UNIVERSITY OF BIRMINGHAM

School of Computer Science

Nature-Inspired Search and Optimisation

PG Aff Computer Science
Author：Wenyi Zhou
Student ID：1887939

Table of Contents

Chapter 1

Exercise 4 Pseudo-code of My Algorithm

Algorithm Time Series Prediction

Require: the dimension of the input vector $n \in N$
Require: the size of the training data *(X,Y)* $m \in N$
Require: Population size $\lambda \in N$
Require: the name of a file containing training data *data* ∈ *file*
Require: the number of seconds to run the algorithm *time_budget* ∈ *N*

```
/*The necessary functions and class of the algorithm*/
define a function class
    initialize attributes of the function class
        the function of the function
        the child number of the function
        the name of the function
define a constant class
    initialize attributes of the constant class
        the value of the constant
        the name of the constant
        the type of the constant is constant
    define evaluation method to evaluate the constant
        Return the value of the constant
define add function (input value_list )
    sum=0
    for i=0 to (the length of value_list)-1 do
        sum=sum+value_list[i]
    end for
    Return sum
define subtract function (input value_list )
    Return value_list[0]- value_list[1]
define multiply function (input value_list )
    Return value_list[0]* value_list[1]
define divide function (input value_list )
    if (value_list[1]=0) do
        Return 0
    else do
        Return value_list[0] / value_list[1]
    end if
define power function (input value_list )
    if (value_list[0]=0 & value_list[1]<=0) do
        Return 0
    else if (value_list[0]^Value_list[1 is not a real number] do
        Return 0
    else do
        Return value_list[0]^Value_list[1]
    end if
define sqrt function (input value_list )
    if (value_list[0]<0) do
        Return 0
    else do
```

```
            Return √value_list[0]
    end if
define log function (input value_list )
    if (value_list[0]≤0) do
            Return 0
    else do
            Return log_2 value_list[0]
    end if
define exp function (input value_list )
    Return e^value_list[0]
define ifleq function (input value_list )
    if (value_list[0] ≤ value_list[1]) do
            Return value_list[2]
    else do
            Return value_list[3]
    end if
define data function (input value_list , X )
    j ≡ |⌊value_list[0]⌋| mod n
    Return X_j
define diff function (input value_list , X )
    k= |⌊value_list[0]⌋| mod n
    l= |⌊value_list[1]⌋| mod n
    Return X_j
define avg function (input value_list , X )
    if (value_list[0] = value_list[1]) do
            Return 0
    else do
            k= |⌊value_list[0]⌋| mod n
            l= |⌊value_list[1]⌋| mod n
            t=min(k,l)
            upper=max(k,l)-1
            sum=0
            for i=t to upper do
                    sum= sum+X_i
            end for
            Return sum/abs(k-l)
    end if
define max function (input value_list )
    Return max(value_list)
Create add function object(the function is add, the child number is 2, the function name is "add")
Create sub function object(the function is sub, the child number is 2, the function name is "sub")
Create mul function object(the function is mul, the child number is 2, the function name is "mul")
Create div function object(the function is div, the child number is 2, the function name is "div")
Create pow function object(the function is pow, the child number is 2, the function name is " pow")
Create sqrt function object(the function is sqrt, the child number is 1, the function name is " sqrt ")
Create log function object(the function is log, the child number is 1, the function name is " log ")
Create exp function object(the function is exp, the child number is 1, the function name is " exp ")
Create max function object(the function is max, the child number is 2, the function name is " max ")
Create ifleq function object(the function is ifleq, the child number is 4, the function name is " ifleq ")
Create data function object(the function is data, the child number is 1, the function name is " data ")
Create diff function object(the function is diff, the child number is 2, the function name is " diff ")
Create avg function object(the function is avg, the child number is 2, the function name is " avg ")
function_list=[ add, sub, mul, div, pow, sqrt, log, exp, max, ifleq, data, diff, avg]
constant_list=list(range(-100, 100)) /* randomly generate constant from -100 to 100 */
define a node class
    initialize attributes of the node class
            the type of the node
            the children of the node
            the name of the node
            the depth of the node
            the value of the node
            the fitness of the node
    define evaluation method to calculate the value of the node
```

```
        if (the type of the node is constant
            Return the value of the constant node
        else do
            for i=0 to (the length of children_list)-1 do
                value_list =[ children_list[i].evaluation_method for i=0 to (the length of children_list)-1 ] /*recursion*/
                Return node.function(value_list,X)
            end for
        end if
    define display method to parse the tree to an expression
        if ( the type of the node is function) do
            expression=expression+ ' ('+node.function.name
        else if ( the type of the node is constant) do
            expression= ' '+node.constant.name
        end if
        if (the node has children) do
            for i=0 to (the length of children_list)-1 do
                expression=expression+ children_list[i].display_method /*recursion*/
            end for
            expression= expression +')'
        end if
        Return expression
define get_fitness method to calculate the fitness of a tree(input data)
    sum=0
    for i=0 to (the length of data)-1 do
        sum=sum+(tree.evaluate_method(data ['X'])- data ['Y'])**2
    end for
    fitness=sum/m
    Return fitness
define an environment class
    initialize attributes of the environment class
        the function_list of the environment class
        the constant_list of the environment class
        the training data of the environment class
        the population size of the environment class
        the population of the environment=population or create_population_method(input λ)
        the max depth of the tree
    define make tree method(input start_depth)
        if start_depth=0 do
            node_pattern=0
        else if start_depth=max_depth-1 do
            node_pattern=1
        else do
            node_pattern=random(0,1)
        end if
        if node_pattern=0 do
            randomly select a function from function_list
            for i=0 to (the number of function's children)-1 do
                child=make_tree_method(start_depth+1) /*recursion*/
                add child to children_list
            end for
            Return node("function", children_list, function)
        else do
            randomly select a constant from constant_list
            Return node("constant", constant_class(constant))
        end if
    define create population method (input λ)
        Return [make_tree_method(input start_depth=0) for i=0 to λ -1 do ]
    define mutation method(input tree, mutation_rate=0.1,startdepth=0)
        if (randomly generate a probability)<=mutation_rate do
            return tree.make_tree_method(input start_depth=0) /*mutate the root of tree*/
        else do
            new_tree=deepcopy(tree)
            if the node type of the tree is function do
```

```
                                new_tree(node).children=[ mutation_method(input tree.children_list[c], mutation_rate=0.1, start_depth+1) for c=0 to (the length of tree.children_list)-1     ]    /*recursion*/
                        end if
                end if
                Return new_tree
        define crossover method (input tree1, tree2,crossover_rate=0.9 , top=1)
                if (randomly generate a probability)<=mutation_rate and not top do
                        Return deepcopy(tree2)
                else do
                        new_tree=deepcopy(tree1)
                        if the type of tree1's node is function and the type of the tree2's node is function do
                                new_tree(node).children=[ crossover_method(input tree1.children_list[c],randomly select a node from tree2,crossover_rate=0.9, 0) for c=0 to (the length of tree1.children_list)-1     ]    /*recursion*/
                        end if
                end if
                Return new_tree
        define roulette selection method
                all_fitness=0
                for i=0 to  λ  -1 do
                        all_fitness=population[i].getfitness(input checkdata)
                end for
                random_probability=random(0,1)*(  λ-1 )
                add_probability=0
                for i=0 to  λ  -1 do
                        add_probability=add_probability+(1.0- population[i].getfitness(input checkdata/ all_fitness)
                        if add_probability≥random_probability do
                                Return population[i]
                        end if
                end for
        define evolve method(input time_budget,min_or_max)
                get start_time
                while True:
                        create a offspring_list to store offspring
                        get end_time
                        if time_budget<( end_time- start_time) do
                                end loop while
                        end if
                        for i=0 to (λ  -1) do
                                father= roulette selection method(input reverse=False)
                                mother= roulette selection method(input reverse=False)
                                offspring1=crossover(father,mother)
                                add offspring1 in offspring_list
                                parent= roulette selection method(input reverse=False)
                                offspring2= mutation method(parent)
                                add offspring2 in offspring_list
                        end for
                        population= offspring_list    /*use offspring_list to replace prevous population*/
                        best_tree=population[0]
                        for j=0 to (λ  -1) do
                                population[j].getfitness(input checkdata) /update the fitness of individual
                                if best_tree.fitness> population[j].fitness
                                        best_tree= population[j]
                                end if
                        end for
                end while
                Return best_tree.display
define read data file method(input filename,m,n)
    f=open(' filename ', ' read)
    create a data_list to store taining data
    for i=1 to length(f) do
                create a dictionary to store each sample of training data
                x=f[i][:n]
                y= f[i][n]
```

```
        dictionary['x'] =x
        dictionary['y'] =y
        add dictionary to data_list
    end for
    Return data_list

/* main function to start the program */
training_data= read data file method(input filename,m,n)
environment_object=environment (input  λ, function_list, constant_list, data_list )
best_expression= environment_object.envolve(input time_budget)
output(best_expression)
```

Chapter 2

Exercise 5-Fittest Solution VS Mutation Rate

2.1 Experiment Parameter

In order to investigate the relationship between the fitness of the fittest solution found within the time budget and mutation rate, we should control variables. The experiment parameters were used as shown below.

2.1.1 Constant

2.1.1.1 Max Depth of Tree

Growth method is used to create initial population with fixed maximum tree depth. Do add random function or terminal nodes until all branches have terminals or are (maxDepth-1) depth. Then add random terminal nodes to all branches without terminals. Each individual (expression) is represented by a tree. The max depth of tree is represented by maxDepth in the source code. In this experiment, ten is chose as the max depth of tree **(i.e., Max Depth of Tree (*maxDepth*)=10)**.

2.1.1.2 Population Size

Population size (i.e., it is represented by λ) is the number of individuals in a population. The value of population size is defined as 100 **(i.e., Population Size (λ)=100)**.

2.1.1.3 Time Budget

Time Budget is defined as the number of seconds to run the algorithm, In this experiment, 20 seconds are chose as the value of time budget **(i.e., Time Budget(*time_budget*)=20)**.

2.1.1.4 Crossover Rate

Crossover rate is essential for the behavior of the algorithm. In this experiment, 0.9 is chose as the value of *crossover rate* **(i.e., *Crossover Rate*=0.9)**.

2.1.1.5 Training Data File

Training Data File (i.e., Created randomly by a python script) contains the training data in the form of m lines, where each line contains n+1 values separated by tab characters. The first n elements in a line represents an input vector x, and the last element in a line represents the output value y. In this experiment, **'data.txt' is the instance to test code on (The figure showed below).**

data.txt

85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
57.64	36.87	89.02	51.57	87.95	75.79	95.99	10.37	87.18	52.43	80.94
81.02	30.11	84.77	28.98	50.04	79.24	83.85	49.90	22.78	18.45	91.94
17.54	57.29	83.04	28.05	5.64	99.18	15.34	82.69	63.60	58.79	22.96
10.04	22.12	6.41	90.44	44.77	85.23	53.96	29.43	39.59	16.05	75.29
6.06	28.41	23.15	64.00	82.22	3.99	84.70	23.35	8.87	53.54	3.72
48.92	96.78	3.69	50.95	11.75	26.03	37.92	74.25	49.05	96.16	55.13
50.91	14.77	90.91	55.71	73.26	46.55	30.85	77.56	39.49	26.55	77.81
25.31	12.84	54.40	58.50	10.64	21.25	57.70	35.14	40.19	43.12	9.04
72.53	42.92	19.48	29.75	41.93	18.12	40.08	35.23	20.13	92.35	93.04

figure 1: Training Data File

2.1.2 Variable

Mutation rate is essential for the behavior of the algorithm. The range of mutation rate is defined from 0.001 to 0.5 **(i.e., Mutation Rate ∈ (0.001, 0.8))**.

