



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

<i>Chapter 1 Exercise4 Pseudo-code of My Algorithm</i>	<i>1</i>
<i>Chapter 2 Exercise5-Fittest Solution VS Mutation Rate.....</i>	<i>6</i>
<i>Chapter 3 Exercise5-Fittest Solution VS Max Depth of Tree</i>	<i>10</i>
<i>Chapter 4 Exercise5-Fittest Solution VS Population Size</i>	<i>14</i>
<i>Chapter 5 Exercise5-Fittest Solution VS Crossover Rate.....</i>	<i>17</i>
<i>Appendix A Fittest Solution vs Mutation Rate</i>	<i>21</i>
<i>Appendix B Fittest Solution vs Max Depth of Tree</i>	<i>29</i>
<i>Appendix C Fittest Solution vs Population Size</i>	<i>38</i>
<i>Appendix D Fittest Solution vs Crossover Rate</i>	<i>47</i>

Chapter 1

Exercise 4 Pseudo-code of My Algorithm

Algorithm Time Series Prediction

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

```
/*The necessary functions and class of the algorithm*/
1: define a function class
2:   initialize attributes of the function class
3:   the function of the function
4:   the child number of the function
5:   the name of the function
6: define a constant class
7:   initialize attributes of the constant class
8:   the value of the constant
9:   the name of the constant
10:  the type of the constant is constant
11:  define evaluation method to evaluate the constant
12:  Return the value of the constant
13: define add function (input value_list)
14:   sum=0
15:   for i=0 to (the length of value_list)-1 do
16:     sum=sum+value_list[i]
17:   end for
18:   Return sum
19: define subtract function (input value_list)
20:   Return value_list[0]- value_list[1]
21: define multiply function (input value_list)
22:   Return value_list[0]*value_list[1]
23: define divide function (input value_list)
24:   if (value_list[1]=0) do
25:     Return 0
26:   else do
27:     Return value_list[0] / value_list[1]
28:   end if
29: define power function (input value_list)
30:   if (value_list[0]=0 & value_list[1]<=0) do
31:     Return 0
32:   else if (value_list[0] value_list[1] is not a real number) do
33:     Return 0
34:   else do
35:     Return value_list[0] value_list[1]
36:   end if
37: define sqrt function (input value_list)
38:   if (value_list[0]<0) do
39:     Return 0
40:   else do
```

```

41:     Return  $\sqrt{value\_list[0]}$ 
42: end if
43: define log function (input value_list)
44: if (value_list[0] ≤ 0) do
45:     Return 0
46: else do
47:     Return  $\log_2 value\_list[0]$ 
48: end if
49: define exp function (input value_list)
50: Return  $e^{value\_list[0]}$ 
51: define ifleq function (input value_list)
52: if (value_list[0] ≤ value_list[1]) do
53:     Return value_list[2]
54: else do
55:     Return value_list[3]
56: end if
57: define data function (input value_list, X)
58:  $j \equiv \lfloor value\_list[0] \rfloor \bmod n$ 
59: Return  $X_j$ 
60: define diff function (input value_list, X)
61:  $k = \lfloor value\_list[0] \rfloor \bmod n$ 
62:  $l = \lfloor value\_list[1] \rfloor \bmod n$ 
63: Return  $X_l$ 
64: define avg function (input value_list, X)
65: if (value_list[0] = value_list[1]) do
66:     Return 0
67: else do
68:      $k = \lfloor value\_list[0] \rfloor \bmod n$ 
69:      $l = \lfloor value\_list[1] \rfloor \bmod n$ 
70:      $t = \min(k, l)$ 
71:     upper =  $\max(k, l) - 1$ 
72:     sum = 0
73:     for i = t to upper do
74:         sum = sum +  $X_i$ 
75:     end for
76:     Return sum/abs(k-l)
77: end if
78: define max function (input value_list)
79: Return max(value_list)
80: Create add function object(the function is add, the child number is 2, the function name is "add")
81: Create sub function object(the function is sub, the child number is 2, the function name is "sub")
82: Create mul function object(the function is mul, the child number is 2, the function name is "mul")
83: Create div function object(the function is div, the child number is 2, the function name is "div")
84: Create pow function object(the function is pow, the child number is 2, the function name is "pow")
85: Create sqrt function object(the function is sqrt, the child number is 1, the function name is "sqrt")
86: Create log function object(the function is log, the child number is 1, the function name is "log")
87: Create exp function object(the function is exp, the child number is 1, the function name is "exp")
88: Create max function object(the function is max, the child number is 2, the function name is "max")
89: Create ifleq function object(the function is ifleq, the child number is 4, the function name is "ifleq")
90: Create data function object(the function is data, the child number is 1, the function name is "data")
91: Create diff function object(the function is diff, the child number is 2, the function name is "diff")
92: Create avg function object(the function is avg, the child number is 2, the function name is "avg")
93: function_list = [ add, sub, mul, div, pow, sqrt, log, exp, max, ifleq, data, diff, avg]
94: constant_list = list(range(-100, 100)) /* randomly generate constant from -100 to 100 */
95: define a node class
96:     initialize attributes of the node class
97:         the type of the node
98:         the children of the node
99:         the name of the node
100:        the depth of the node
101:        the value of the node
102:        the fitness of the node
103: define evaluation method to calculate the value of the node

```

```

104:     if (the type of the node is constant
105:         Return the value of the constant node
106:     else do
107:         for i=0 to (the length of children_list)-1 do
108:             value_list = [ children_list[i].evaluation_method for i=0 to (the length of children_list)-1 ] /*recursion*/
109:             Return node.function(value_list,X)
110:         end for
111:     end if
112: define display method to parse the tree to an expression
113:     if ( the type of the node is function) do
114:         expression=expression+ ' ('+node.function.name
115:     else if ( the type of the node is constant) do
116:         expression= ' '+node.constant.name
117:     end if
118:     if (the node has children) do
119:         for i=0 to (the length of children_list)-1 do
120:             expression=expression+ children_list[i].display_method /*recursion*/
121:         end for
122:         expression= expression + ')'
123:     end if
124:     Return expression
125: define get_fitness method to calculate the fitness of a tree(input data)
126:     sum=0
127:     for i=0 to (the length of data)-1 do
128:         sum=sum+(tree.evaluate_method(data ['X'])- data ['Y'])**2
129:     end for
130:     fitness=sum/m
131:     Return fitness
132: define an environment class
133:     initialize attributes of the environment class
134:         the function_list of the environment class
135:         the constant_list of the environment class
136:         the training data of the environment class
137:         the population size of the environment class
138:         the population of the environment=create_population_method(input λ)
139:         the max depth of the tree
140: define make tree method(input start_depth)
141:     if start_depth=0 do
142:         node_pattern=0
143:     else if start_depth=max_depth-1 do
144:         node_pattern=1
145:     else do
146:         node_pattern=random(0,1)
147:     end if
148:     if node_pattern=0 do
149:         randomly select a function from function_list
150:         for i=0 to (the number of function's children)-1 do
151:             child=make_tree_method(start_depth+1) /*recursion*/
152:             add child to children_list
153:         end for
154:         Return node("function", children_list, function)
155:     else do
156:         randomly select a constant from constant_list
157:         Return node("constant", constant_class(constant))
158:     end if
159: define create population method (input λ)
160:     Return [make_tree_method(input start_depth=0) for i=0 to λ -1 do ]
161: define mutation method(input tree, mutation_rate=0.1,startdepth=0)
162:     if (randomly generate a probability)<=mutation_rate do
163:         return tree.make_tree_method(input start_depth=0) /*mutate the root of tree*/
164:     else do
165:         new_tree=deepcopy(tree)
166:         if the node type of the tree is function do

```

```

167:         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*/
168:     end if
169: end if
170: Return new_tree
171: define crossover method (input tree1, tree2,crossover_rate=0.9 , top=1)
172:     if (randomly generate a probability)<=mutation_rate and not top do
173:         Return deepcopy(tree2)
174:     else do
175:         new_tree=deepcopy(tree1)
176:         if the type of tree1's node is function and the type of the tree2's node is function do
177:             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*/
178:         end if
179:     end if
180: Return new_tree
181: define roulette selection method
182:     all_fitness=0
183:     for i=0 to  $\lambda$  -1 do
184:         all_fitness=population[i].getfitness(input checkdata)
185:     end for
186:     random_probability=random(0,1)*(  $\lambda$ -1 )
187:     add_probability=0
188:     for i=0 to  $\lambda$  -1 do
189:         add_probability=add_probability+(1.0- population[i].getfitness(input checkdata)/ all_fitness)
190:         if add_probability $\geq$  random_probability do
191:             Return population[i]
192:         end if
193:     end for
194: define evolve method(input time_budget,min_or_max)
195:     get start_time
196:     while True:
197:         create a offspring_list to store offspring
198:         get end_time
199:         if time_budget<( end_time- start_time) do
200:             end loop while
201:         end if
202:         for i=0 to ( $\lambda$  -1) do
203:             father= roulette selection method(input reverse=False)
204:             mother= roulette selection method(input reverse=False)
205:             offspring1=crossover(father,mother)
206:             add offspring1 in offspring_list
207:             parent= roulette selection method(input reverse=False)
208:             offspring2= mutation method(parent)
209:             add offspring2 in offspring_list
210:         end for
211:         population= offspring_list /*use offspring_list to replace previous population*/
212:         best_tree=population[0]
213:         for j=0 to ( $\lambda$  -1) do
214:             population[j].getfitness(input checkdata) /update the fitness of individual
215:             if best_tree.fitness> population[j].fitness
216:                 best_tree= population[j]
217:             end if
218:         end for
219:     end while
220:     Return best_tree.display
221: define read data file method(input filename,m,n)
222:     f=open( ' filename ', ' read)
223:     create a data_list to store taining data
224:     for i=1 to length(f) do
225:         create a dictionary to store each sample of training data
226:         x=f[i][:n]
227:         y= f[i][n]

