)
# res_rrhc = random_restart_hill_climber(num_eval=num_evaluation, x_s=x_s,
y_s=y_s, depth=8, num_tries=10)
# res_rmhc = random_mutation_hill_climber(num=num_evaluation, x_s=x_s,
y_s=y_s, depth=8)
# res_rs = random_search(num=num_evaluation, x_s=x_s, y_s=y_s, depth=8)




# plt.figure(figsize=(10,10))
# plt.scatter(res_rs[0], res_rs[1], color='purple', label='random search')
# plt.plot(res_rs[0], res_rs[1], color='purple')
```

```
# plt.scatter(res_rmhc[0], res_rmhc[1], color='orange', label='random
mutation hill climber')
# plt.plot(res_rmhc[0], res_rmhc[1], color='orange')


# plt.scatter(res_rrhc[0], res_rrhc[1], color='blue', label='random restart
hill climber')
# plt.plot(res_rrhc[0], res_rrhc[1], color='blue')


# plt.scatter(res_ea[0], res_ea[1], color='green', label='EA')
# plt.plot(res_ea[0], res_ea[1], color='green')


# plt.scatter(res_ea_div[0], res_ea_div[1], color='pink', label='EA with
higher diversity')
# plt.plot(res_ea_div[0], res_ea_div[1], color='pink')


# plt.scatter(res_ea_div_mut[0], res_ea_div_mut[1], color='red',
label='EA_mut_div')
# plt.plot(res_ea_div_mut[0], res_ea_div_mut[1], color='red')


# plt.legend()


# plt.ylim(0, 100)
# plt.xscale('log')
# plt.ylabel("Fitness")
# plt.xlabel("Evaluation");
# plt.show()



# plt.figure(figsize=(10,10))
# plt.plot(x_s, y_s, label="acutal data");
# # plt.plot(x_s, res_rs[3], label="random search");
# # plt.plot(x_s, res_rrhc[3], label="random restart HC");
# # plt.plot(x_s, res_rmhc[3], label="random mutation HC");
# # plt.plot(x_s, res_ea[3], label="EA");
# plt.plot(x_s, res_ea[3], label="EA_div");
# # plt.plot(x_s, res_ea_div_mut[3], label="EA_div_mut");
# plt.legend()
# plt.show()




def save_data(result, title):
```

```
                path = "/home/dxing/Desktop/ea_symbolicRegression/tmp/"


                df_plot = pd.DataFrame(data={'evaluation': result[0], 'mse': result[1]})

                df_graph = pd.DataFrame(data={"y_cal": result[3]})


                with open(f"{path}{d}_{title}_e{num_evaluation}_final_equation.txt",'w')
as e:
                        for ele in result[2]:

                            e.write(str(ele))

                            e.write('\n')




                df_plot.to_csv(f"{path}{d}_{title}_e{num_evaluation}_plot.csv",
index=False)
                df_graph.to_csv(f"{path}{d}_{title}_e{num_evaluation}_graph.csv",
index=False)




# In[610]:




# save_data(res_rs,title="rs")

# save_data(res_rmhc,title="rmhc")

# save_data(res_rrhc,title="rrhc")

# save_data(res_ea,title="ea")

# save_data(res_ea_div,title="ea with div")

# save_data(res_ea_div_mut,title="ea_div_mut")


def mutate_incremental(binary_heap, mutate_rate=0.01):

    binaryheap = copy.deepcopy(binary_heap)


    #print(binaryheap)
    #print("that was before binary heap")
    for n in range(len(binaryheap)//2):   #len(binaryheap)//2):
        ptr = random.randint(1, len(binaryheap) - 1)     #   len(binaryheap)//2,
len(binaryheap) - 1)
        #print(ptr)
        if ptr < len(binaryheap) // 2:
            pick = random.choice(operators1)

            if (pick == binaryheap[ptr - 1]) and (binaryheap[ptr // 2] == '-'):  # To ensure
that operators aren't 0.
                tmp = copy.deepcopy(operators1)
                tmp.remove(binaryheap[ptr - 1])
                pick = random.choice(tmp)

        else:
            pick = random.choice(operators2)

            if (pick == binaryheap[ptr - 1]) and (binaryheap[ptr // 2] == '-'):  # To ensure
that operators aren't 0.
                tmp = copy.deepcopy(operators2)
```