2.2 Experiment Results and Analysis

Each experiment has been repeated for 100 times. The results are shown as boxplots. In order to display the overall trends and each individual case, the increment of mutation rate was divided into three parts in one figure. The increment of first part is 10 times than previous one , the increment of second part is 0.05 and the increment of the third part is 0.1 **(i.e., Mutation Rate ∈ Set {** 0.001, 0.01, 0.05, 0.1, 0.15, 0.2, 0.3, 0.4, 0.5, 0.8 **}).**

2.2.1 Boxplot of Results

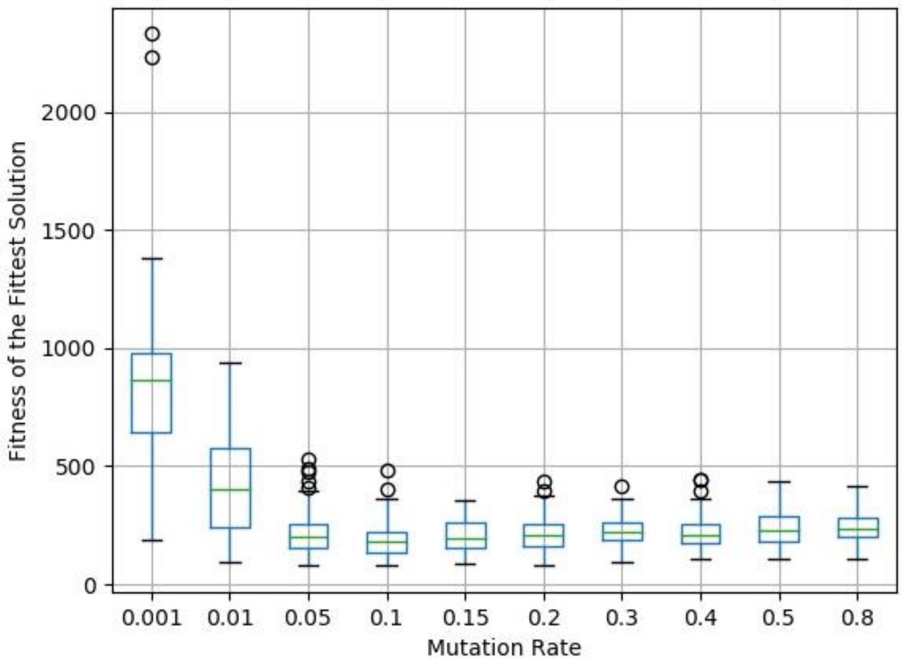


figure 2: Fittest Solution VS Mutation Rate

2.2.2 Analysis of Boxplot

The boxplots summarize the impact of mutation rate on the quality (i.e., fitness of fittest solution) of the solutions obtained.

Overall, when mutation rate is below 0.1, the upper edge (i.e., the maximum value), the upper quartile (i.e., the third quartile), the median, the lower quartile (i.e., the first quartile) and the lower edge (i.e., the minimum value) all have an approximate linear decreasing trend with the increment of mutation rate. However, when the mutation rate is above 0.1, they have a slight upward trend. It means when the mutation rate is 0.1, the fitness of the fittest solution is lowest.

It is clear that the outliers almost appear above the upper edge (i.e., the maximum value) in the boxplots. Because the quality of the initial population will affect the probability of outliers occurring. Therefore, the quality of initial population may be worse.

In conclusion, the fitness of the solution represents the mean-square error of the prediction. The fittest solution has the lowest fitness. Therefore, setting mutation rate to 0.1 is the best choice. It can not only get the fittest solution, but also can keep the genes diverse.

Chapter 3

Experiment 2- Fittest Solution VS Max Depth of Tree

3.1 Experiment Parameter

In order to investigate the relationship between the fitness of the fittest solution found within the time budget and the max depth of tree (i.e., maxDepth), we should control variables. The experiment parameters were used as shown below.

3.1.1 Constant

3.1.1.1 Mutation Rate

Mutation rate is essential for the behavior of the algorithm. In this experiment, 0.1 is chose as the value of *mutation rate* **(i.e., *Mutation Rate*=0.1)**.

3.1.1.2 Population Size

Population size (i.e., it is represented by λ) is the number of individuals in a population. The value of population size is defined as one hundred **(i.e., Population Size (λ)=100)**.

3.1.1.3 Time Budget

Time Budget is defined as the number of seconds to run the algorithm, In this experiment, 20 seconds are chose as the value of time budget **(i.e., Time Budget(*time_budget*)=20)**.

3.1.1.4 Crossover Rate

Crossover rate is essential for the behavior of the algorithm. In this experiment, 0.9 is chose as the value of *crossover rate* **(i.e., *Crossover Rate*=0.9)**.

3.1.1.5 Training Data File

Training Data File (i.e., Created randomly by a python script) contains the training data in the form of m lines, where each line contains n+1 values separated by tab characters. The first n elements in a line represents an input vector x, and the last element in a line represents the output value y. In this experiment, **'data.txt' is the instance to test code on (The figure showed below).**

data.txt

85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
57.64	36.87	89.02	51.57	87.95	75.79	95.99	10.37	87.18	52.43	80.94
81.02	30.11	84.77	28.98	50.04	79.24	83.85	49.90	22.78	18.45	91.94
17.54	57.29	83.04	28.05	5.64	99.18	15.34	82.69	63.60	58.79	22.96
10.04	22.12	6.41	90.44	44.77	85.23	53.96	29.43	39.59	16.05	75.29
6.06	28.41	23.15	64.00	82.22	3.99	84.70	23.35	8.87	53.54	3.72
48.92	96.78	3.69	50.95	11.75	26.03	37.92	74.25	49.05	96.16	55.13
50.91	14.77	90.91	55.71	73.26	46.55	30.85	77.56	39.49	26.55	77.81
25.31	12.84	54.40	58.50	10.64	21.25	57.70	35.14	40.19	43.12	9.04
72.53	42.92	19.48	29.75	41.93	18.12	40.08	35.23	20.13	92.35	93.04

figure 3: Training Data File

3.1.2 Variable

Growth method is used to create initial population with fixed maximum tree depth. Do add random function or terminal nodes until all branches have terminals or are (maxDepth-1) depth. Then add random terminal nodes to all branches without terminals. Each individual (expression) is represented by a tree. The max depth of tree is represented by maxDepth in the source code. In this experiment, the max depth of tree is ranging from ten to ninety **(i.e., Max Depth of Tree (*maxDepth*) $\in$ (2 , 90))**.

3.2 Experiment Results and Analysis

Each experiment has been repeated for 100 times. The results are shown as boxplots. In order to display the overall trend and each individual case, the increment of max depth was divided into two parts in two figures. In one figure the increments of 2 for the max depth was tried in this experiment and in another figure, the increment of max depth is 10 **(i.e., Max Depth of Tree (*maxDepth*) $\in$ Set {2, 4, 6, 8,10, 20, 30, 40,50, 60, 70, 80, 90})**.

3.2.1 Boxplot of Results

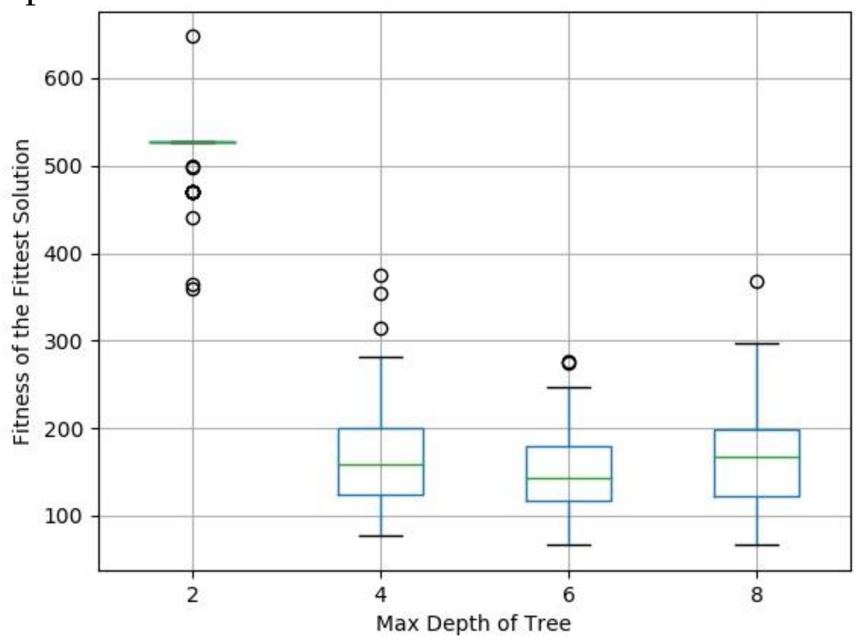


figure 4: Fittest Solution VS Max Depth of Tree

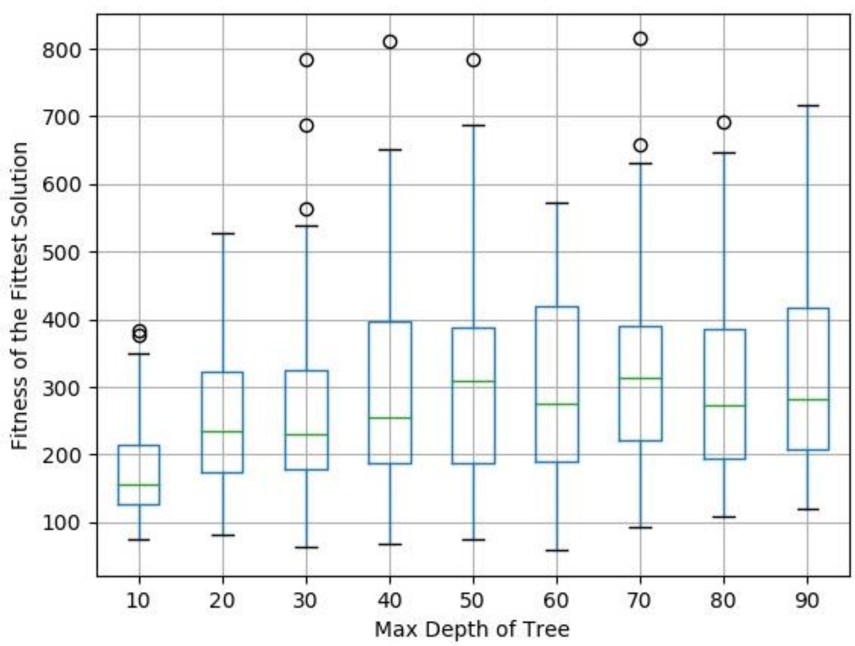


figure 5: Fittest Solution VS Max Depth of Tree

3.2.2 Analysis of Boxplot

The boxplots summarize the impact of max depth on the quality (i.e., fitness of fittest solution) of the solutions obtained.