```

```

228:     dictionary['x'] =x
229:     dictionary['y'] =y
230:     add dictionary to data_list
231: end for
232: Return data_list

/* main function to start the program */
233: training_data= read data file method(input filename,m,n)
234: environment_object=environment (input  $\lambda$ , function_list, constant_list, data_list )
235: best_expression= environment_object.envolve(input time_budget)
236: 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).

	85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
1	85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
2	57.64	36.87	89.02	51.57	87.95	75.79	95.99	10.37	87.18	52.43	80.94
3	81.02	30.11	84.77	28.98	50.04	79.24	83.85	49.90	22.78	18.45	91.94
4	17.54	57.29	83.04	28.05	5.64	99.18	15.34	82.69	63.60	58.79	22.96
5	10.04	22.12	6.41	90.44	44.77	85.23	53.96	29.43	39.59	16.05	75.29
6	6.06	28.41	23.15	64.00	82.22	3.99	84.70	23.35	8.87	53.54	3.72
7	48.92	96.78	3.69	50.95	11.75	26.03	37.92	74.25	49.05	96.16	55.13
8	50.91	14.77	90.91	55.71	73.26	46.55	30.85	77.56	39.49	26.55	77.81
9	25.31	12.84	54.40	58.50	10.64	21.25	57.70	35.14	40.19	43.12	9.04
10	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** \in (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** \in 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

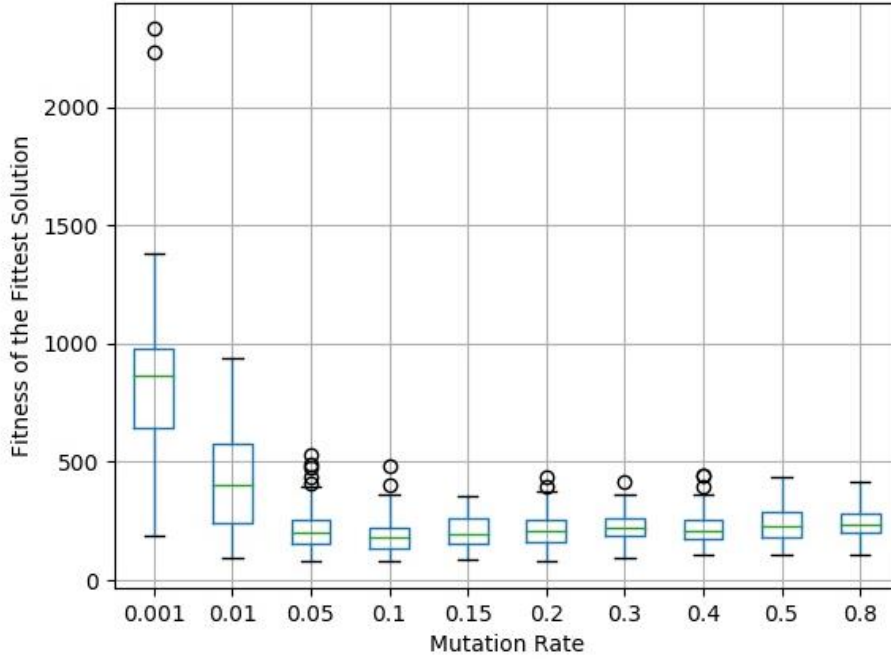


figure 2: Fittest Solution VS Mutation Rate

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).

	85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
1	85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
2	57.64	36.87	89.02	51.57	87.95	75.79	95.99	10.37	87.18	52.43	80.94
3	81.02	30.11	84.77	28.98	50.04	79.24	83.85	49.90	22.78	18.45	91.94
4	17.54	57.29	83.04	28.05	5.64	99.18	15.34	82.69	63.60	58.79	22.96
5	10.04	22.12	6.41	90.44	44.77	85.23	53.96	29.43	39.59	16.05	75.29
6	6.06	28.41	23.15	64.00	82.22	3.99	84.70	23.35	8.87	53.54	3.72
7	48.92	96.78	3.69	50.95	11.75	26.03	37.92	74.25	49.05	96.16	55.13
8	50.91	14.77	90.91	55.71	73.26	46.55	30.85	77.56	39.49	26.55	77.81
9	25.31	12.84	54.40	58.50	10.64	21.25	57.70	35.14	40.19	43.12	9.04
10	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 ($maxDepth$) $\in \text{Set } \{2, 4, 6, 8, 10, 20, 30, 40, 50, 60, 70, 80, 90\}$**).

3.2.1 Boxplot of Results

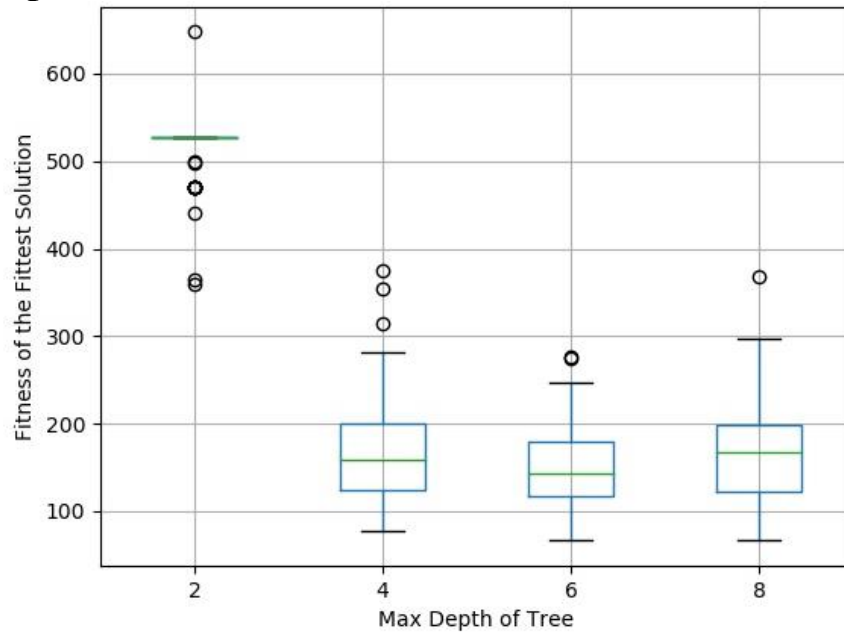


figure 4: Fittest Solution VS Max Depth of Tree

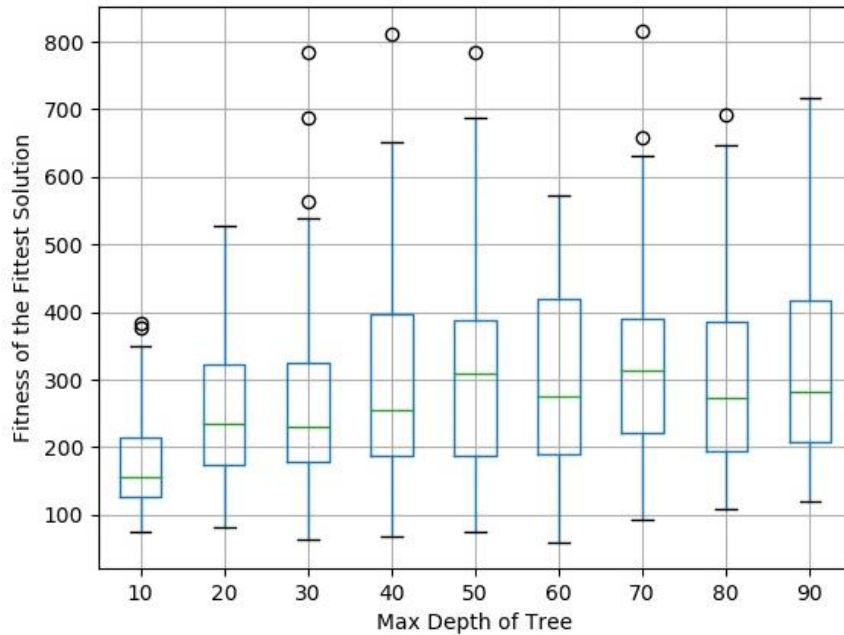


figure 5: Fittest Solution VS Max Depth of Tree

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***).

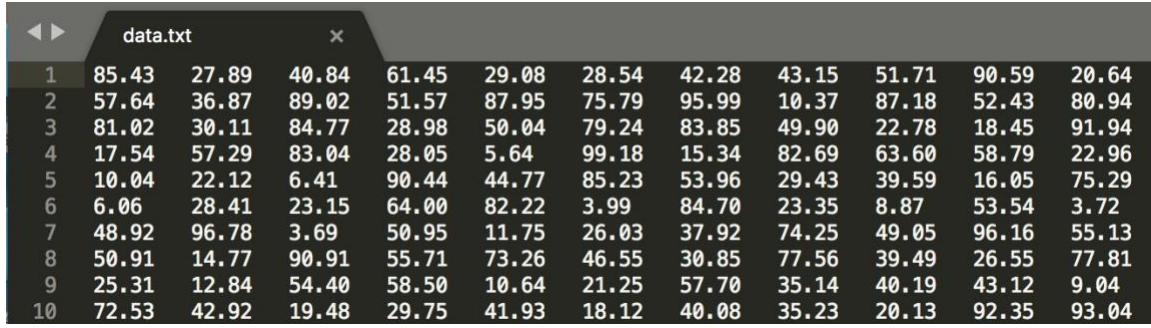
4.1.1.4 Crossover Rate

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).



	85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
1	85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
2	57.64	36.87	89.02	51.57	87.95	75.79	95.99	10.37	87.18	52.43	80.94
3	81.02	30.11	84.77	28.98	50.04	79.24	83.85	49.90	22.78	18.45	91.94
4	17.54	57.29	83.04	28.05	5.64	99.18	15.34	82.69	63.60	58.79	22.96
5	10.04	22.12	6.41	90.44	44.77	85.23	53.96	29.43	39.59	16.05	75.29
6	6.06	28.41	23.15	64.00	82.22	3.99	84.70	23.35	8.87	53.54	3.72
7	48.92	96.78	3.69	50.95	11.75	26.03	37.92	74.25	49.05	96.16	55.13
8	50.91	14.77	90.91	55.71	73.26	46.55	30.85	77.56	39.49	26.55	77.81
9	25.31	12.84	54.40	58.50	10.64	21.25	57.70	35.14	40.19	43.12	9.04
10	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

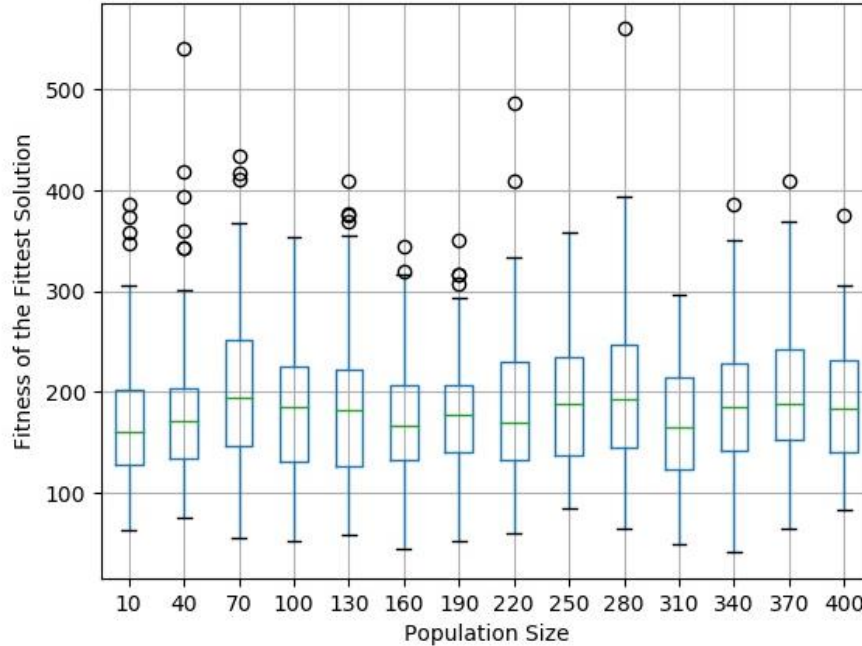


figure 7: Fittest Solution VS Population Size

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***).

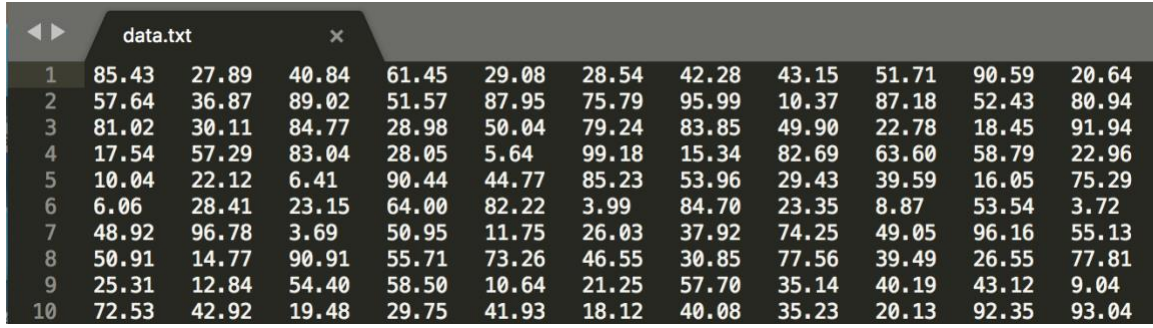
5.1.1.4 Time Budget

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)**.



	85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
1	85.43	27.89	40.84	61.45	29.08	28.54	42.28	43.15	51.71	90.59	20.64
2	57.64	36.87	89.02	51.57	87.95	75.79	95.99	10.37	87.18	52.43	80.94
3	81.02	30.11	84.77	28.98	50.04	79.24	83.85	49.90	22.78	18.45	91.94
4	17.54	57.29	83.04	28.05	5.64	99.18	15.34	82.69	63.60	58.79	22.96
5	10.04	22.12	6.41	90.44	44.77	85.23	53.96	29.43	39.59	16.05	75.29
6	6.06	28.41	23.15	64.00	82.22	3.99	84.70	23.35	8.87	53.54	3.72
7	48.92	96.78	3.69	50.95	11.75	26.03	37.92	74.25	49.05	96.16	55.13
8	50.91	14.77	90.91	55.71	73.26	46.55	30.85	77.56	39.49	26.55	77.81
9	25.31	12.84	54.40	58.50	10.64	21.25	57.70	35.14	40.19	43.12	9.04
10	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 $\in [0, 1]$**).

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 $\in \text{Set } \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$**).

5.2.1 Boxplot of Results

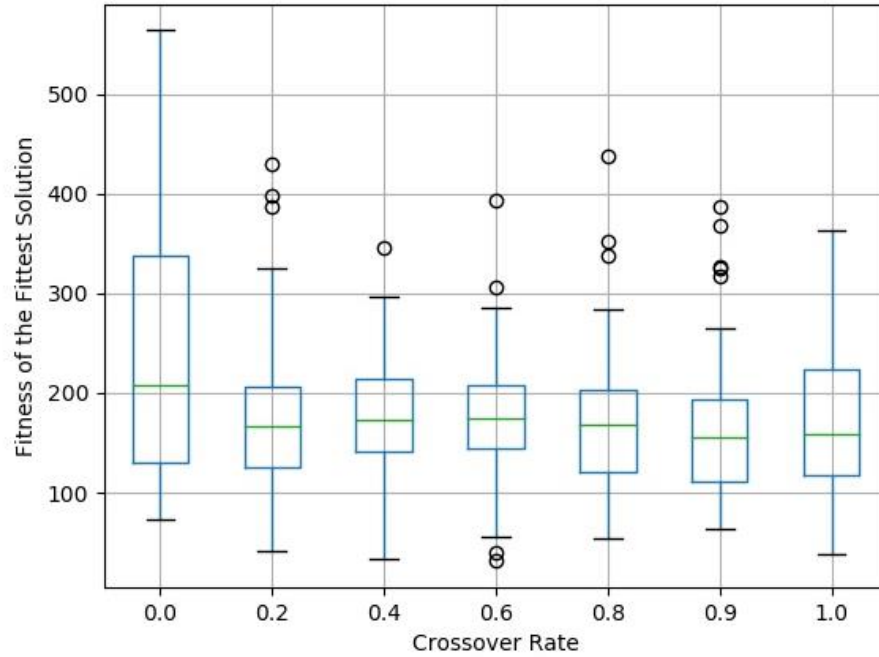


figure 5: figure 9: Fittest Solution VS Crossover Rate

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

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # Author: Janet Chou
4  #!/usr/bin/env python
5  # -*- coding: utf-8 -*-
6  # Author: Janet Chou
7  from random import random, randint, choice
8  from copy import deepcopy
9  from PIL import Image, ImageDraw
10 import numpy as np
11 import pandas as pd
12 import matplotlib.pyplot as plt
13 import time
14
15 class funwrapper:
16     def __init__(self, function, childcount, name):
17         self.function = function
18         self.childcount = childcount
19         self.name = name
20
21 class variable:
22     def __init__(self, var, value=0):
23         self.var = var
24         self.value = value
25         self.name = str(var)
26         self.type = "variable"
27
28     def evaluate(self):
29         return self.var.value
30
31     def setvar(self, value):
32         self.value = value
33
34     def display(self, indent=0):
35         print('%s%s' % (' '*indent, self.var))
36
37 class const:
38     def __init__(self, value):
39         self.value = value
40         self.name = str(value)
41         self.type = "constant"
42
43     def evaluate(self):
44         return self.value
45
46     def display(self, indent=0):
47         print('%s%d' % (' '*indent, self.value))
48
```

```

49 class node:
50     def __init__(self, type, children, funwrap, var=None,
51 const=None):
52         self.type = type
53         self.children = children
54         self.funwrap = funwrap
55         self.variable = var
56         self.const = const
57         self.depth = self.refreshdepth()
58         self.value = 0
59         self.fitness = 0
60
61     def eval(self,x):
62         if self.type == "variable":
63             return self.variable.value
64         elif self.type == "constant":
65             return self.const.value
66         else:
67             for c in self.children:
68                 result = [c.eval(x) for c in self.children]
69                 return self.funwrap.function(result,x)
70
71     def getfitness(self, checkdata):#checkdata like
72 {"x":1,"result":3}
73     diff = 0
74     #set variable value
75     for data in checkdata:
76         self.setvariablevalue(data)
77         diff += (self.eval(data['x']) - data["result"])**2
78     self.fitness = diff/len(checkdata)
79
80     def setvariablevalue(self, value):
81         if self.type == "variable":
82             if self.variable.var in value:
83                 self.variable.setvar(value[self.variable.var])
84             else:
85                 print("There is no value for variable:",
86 self.variable.var)
87             return
88         if self.type == "constant":
89             pass
90         if self.children:#function node
91             for child in self.children:
92                 child.setvariablevalue(value)
93
94     def refreshdepth(self):
95         if self.type == "constant" or self.type == "variable":
96             return 0
97         else:
98             depth = []
99             for c in self.children:
100                 depth.append(c.refreshdepth())
101             return max(depth) + 1
102