```
                tmp.remove(binaryheap[ptr - 1])
                pick = random.choice(tmp)
        if pick == "const":  # To ensure constant numbers have trails that don't result in
"0".
            pick = round(random.random() * 10 + 0.1, 3)

        binaryheap[ptr] = pick


def mutate_incremental_lesser(binary_heap, mutate_rate=0.01):

    binaryheap = copy.deepcopy(binary_heap)


    #print(binaryheap)
    #print("that was before binary heap")
    for n in range(len(binaryheap)//4):   #len(binaryheap)//2):
        ptr = random.randint(1, len(binaryheap) - 1)     #   len(binaryheap)//2,
len(binaryheap) - 1)
        #print(ptr)
        if ptr < len(binaryheap) // 2:
            pick = random.choice(operators1)

            if (pick == binaryheap[ptr - 1]) and (binaryheap[ptr // 2] == '-'):  # To ensure
that operators aren't 0.
                tmp = copy.deepcopy(operators1)
                tmp.remove(binaryheap[ptr - 1])
                pick = random.choice(tmp)

        else:
            pick = random.choice(operators2)

            if (pick == binaryheap[ptr - 1]) and (binaryheap[ptr // 2] == '-'):  # To ensure
that operators aren't 0.
                tmp = copy.deepcopy(operators2)
                tmp.remove(binaryheap[ptr - 1])
                pick = random.choice(tmp)
        if pick == "const":  # To ensure constant numbers have trails that don't result in
"0".
            pick = round(random.random() * 10 + 0.1, 3)

        binaryheap[ptr] = pick



def genetic_algorithm_incremental(num_eval=num_evaluation, x_s=x_s, y_s=y_s, depth=8,
num_tries=100):
    evaluations = []
    errors = []
    final_equation = []
    y_calculated = []
    counter = num_eval
    firstsuccess = 0
    bestmseTOTAL = 1000
    msebest = 500
    iterationdatatotal = np.array([0, 0])
    dotdatatotal = np.array([0, 0])
    complexitydatatotal = np.array([0, 0])
    averagecomplexitytotal = np.array([0, 0])
    iters = 0
    itcount = 0

    for iterations in range(1):
        print("Starting parent child iteration #: " +str(iterations))


        for parent in range(2):
            print("This is parent: " + str(parent))
```

```
            newmse = 1000
            bestmse = 1000
            BH1 = random_generate(depth=3)
            BH1 = mutate(BH1)
            #print(len(BH1))
            lenOrig = len(BH1)
            if parent == 1:
                firstsuccess = 1

            #print(BH1)

            mse = calculate_mse(x_s, y_s, BH1)
            #print("This is MSE: " + str(mse))

            BHnah = ['nah' for x in range(256-lenOrig)]

            BH1new = BH1 + BHnah

            #print(BH1new)

            L0BH1 = ['nah']
            L1BH1 = BH1new[1:2]
            #print(L1BH1)
            L2BH1 = BH1new[2:4]
            #print(L2BH1)
            L3BH1 = BH1new[4:8]
            #print(L3BH1)
            L4BH1 = BH1new[8:16]
            L5BH1 = BH1new[16:32]
            L6BH1 = BH1new[32:64]
            L7BH1 = BH1new[64:128]
            L8BH1 = BH1new[128:256]

            # Create a new F type to be able to modify and compile later as a "new" array

            FL0BH1 = L0BH1
            FL1BH1 = L1BH1
            FL2BH1 = L2BH1
            FL3BH1 = L3BH1
            FL4BH1 = L4BH1
            FL5BH1 = L5BH1
            FL6BH1 = L6BH1
            FL7BH1 = L7BH1
            FL8BH1 = L8BH1