Overall, after the upper edge (i.e., the maximum value), the upper quartile (i.e., the third quartile), the median, the lower quartile (i.e., the first quartile) and the lower edge (i.e., the minimum value) all have a downward trend with the max depth varying from 2 to 6, they approximately have a slight upward trend with the increment of max depth. It seems like when the max depth is 6, the fitness of the fittest solution is lowest and the height of the boxplot is smallest. It means the distribution of normal values is concentrated.

It is clear that the outliers almost appear above the upper edge (i.e., the maximum value) in the boxplots. Because the quality of the initial population will affect the probability of outliers occurring. Therefore, the quality of initial population may be worse.

In conclusion, the fitness of the solution represents the mean-square error of the prediction. The fittest solution has the lowest fitness. Therefore, setting max depth to 6 is the best choice in this experiment.

Chapter 4

Exercise 5- Fittest Solution VS Population Size

4.1 Experiment Parameter

In order to investigate the relationship between the fitness of the fittest solution found within the time budget and population size, we should control variables. The experiment parameters were used as shown below.

4.1.1 Constant

4.1.1.1 Mutation Rate

Mutation rate is essential for the behavior of the algorithm. In this experiment, 0.1 is chose as the value of *mutation rate* **(i.e.,** ***Mutation Rate*****=0.1)**.

4.1.1.2 Max Depth of Tree

Growth method is used to create initial population with fixed maximum tree depth. Do add random function or terminal nodes until all branches have terminals or are (maxDepth-1) depth. Then add random terminal nodes to all branches without terminals. Each individual (expression) is represented by a tree. The max depth of tree is represented by maxDepth in the source code. In this experiment, ten is chose as the max depth of tree **(i.e., Max Depth of Tree (*****maxDepth*****)=10)**.

4.1.1.3 Time Budget

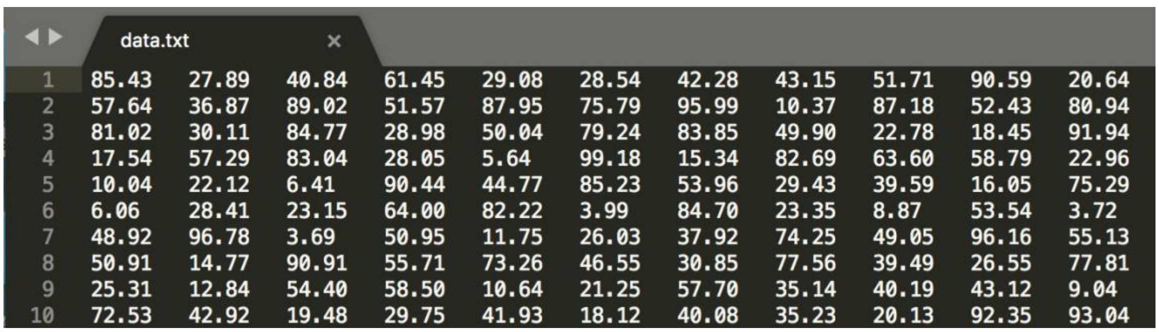
Time Budget is defined as the number of seconds to run the algorithm, In this experiment, 20 seconds are chose as the value of time budget **(i.e., Time Budget(*****time_budget*****)=20)**.

4.1.1.4 Crossover Rate

Crossover rate is essential for the behavior of the algorithm. In this experiment, 0.9 is chose as the value of *crossover rate* **(i.e., *Crossover Rate*=0.9)**.

4.1.1.5 Training Data File

Training Data File (i.e., Created randomly by a python script) contains the training data in the form of m lines, where each line contains n+1 values separated by tab characters. The first n elements in a line represents an input vector x, and the last element in a line represents the output value y. In this experiment, **'data.txt' is the instance to test code on (The figure showed below).**


data.txt

85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
57.64	36.87	89.02	51.57	87.95	75.79	95.99	10.37	87.18	52.43	80.94
81.02	30.11	84.77	28.98	50.04	79.24	83.85	49.90	22.78	18.45	91.94
17.54	57.29	83.04	28.05	5.64	99.18	15.34	82.69	63.60	58.79	22.96
10.04	22.12	6.41	90.44	44.77	85.23	53.96	29.43	39.59	16.05	75.29
6.06	28.41	23.15	64.00	82.22	3.99	84.70	23.35	8.87	53.54	3.72
48.92	96.78	3.69	50.95	11.75	26.03	37.92	74.25	49.05	96.16	55.13
50.91	14.77	90.91	55.71	73.26	46.55	30.85	77.56	39.49	26.55	77.81
25.31	12.84	54.40	58.50	10.64	21.25	57.70	35.14	40.19	43.12	9.04
72.53	42.92	19.48	29.75	41.93	18.12	40.08	35.23	20.13	92.35	93.04

figure 6: Training Data File

4.1.2 Variable

Population size (i.e., it is represented by λ) is the number of individuals in a population. The range of population size is defined from ten to four hundred **(i.e., Population Size (λ) $\in$ [10 , 400])**.

4.2 Experiment Results and Analysis

Each experiment has been repeated for 100 times. The results are shown as boxplots. In order to display the overall trend and each individual case, the increments of 30 for population size was tried in this experiment **(i.e., Population Size (λ) $\in$ Set {10, 40, 70, 100, 130, 160, 190, 220, 250, 280, 310, 340, 370, 400, })**.

4.2.1 Boxplot of Results

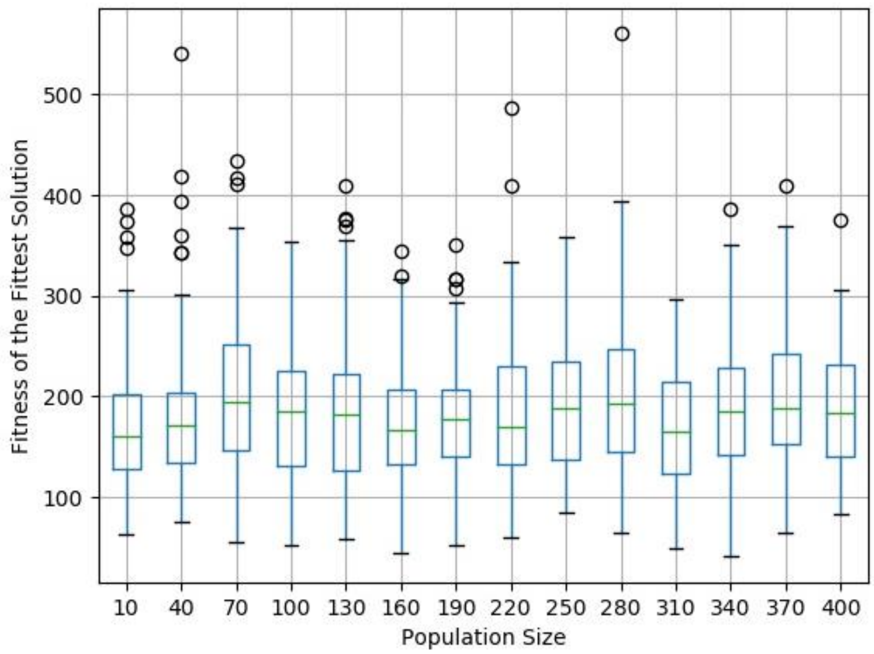


figure 7: Fittest Solution VS Population Size

4.2.2 Analysis of Boxplot

The boxplots summarize the impact of population size on the quality (i.e., fitness of fittest solution) of the solutions obtained.

Overall, the upper edge (i.e., the maximum value), the upper quartile (i.e., the third quartile), the median, the lower quartile (i.e., the first quartile) and the lower edge (i.e., the minimum value) all fluctuates slightly with the increment of population size. It seems the population size has a slight influence on the efficiency of the algorithm. However, when the population size is 10, the fitness of the fittest solution is lowest.

It is clear that the outliers almost appear above the upper edge (i.e., the maximum value) in the boxplots. Because the quality of the initial population will affect the probability of outliers occurring. Therefore, the quality of initial population may be worse.

In conclusion, Although the population size doesn't affect the efficiency of the algorithm too much. Setting population size to 10 is the best choice in this experiment.

Chapter 5

Exercise 5- Fittest Solution VS Crossover Rate

5.1 Experiment Parameter

In order to investigate the relationship between the fitness of the fittest solution found within the time budget and crossover rate, we should control variables. The experiment parameters were used as shown below.

5.1.1 Constant

5.1.1.1 Mutation Rate

Mutation rate is essential for the behavior of the algorithm. In this experiment, 0.1 is chose as the value of *mutation rate* **(i.e., *Mutation Rate*=0.1)**.

5.1.1.2 Max Depth of Tree

Growth method is used to create initial population with fixed maximum tree depth. Do add random function or terminal nodes until all branches have terminals or are (maxDepth-1) depth. Then add random terminal nodes to all branches without terminals. Each individual (expression) is represented by a tree. The max depth of tree is represented by maxDepth in the source code. In this experiment, ten is chose as the max depth of tree **(i.e., Max Depth of Tree (*maxDepth*)=10)**.

5.1.1.3 Population Size

Population size (i.e., it is represented by λ) is the number of individuals in a population. The value of population size is defined as one hundred **(i.e., Population Size (λ)=100)**.

5.1.1.4 Time Budget

Time Budget is defined as the number of seconds to run the algorithm, In this experiment, 20 seconds are chose as the value of time budget **(i.e., Time Budget(*time_budget*)=20)**.

5.1.1.5 Training Data File

Training Data File (i.e., Created randomly by a python script) contains the training data in the form of m lines, where each line contains n+1 values separated by tab characters. The first n elements in a line represents an input vector x, and the last element in a line represents the output value y. In this experiment, **'data.txt' is the instance to test code on (The figure showed below).**

data.txt

85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
57.64	36.87	89.02	51.57	87.95	75.79	95.99	10.37	87.18	52.43	80.94
81.02	30.11	84.77	28.98	50.04	79.24	83.85	49.90	22.78	18.45	91.94
17.54	57.29	83.04	28.05	5.64	99.18	15.34	82.69	63.60	58.79	22.96
10.04	22.12	6.41	90.44	44.77	85.23	53.96	29.43	39.59	16.05	75.29
6.06	28.41	23.15	64.00	82.22	3.99	84.70	23.35	8.87	53.54	3.72
48.92	96.78	3.69	50.95	11.75	26.03	37.92	74.25	49.05	96.16	55.13
50.91	14.77	90.91	55.71	73.26	46.55	30.85	77.56	39.49	26.55	77.81
25.31	12.84	54.40	58.50	10.64	21.25	57.70	35.14	40.19	43.12	9.04
72.53	42.92	19.48	29.75	41.93	18.12	40.08	35.23	20.13	92.35	93.04

figure 8: Training Data File

5.1.2 Variable

Crossover rate is essential for the behavior of the algorithm. The range of crossover rate is defined from zero to one **(i.e., Crossover Rate ∈ [0 , 1])**.