```



```

103
104 def display(self, indent=0):
105     if self.type == "function":
106         print ((' '*indent) + self.funwrap.name)
107     elif self.type == "variable":
108         print ((' '*indent) + self.variable.name)
109     elif self.type == "constant":
110         print ((' '*indent) + self.const.name)
111     if self.children:
112         for c in self.children:
113             c.display(indent + 1)
114     ##for draw node
115 def getwidth(self):
116     if self.type == "variable" or self.type == "constant":
117         return 1
118     else:
119         result = 0
120         for i in range(0, len(self.children)):
121             result += self.children[i].getwidth()
122         return result
123 def drawnode(self, draw, x, y):
124     if self.type == "function":
125         allwidth = 0
126         for c in self.children:
127             allwidth += c.getwidth()*100
128         left = x - allwidth / 2
129         #draw the function name
130         draw.text((x - 10, y - 10), self.funwrap.name, (0, 0, 0))
131         #draw the children
132         for c in self.children:
133             wide = c.getwidth()*100
134             draw.line((x, y, left + wide / 2, y + 100), fill=(255, 0,
135 0))
136             c.drawnode(draw, left + wide / 2, y + 100)
137             left = left + wide
138         elif self.type == "variable":
139             draw.text((x - 5, y), self.variable.name, (0, 0, 0))
140         elif self.type == "constant":
141             draw.text((x - 5, y), self.const.name, (0, 0, 0))
142
143 def drawtree(self, jpeg="tree.png"):
144     w = self.getwidth()*100
145     h = self.depth * 100 + 120
146
147     img = Image.new('RGB', (w, h), (255, 255, 255))
148     draw = ImageDraw.Draw(img)
149     self.drawnode(draw, w / 2, 20)
150     img.save(jpeg, 'PNG')
151
152 class enviroment:
153     def __init__(self, mutationrate, funwraplist, variablelist,
154 constantlist,
155     checkdata,
156