            #print(type(L2BH1)) #
list(L0BH1)+list(L1BH1))#+L2BH1+L3BH1+L4BH1+L5BH1+L6BH1+L7BH1+L8BH1))

            alllevelsBH1 = L0BH1 + L1BH1 + L2BH1 + L3BH1 + L4BH1 + L5BH1 + L6BH1 + L7BH1 +
L8BH1
            #alllevelsBH1 = alllevelsBH1.extend(L0BH1, L1BH1, L2BH1, L3BH1, L4BH1, L5BH1,
L6BH1, L7BH1, L8BH1)
            #print(alllevelsBH1)
            #print("This was alllevelsBH1")

            #BH1new = BH1new[:lenOrig] #This deletes all the 'nah'.
            #print(BH1new)
            newmse = calculate_mse(x_s, y_s, BH1new)

            #print(mse)
            depthcounter = 3
```

```
 #print(depthcounter)
 #print(L2BH1)
 #print("This was old L2BH1")
#FL2BH1=L2BH1

 oldval = FL3BH1
 iterationvalue = 5000



 if depthcounter ==3:
     BH1 = FL0BH1 + FL1BH1 + FL2BH1 + FL3BH1
     oldmse = calculate_mse(x_s, y_s, BH1)
     maxmse = 1000
     successcounter = 0

     for iterations in range(2000):

         BH2 = mutate_incremental(BH1)
         BH3 = random_generate(depth=3)

         iters += 1
         newmse2 = calculate_mse(x_s, y_s, BH2)
         newmse3 = calculate_mse(x_s, y_s, BH3)





         if newmse2 < maxmse:
             BH1 = BH2
             maxmse = newmse2
             bestBH = BH2
             print("Success from Mutate! New MSE = " + str(maxmse))
             successcounter = 1
         if newmse3 < maxmse:
             BH1 = BH3
             maxmse = newmse3
             bestBH = BH3
             print("Success from Random! New MSE = " + str(maxmse))
             successcounter = 1

         else:
             bestBH = BH1


         if iterations % 100 == 0:
             print(iterations)



         if firstsuccess == 0:
             msebest = maxmse

         iterationdatanew = [iters, msebest]
         iterationdatatotal = np.vstack((iterationdatatotal, iterationdatanew))

         dotdatanew = [iters, newmse2]
         dotdatatotal = np.vstack((dotdatatotal, dotdatanew))

         complexitydatanew = [maxmse, len(bestBH)]
         complexitydatatotal = np.vstack((complexitydatatotal, complexitydatanew))
```

```
                    #print(complexitydatatotal[:,1])

                    averagecomplexitynew = [iters, np.mean(complexitydatatotal[:,1])]
                    averagecomplexitytotal = np.vstack((averagecomplexitytotal,
averagecomplexitynew))
                    #print(averagecomplexitynew)

                    #print(BH1)
                    #print(BH2)

                depthcounter = 4

                print("Depth now at layer 4.")


            if depthcounter ==4 and successcounter ==1:
                successcounter = 0
                print(BH1)
                BH1 = BH1 + FL4BH1
                BH1R = random_generate(depth=4)
                BH1[4:16] = BH1R[4:16]
                print(BH1)
                oldmse = calculate_mse(x_s, y_s, BH1)

                for iterations in range(3000):
                    iters += 1
                    BH2 = mutate_incremental_lesser(BH1)
                    BH3 = mutate_incremental(BH1)

                    if iterations % 10 == 0:  #This tries a completely new random set every 10
iterations.
                        BH3R = random_generate(depth=4)
                        BH3 = BH1[0:4] + BH3R[4:16]

                        if iterations % 50 == 0:
                            BH3 =BH3R

                    newmse2 = calculate_mse(x_s, y_s, BH2)
                    newmse3 = calculate_mse(x_s, y_s, BH3)

                    if newmse2 < maxmse:
                        BH1 = BH2
                        maxmse = newmse2
                        bestBH = BH2
                        print("Success from Mutate! New MSE = " + str(maxmse))
                        successcounter = 1

                    if newmse3 < maxmse:
                        BH1 = BH3
                        maxmse = newmse3
                        bestBH = BH3
                        print("Success from Random! New MSE = " + str(maxmse))
                        successcounter = 1

                    if iterations % 100 == 0:
                        print(iterations)