5.2 Experiment Results and Analysis

Each experiment has been repeated for 100 times. The results are shown as boxplots. In order to display the overall trends and each individual case, the increments of 3 for crossover rate was tried in this experiment **(i.e., Crossover Rate ∈ Set {0, 0.1, 0.2, 0.3, 0.4, 0.5,0.6, 0.7, 0.8, 0.9, 1.0 })**.

5.2.1 Boxplot of Results

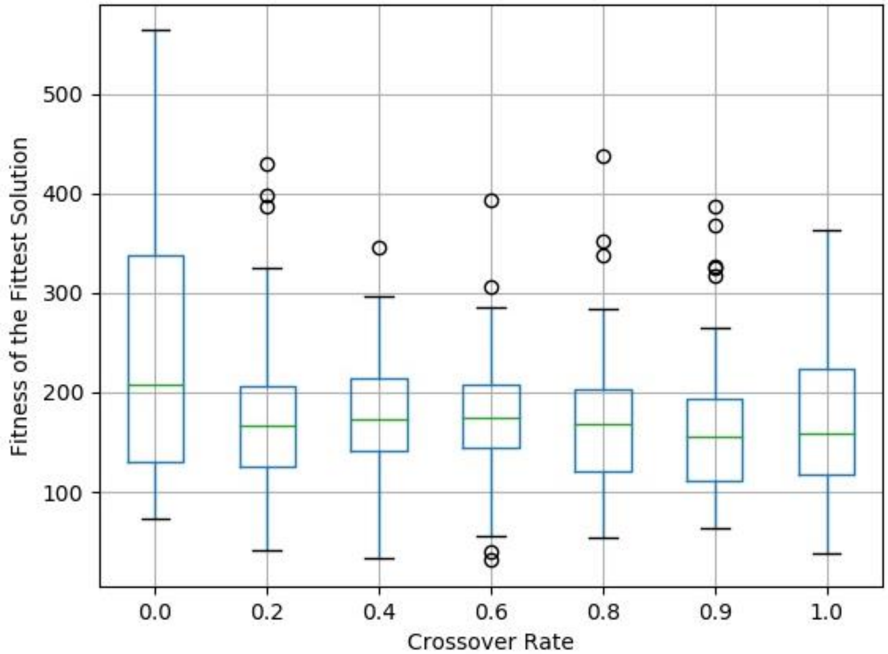


figure 5: figure 9: Fittest Solution VS Crossover Rate

5.2.2 Analysis of Boxplot

The boxplots summarize the impact of crossover rate on the quality (i.e., fitness of fittest solution) of the solutions obtained.

Overall, When the crossover rate is 0, the upper edge (i.e., the maximum value), the upper quartile (i.e., the third quartile), the median, the lower quartile (i.e., the first quartile) and the lower edge (i.e., the minimum value) all have the highest value. However, when the crossover rate is above 0.2, they all fluctuate slightly with the increment of crossover rate. It is obvious that when the crossover rate is 0.9, they all have the lowest value.

It is clear that when the crossover rate is 0, the height of the boxplot is much larger than others. It reveals that the distribution of normal values will become dispersive if the algorithm is lack of crossover operator.

In conclusion, the fitness of the solution represents the mean-square error of the prediction. The fittest solution has the lowest fitness. Therefore, setting crossover rate to 0.9 is the best choice in this experiment.

Appendix A

Fittest Solution vs Mutation Rate

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: Janet Chou
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: Janet Chou
from random import random, randint, choice
from copy import deepcopy
from PIL import Image, ImageDraw
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time

class funwrapper:
  def __init__(self, function, childcount, name):
    self.function = function
    self.childcount = childcount
    self.name = name

class variable:
  def __init__(self, var, value=0):
    self.var = var
    self.value = value
    self.name = str(var)
    self.type = "variable"

  def evaluate(self):
    return self.var.value

  def setvar(self, value):
    self.value = value

  def display(self, indent=0):
    print('%s%s' % (' '*indent, self.var))

class const:
  def __init__(self, value):
    self.value = value
    self.name = str(value)
    self.type = "constant"

  def evaluate(self):
    return self.value

  def display(self, indent=0):
    print('%s%d' % (' '*indent, self.value))

```

```
class node:
  def __init__(self, type, children, funwrap, var=None,
const=None):
    self.type = type
    self.children = children
    self.funwrap = funwrap
    self.variable = var
    self.const = const
    self.depth = self.refreshdepth()
    self.value = 0
    self.fitness = 0

  def eval(self,x):
    if self.type == "variable":
      return self.variable.value
    elif self.type == "constant":
      return self.const.value
    else:
      for c in self.children:
        result = [c.eval(x) for c in self.children]
        return self.funwrap.function(result,x)

  def getfitness(self, checkdata):#checkdata like
{"x":1,"result":3"}
    diff = 0
    #set variable value
    for data in checkdata:
      self.setvariablevalue(data)
      diff += (self.eval(data['x']) - data["result"])**2
    self.fitness = diff/len(checkdata)

  def setvariablevalue(self, value):
    if self.type == "variable":
      if self.variable.var in value:
        self.variable.setvar(value[self.variable.var])
      else:
        print("There is no value for variable:",
self.variable.var)
        return
    if self.type == "constant":
      pass
    if self.children:#function node
      for child in self.children:
        child.setvariablevalue(value)

  def refreshdepth(self):
    if self.type == "constant" or self.type == "variable":
      return 0
    else:
      depth = []
      for c in self.children:
        depth.append(c.refreshdepth())
      return max(depth) + 1
```

```

  def display(self, indent=0):
    if self.type == "function":
      print ((' '*indent) + self.funwrap.name)
    elif self.type == "variable":
      print ((' '*indent) + self.variable.name)
    elif self.type == "constant":
      print ((' '*indent) + self.const.name)
    if self.children:
      for c in self.children:
        c.display(indent + 1)
  ##for draw node
  def getwidth(self):
    if self.type == "variable" or self.type == "constant":
      return 1
    else:
      result = 0
      for i in range(0, len(self.children)):
        result += self.children[i].getwidth()
      return result
  def drawnode(self, draw, x, y):
    if self.type == "function":
      allwidth = 0
      for c in self.children:
        allwidth += c.getwidth()*100
      left = x - allwidth / 2
      #draw the function name
      draw.text((x - 10, y - 10), self.funwrap.name, (0, 0, 0))
      #draw the children
      for c in self.children:
        wide = c.getwidth()*100
        draw.line((x, y, left + wide / 2, y + 100), fill=(255, 0,
0))
        c.drawnode(draw, left + wide / 2, y + 100)
        left = left + wide
    elif self.type == "variable":
      draw.text((x - 5 , y), self.variable.name, (0, 0, 0))
    elif self.type == "constant":
      draw.text((x - 5 , y), self.const.name, (0, 0, 0))

  def drawtree(self, jpeg="tree.png"):
    w = self.getwidth()*100
    h = self.depth * 100 + 120

    img = Image.new('RGB', (w, h), (255, 255, 255))
    draw = ImageDraw.Draw(img)
    self.drawnode(draw, w / 2, 20)
    img.save(jpeg, 'PNG')

class enviroment:
  def __init__(self, mutationrate,funwraplist, variablelist,
constantlist,
  checkdata,

```

```
              minimaxtype="min", population=None,
size=10,maxdepth=10,
              maxgen=500, crossrate=0.9, newbirthrate=1):
    self.funwraplist = funwraplist
    self.variablelist = variablelist
    self.constantlist = constantlist
    self.checkdata = checkdata
    self.minimaxtype = minimaxtype
    self.maxdepth = maxdepth
    self.population = population or self._makepopulation(size)
    self.size = size
    self.maxgen = maxgen
    self.crossrate = crossrate
    self.mutationrate = mutationrate
    self.newbirthrate = newbirthrate

    self.besttree = self.population[0]
    for i in range(0, self.size):
      self.population[i].depth=self.population[i].refreshdepth()
      self.population[i].getfitness(checkdata)
      if self.minimaxtype == "min":
        if self.population[i].fitness < self.besttree.fitness:
          self.besttree = self.population[i]
      elif self.minimaxtype == "max":
        if self.population[i].fitness > self.besttree.fitness:
          self.besttree = self.population[i]

  def _makepopulation(self, popsize):
    return [self._maketree(0) for i in range(0, popsize)]

  def _maketree(self, startdepth):
    if startdepth == 0:
      #make a new tree
      nodepattern = 0#function
    elif startdepth == self.maxdepth:
      nodepattern = 1#variable or constant
    else:
      nodepattern = randint(0, 1)
    if nodepattern == 0:
      childlist = []
      selectedfun = randint(0, len(self.funwraplist) - 1)
      for i in range(0, self.funwraplist[selectedfun].childcount):
        child = self._maketree(startdepth + 1)
        childlist.append(child)
      return node("function", childlist,
self.funwraplist[selectedfun])
    else:
        selectedconstant = randint(0, len(self.constantlist) - 1)
        return node("constant", None, None, None,
              const(self.constantlist[selectedconstant]))

  def mutate(self, tree, probchange, startdepth=0):
    if random() < probchange:
      return self._maketree(startdepth)
```

```
    else:
      result = deepcopy(tree)
      if result.type == "function":
        result.children = [self.mutate(c, probchange, startdepth +
1) \
                      for c in tree.children]
    return result

  def crossover(self, tree1, tree2, probswap=1, top=1):
    if random() < probswap and not top:
      return deepcopy(tree2)
    else:
      result = deepcopy(tree1)
      if tree1.type == "function" and tree2.type == "function":
        result.children = [self.crossover(c,
choice(tree2.children),
                      probswap, 0) for c in tree1.children]
    return result

  def envolve(self, mutationrate,maxgen=1000, crossrate=0.9):
    timebudget=50
    start=time.clock()
    while True:
      # print("generation no.", i)
      child = []
      end=time.clock()
      if timebudget<(end-start):
        # print(end-start)
        break
      for j in range(0, int(self.size * self.newbirthrate / 2)):
        parent1, p1 = self.roulettewheelsel()
        parent2, p2 = self.roulettewheelsel()
        newchild = self.crossover(parent1, parent2)
        child.append(newchild)#generate new tree
        parent, p3 = self.roulettewheelsel()
        newchild = self.mutate(parent, mutationrate)
        child.append(newchild)
      #refresh all tree's fitness
      for j in range(0, int(self.size * self.newbirthrate)):
        replacedtree, replacedindex =
self.roulettewheelsel(reverse=True)
        #replace bad tree with child
        self.population[replacedindex] = child[j]

      for k in range(0, self.size):
        self.population[k].getfitness(self.checkdata)
        self.population[k].depth=self.population[k].refreshdepth()
        if self.minimaxtype == "min":
          if self.population[k].fitness < self.besttree.fitness:
            self.besttree = self.population[k]
        elif self.minimaxtype == "max":
          if self.population[k].fitness > self.besttree.fitness:
            self.besttree = self.population[k]
    return self.besttree.fitness
```