```

```

157         minimaxtype="min", population=None,
158 size=10,maxdepth=10,
159         maxgen=500, crossrate=0.9, newbirthrate=1):
160     self.funwraplist = funwraplist
161     self.variablelist = variablelist
162     self.constantlist = constantlist
163     self.checkdata = checkdata
164     self.minimaxtype = minimaxtype
165     self.maxdepth = maxdepth
166     self.population = population or self._makepopulation(size)
167     self.size = size
168     self.maxgen = maxgen
169     self.crossrate = crossrate
170     self.mutationrate = mutationrate
171     self.newbirthrate = newbirthrate
172
173     self.besttree = self.population[0]
174     for i in range(0, self.size):
175         self.population[i].depth=self.population[i].refreshdepth()
176         self.population[i].getfitness(checkdata)
177         if self.minimaxtype == "min":
178             if self.population[i].fitness < self.besttree.fitness:
179                 self.besttree = self.population[i]
180         elif self.minimaxtype == "max":
181             if self.population[i].fitness > self.besttree.fitness:
182                 self.besttree = self.population[i]
183
184     def makepopulation(self, popsize):
185         return [self.maketree(0) for i in range(0, popsize)]
186
187     def maketree(self, startdepth):
188         if startdepth == 0:
189             #make a new tree
190             nodepattern = 0#function
191         elif startdepth == self.maxdepth:
192             nodepattern = 1#variable or constant
193         else:
194             nodepattern = randint(0, 1)
195         if nodepattern == 0:
196             childlist = []
197             selectedfun = randint(0, len(self.funwraplist) - 1)
198             for i in range(0, self.funwraplist[selectedfun].childcount):
199                 child = self.maketree(startdepth + 1)
200                 childlist.append(child)
201             return node("function", childlist,
202 self.funwraplist[selectedfun])
203         else:
204             selectedconstant = randint(0, len(self.constantlist) - 1)
205             return node("constant", None, None, None,
206                 const(self.constantlist[selectedconstant]))
207
208     def mutate(self, tree, probchange, startdepth=0):
209         if random() < probchange:
210             return self._maketree(startdepth)

```

```

211     else:
212         result = deepcopy(tree)
213         if result.type == "function":
214             result.children = [self.mutate(c, probchange, startdepth +
215 1) \
216                               for c in tree.children]
217     return result
218
219 def crossover(self, tree1, tree2, probswap=1, top=1):
220     if random() < probswap and not top:
221         return deepcopy(tree2)
222     else:
223         result = deepcopy(tree1)
224         if tree1.type == "function" and tree2.type == "function":
225             result.children = [self.crossover(c,
226 choice(tree2.children),
227                               probswap, 0) for c in tree1.children]
228     return result
229
230 def evolve(self, mutationrate, maxgen=1000, crossrate=0.9):
231     timebudget=50
232     start=time.clock()
233     while True:
234         # print("generation no.", i)
235         child = []
236         end=time.clock()
237         if timebudget<(end-start):
238             # print(end-start)
239             break
240         for j in range(0, int(self.size * self.newbirthrate / 2)):
241             parent1, p1 = self.roulettewheelssel()
242             parent2, p2 = self.roulettewheelssel()
243             newchild = self.crossover(parent1, parent2)
244             child.append(newchild)#generate new tree
245             parent, p3 = self.roulettewheelssel()
246             newchild = self.mutate(parent, mutationrate)
247             child.append(newchild)
248             #refresh all tree's fitness
249             for j in range(0, int(self.size * self.newbirthrate)):
250                 replacedtree, replacedindex =
251 self.roulettewheelssel(reverse=True)
252                 #replace bad tree with child
253                 self.population[replacedindex] = child[j]
254
255             for k in range(0, self.size):
256                 self.population[k].getfitness(self.checkdata)
257                 self.population[k].depth=self.population[k].refreshdepth()
258                 if self.minimaxtype == "min":
259                     if self.population[k].fitness < self.besttree.fitness:
260                         self.besttree = self.population[k]
261                 elif self.minimaxtype == "max":
262                     if self.population[k].fitness > self.besttree.fitness:
263                         self.besttree = self.population[k]
264             return self.besttree.fitness