                    #print(BH1)
                    #print(BH2)

                    if firstsuccess == 0:
```

```
                    bestmseTOTAL = maxmse
                iterationdatanew = [iters, bestmseTOTAL]
                iterationdatatotal = np.vstack((iterationdatatotal, iterationdatanew))

                dotdatanew = [iters, newmse2]
                dotdatatotal = np.vstack((dotdatatotal, dotdatanew))

                complexitydatanew = [maxmse, len(bestBH)]
                complexitydatatotal = np.vstack((complexitydatatotal, complexitydatanew))

                averagecomplexitynew = [iters, np.mean(complexitydatatotal[:,1])]
                averagecomplexitytotal = np.vstack((averagecomplexitytotal,
averagecomplexitynew))

            depthcounter = 5
            print("Depth now at layer 5.")

        if depthcounter == 5 and successcounter == 1:
            successcounter = 0
            print(BH1)
            BH1 = BH1 + FL5BH1
            BH1R = random_generate(depth=5)
            BH1[16:32] = BH1R[16:32]
            print(BH1)

            for iterations in range(3000):
                iters += 1
                BH2 = mutate_incremental_lesser(BH1)
                BH3 = mutate_incremental(BH1)

                if iterations % 50 == 0:  #This tries a completely new random set every 10
iterations.
                    BH3R = random_generate(depth=5)
                    BH3 = BH1[0:16] + BH3R[16:32]

                    if iterations % 100 == 0:
                        BH3 =BH3R

                newmse2 = calculate_mse(x_s, y_s, BH2)
                newmse3 = calculate_mse(x_s, y_s, BH3)

                if newmse2 < maxmse:
                    BH1 = BH2
                    maxmse = newmse2
                    bestBH = BH2
                    print("Success from Mutate! New MSE = " + str(maxmse))
                    successcounter = 1

                if newmse3 < maxmse:
                    BH1 = BH3
                    maxmse = newmse3
                    bestBH = BH3
                    print("Success from Random! New MSE = " + str(maxmse))
                    successcounter = 1

                if iterations % 100 == 0:
                    print(iterations)

                # print(BH1)
                # print(BH2)

                if firstsuccess == 0:
```

```
                        bestmseTOTAL = maxmse
                    iterationdatanew = [iters, bestmseTOTAL]
                    iterationdatatotal = np.vstack((iterationdatatotal, iterationdatanew))

                    dotdatanew = [iters, newmse2]
                    dotdatatotal = np.vstack((dotdatatotal, dotdatanew))

                    complexitydatanew = [maxmse, len(bestBH)]
                    complexitydatatotal = np.vstack((complexitydatatotal, complexitydatanew))

                    averagecomplexitynew = [iters, np.mean(complexitydatatotal[:,1])]
                    averagecomplexitytotal = np.vstack((averagecomplexitytotal,
averagecomplexitynew))

                depthcounter = 6
                print("Depth now at layer 6.")

            if depthcounter == 6 and successcounter == 1:
                successcounter = 0
                print(BH1)
                BH1 = BH1 + FL6BH1
                BH1R = random_generate(depth=6)
                BH1[32:64] = BH1R[32:64]
                print(BH1)
                ###########################
                oldmse = calculate_mse(x_s, y_s, BH1)

                for iterations in range(3000):
                    iters += 1
                    BH2 = mutate_incremental_lesser(BH1)
                    BH3 = mutate_incremental(BH1)

                    if iterations % 10 == 0:  #This tries a completely new random set every 10
iterations.
                        BH3R = random_generate(depth=6)
                        BH3 = BH1[0:32] + BH3R[32:64]

                        if iterations % 50 == 0:
                            BH3 =BH3R


                    newmse2 = calculate_mse(x_s, y_s, BH2)
                    newmse3 = calculate_mse(x_s, y_s, BH3)

                    if newmse2 < maxmse:
                        BH1 = BH2
                        maxmse = newmse2
                        bestBH = BH2
                        print("Success from Mutate! New MSE = " + str(maxmse))
                        successcounter = 1

                    if newmse3 < maxmse:
                        BH1 = BH3
                        maxmse = newmse3
                        bestBH = BH3
                        print("Success from Random! New MSE = " + str(maxmse))
                        successcounter = 1

                    if iterations % 100 == 0:
                        print(iterations)