```python


  def roulettewheelsel(self, reverse=False):
    if reverse == False:
      allfitness = 0
      for i in range(0, self.size):
        allfitness += self.population[i].fitness
      randomnum = random()*(self.size - 1)
      check = 0
      for i in range(0, self.size):
        # print(self.population[i].fitness)
        check += (1.0 - self.population[i].fitness / allfitness)
        # print('---')
        # print(check)
        if check >= randomnum:
          return self.population[i], i
    if reverse == True:
      allfitness = 0
      for i in range(0, self.size):
        allfitness += self.population[i].fitness
      randomnum = random()
      check = 0
      for i in range(0, self.size):
        check += self.population[i].fitness * 1.0 / allfitness
        if check >= randomnum:
          return self.population[i], i
#################################################################

def add(ValuesList,x):
    sumtotal = 0
    for val in ValuesList:
      sumtotal = sumtotal + val
    return sumtotal

def sub(ValuesList,x):
    return ValuesList[0] - ValuesList[1]

def mul(ValuesList,x):
    return ValuesList[0] * ValuesList[1]

def div(ValuesList,x):
    if ValuesList[1] == 0:
        return 1
    return ValuesList[0] / ValuesList[1]

def pow(ValuesList,x):
    if ValuesList[0]==0 and ValuesList[1]==0:
        return 0
    else:
        return float(ValuesList[0] ** ValuesList[1])

def sqrt(ValuesList,x):
    if ValuesList[0]<0:
        return 0
```

```
    else:
        return np.sqrt(float(ValuesList[0]))
def log(ValuesList,x):
    if ValuesList[0] <=0:
        return 0
    else:
        return np.log2(float(ValuesList[0]))
def exp(ValuesList,x):
    return round(np.exp(float(ValuesList[0])),2)

def maximum(ValuesList,x):
    return max(ValuesList)

def ifleq(ValuesList,x):
    if ValuesList[0]<=ValuesList[1]:
        return ValuesList[2]
    else:
        return ValuesList[3]

def data(ValuesList,x):
  index_x=np.mod(round(float(ValuesList[0])),n)
  return x[int(index_x)]

def diff(ValuesList,x):
  k = np.mod(round(float(ValuesList[0])), n)
  l = np.mod(round(float(ValuesList[1])), n)
  result = np.subtract(x[int(k)], x[int(l)])
  return result

def avg(ValuesList,x):
  sum_x = 0
  k = np.mod(round(float(ValuesList[0])), n)
  l = np.mod(round(float(ValuesList[1])), n)
  if k == l:
    return 0
  else:
    t = np.minimum(k, l)
    upper = np.maximum(k, l) - 1
    for z in range(int(t), int(upper) + 1):
      sum_x = sum_x + x[z]
    result = np.divide(sum_x, (k - l))
    return result

addwrapper = funwrapper(add, 2, "add")
subwrapper = funwrapper(sub, 2, "sub")
mulwrapper = funwrapper(mul, 2, "mul")
divwrapper = funwrapper(div, 2, "div")

powwrapper = funwrapper(pow, 2, "pow")
sqrtwrapper = funwrapper(sqrt, 1, "sqrt")
logwrapper = funwrapper(log, 1, "log")
expwrapper = funwrapper(exp, 1, "exp")
maximumwrapper = funwrapper(maximum, 2, "maximum")
ifleqwrapper = funwrapper(ifleq, 4, "ifleq")
```

```
datawrapper = funwrapper(data, 1, "data")
diffwrapper = funwrapper(diff, 2, "diff")
avgwrapper = funwrapper(avg, 2, "avg")

def constructcheckdata(filename,m,n):
  with open(filename,'r') as f:
    checkdata=list()
    for i in f:
      dic=dict()
      train_data = list(map(float,
i.strip('\n').strip('\t').split('\t')))
      x = train_data[:n]
      result= train_data[n]
      dic['x'] = x
      dic['result'] = result
      checkdata.append(dic)
    return checkdata
  pass

if __name__ == "__main__":
  filename='data.txt'
  m=10
  n=10
  checkdata = constructcheckdata(filename,m,n)
  # print(checkdata)
  dic_nsat=dict()
  mutation_list = [ 0.001, 0.01, 0.05, 0.1, 0.15, 0.2, 0.3,
0.4,0.5,0.8]
  for mutationrate in mutation_list:
      list_nsat=list()
      for i in range(80):
          env = enviroment(mutationrate,[addwrapper, subwrapper,
mulwrapper,
          divwrapper,sqrtwrapper,logwrapper,maximumwrapper,