```

```

265
266
267 def roulettewheelself(self, reverse=False):
268     if reverse == False:
269         allfitness = 0
270         for i in range(0, self.size):
271             allfitness += self.population[i].fitness
272         randomnum = random()*(self.size - 1)
273         check = 0
274         for i in range(0, self.size):
275             # print(self.population[i].fitness)
276             check += (1.0 - self.population[i].fitness / allfitness)
277             # print('---')
278             # print(check)
279             if check >= randomnum:
280                 return self.population[i], i
281     if reverse == True:
282         allfitness = 0
283         for i in range(0, self.size):
284             allfitness += self.population[i].fitness
285         randomnum = random()
286         check = 0
287         for i in range(0, self.size):
288             check += self.population[i].fitness * 1.0 / allfitness
289             if check >= randomnum:
290                 return self.population[i], i
291     #####
292
293 def add(ValuesList,x):
294     sumtotal = 0
295     for val in ValuesList:
296         sumtotal = sumtotal + val
297     return sumtotal
298
299 def sub(ValuesList,x):
300     return ValuesList[0] - ValuesList[1]
301
302 def mul(ValuesList,x):
303     return ValuesList[0] * ValuesList[1]
304
305 def div(ValuesList,x):
306     if ValuesList[1] == 0:
307         return 1
308     return ValuesList[0] / ValuesList[1]
309
310 def pow(ValuesList,x):
311     if ValuesList[0]==0 and ValuesList[1]==0:
312         return 0
313     else:
314         return float(ValuesList[0] ** ValuesList[1])
315
316 def sqrt(ValuesList,x):
317     if ValuesList[0]<0:
318         return 0

```

```

319     else:
320         return np.sqrt(float(ValuesList[0]))
321 def log(ValuesList,x):
322     if ValuesList[0] <=0:
323         return 0
324     else:
325         return np.log2(float(ValuesList[0]))
326 def exp(ValuesList,x):
327     return round(np.exp(float(ValuesList[0])),2)
328
329 def maximum(ValuesList,x):
330     return max(ValuesList)
331
332 def ifleq(ValuesList,x):
333     if ValuesList[0]<=ValuesList[1]:
334         return ValuesList[2]
335     else:
336         return ValuesList[3]
337
338 def data(ValuesList,x):
339     index x=np.mod(round(float(ValuesList[0])),n)
340     return x[int(index x)]
341
342 def diff(ValuesList,x):
343     k = np.mod(round(float(ValuesList[0])), n)
344     l = np.mod(round(float(ValuesList[1])), n)
345     result = np.subtract(x[int(k)], x[int(l)])
346     return result
347
348 def avg(ValuesList,x):
349     sum x = 0
350     k = np.mod(round(float(ValuesList[0])), n)
351     l = np.mod(round(float(ValuesList[1])), n)
352     if k == l:
353         return 0
354     else:
355         t = np.minimum(k, l)
356         upper = np.maximum(k, l) - 1
357         for z in range(int(t), int(upper) + 1):
358             sum x = sum x + x[z]
359         result = np.divide(sum x, (k - l))
360         return result
361
362 addwrapper = funwrapper(add, 2, "add")
363 subwrapper = funwrapper(sub, 2, "sub")
364 mulwrapper = funwrapper(mul, 2, "mul")
365 divwrapper = funwrapper(div, 2, "div")
366
367 powwrapper = funwrapper(pow, 2, "pow")
368 sqrtwrapper = funwrapper(sqrt, 1, "sqrt")
369 logwrapper = funwrapper(log, 1, "log")
370 expwrapper = funwrapper(exp, 1, "exp")
371 maximumwrapper = funwrapper(maximum, 2, "maximum")
372 ifleqwrapper = funwrapper(ifleq, 4, "ifleq")

```

```

373
374 datawrapper = funwrapper(data, 1, "data")
375 diffwrapper = funwrapper(diff, 2, "diff")
376 avgwrapper = funwrapper(avg, 2, "avg")
377
378 def constructcheckdata(filename,m,n):
379     with open(filename,'r') as f:
380         checkdata=list()
381         for i in f:
382             dic=dict()
383             train data = list(map(float,
384 i.strip('\n').strip('\t').split('\t')))
385             x = train data[:n]
386             result= train data[n]
387             dic['x'] = x
388             dic['result'] = result
389             checkdata.append(dic)
390         return checkdata
391     pass
392
393 if name == " main ":
394     filename='data.txt'
395     m=10
396     n=10
397     checkdata = constructcheckdata(filename,m,n)
398     # print(checkdata)
399     dic nsat=dict()
400     mutation list = [ 0.001, 0.01, 0.05, 0.1, 0.15, 0.2, 0.3,
401 0.4,0.5,0.8]
402     for mutationrate in mutation list:
403         list nsat=list()
404         for i in range(80):
405             env = enviroment(mutationrate,[addwrapper, subwrapper,
406 mulwrapper,
407 divwrapper,sqrtwrapper,logwrapper,maximumwrapper,
408 ifleqwrapper,datawrapper,diffwrapper,avgwrapper],
409 ["x", "y"],
410 list(range(-100,100)), checkdata)
411             best fitness=env.envolve(mutationrate)
412             list nsat.append(best fitness)
413             print(best fitness)
414             # print('---')
415             dic_nsat[mutationrate]=list_nsat
416 data = pd.DataFrame(dic_nsat)
417 data.boxplot()
418 plt.ylabel('Fitness of the Fittest Solution')
419 plt.xlabel('Mutation Rate')
420 plt.savefig('figure4.jpg')

```

Appendix B

Fittest Solution VS Max Depth of Tree

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # Author: Janet Chou
4  from random import random, randint, choice
5  from copy import deepcopy
6  from PIL import Image, ImageDraw
7  import numpy as np
8  import pandas as pd
9  import matplotlib.pyplot as plt
10 import time
11
12 class funwrapper:
13     def __init__(self, function, childcount, name):
14         self.function = function
15         self.childcount = childcount
16         self.name = name
17
18 class variable:
19     def __init__(self, var, value=0):
20         self.var = var
21         self.value = value
22         self.name = str(var)
23         self.type = "variable"
24
25     def evaluate(self):
26         return self.var.value
27
28     def setvar(self, value):
29         self.value = value
30
31     def display(self, indent=0):
32         print('%s%s' % (' '*indent, self.var))
33
34 class const:
35     def __init__(self, value):
36         self.value = value
37         self.name = str(value)
38         self.type = "constant"
39
40     def evaluate(self):
41         return self.value
42
43     def display(self, indent=0):
44         print('%s%d' % (' '*indent, self.value))
45
46 class node:
47     def __init__(self, type, children, funwrap, var=None,
48 const=None):
49         self.type = type
```

```

50     self.children = children
51     self.funwrap = funwrap
52     self.variable = var
53     self.const = const
54     self.depth = self.refreshdepth()
55     self.value = 0
56     self.fitness = 0
57
58     def eval(self,x):
59         if self.type == "variable":
60             return self.variable.value
61         elif self.type == "constant":
62             return self.const.value
63         else:
64             for c in self.children:
65                 result = [c.eval(x) for c in self.children]
66                 return self.funwrap.function(result,x)
67
68     def getfitness(self, checkdata):#checkdata like
69     {"x":1,"result":3}
70     diff = 0
71     #set variable value
72     for data in checkdata:
73         self.setvariablevalue(data)
74         diff += (self.eval(data['x']) - data["result"])**2
75     self.fitness = diff/len(checkdata)
76
77     def setvariablevalue(self, value):
78         if self.type == "variable":
79             if self.variable.var in value:
80                 self.variable.setvar(value[self.variable.var])
81             else:
82                 print("There is no value for variable:",
83 self.variable.var)
84             return
85         if self.type == "constant":
86             pass
87         if self.children:#function node
88             for child in self.children:
89                 child.setvariablevalue(value)
90
91     def refreshdepth(self):
92         if self.type == "constant" or self.type == "variable":
93             return 0
94         else:
95             depth = []
96             for c in self.children:
97                 depth.append(c.refreshdepth())
98             return max(depth) + 1
99
100
101     def display(self, indent=0):
102         if self.type == "function":
103             print ((' '*indent) + self.funwrap.name)

```



```

104     elif self.type == "variable":
105         print ((' '*indent) + self.variable.name)
106     elif self.type == "constant":
107         print ((' '*indent) + self.const.name)
108     if self.children:
109         for c in self.children:
110             c.display(indent + 1)
111     ##for draw node
112     def getwidth(self):
113         if self.type == "variable" or self.type == "constant":
114             return 1
115         else:
116             result = 0
117             for i in range(0, len(self.children)):
118                 result += self.children[i].getwidth()
119             return result
120     def drawnode(self, draw, x, y):
121         if self.type == "function":
122             allwidth = 0
123             for c in self.children:
124                 allwidth += c.getwidth()*100
125             left = x - allwidth / 2
126             #draw the function name
127             draw.text((x - 10, y - 10), self.funwrap.name, (0, 0, 0))
128             #draw the children
129             for c in self.children:
130                 wide = c.getwidth()*100
131                 draw.line((x, y, left + wide / 2, y + 100), fill=(255, 0,
132 0))
133                 c.drawnode(draw, left + wide / 2, y + 100)
134                 left = left + wide
135             elif self.type == "variable":
136                 draw.text((x - 5, y), self.variable.name, (0, 0, 0))
137             elif self.type == "constant":
138                 draw.text((x - 5, y), self.const.name, (0, 0, 0))
139
140     def drawtree(self, jpeg="tree.png"):
141         w = self.getwidth()*100
142         h = self.depth * 100 + 120
143
144         img = Image.new('RGB', (w, h), (255, 255, 255))
145         draw = ImageDraw.Draw(img)
146         self.drawnode(draw, w / 2, 20)
147         img.save(jpeg, 'PNG')
148
149     class enviroment:
150         def __init__(self, maxdepth, funwraplist, variablelist,
151 constantlist, checkdata,
152             minimaxtype="min", population=None, size=10,
153             maxgen=500, crossrate=0.9, mutationrate=0.1,
154 newbirthrate=1):
155             self.funwraplist = funwraplist
156             self.variablelist = variablelist
157             self.constantlist = constantlist

```