                    # print(BH1)
```

```
                    # print(BH2)

                    if firstsuccess == 0:
                        bestmseTOTAL = maxmse
                    iterationdatanew = [iters, bestmseTOTAL]
                    iterationdatatotal = np.vstack((iterationdatatotal, iterationdatanew))

                    dotdatanew = [iters, newmse2]
                    dotdatatotal = np.vstack((dotdatatotal, dotdatanew))

                    complexitydatanew = [maxmse, len(bestBH)]
                    complexitydatatotal = np.vstack((complexitydatatotal, complexitydatanew))

                    averagecomplexitynew = [iters, np.mean(complexitydatatotal[:,1])]
                    averagecomplexitytotal = np.vstack((averagecomplexitytotal,
averagecomplexitynew))

                depthcounter = 7
                print("Depth now at layer 7.")

            if depthcounter == 7 and successcounter == 1:
                successcounter = 0
                print(BH1)
                BH1 = BH1 + FL7BH1   #Adds a new layer of "Nah"
                BH1R = random_generate(depth=7)
                BH1[64:128] = BH1R[64:128]
                print(BH1)

                for iterations in range(2000):
                    iters += 1
                    BH2 = mutate_incremental_lesser(BH1)
                    BH3 = mutate_incremental(BH1)

                    if iterations % 10 == 0:  # This tries a completely new random set every
10 iterations.
                        BH3R = random_generate(depth=7)
                        BH3 = BH1[0:64] + BH3R[64:128]

                        if iterations % 50 == 0:
                            BH3 = BH3R

                    newmse2 = calculate_mse(x_s, y_s, BH2)
                    newmse3 = calculate_mse(x_s, y_s, BH3)

                    if newmse2 < maxmse:
                        BH1 = BH2
                        maxmse = newmse2
                        bestBH = BH2
                        print("Success from Mutate! New MSE = " + str(maxmse))
                        successcounter = 1

                    if newmse3 < maxmse:
                        BH1 = BH3
                        maxmse = newmse3
                        bestBH = BH3
                        print("Success from Random! New MSE = " + str(maxmse))
                        successcounter = 1

                    if iterations % 100 == 0:
                        print(iterations)

                    # print(BH1)
```

```
                    # print(BH2)

                    if firstsuccess == 0:
                        bestmseTOTAL = maxmse
                        bestBHTOTALpre = bestBH
                    iterationdatanew = [iters, bestmseTOTAL]
                    iterationdatatotal = np.vstack((iterationdatatotal, iterationdatanew))

                    dotdatanew = [iters, newmse2]
                    dotdatatotal = np.vstack((dotdatatotal, dotdatanew))

                    complexitydatanew = [maxmse, len(bestBH)]
                    complexitydatatotal = np.vstack((complexitydatatotal, complexitydatanew))

                    averagecomplexitynew = [iters, np.mean(complexitydatatotal[:,1])]
                    averagecomplexitytotal = np.vstack((averagecomplexitytotal,
averagecomplexitynew))

                depthcounter = 8
                print("Depth now at layer 8.")

            if depthcounter == 8 and successcounter ==1:

                print(BH1)
                BH1 = BH1 + FL7BH1   #Adds a new layer of "Nah"
                BH1R = random_generate(depth=8)
                BH1[128:256] = BH1R[128:256]
                print(BH1)

                for iterations in range(1000):
                    iters += 1
                    BH2 = mutate_incremental_lesser(BH1)
                    BH3 = mutate_incremental(BH1)

                    if iterations % 10 == 0:  # This tries a completely new random set every
10 iterations.
                        BH3R = random_generate(depth=8)
                        BH3 = BH1[0:128] + BH3R[128:256]