ifleqwrapper,datawrapper,diffwrapper,avgwrapper],
                            ["x", "y"],
                           list(range(-100,100)), checkdata)
          best_fitness=env.envolve(mutationrate)
          list_nsat.append(best_fitness)
          print(best_fitness)
      # print('---')
      dic_nsat[mutationrate]=list_nsat
  data = pd.DataFrame(dic_nsat)
  data.boxplot()
  plt.ylabel('Fitness of the Fittest Solution')
  plt.xlabel('Mutation Rate')
  plt.savefig('figure4.jpg')
```

Appendix B

Fittest Solution VS Max Depth of Tree

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: Janet Chou
from random import random, randint, choice
from copy import deepcopy
from PIL import Image, ImageDraw
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time

class funwrapper:
  def __init__(self, function, childcount, name):
    self.function = function
    self.childcount = childcount
    self.name = name

class variable:
  def __init__(self, var, value=0):
    self.var = var
    self.value = value
    self.name = str(var)
    self.type = "variable"

  def evaluate(self):
    return self.var.value

  def setvar(self, value):
    self.value = value

  def display(self, indent=0):
    print('%s%s' % (' '*indent, self.var))

class const:
  def __init__(self, value):
    self.value = value
    self.name = str(value)
    self.type = "constant"

  def evaluate(self):
    return self.value

  def display(self, indent=0):
    print('%s%d' % (' '*indent, self.value))

class node:
  def __init__(self, type, children, funwrap, var=None,
const=None):
    self.type = type
```

```
    self.children = children
    self.funwrap = funwrap
    self.variable = var
    self.const = const
    self.depth = self.refreshdepth()
    self.value = 0
    self.fitness = 0

  def eval(self,x):
    if self.type == "variable":
      return self.variable.value
    elif self.type == "constant":
      return self.const.value
    else:
      for c in self.children:
        result = [c.eval(x) for c in self.children]
        return self.funwrap.function(result,x)

  def getfitness(self, checkdata):#checkdata like
{"x":1,"result":3"}
    diff = 0
    #set variable value
    for data in checkdata:
      self.setvariablevalue(data)
      diff += (self.eval(data['x']) - data["result"])**2
    self.fitness = diff/len(checkdata)

  def setvariablevalue(self, value):
    if self.type == "variable":
      if self.variable.var in value:
        self.variable.setvar(value[self.variable.var])
      else:
        print("There is no value for variable:",
self.variable.var)
        return
    if self.type == "constant":
      pass
    if self.children:#function node
      for child in self.children:
        child.setvariablevalue(value)

  def refreshdepth(self):
    if self.type == "constant" or self.type == "variable":
      return 0
    else:
      depth = []
      for c in self.children:
        depth.append(c.refreshdepth())
      return max(depth) + 1


  def display(self, indent=0):
    if self.type == "function":
      print (('  '*indent) + self.funwrap.name)
```

```
    elif self.type == "variable":
      print (('  '*indent) + self.variable.name)
    elif self.type == "constant":
      print (('  '*indent) + self.const.name)
    if self.children:
      for c in self.children:
        c.display(indent + 1)
  ##for draw node
  def getwidth(self):
    if self.type == "variable" or self.type == "constant":
      return 1
    else:
      result = 0
      for i in range(0, len(self.children)):
        result += self.children[i].getwidth()
      return result
  def drawnode(self, draw, x, y):
    if self.type == "function":
      allwidth = 0
      for c in self.children:
        allwidth += c.getwidth()*100
      left = x - allwidth / 2
      #draw the function name
      draw.text((x - 10, y - 10), self.funwrap.name, (0, 0, 0))
      #draw the children
      for c in self.children:
        wide = c.getwidth()*100
        draw.line((x, y, left + wide / 2, y + 100), fill=(255, 0,
0))
        c.drawnode(draw, left + wide / 2, y + 100)
        left = left + wide
    elif self.type == "variable":
      draw.text((x - 5 , y), self.variable.name, (0, 0, 0))
    elif self.type == "constant":
      draw.text((x - 5 , y), self.const.name, (0, 0, 0))

  def drawtree(self, jpeg="tree.png"):
    w = self.getwidth()*100
    h = self.depth * 100 + 120

    img = Image.new('RGB', (w, h), (255, 255, 255))
    draw = ImageDraw.Draw(img)
    self.drawnode(draw, w / 2, 20)
    img.save(jpeg, 'PNG')

class enviroment:
  def __init__(self, maxdepth,funwraplist, variablelist,
constantlist, checkdata,
            minimaxtype="min", population=None, size=10,
            maxgen=500, crossrate=0.9, mutationrate=0.1,
newbirthrate=1):
    self.funwraplist = funwraplist
    self.variablelist = variablelist
    self.constantlist = constantlist
```

```
    self.checkdata = checkdata
    self.minimaxtype = minimaxtype
    self.maxdepth = maxdepth
    self.population = population or self._makepopulation(size)
    self.size = size
    self.maxgen = maxgen
    self.crossrate = crossrate
    self.mutationrate = mutationrate
    self.newbirthrate = newbirthrate

    self.besttree = self.population[0]
    for i in range(0, self.size):
      self.population[i].depth=self.population[i].refreshdepth()
      self.population[i].getfitness(checkdata)
      if self.minimaxtype == "min":
        if self.population[i].fitness < self.besttree.fitness:
          self.besttree = self.population[i]
      elif self.minimaxtype == "max":
        if self.population[i].fitness > self.besttree.fitness:
          self.besttree = self.population[i]

  def _makepopulation(self, popsize):
    return [self._maketree(0) for i in range(0, popsize)]

  def _maketree(self, startdepth):
    if startdepth == 0:
      #make a new tree
      nodepattern = 0#function
    elif startdepth == self.maxdepth:
      nodepattern = 1#variable or constant
    else:
      nodepattern = randint(0, 1)
    if nodepattern == 0:
      childlist = []
      selectedfun = randint(0, len(self.funwraplist) - 1)
      for i in range(0, self.funwraplist[selectedfun].childcount):
        child = self._maketree(startdepth + 1)
        childlist.append(child)
      return node("function", childlist,
self.funwraplist[selectedfun])
    else:
      # if randint(0, 1) == 0:#variable
      #   selectedvariable = randint(0, len(self.variablelist) - 1)
      #   return node("variable", None, None,
      #         variable(self.variablelist[selectedvariable]),
None)
      # else:
        selectedconstant = randint(0, len(self.constantlist) - 1)
        return node("constant", None, None, None,
              const(self.constantlist[selectedconstant]))

  def mutate(self, tree, probchange=0.1, startdepth=0):
    if random() < probchange:
      return self._maketree(startdepth)
```

```
    else:
      result = deepcopy(tree)
      if result.type == "function":
        result.children = [self.mutate(c, probchange, startdepth +
1) \
                      for c in tree.children]
    return result

  def crossover(self, tree1, tree2, probswap=1, top=1):
    if random() < probswap and not top:
      return deepcopy(tree2)
    else:
      result = deepcopy(tree1)
      if tree1.type == "function" and tree2.type == "function":
        result.children = [self.crossover(c,
choice(tree2.children),
                      probswap, 0) for c in tree1.children]
    return result

  def envolve(self, maxgen=1000, crossrate=0.9,
mutationrate=0.1):
    timebudget=50
    start=time.clock()
    while True:
      # print("generation no.", i)
      child = []
      end=time.clock()
      if timebudget<(end-start):
        print(end-start)
        break
      for j in range(0, int(self.size * self.newbirthrate / 2)):
        parent1, p1 = self.roulettewheelsel()
        parent2, p2 = self.roulettewheelsel()
        newchild = self.crossover(parent1, parent2)
        child.append(newchild)#generate new tree
        parent, p3 = self.roulettewheelsel()
        newchild = self.mutate(parent, mutationrate)
        child.append(newchild)
      #refresh all tree's fitness
      for j in range(0, int(self.size * self.newbirthrate)):
        replacedtree, replacedindex =
self.roulettewheelsel(reverse=True)
        #replace bad tree with child
        self.population[replacedindex] = child[j]

      for k in range(0, self.size):
        self.population[k].getfitness(self.checkdata)
        self.population[k].depth=self.population[k].refreshdepth()
        if self.minimaxtype == "min":
          if self.population[k].fitness < self.besttree.fitness:
            self.besttree = self.population[k]
        elif self.minimaxtype == "max":
          if self.population[k].fitness > self.besttree.fitness:
            self.besttree = self.population[k]
```

```
   # print("best tree's fitbess..",self.besttree.fitness)
  # self.besttree.display()
  # self.besttree.drawtree()
  return self.besttree.fitness


 def roulettewheelsel(self, reverse=False):
  if reverse == False:
   allfitness = 0
   for i in range(0, self.size):
    allfitness += self.population[i].fitness
   randomnum = random()*(self.size - 1)
   check = 0
   for i in range(0, self.size):
    # print(self.population[i].fitness)
    check += (1.0 - self.population[i].fitness / allfitness)
    # print('---')
    # print(check)
    if check >= randomnum:
     return self.population[i], i
  if reverse == True:
   allfitness = 0
   for i in range(0, self.size):
    allfitness += self.population[i].fitness
   randomnum = random()
   check = 0
   for i in range(0, self.size):
    check += self.population[i].fitness * 1.0 / allfitness
    if check >= randomnum:
     return self.population[i], i


#############################################################

def add(ValuesList,x):
  sumtotal = 0
  for val in ValuesList:
   sumtotal = sumtotal + val
  return sumtotal

def sub(ValuesList,x):
  return ValuesList[0] - ValuesList[1]

def mul(ValuesList,x):
  return ValuesList[0] * ValuesList[1]

def div(ValuesList,x):
  if ValuesList[1] == 0:
    return 1
  return ValuesList[0] / ValuesList[1]

def pow(ValuesList,x):
  if ValuesList[0]==0 and ValuesList[1]==0:
    return 0
```

```
    else:
        return float(ValuesList[0] ** ValuesList[1])

def sqrt(ValuesList,x):
    if ValuesList[0]<0:
        return 0
    else:
        return np.sqrt(float(ValuesList[0]))
def log(ValuesList,x):
    if ValuesList[0] <=0:
        return 0
    else:
        return np.log2(float(ValuesList[0]))
def exp(ValuesList,x):
    return round(np.exp(float(ValuesList[0])),2)

def maximum(ValuesList,x):
    return max(ValuesList)

def ifleq(ValuesList,x):
    if ValuesList[0]<=ValuesList[1]:
        return ValuesList[2]
    else:
        return ValuesList[3]


def data(ValuesList,x):
  index_x=np.mod(round(float(ValuesList[0])),n)
  return x[int(index_x)]

def diff(ValuesList,x):
  k = np.mod(round(float(ValuesList[0])), n)
  l = np.mod(round(float(ValuesList[1])), n)
  result = np.subtract(x[int(k)], x[int(l)])
  return result

def avg(ValuesList,x):
  sum x = 0
  k = np.mod(round(float(ValuesList[0])), n)
  l = np.mod(round(float(ValuesList[1])), n)
  if k == l:
    return 0
  else:
    t = np.minimum(k, l)
    upper = np.maximum(k, l) - 1
    for z in range(int(t), int(upper) + 1):
      sum_x = sum_x + x[z]
    result = np.divide(sum_x, (k - l))
    return result



addwrapper = funwrapper(add, 2, "add")
subwrapper = funwrapper(sub, 2, "sub")
```

```
mulwrapper = funwrapper(mul, 2, "mul")
divwrapper = funwrapper(div, 2, "div")

powwrapper = funwrapper(pow, 2, "pow")
sqrtwrapper = funwrapper(sqrt, 1, "sqrt")
logwrapper = funwrapper(log, 1, "log")
expwrapper = funwrapper(exp, 1, "exp")
maximumwrapper = funwrapper(maximum, 2, "maximum")
ifleqwrapper = funwrapper(ifleq, 4, "ifleq")

datawrapper = funwrapper(data, 1, "data")
diffwrapper = funwrapper(diff, 2, "diff")
avgwrapper = funwrapper(avg, 2, "avg")

def constructcheckdata(filename,m,n):
  with open(filename,'r') as f:
    checkdata=list()
    for i in f:
      dic=dict()
      train_data = list(map(float,
i.strip('\n').strip('\t').split('\t')))
      x = train_data[:n]
      result= train_data[n]
      dic['x'] = x
      dic['result'] = result
      checkdata.append(dic)
    return checkdata

  pass

if __name__ == "__main__":
  filename='data.txt'
  m=10
  n=10
  checkdata = constructcheckdata(filename,m,n)
  # print(checkdata)
  dic_nsat=dict()
  for maxdepth in range(10,100,10):
      list_nsat=list()
      for i in range(80):
          env = enviroment(maxdepth,[addwrapper, subwrapper,
mulwrapper,divwrapper,
          sqrtwrapper,logwrapper,maximumwrapper,

ifleqwrapper,datawrapper,diffwrapper,avgwrapper],
                             ["x", "y"],
                           list(range(-100,100)), checkdata)
          best_fitness=env.envolve()
          list_nsat.append(best_fitness)
          print(best_fitness)
      # print('---')
      dic_nsat[maxdepth]=list_nsat
  data = pd.DataFrame(dic_nsat)
  data.boxplot()
```

```
plt.ylabel('Fitness of the Fittest Solution')
plt.xlabel('Max Depth of Tree')
plt.savefig('figure3.jpg')
```

Appendix C

Fittest Solution VS Population Size

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: Janet Chou
from random import random, randint, choice
from copy import deepcopy
from PIL import Image, ImageDraw
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time

class funwrapper:
  def __init__(self, function, childcount, name):
    self.function = function
    self.childcount = childcount
    self.name = name

class variable:
  def __init__(self, var, value=0):
    self.var = var
    self.value = value
    self.name = str(var)
    self.type = "variable"

  def evaluate(self):
    return self.var.value

  def setvar(self, value):
    self.value = value

  def display(self, indent=0):
    print('%s%s' % (' '*indent, self.var))

class const:
  def __init__(self, value):
    self.value = value
    self.name = str(value)
    self.type = "constant"

  def evaluate(self):
    return self.value

  def display(self, indent=0):
    print('%s%d' % (' '*indent, self.value))

class node:
  def __init__(self, type, children, funwrap, var=None,
const=None):
```

```
    self.type = type
    self.children = children
    self.funwrap = funwrap
    self.variable = var
    self.const = const
    self.depth = self.refreshdepth()
    self.value = 0
    self.fitness = 0

  def eval(self,x):
    if self.type == "variable":
      return self.variable.value
    elif self.type == "constant":
      return self.const.value
    else:
      for c in self.children:
        result = [c.eval(x) for c in self.children]
        return self.funwrap.function(result,x)

  def getfitness(self, checkdata):#checkdata like
{"x":1,"result":3"}
    diff = 0
    #set variable value
    for data in checkdata:
      self.setvariablevalue(data)
      diff += (self.eval(data['x']) - data["result"])**2
    self.fitness = diff/len(checkdata)

  def setvariablevalue(self, value):
    if self.type == "variable":
      if self.variable.var in value:
        self.variable.setvar(value[self.variable.var])
      else:
        print("There is no value for variable:",
self.variable.var)
        return
    if self.type == "constant":
      pass
    if self.children:#function node
      for child in self.children:
        child.setvariablevalue(value)

  def refreshdepth(self):
    if self.type == "constant" or self.type == "variable":
      return 0
    else:
      depth = []
      for c in self.children:
        depth.append(c.refreshdepth())
      return max(depth) + 1


  def display(self, indent=0):
    if self.type == "function":
```

```
      print (('  '*indent) + self.funwrap.name)
    elif self.type == "variable":
      print (('  '*indent) + self.variable.name)
    elif self.type == "constant":
      print (('  '*indent) + self.const.name)
    if self.children:
      for c in self.children:
        c.display(indent + 1)
  ##for draw node
  def getwidth(self):
    if self.type == "variable" or self.type == "constant":
      return 1
    else:
      result = 0
      for i in range(0, len(self.children)):
        result += self.children[i].getwidth()
      return result
  def drawnode(self, draw, x, y):
    if self.type == "function":
      allwidth = 0
      for c in self.children:
        allwidth += c.getwidth()*100
      left = x - allwidth / 2
      #draw the function name
      draw.text((x - 10, y - 10), self.funwrap.name, (0, 0, 0))
      #draw the children
      for c in self.children:
        wide = c.getwidth()*100
        draw.line((x, y, left + wide / 2, y + 100), fill=(255, 0,
0))
        c.drawnode(draw, left + wide / 2, y + 100)
        left = left + wide
    elif self.type == "variable":
      draw.text((x - 5 , y), self.variable.name, (0, 0, 0))
    elif self.type == "constant":
      draw.text((x - 5 , y), self.const.name, (0, 0, 0))

  def drawtree(self, jpeg="tree.png"):
    w = self.getwidth()*100
    h = self.depth * 100 + 120

    img = Image.new('RGB', (w, h), (255, 255, 255))
    draw = ImageDraw.Draw(img)
    self.drawnode(draw, w / 2, 20)
    img.save(jpeg, 'PNG')

class enviroment:
  def __init__(self, size,funwraplist, variablelist,
constantlist, checkdata,
               minimaxtype="min", population=None, maxdepth=10,
               maxgen=500, crossrate=0.9, mutationrate=0.1,
newbirthrate=1):
    self.funwraplist = funwraplist
    self.variablelist = variablelist
```