```

158     self.checkdata = checkdata
159     self.minimaxtype = minimaxtype
160     self.maxdepth = maxdepth
161     self.population = population or self._makepopulation(size)
162     self.size = size
163     self.maxgen = maxgen
164     self.crossrate = crossrate
165     self.mutationrate = mutationrate
166     self.newbirthrate = newbirthrate
167
168     self.besttree = self.population[0]
169     for i in range(0, self.size):
170         self.population[i].depth=self.population[i].refreshdepth()
171         self.population[i].getfitness(checkdata)
172         if self.minimaxtype == "min":
173             if self.population[i].fitness < self.besttree.fitness:
174                 self.besttree = self.population[i]
175         elif self.minimaxtype == "max":
176             if self.population[i].fitness > self.besttree.fitness:
177                 self.besttree = self.population[i]
178
179     def makepopulation(self, popsize):
180         return [self.maketree(0) for i in range(0, popsize)]
181
182     def maketree(self, startdepth):
183         if startdepth == 0:
184             #make a new tree
185             nodepattern = 0#function
186         elif startdepth == self.maxdepth:
187             nodepattern = 1#variable or constant
188         else:
189             nodepattern = randint(0, 1)
190         if nodepattern == 0:
191             childlist = []
192             selectedfun = randint(0, len(self.funwraplist) - 1)
193             for i in range(0, self.funwraplist[selectedfun].childcount):
194                 child = self.maketree(startdepth + 1)
195                 childlist.append(child)
196             return node("function", childlist,
197 self.funwraplist[selectedfun])
198         else:
199             # if randint(0, 1) == 0:#variable
200             #     selectedvariable = randint(0, len(self.variablelist) - 1)
201             #     return node("variable", None, None,
202             #         variable(self.variablelist[selectedvariable]),
203             None)
204             # else:
205             selectedconstant = randint(0, len(self.constantlist) - 1)
206             return node("constant", None, None, None,
207                 const(self.constantlist[selectedconstant]))
208
209     def mutate(self, tree, probchange=0.1, startdepth=0):
210         if random() < probchange:
211             return self._maketree(startdepth)

```

```

212     else:
213         result = deepcopy(tree)
214         if result.type == "function":
215             result.children = [self.mutate(c, probchange, startdepth +
216 1) \
217                             for c in tree.children]
218     return result
219
220 def crossover(self, tree1, tree2, probswap=1, top=1):
221     if random() < probswap and not top:
222         return deepcopy(tree2)
223     else:
224         result = deepcopy(tree1)
225         if tree1.type == "function" and tree2.type == "function":
226             result.children = [self.crossover(c,
227 choice(tree2.children),
228                             probswap, 0) for c in tree1.children]
229     return result
230
231 def evolve(self, maxgen=1000, crossrate=0.9,
232 mutationrate=0.1):
233     timebudget=50
234     start=time.clock()
235     while True:
236         # print("generation no.", i)
237         child = []
238         end=time.clock()
239         if timebudget<(end-start):
240             print(end-start)
241             break
242         for j in range(0, int(self.size * self.newbirthrate / 2)):
243             parent1, p1 = self.roulettewheelsel()
244             parent2, p2 = self.roulettewheelsel()
245             newchild = self.crossover(parent1, parent2)
246             child.append(newchild)#generate new tree
247             parent, p3 = self.roulettewheelsel()
248             newchild = self.mutate(parent, mutationrate)
249             child.append(newchild)
250         #refresh all tree's fitness
251         for j in range(0, int(self.size * self.newbirthrate)):
252             replacedtree, replacedindex =
253 self.roulettewheelsel(reverse=True)
254             #replace bad tree with child
255             self.population[replacedindex] = child[j]
256
257         for k in range(0, self.size):
258             self.population[k].getfitness(self.checkdata)
259             self.population[k].depth=self.population[k].refreshdepth()
260             if self.minimaxtype == "min":
261                 if self.population[k].fitness < self.besttree.fitness:
262                     self.besttree = self.population[k]
263             elif self.minimaxtype == "max":
264                 if self.population[k].fitness > self.besttree.fitness:
265                     self.besttree = self.population[k]

```

```

266     # print("best tree's fitness..",self.besttree.fitness)
267     # self.besttree.display()
268     # self.besttree.drawtree()
269     return self.besttree.fitness
270
271
272     def roulettewheelself(self, reverse=False):
273         if reverse == False:
274             allfitness = 0
275             for i in range(0, self.size):
276                 allfitness += self.population[i].fitness
277             randomnum = random()*(self.size - 1)
278             check = 0
279             for i in range(0, self.size):
280                 # print(self.population[i].fitness)
281                 check += (1.0 - self.population[i].fitness / allfitness)
282                 # print('---')
283                 # print(check)
284                 if check >= randomnum:
285                     return self.population[i], i
286         if reverse == True:
287             allfitness = 0
288             for i in range(0, self.size):
289                 allfitness += self.population[i].fitness
290             randomnum = random()
291             check = 0
292             for i in range(0, self.size):
293                 check += self.population[i].fitness * 1.0 / allfitness
294                 if check >= randomnum:
295                     return self.population[i], i
296
297
298     #####
299
300     def add(ValuesList,x):
301         sumtotal = 0
302         for val in ValuesList:
303             sumtotal = sumtotal + val
304         return sumtotal
305
306     def sub(ValuesList,x):
307         return ValuesList[0] - ValuesList[1]
308
309     def mul(ValuesList,x):
310         return ValuesList[0] * ValuesList[1]
311
312     def div(ValuesList,x):
313         if ValuesList[1] == 0:
314             return 1
315         return ValuesList[0] / ValuesList[1]
316
317     def pow(ValuesList,x):
318         if ValuesList[0]==0 and ValuesList[1]==0:
319             return 0

```

```

320     else:
321         return float(ValuesList[0] ** ValuesList[1])
322
323 def sqrt(ValuesList,x):
324     if ValuesList[0]<0:
325         return 0
326     else:
327         return np.sqrt(float(ValuesList[0]))
328 def log(ValuesList,x):
329     if ValuesList[0] <=0:
330         return 0
331     else:
332         return np.log2(float(ValuesList[0]))
333 def exp(ValuesList,x):
334     return round(np.exp(float(ValuesList[0])),2)
335
336 def maximum(ValuesList,x):
337     return max(ValuesList)
338
339 def ifleq(ValuesList,x):
340     if ValuesList[0]<=ValuesList[1]:
341         return ValuesList[2]
342     else:
343         return ValuesList[3]
344
345
346 def data(ValuesList,x):
347     index x=np.mod(round(float(ValuesList[0])),n)
348     return x[int(index x)]
349
350 def diff(ValuesList,x):
351     k = np.mod(round(float(ValuesList[0])), n)
352     l = np.mod(round(float(ValuesList[1])), n)
353     result = np.subtract(x[int(k)], x[int(l)])
354     return result
355
356 def avg(ValuesList,x):
357     sum x = 0
358     k = np.mod(round(float(ValuesList[0])), n)
359     l = np.mod(round(float(ValuesList[1])), n)
360     if k == l:
361         return 0
362     else:
363         t = np.minimum(k, l)
364         upper = np.maximum(k, l) - 1
365         for z in range(int(t), int(upper) + 1):
366             sum_x = sum_x + x[z]
367         result = np.divide(sum_x, (k - l))
368         return result
369
370
371
372 addwrapper = funwrapper(add, 2, "add")
373 subwrapper = funwrapper(sub, 2, "sub")

```