                        if iterations % 50 == 0:
                            BH3 = BH3R

                    newmse2 = calculate_mse(x_s, y_s, BH2)
                    newmse3 = calculate_mse(x_s, y_s, BH3)

                    if newmse2 < maxmse:
                        BH1 = BH2
                        maxmse = newmse2
                        bestBH = BH2
                        print("Success from Mutate! New MSE = " + str(maxmse))

                    if newmse3 < maxmse:
                        BH1 = BH3
                        maxmse = newmse3
                        bestBH = BH3
                        print("Success from Random! New MSE = " + str(maxmse))

                    if iterations % 100 == 0:
                        print(iterations)

                    if firstsuccess == 0:
                        bestmseTOTAL = maxmse
```

```
                    iterationdatanew = [iters, bestmseTOTAL]
                    iterationdatatotal = np.vstack((iterationdatatotal, iterationdatanew))

                    dotdatanew = [iters, newmse2]
                    dotdatatotal = np.vstack((dotdatatotal, dotdatanew))

                    complexitydatanew = [maxmse, len(bestBH)]
                    complexitydatatotal = np.vstack((complexitydatatotal, complexitydatanew))

                    averagecomplexitynew = [iters, np.mean(complexitydatatotal[:,1])]
                    averagecomplexitytotal = np.vstack((averagecomplexitytotal,
averagecomplexitynew))

                    # print(BH1)
                    # print(BH2)

                depthcounter = 8

            np.savetxt("Complexity Data.csv", complexitydatatotal, delimiter=",")
            np.savetxt("Average Complexity.csv", averagecomplexitytotal, delimiter=",")


            y_calculated = calculate_y(x_s, equation=bestBH)
            print("Done!")
            print(maxmse)
            print(bestBH)

            if firstsuccess == 0:
                bestBHTOTAL = bestBH
                bestBHTOTALpre = bestBHTOTAL

            if maxmse < bestmse:
                bestmse = maxmse

            if parent == 0:
                FatherBH = bestBH
                Fathermse = maxmse
                print("Solved for Father: " +str(Fathermse) + str(FatherBH))

            if parent == 1:
                MotherBH = bestBH
                Mothermse = maxmse
                print("Solved for Mother: " + str(Mothermse) + str(MotherBH))




        BH1 = FatherBH
        BH2 = MotherBH

        BHnahfa = ['nah' for x in range(256 - len(BH1))]
        BH1 = BH1 + BHnahfa

        BHnahma = ['nah' for x in range(256 - len(BH2))]
        BH2 = BH2 + BHnahma

        # The following splits every line into its own array of strings.

        # For BH1:

        L0BH1 = [BH1[0]]
        L1BH1 = [BH1[1]]
```

```
        L2BH1 = BH1[2:4]
        L3BH1 = BH1[4:8]
        L4BH1 = BH1[8:16]
        L5BH1 = BH1[16:32]
        L6BH1 = BH1[32:64]
        L7BH1 = BH1[64:128]
        L8BH1 = BH1[128:256]

        # For BH2:

        L0BH2 = [BH2[0]]
        L1BH2 = [BH2[1]]
        L2BH2 = BH2[2:4]
        L3BH2 = BH2[4:8]
        L4BH2 = BH2[8:16]
        L5BH2 = BH2[16:32]
        L6BH2 = BH2[32:64]
        L7BH2 = BH2[64:128]
        L8BH2 = BH2[128:256]

        # Now to split each of these into each set of child operators/numbers:

        # Selecting a point, let's say BH[2], so the first 50% of each element will be under
it.

        # This copies everything on the left side of the first binary heap and pastes it on
top of everything on the left side of the second binary heap.