```
    self.constantlist = constantlist
    self.checkdata = checkdata
    self.minimaxtype = minimaxtype
    self.maxdepth = maxdepth
    self.population = population or self._makepopulation(size)
    self.size = size
    self.maxgen = maxgen
    self.crossrate = crossrate
    self.mutationrate = mutationrate
    self.newbirthrate = newbirthrate

    self.besttree = self.population[0]
    for i in range(0, self.size):
      self.population[i].depth=self.population[i].refreshdepth()
      self.population[i].getfitness(checkdata)
      if self.minimaxtype == "min":
        if self.population[i].fitness < self.besttree.fitness:
          self.besttree = self.population[i]
      elif self.minimaxtype == "max":
        if self.population[i].fitness > self.besttree.fitness:
          self.besttree = self.population[i]

  def _makepopulation(self, popsize):
    return [self._maketree(0) for i in range(0, popsize)]

  def _maketree(self, startdepth):
    if startdepth == 0:
      #make a new tree
      nodepattern = 0#function
    elif startdepth == self.maxdepth:
      nodepattern = 1#variable or constant
    else:
      nodepattern = randint(0, 1)
    if nodepattern == 0:
      childlist = []
      selectedfun = randint(0, len(self.funwraplist) - 1)
      for i in range(0, self.funwraplist[selectedfun].childcount):
        child = self._maketree(startdepth + 1)
        childlist.append(child)
      return node("function", childlist,
self.funwraplist[selectedfun])
    else:
      # if randint(0, 1) == 0:#variable
      #   selectedvariable = randint(0, len(self.variablelist) - 1)
      #   return node("variable", None, None,
      #         variable(self.variablelist[selectedvariable]),
None)
      # else:
        selectedconstant = randint(0, len(self.constantlist) - 1)
        return node("constant", None, None,
               const(self.constantlist[selectedconstant]))

  def mutate(self, tree, probchange=0.1, startdepth=0):
    if random() < probchange:
```

```
      return self._maketree(startdepth)
    else:
      result = deepcopy(tree)
      if result.type == "function":
        result.children = [self.mutate(c, probchange, startdepth +
1) \
                          for c in tree.children]
    return result

  def crossover(self, tree1, tree2, probswap=1, top=1):
    if random() < probswap and not top:
      return deepcopy(tree2)
    else:
      result = deepcopy(tree1)
      if tree1.type == "function" and tree2.type == "function":
        result.children = [self.crossover(c,
choice(tree2.children),
                          probswap, 0) for c in tree1.children]
    return result

  def envolve(self, maxgen=1000, crossrate=0.9,
mutationrate=0.1):
    timebudget=50
    start=time.clock()
    while True:
      # print("generation no.", i)
      child = []
      end=time.clock()
      if timebudget<(end-start):
        break
      for j in range(0, int(self.size * self.newbirthrate / 2)):
        parent1, p1 = self.roulettewheelsel()
        parent2, p2 = self.roulettewheelsel()
        newchild = self.crossover(parent1, parent2)
        child.append(newchild)#generate new tree
        parent, p3 = self.roulettewheelsel()
        newchild = self.mutate(parent, mutationrate)
        child.append(newchild)
      #refresh all tree's fitness
      for j in range(0, int(self.size * self.newbirthrate)):
        replacedtree, replacedindex =
self.roulettewheelsel(reverse=True)
        #replace bad tree with child
        self.population[replacedindex] = child[j]

      for k in range(0, self.size):
        self.population[k].getfitness(self.checkdata)
        self.population[k].depth=self.population[k].refreshdepth()
        if self.minimaxtype == "min":
          if self.population[k].fitness < self.besttree.fitness:
            self.besttree = self.population[k]
        elif self.minimaxtype == "max":
          if self.population[k].fitness > self.besttree.fitness:
            self.besttree = self.population[k]
```

```
      # print("best tree's fitbess..",self.besttree.fitness)
    # self.besttree.display()
    # self.besttree.drawtree()
    return self.besttree.fitness


  def roulettewheelsel(self, reverse=False):
    if reverse == False:
      allfitness = 0
      for i in range(0, self.size):
        allfitness += self.population[i].fitness
      randomnum = random()*(self.size - 1)
      check = 0
      for i in range(0, self.size):
        # print(self.population[i].fitness)
        check += (1.0 - self.population[i].fitness / allfitness)
        # print('---')
        # print(check)
        if check >= randomnum:
          return self.population[i], i
    if reverse == True:
      allfitness = 0
      for i in range(0, self.size):
        allfitness += self.population[i].fitness
      randomnum = random()
      check = 0
      for i in range(0, self.size):
        check += self.population[i].fitness * 1.0 / allfitness
        if check >= randomnum:
          return self.population[i], i


############################################################

def add(ValuesList,x):
    sumtotal = 0
    for val in ValuesList:
      sumtotal = sumtotal + val
    return sumtotal

def sub(ValuesList,x):
    return ValuesList[0] - ValuesList[1]

def mul(ValuesList,x):
    return ValuesList[0] * ValuesList[1]

def div(ValuesList,x):
    if ValuesList[1] == 0:
        return 1
    return ValuesList[0] / ValuesList[1]

def pow(ValuesList,x):
    if ValuesList[0]==0 and ValuesList[1]==0:
        return 0
```

```
    else:
        return float(ValuesList[0] ** ValuesList[1])

def sqrt(ValuesList,x):
    if ValuesList[0]<0:
        return 0
    else:
        return np.sqrt(float(ValuesList[0]))
def log(ValuesList,x):
    if ValuesList[0] <=0:
        return 0
    else:
        return np.log2(float(ValuesList[0]))
def exp(ValuesList,x):
    return round(np.exp(float(ValuesList[0])),2)

def maximum(ValuesList,x):
    return max(ValuesList)

def ifleq(ValuesList,x):
    if ValuesList[0]<=ValuesList[1]:
        return ValuesList[2]
    else:
        return ValuesList[3]


def data(ValuesList,x):
  index_x=np.mod(round(float(ValuesList[0])),n)
  return x[int(index_x)]

def diff(ValuesList,x):
  k = np.mod(round(float(ValuesList[0])), n)
  l = np.mod(round(float(ValuesList[1])), n)
  result = np.subtract(x[int(k)], x[int(l)])
  return result

def avg(ValuesList,x):
  sum x = 0
  k = np.mod(round(float(ValuesList[0])), n)
  l = np.mod(round(float(ValuesList[1])), n)
  if k == l:
    return 0
  else:
    t = np.minimum(k, l)
    upper = np.maximum(k, l) - 1
    for z in range(int(t), int(upper) + 1):
      sum_x = sum_x + x[z]
    result = np.divide(sum_x, (k - l))
    return result



addwrapper = funwrapper(add, 2, "add")
subwrapper = funwrapper(sub, 2, "sub")
```

```
mulwrapper = funwrapper(mul, 2, "mul")
divwrapper = funwrapper(div, 2, "div")

powwrapper = funwrapper(pow, 2, "pow")
sqrtwrapper = funwrapper(sqrt, 1, "sqrt")
logwrapper = funwrapper(log, 1, "log")
expwrapper = funwrapper(exp, 1, "exp")
maximumwrapper = funwrapper(maximum, 2, "maximum")
ifleqwrapper = funwrapper(ifleq, 4, "ifleq")

datawrapper = funwrapper(data, 1, "data")
diffwrapper = funwrapper(diff, 2, "diff")
avgwrapper = funwrapper(avg, 2, "avg")

def constructcheckdata(filename,m,n):
  with open(filename,'r') as f:
    checkdata=list()
    for i in f:
      dic=dict()
      train_data = list(map(float,
i.strip('\n').strip('\t').split('\t')))
      x = train_data[:n]
      result= train_data[n]
      dic['x'] = x
      dic['result'] = result
      checkdata.append(dic)
    return checkdata


  pass

# def examplefun(x, y):
#   return x * x + x + 2 * y + 1
# def constructcheckdata(count=10):
#   checkdata = []
#   for i in range(0, count):
#     dic = {}
#     x = randint(0, 10)
#     y = randint(0, 10)
#     dic['x'] = x
#     dic['y'] = y
#     dic['result'] = examplefun(x, y)
#     checkdata.append(dic)
#   return checkdata

if __name__ == "__main__":
  filename='data.txt'
  m=10
  n=10
  checkdata = constructcheckdata(filename,m,n)
  # print(checkdata)
  dic_nsat=dict()
  for population_size in range(10,430,30):
      list_nsat=list()
```

```
        for i in range(80):
            env = enviroment(population_size,[addwrapper, subwrapper,
mulwrapper
            ,divwrapper,sqrtwrapper,logwrapper,maximumwrapper,