        L0C1 = L0BH1
        L1C1 = L1BH1
        L2C1 = L2BH1[:(len(L2BH1) // 2)] + L2BH2[(len(L2BH2) // 2):]
        L3C1 = L3BH1[:(len(L2BH1) // 2)] + L3BH2[(len(L2BH2) // 2):]
        L4C1 = L4BH1[:(len(L2BH1) // 2)] + L4BH2[(len(L2BH2) // 2):]
        L5C1 = L5BH1[:(len(L2BH1) // 2)] + L5BH2[(len(L2BH2) // 2):]
        L6C1 = L6BH1[:(len(L2BH1) // 2)] + L6BH2[(len(L2BH2) // 2):]
        L7C1 = L7BH1[:(len(L2BH1) // 2)] + L7BH2[(len(L2BH2) // 2):]
        L8C1 = L8BH1[:(len(L2BH1) // 2)] + L8BH2[(len(L2BH2) // 2):]

        # This copies everything on the left side of the first binary heap and pastes it on
top of everything on the left side of the second binary heap.

        L0C2 = L0BH2
        L1C2 = L1BH2
        L2C2 = L2BH2[:(len(L2BH2) // 2)] + L2BH1[(len(L2BH1) // 2):]
        L3C2 = L3BH2[:(len(L2BH2) // 2)] + L3BH1[(len(L2BH1) // 2):]
        L4C2 = L4BH2[:(len(L2BH2) // 2)] + L4BH1[(len(L2BH1) // 2):]
        L5C2 = L5BH2[:(len(L2BH2) // 2)] + L5BH1[(len(L2BH1) // 2):]
        L6C2 = L6BH2[:(len(L2BH2) // 2)] + L6BH1[(len(L2BH1) // 2):]
        L7C2 = L7BH2[:(len(L2BH2) // 2)] + L7BH1[(len(L2BH1) // 2):]
        L8C2 = L8BH2[:(len(L2BH2) // 2)] + L8BH1[(len(L2BH1) // 2):]

        FL0C2 =  L0C2
        FL1C2 =  L1C2
        FL2C2 =  L2C2
        FL3C2 =  L3C2
        FL4C2 =  L4C2
        FL5C2 =  L5C2
        FL6C2 =  L6C2
        FL7C2 =  L7C2
        FL8C2 =  L8C2
```

```
        # BH1N = [str(L0BH2), str(L1BH2), str(L2BH2), str(L3BH2), str(L4BH2), str(L5BH2), 
str(L6BH2), str(L7BH2), str(L8BH2)]
        #Child1BH = [str(L0C1) + str(L1C1) + str(L2C1) + str(L3C1) + str(L4C1) + str(L5C1) + 
str(L6C1) + str(L7C1) + str(L8C1)]
        Child1BH = L0C1 + L1C1 + L2C1 + L3C1 + L4C1 + L5C1 + L6C1 + L7C1 + L8C1
        #Child1BH = list(Child1BH)

        #Child2BH = [str(L0C2) + str(L1C2) + str(L2C2) + str(L3C2) + str(L4C2) + str(L5C2) + 
str(L6C2) + str(L7C2) + str(L8C2)]
        Child2BH = L0C2 + L1C2 + L2C2 + L3C2 + L4C2 + L5C2 + L6C2 + L7C2 + L8C2




        Fathermse = calculate_mse(x_s, y_s, FatherBH)
        #print(Fathermse)
        Mothermse = calculate_mse(x_s, y_s, MotherBH)
        #print(Mothermse)




        #print("DONE DONE DONE DONE DONE DONE")

        Child1BH, Child2BH = crossover(FatherBH, MotherBH)
        Child1mse = calculate_mse(x_s, y_s, Child1BH)
        #print(Child1BH)
        Child2mse = calculate_mse(x_s, y_s, Child2BH)
        #print(Child2BH)

        if firstsuccess == 0:
            bestBHTOTALpre = bestBHTOTAL



        if Fathermse < Mothermse and Fathermse < bestmseTOTAL:
            bestBHTOTALpre = bestBHTOTAL
            bestBHTOTAL = FatherBH
            bestmseTOTAL = Fathermse
            print("Father wins!")
        if Mothermse < Fathermse and Mothermse < bestmseTOTAL:
            bestBHTOTALpre = bestBHTOTAL
            bestBHTOTAL = MotherBH
            bestmseTOTAL = Mothermse
            print("Mother wins!")
        if Child1mse < Fathermse and Child1mse < Mothermse and Child1mse < Child2mse and 
Child1mse < bestmseTOTAL:
            bestBHTOTALpre = bestBHTOTAL
            bestBHTOTAL = Child1BH
            bestmseTOTAL = Child1mse
            print("Child 1 wins!")
        if Child2mse < Fathermse and Child2mse < Mothermse and Child2mse < Child1mse and 
Child2mse < bestmseTOTAL:
            bestBHTOTALpre = bestBHTOTAL
            bestBHTOTAL = Child1BH
            bestmseTOTAL = Child2mse
            print("Child 2 wins!")