ifleqwrapper,datawrapper,diffwrapper,avgwrapper],
                            ["x", "y"],
                           list(range(-100,100)), checkdata)
            best_fitness=env.envolve()
            list_nsat.append(best_fitness)
            print(best_fitness)
        # print('---')
        dic_nsat[population_size]=list_nsat
  data = pd.DataFrame(dic_nsat)
  data.boxplot()
  plt.ylabel('Fitness of the Fittest Solution')
  plt.xlabel('Population Size')
  plt.savefig('figure2.jpg')
```

Appendix D

Fittest Solution VS Crossover Rate

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: Janet Chou
from random import random, randint, choice
from copy import deepcopy
from PIL import Image, ImageDraw
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time

class funwrapper:
  def __init__(self, function, childcount, name):
    self.function = function
    self.childcount = childcount
    self.name = name

class variable:
  def __init__(self, var, value=0):
    self.var = var
    self.value = value
    self.name = str(var)
    self.type = "variable"

  def evaluate(self):
    return self.var.value

  def setvar(self, value):
    self.value = value

  def display(self, indent=0):
    print('%s%s' % (' '*indent, self.var))

class const:
  def __init__(self, value):
    self.value = value
    self.name = str(value)
    self.type = "constant"

  def evaluate(self):
    return self.value

  def display(self, indent=0):
    print('%s%d' % (' '*indent, self.value))

class node:
  def __init__(self, type, children, funwrap, var=None,
const=None):
    self.type = type
```

```
    self.children = children
    self.funwrap = funwrap
    self.variable = var
    self.const = const
    self.depth = self.refreshdepth()
    self.value = 0
    self.fitness = 0

  def eval(self,x):
    if self.type == "variable":
      return self.variable.value
    elif self.type == "constant":
      return self.const.value
    else:
      for c in self.children:
        result = [c.eval(x) for c in self.children]
        return self.funwrap.function(result,x)

  def getfitness(self, checkdata):#checkdata like
{"x":1,"result":3"}
    diff = 0
    #set variable value
    for data in checkdata:
      self.setvariablevalue(data)
      diff += (self.eval(data['x']) - data["result"])**2
    self.fitness = diff/len(checkdata)

  def setvariablevalue(self, value):
    if self.type == "variable":
      if self.variable.var in value:
        self.variable.setvar(value[self.variable.var])
      else:
        print("There is no value for variable:",
self.variable.var)
        return
    if self.type == "constant":
      pass
    if self.children:#function node
      for child in self.children:
        child.setvariablevalue(value)

  def refreshdepth(self):
    if self.type == "constant" or self.type == "variable":
      return 0
    else:
      depth = []
      for c in self.children:
        depth.append(c.refreshdepth())
      return max(depth) + 1


  def display(self, indent=0):
    if self.type == "function":
      print (('  '*indent) + self.funwrap.name)
```

```
    elif self.type == "variable":
      print (('  '*indent) + self.variable.name)
    elif self.type == "constant":
      print (('  '*indent) + self.const.name)
    if self.children:
      for c in self.children:
        c.display(indent + 1)
  ##for draw node
  def getwidth(self):
    if self.type == "variable" or self.type == "constant":
      return 1
    else:
      result = 0
      for i in range(0, len(self.children)):
        result += self.children[i].getwidth()
      return result
  def drawnode(self, draw, x, y):
    if self.type == "function":
      allwidth = 0
      for c in self.children:
        allwidth += c.getwidth()*100
      left = x - allwidth / 2
      #draw the function name
      draw.text((x - 10, y - 10), self.funwrap.name, (0, 0, 0))
      #draw the children
      for c in self.children:
        wide = c.getwidth()*100
        draw.line((x, y, left + wide / 2, y + 100), fill=(255, 0,
0))
        c.drawnode(draw, left + wide / 2, y + 100)
        left = left + wide
    elif self.type == "variable":
      draw.text((x - 5 , y), self.variable.name, (0, 0, 0))
    elif self.type == "constant":
      draw.text((x - 5 , y), self.const.name, (0, 0, 0))

  def drawtree(self, jpeg="tree.png"):
    w = self.getwidth()*100
    h = self.depth * 100 + 120

    img = Image.new('RGB', (w, h), (255, 255, 255))
    draw = ImageDraw.Draw(img)
    self.drawnode(draw, w / 2, 20)
    img.save(jpeg, 'PNG')

class enviroment:
  def __init__(self,crossrate, funwraplist, variablelist,
constantlist, checkdata,
             minimaxtype="min", population=None,
size=10,mutationrate=0.1,
             maxdepth=10,
             maxgen=500,  newbirthrate=1):
    self.funwraplist = funwraplist
    self.variablelist = variablelist
```

```
    self.constantlist = constantlist
    self.checkdata = checkdata
    self.minimaxtype = minimaxtype
    self.maxdepth = maxdepth
    self.population = population or self._makepopulation(size)
    self.size = size
    self.maxgen = maxgen
    self.crossrate = crossrate
    self.mutationrate = mutationrate
    self.newbirthrate = newbirthrate

    self.besttree = self.population[0]
    for i in range(0, self.size):
      self.population[i].depth=self.population[i].refreshdepth()
      self.population[i].getfitness(checkdata)
      if self.minimaxtype == "min":
        if self.population[i].fitness < self.besttree.fitness:
          self.besttree = self.population[i]
      elif self.minimaxtype == "max":
        if self.population[i].fitness > self.besttree.fitness:
          self.besttree = self.population[i]

  def _makepopulation(self, popsize):
    return [self._maketree(0) for i in range(0, popsize)]

  def _maketree(self, startdepth):
    if startdepth == 0:
      #make a new tree
      nodepattern = 0#function
    elif startdepth == self.maxdepth:
      nodepattern = 1#variable or constant
    else:
      nodepattern = randint(0, 1)
    if nodepattern == 0:
      childlist = []
      selectedfun = randint(0, len(self.funwraplist) - 1)
      for i in range(0, self.funwraplist[selectedfun].childcount):
        child = self._maketree(startdepth + 1)
        childlist.append(child)
      return node("function", childlist,
self.funwraplist[selectedfun])
    else:
      # if randint(0, 1) == 0:#variable
      #   selectedvariable = randint(0, len(self.variablelist) - 1)
      #   return node("variable", None, None,
      #         variable(self.variablelist[selectedvariable]),
None)
      # else:
        selectedconstant = randint(0, len(self.constantlist) - 1)
        return node("constant", None, None, None,
              const(self.constantlist[selectedconstant]))

  def mutate(self, tree, probchange=0.1, startdepth=0):
    if random() < probchange:
```

```
      return self._maketree(startdepth)
    else:
      result = deepcopy(tree)
      if result.type == "function":
        result.children = [self.mutate(c, probchange, startdepth +
1) \
                      for c in tree.children]
    return result

  def crossover(self, tree1, tree2, probswap, top=1):
    if random() < probswap and not top:
      return deepcopy(tree2)
    else:
      result = deepcopy(tree1)
      if tree1.type == "function" and tree2.type == "function":
        result.children = [self.crossover(c,
choice(tree2.children),
                      probswap, 0) for c in tree1.children]
    return result

  def envolve(self,crossrate,mutationrate=0.1,maxgen=1000):
    timebudget=50
    start=time.clock()
    while True:
      # print("generation no.", i)
      child = []
      end=time.clock()
      if timebudget<(end-start):
        # print(end-start)
        break
      for j in range(0, int(self.size * self.newbirthrate / 2)):
        parent1, p1 = self.roulettewheelsel()
        parent2, p2 = self.roulettewheelsel()
        newchild = self.crossover(parent1, parent2,crossrate)
        child.append(newchild)#generate new tree
        parent, p3 = self.roulettewheelsel()
        newchild = self.mutate(parent, mutationrate)
        child.append(newchild)
      #refresh all tree's fitness
      for j in range(0, int(self.size * self.newbirthrate)):
        replacedtree, replacedindex =
self.roulettewheelsel(reverse=True)
        #replace bad tree with child
        self.population[replacedindex] = child[j]

      for k in range(0, self.size):
        self.population[k].getfitness(self.checkdata)
        self.population[k].depth=self.population[k].refreshdepth()
        if self.minimaxtype == "min":
          if self.population[k].fitness < self.besttree.fitness:
            self.besttree = self.population[k]
        elif self.minimaxtype == "max":
          if self.population[k].fitness > self.besttree.fitness:
            self.besttree = self.population[k]
```

```
      # print("best tree's fitbess..",self.besttree.fitness)
    # self.besttree.display()
    # self.besttree.drawtree()
    return self.besttree.fitness


  def roulettewheelsel(self, reverse=False):
    if reverse == False:
      allfitness = 0
      for i in range(0, self.size):
        allfitness += self.population[i].fitness
      randomnum = random()*(self.size - 1)
      check = 0
      for i in range(0, self.size):
        # print(self.population[i].fitness)
        check += (1.0 - self.population[i].fitness / allfitness)
        # print('---')
        # print(check)
        if check >= randomnum:
          return self.population[i], i
    if reverse == True:
      allfitness = 0
      for i in range(0, self.size):
        allfitness += self.population[i].fitness
      randomnum = random()
      check = 0
      for i in range(0, self.size):
        check += self.population[i].fitness * 1.0 / allfitness
        if check >= randomnum:
          return self.population[i], i


############################################################

def add(ValuesList,x):
    sumtotal = 0
    for val in ValuesList:
      sumtotal = sumtotal + val
    return sumtotal

def sub(ValuesList,x):
    return ValuesList[0] - ValuesList[1]

def mul(ValuesList,x):
    return ValuesList[0] * ValuesList[1]

def div(ValuesList,x):
    if ValuesList[1] == 0:
        return 1
    return ValuesList[0] / ValuesList[1]

def pow(ValuesList,x):
    if ValuesList[0]==0 and ValuesList[1]==0:
        return 0
```

```
    else:
        return float(ValuesList[0] ** ValuesList[1])

def sqrt(ValuesList,x):
    if ValuesList[0]<0:
        return 0
    else:
        return np.sqrt(float(ValuesList[0]))
def log(ValuesList,x):
    if ValuesList[0] <=0:
        return 0
    else:
        return np.log2(float(ValuesList[0]))
def exp(ValuesList,x):
    return round(np.exp(float(ValuesList[0])),2)

def maximum(ValuesList,x):
    return max(ValuesList)

def ifleq(ValuesList,x):
    if ValuesList[0]<=ValuesList[1]:
        return ValuesList[2]
    else:
        return ValuesList[3]


def data(ValuesList,x):
  index_x=np.mod(round(float(ValuesList[0])),n)
  return x[int(index_x)]

def diff(ValuesList,x):
  k = np.mod(round(float(ValuesList[0])), n)
  l = np.mod(round(float(ValuesList[1])), n)
  result = np.subtract(x[int(k)], x[int(l)])
  return result

def avg(ValuesList,x):
  sum_x = 0
  k = np.mod(round(float(ValuesList[0])), n)
  l = np.mod(round(float(ValuesList[1])), n)
  if k == l:
    return 0
  else:
    t = np.minimum(k, l)
    upper = np.maximum(k, l) - 1
    for z in range(int(t), int(upper) + 1):
      sum_x = sum_x + x[z]
    result = np.divide(sum_x, (k - l))
    return result



addwrapper = funwrapper(add, 2, "add")
subwrapper = funwrapper(sub, 2, "sub")
```

```
mulwrapper = funwrapper(mul, 2, "mul")
divwrapper = funwrapper(div, 2, "div")

powwrapper = funwrapper(pow, 2, "pow")
sqrtwrapper = funwrapper(sqrt, 1, "sqrt")
logwrapper = funwrapper(log, 1, "log")
expwrapper = funwrapper(exp, 1, "exp")
maximumwrapper = funwrapper(maximum, 2, "maximum")
ifleqwrapper = funwrapper(ifleq, 4, "ifleq")

datawrapper = funwrapper(data, 1, "data")
diffwrapper = funwrapper(diff, 2, "diff")
avgwrapper = funwrapper(avg, 2, "avg")

def constructcheckdata(filename,m,n):
  with open(filename,'r') as f:
    checkdata=list()
    for i in f:
      dic=dict()
      train_data = list(map(float,
i.strip('\n').strip('\t').split('\t')))
      x = train_data[:n]
      result= train_data[n]
      dic['x'] = x
      dic['result'] = result
      checkdata.append(dic)
    return checkdata


  pass

# def examplefun(x, y):
#   return x * x + x + 2 * y + 1
# def constructcheckdata(count=10):
#   checkdata = []
#   for i in range(0, count):
#     dic = {}
#     x = randint(0, 10)
#     y = randint(0, 10)
#     dic['x'] = x
#     dic['y'] = y
#     dic['result'] = examplefun(x, y)
#     checkdata.append(dic)
#   return checkdata

if __name__ == "__main__":
  filename='data.txt'
  m=10
  n=10
  checkdata = constructcheckdata(filename,m,n)
  # print(checkdata)
  dic_nsat=dict()
  crossrate_list = [ 0,0.2,0.4,0.6,0.8,0.9,1]
  for crossrate in crossrate_list:
```

```
      list_nsat=list()
      for i in range(80):
          env = enviroment(crossrate,[addwrapper, subwrapper,
mulwrapper,divwrapper,
          sqrtwrapper,logwrapper,maximumwrapper,

ifleqwrapper,datawrapper,diffwrapper,avgwrapper],
                             ["x", "y"],
                            list(range(-100,100)), checkdata)
          best_fitness=env.envolve(crossrate)
          list_nsat.append(best_fitness)
          print(best_fitness)
      # print('---')
      dic_nsat[crossrate]=list_nsat
  data = pd.DataFrame(dic_nsat)
  data.boxplot()
  plt.ylabel('Fitness of the Fittest Solution')
  plt.xlabel('Crossover Rate')
  plt.savefig('figure5.jpg')
```