        champion1, champion2 = crossover(bestBHTOTALpre, bestBHTOTAL)
        champion1mse = calculate_mse(x_s, y_s, champion1)
```

```
        champion2mse = calculate_mse(x_s, y_s, champion2)

        if champion1mse < bestmseTOTAL:
            bestBHTOTAL = champion1
            bestmseTOTAL = champion1mse
            print("CHAMPION 1 CROSSOVER WINS")

        if champion2mse < bestmseTOTAL:
            bestBHTOTAL = champion2
            bestmseTOTAL = champion2mse
            print("CHAMPION 2 CROSSOVER WINS")

        print(Fathermse, Mothermse, Child1mse, Child2mse, champion1mse, champion2mse,
bestmseTOTAL)

        firstsuccess = 1



    np.savetxt("trial4.csv", iterationdatatotal, delimiter=",")
    np.savetxt("trial5.csv", dotdatatotal, delimiter=",")

    y_calculated = calculate_y(x_s, equation=bestBHTOTAL)
    print("Final Data: MSE = " + str(bestmseTOTAL) + "; Best BH: " + str(bestBHTOTAL))


    plt.xlabel("X value")
    plt.ylabel("Y value")
    plt.title("Equation 3X+5 Ran on EA")

    plt.figure(figsize=(10, 10))
    plt.plot(x_s, y_s, label="actual data");
    # plt.plot(x_s, res_rs[3], label="random search");
    plt.plot(x_s, y_calculated, label="Incremental Evolutionary Algorithm");
    plt.xlabel("X value")
    plt.ylabel("Y value")
    plt.title("Equation 3X+5 Ran on EA")
    # plt.plot(x_s, res_rmhc[3], label="random mutation HC");
    plt.legend()
    plt.show()
```

Automatically draw tree representing best solution

```
from graphics import *
import math
import random

# filename = "ea_dm_equation.txt"
filename = "test.txt"
levels = [512, 256, 128, 64, 32, 16, 8, 4, 2, 1]


def read_equation(filename):
    equations = []
    with open(f'{filename}', 'r') as f:

        for line in f.read().splitlines():
            equations.append(line)

    return equations
```

```python
def build_tree(binary_heap, win):

    ptr = 1
    radius = 20

    locations = [[0, 0], [2500, 50]]

    print(locations[ptr][0])
    print(locations[ptr][1])

    while ptr < len(binary_heap):
        x = locations[ptr][0]
        y = locations[ptr][1]
        # print(f"x = {x}, y = {y}")
        pt = Point(x, y)
        cir = Circle(pt, radius)
        txt = Text(pt, binary_heap[ptr])

        level = math.floor(math.log2(ptr))

        print(binary_heap[ptr], level)
        cir.draw(win)
        txt.draw(win)

        child_lt = [x - (1 * levels[level]), y + 100]
        child_rt = [x + (1 * levels[level]), y + 100]

        if ptr < len(binary_heap) // 2:
            line_lt = Line(Point(x,y+radius), Point(child_lt[0], child_lt[1]-radius))
            line_rt = Line(Point(x,y+radius), Point(child_rt[0], child_rt[1]-radius))

            line_lt.draw(win)
            line_rt.draw(win)

        locations.append(child_lt)
        locations.append(child_rt)

        ptr += 1
    # print(len(locations))


def main():
    win = GraphWin("My window", 5000, 2000)

    win.setBackground(color_rgb(255, 255, 255))
    bh = read_equation(filename=filename)
    build_tree(bh, win=win)

    win.getMouse()
    win.close()

main()
```