```

374 mulwrapper = funwrapper(mul, 2, "mul")
375 divwrapper = funwrapper(div, 2, "div")
376
377 powwrapper = funwrapper(pow, 2, "pow")
378 sqrtwrapper = funwrapper(sqrt, 1, "sqrt")
379 logwrapper = funwrapper(log, 1, "log")
380 expwrapper = funwrapper(exp, 1, "exp")
381 maximumwrapper = funwrapper(maximum, 2, "maximum")
382 ifleqwrapper = funwrapper(ifleq, 4, "ifleq")
383
384 datawrapper = funwrapper(data, 1, "data")
385 diffwrapper = funwrapper(diff, 2, "diff")
386 avgwrapper = funwrapper(avg, 2, "avg")
387
388 def constructcheckdata(filename,m,n):
389     with open(filename,'r') as f:
390         checkdata=list()
391         for i in f:
392             dic=dict()
393             train data = list(map(float,
394 i.strip('\n').strip('\t').split('\t')))
395             x = train data[:n]
396             result= train data[n]
397             dic['x'] = x
398             dic['result'] = result
399             checkdata.append(dic)
400         return checkdata
401
402     pass
403
404 if name == " main ":
405     filename='data.txt'
406     m=10
407     n=10
408     checkdata = constructcheckdata(filename,m,n)
409     # print(checkdata)
410     dic nsat=dict()
411     for maxdepth in range(10,100,10):
412         list nsat=list()
413         for i in range(80):
414             env = enviroment(maxdepth,[addwrapper, subwrapper,
415 mulwrapper,divwrapper,
416 sqrtwrapper,logwrapper,maximumwrapper,
417 ifleqwrapper,datawrapper,diffwrapper,avgwrapper],
418 ["x", "y"],
419 list(range(-100,100)), checkdata)
420             best_fitness=env.envolve()
421             list nsat.append(best_fitness)
422             print(best_fitness)
423         # print('---')
424         dic_nsat[maxdepth]=list_nsat
425     data = pd.DataFrame(dic_nsat)
426     data.boxplot()

```

```
428 plt.ylabel('Fitness of the Fittest Solution')
429 plt.xlabel('Max Depth of Tree')
430 plt.savefig('figure3.jpg')
431
432
433
434
435
436
437
438
439
440
```

Appendix C

Fittest Solution VS Population Size

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # Author: Janet Chou
4  from random import random, randint, choice
5  from copy import deepcopy
6  from PIL import Image, ImageDraw
7  import numpy as np
8  import pandas as pd
9  import matplotlib.pyplot as plt
10 import time
11
12 class funwrapper:
13     def __init__(self, function, childcount, name):
14         self.function = function
15         self.childcount = childcount
16         self.name = name
17
18 class variable:
19     def __init__(self, var, value=0):
20         self.var = var
21         self.value = value
22         self.name = str(var)
23         self.type = "variable"
24
25     def evaluate(self):
26         return self.var.value
27
28     def setvar(self, value):
29         self.value = value
30
31     def display(self, indent=0):
32         print('%s%s' % (' '*indent, self.var))
33
34 class const:
35     def __init__(self, value):
36         self.value = value
37         self.name = str(value)
38         self.type = "constant"
39
40     def evaluate(self):
41         return self.value
42
43     def display(self, indent=0):
44         print('%s%d' % (' '*indent, self.value))
45
46 class node:
47     def __init__(self, type, children, funwrap, var=None,
48 const=None):
```



```

49     self.type = type
50     self.children = children
51     self.funwrap = funwrap
52     self.variable = var
53     self.const = const
54     self.depth = self.refreshdepth()
55     self.value = 0
56     self.fitness = 0
57
58     def eval(self,x):
59         if self.type == "variable":
60             return self.variable.value
61         elif self.type == "constant":
62             return self.const.value
63         else:
64             for c in self.children:
65                 result = [c.eval(x) for c in self.children]
66                 return self.funwrap.function(result,x)
67
68     def getfitness(self, checkdata):#checkdata like
69     {"x":1,"result":3}
70     diff = 0
71     #set variable value
72     for data in checkdata:
73         self.setvariablevalue(data)
74         diff += (self.eval(data['x']) - data["result"])**2
75     self.fitness = diff/len(checkdata)
76
77     def setvariablevalue(self, value):
78         if self.type == "variable":
79             if self.variable.var in value:
80                 self.variable.setvar(value[self.variable.var])
81             else:
82                 print("There is no value for variable:",
83 self.variable.var)
84             return
85         if self.type == "constant":
86             pass
87         if self.children:#function node
88             for child in self.children:
89                 child.setvariablevalue(value)
90
91     def refreshdepth(self):
92         if self.type == "constant" or self.type == "variable":
93             return 0
94         else:
95             depth = []
96             for c in self.children:
97                 depth.append(c.refreshdepth())
98             return max(depth) + 1
99
100
101     def display(self, indent=0):
102         if self.type == "function":

```

```

1103     print ((' '*indent) + self.funwrap.name)
1104     elif self.type == "variable":
1105         print ((' '*indent) + self.variable.name)
1106     elif self.type == "constant":
1107         print ((' '*indent) + self.const.name)
1108     if self.children:
1109         for c in self.children:
1110             c.display(indent + 1)
1111     ##for draw node
1112     def getwidth(self):
1113         if self.type == "variable" or self.type == "constant":
1114             return 1
1115         else:
1116             result = 0
1117             for i in range(0, len(self.children)):
1118                 result += self.children[i].getwidth()
1119             return result
1120     def drawnode(self, draw, x, y):
1121         if self.type == "function":
1122             allwidth = 0
1123             for c in self.children:
1124                 allwidth += c.getwidth()*100
1125             left = x - allwidth / 2
1126             #draw the function name
1127             draw.text((x - 10, y - 10), self.funwrap.name, (0, 0, 0))
1128             #draw the children
1129             for c in self.children:
1130                 wide = c.getwidth()*100
1131                 draw.line((x, y, left + wide / 2, y + 100), fill=(255, 0,
1132 0))
1133                 c.drawnode(draw, left + wide / 2, y + 100)
1134                 left = left + wide
1135             elif self.type == "variable":
1136                 draw.text((x - 5, y), self.variable.name, (0, 0, 0))
1137             elif self.type == "constant":
1138                 draw.text((x - 5, y), self.const.name, (0, 0, 0))
1139
1140     def drawtree(self, jpeg="tree.png"):
1141         w = self.getwidth()*100
1142         h = self.depth * 100 + 120
1143
1144         img = Image.new('RGB', (w, h), (255, 255, 255))
1145         draw = ImageDraw.Draw(img)
1146         self.drawnode(draw, w / 2, 20)
1147         img.save(jpeg, 'PNG')
1148
1149     class enviroment:
1150         def __init__(self, size, funwraplist, variablelist,
1151 constantlist, checkdata,
1152             minimaxtype="min", population=None, maxdepth=10,
1153             maxgen=500, crossrate=0.9, mutationrate=0.1,
1154 newbirthrate=1):
1155             self.funwraplist = funwraplist
1156             self.variablelist = variablelist

```

```

157     self.constantlist = constantlist
158     self.checkdata = checkdata
159     self.minimaxtype = minimaxtype
160     self.maxdepth = maxdepth
161     self.population = population or self._makepopulation(size)
162     self.size = size
163     self.maxgen = maxgen
164     self.crossrate = crossrate
165     self.mutationrate = mutationrate
166     self.newbirthrate = newbirthrate
167
168     self.besttree = self.population[0]
169     for i in range(0, self.size):
170         self.population[i].depth=self.population[i].refreshdepth()
171         self.population[i].getfitness(checkdata)
172         if self.minimaxtype == "min":
173             if self.population[i].fitness < self.besttree.fitness:
174                 self.besttree = self.population[i]
175         elif self.minimaxtype == "max":
176             if self.population[i].fitness > self.besttree.fitness:
177                 self.besttree = self.population[i]
178
179     def makepopulation(self, popsize):
180         return [self.maketree(0) for i in range(0, popsize)]
181
182     def maketree(self, startdepth):
183         if startdepth == 0:
184             #make a new tree
185             nodepattern = 0#function
186         elif startdepth == self.maxdepth:
187             nodepattern = 1#variable or constant
188         else:
189             nodepattern = randint(0, 1)
190         if nodepattern == 0:
191             childlist = []
192             selectedfun = randint(0, len(self.funwraplist) - 1)
193             for i in range(0, self.funwraplist[selectedfun].childcount):
194                 child = self.maketree(startdepth + 1)
195                 childlist.append(child)
196             return node("function", childlist,
197 self.funwraplist[selectedfun])
198         else:
199             # if randint(0, 1) == 0:#variable
200             #     selectedvariable = randint(0, len(self.variablelist) - 1)
201             #     return node("variable", None, None,
202             #         variable(self.variablelist[selectedvariable]),
203             None)
204             # else:
205             selectedconstant = randint(0, len(self.constantlist) - 1)
206             return node("constant", None, None, None,
207                 const(self.constantlist[selectedconstant]))
208
209     def mutate(self, tree, probchange=0.1, startdepth=0):
210         if random() < probchange:

```

```

211     return self._maketree(startdepth)
212 else:
213     result = deepcopy(tree)
214     if result.type == "function":
215         result.children = [self.mutate(c, probchange, startdepth +
216 1) \
217                             for c in tree.children]
218     return result
219
220 def crossover(self, tree1, tree2, probswap=1, top=1):
221     if random() < probswap and not top:
222         return deepcopy(tree2)
223     else:
224         result = deepcopy(tree1)
225         if tree1.type == "function" and tree2.type == "function":
226             result.children = [self.crossover(c,
227 choice(tree2.children),
228                             probswap, 0) for c in tree1.children]
229         return result
230
231 def evolve(self, maxgen=1000, crossrate=0.9,
232 mutationrate=0.1):
233     timebudget=50
234     start=time.clock()
235     while True:
236         # print("generation no.", i)
237         child = []
238         end=time.clock()
239         if timebudget<(end-start):
240             break
241         for j in range(0, int(self.size * self.newbirthrate / 2)):
242             parent1, p1 = self.roulettewheelssel()
243             parent2, p2 = self.roulettewheelssel()
244             newchild = self.crossover(parent1, parent2)
245             child.append(newchild)#generate new tree
246             parent, p3 = self.roulettewheelssel()
247             newchild = self.mutate(parent, mutationrate)
248             child.append(newchild)
249         #refresh all tree's fitness
250         for j in range(0, int(self.size * self.newbirthrate)):
251             replacetre, replacedindex =
252 self.roulettewheelssel(reverse=True)
253             #replace bad tree with child
254             self.population[replacedindex] = child[j]
255
256         for k in range(0, self.size):
257             self.population[k].getfitness(self.checkdata)
258             self.population[k].depth=self.population[k].refreshdepth()
259             if self.minimaxtype == "min":
260                 if self.population[k].fitness < self.besttree.fitness:
261                     self.besttree = self.population[k]
262             elif self.minimaxtype == "max":
263                 if self.population[k].fitness > self.besttree.fitness:
264                     self.besttree = self.population[k]

```

```

265     # print("best tree's fitness..",self.besttree.fitness)
266     # self.besttree.display()
267     # self.besttree.drawtree()
268     return self.besttree.fitness
269
270
271 def roulettewheelself(self, reverse=False):
272     if reverse == False:
273         allfitness = 0
274         for i in range(0, self.size):
275             allfitness += self.population[i].fitness
276         randomnum = random()*(self.size - 1)
277         check = 0
278         for i in range(0, self.size):
279             # print(self.population[i].fitness)
280             check += (1.0 - self.population[i].fitness / allfitness)
281             # print('---')
282             # print(check)
283             if check >= randomnum:
284                 return self.population[i], i
285     if reverse == True:
286         allfitness = 0
287         for i in range(0, self.size):
288             allfitness += self.population[i].fitness
289         randomnum = random()
290         check = 0
291         for i in range(0, self.size):
292             check += self.population[i].fitness * 1.0 / allfitness
293             if check >= randomnum:
294                 return self.population[i], i
295
296
297 #####
298
299 def add(ValuesList,x):
300     sumtotal = 0
301     for val in ValuesList:
302         sumtotal = sumtotal + val
303     return sumtotal
304
305 def sub(ValuesList,x):
306     return ValuesList[0] - ValuesList[1]
307
308 def mul(ValuesList,x):
309     return ValuesList[0] * ValuesList[1]
310
311 def div(ValuesList,x):
312     if ValuesList[1] == 0:
313         return 1
314     return ValuesList[0] / ValuesList[1]
315
316 def pow(ValuesList,x):
317     if ValuesList[0]==0 and ValuesList[1]==0:
318         return 0

```

```

319     else:
320         return float(ValuesList[0] ** ValuesList[1])
321
322 def sqrt(ValuesList,x):
323     if ValuesList[0]<0:
324         return 0
325     else:
326         return np.sqrt(float(ValuesList[0]))
327 def log(ValuesList,x):
328     if ValuesList[0] <=0:
329         return 0
330     else:
331         return np.log2(float(ValuesList[0]))
332 def exp(ValuesList,x):
333     return round(np.exp(float(ValuesList[0])),2)
334
335 def maximum(ValuesList,x):
336     return max(ValuesList)
337
338 def ifleq(ValuesList,x):
339     if ValuesList[0]<=ValuesList[1]:
340         return ValuesList[2]
341     else:
342         return ValuesList[3]
343
344
345 def data(ValuesList,x):
346     index x=np.mod(round(float(ValuesList[0])),n)
347     return x[int(index x)]
348
349 def diff(ValuesList,x):
350     k = np.mod(round(float(ValuesList[0])), n)
351     l = np.mod(round(float(ValuesList[1])), n)
352     result = np.subtract(x[int(k)], x[int(l)])
353     return result
354
355 def avg(ValuesList,x):
356     sum x = 0
357     k = np.mod(round(float(ValuesList[0])), n)
358     l = np.mod(round(float(ValuesList[1])), n)
359     if k == l:
360         return 0
361     else:
362         t = np.minimum(k, l)
363         upper = np.maximum(k, l) - 1
364         for z in range(int(t), int(upper) + 1):
365             sum_x = sum_x + x[z]
366         result = np.divide(sum_x, (k - l))
367         return result
368
369
370
371 addwrapper = funwrapper(add, 2, "add")
372 subwrapper = funwrapper(sub, 2, "sub")

```

```

373 mulwrapper = funwrapper(mul, 2, "mul")
374 divwrapper = funwrapper(div, 2, "div")
375
376 powwrapper = funwrapper(pow, 2, "pow")
377 sqrtwrapper = funwrapper(sqrt, 1, "sqrt")
378 logwrapper = funwrapper(log, 1, "log")
379 expwrapper = funwrapper(exp, 1, "exp")
380 maximumwrapper = funwrapper(maximum, 2, "maximum")
381 ifleqwrapper = funwrapper(ifleq, 4, "ifleq")
382
383 datawrapper = funwrapper(data, 1, "data")
384 diffwrapper = funwrapper(diff, 2, "diff")
385 avgwrapper = funwrapper(avg, 2, "avg")
386
387 def constructcheckdata(filename,m,n):
388     with open(filename,'r') as f:
389         checkdata=list()
390         for i in f:
391             dic=dict()
392             train data = list(map(float,
393 i.strip('\n').strip('\t').split('\t')))
394             x = train data[:n]
395             result= train data[n]
396             dic['x'] = x
397             dic['result'] = result
398             checkdata.append(dic)
399         return checkdata
400
401
402     pass
403
404 # def examplefun(x, y):
405 #     return x * x + x + 2 * y + 1
406 # def constructcheckdata(count=10):
407 #     checkdata = []
408 #     for i in range(0, count):
409 #         dic = {}
410 #         x = randint(0, 10)
411 #         y = randint(0, 10)
412 #         dic['x'] = x
413 #         dic['y'] = y
414 #         dic['result'] = examplefun(x, y)
415 #         checkdata.append(dic)
416 #     return checkdata
417
418 if __name__ == "__main__":
419     filename='data.txt'
420     m=10
421     n=10
422     checkdata = constructcheckdata(filename,m,n)
423     # print(checkdata)
424     dic_nsar=dict()
425     for population_size in range(10,430,30):
426         list_nsar=list()

```

```

427     for i in range(80):
428         env = enviroment(population_size,[addwrapper, subwrapper,
429 mulwrapper
430         ,divwrapper,sqrtwrapper,logwrapper,maximumwrapper,
431
432 ifleqwrapper,datawrapper,diffwrapper,avgwrapper],
433         ["x", "y"],
434         list(range(-100,100)), checkdata)
435         best_fitness=env.envolve()
436         list_nsat.append(best_fitness)
437         print(best_fitness)
438     # print('---')
439     dic nsat[population size]=list nsat
440     data = pd.DataFrame(dic nsat)
441     data.boxplot()
442     plt.ylabel('Fitness of the Fittest Solution')
443     plt.xlabel('Population Size')
444     plt.savefig('figure2.jpg')
445
446
447
448
449
450
451
452

```


Appendix D

Fittest Solution VS Crossover Rate

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # Author: Janet Chou
4  from random import random, randint, choice
5  from copy import deepcopy
6  from PIL import Image, ImageDraw
7  import numpy as np
8  import pandas as pd
9  import matplotlib.pyplot as plt
10 import time
11
12 class funwrapper:
13     def __init__(self, function, childcount, name):
14         self.function = function
15         self.childcount = childcount
16         self.name = name
17
18 class variable:
19     def __init__(self, var, value=0):
20         self.var = var
21         self.value = value
22         self.name = str(var)
23         self.type = "variable"
24
25     def evaluate(self):
26         return self.var.value
27
28     def setvar(self, value):
29         self.value = value
30
31     def display(self, indent=0):
32         print('%s%s' % (' '*indent, self.var))
33
34 class const:
35     def __init__(self, value):
36         self.value = value
37         self.name = str(value)
38         self.type = "constant"
39
40     def evaluate(self):
41         return self.value
42
43     def display(self, indent=0):
44         print('%s%d' % (' '*indent, self.value))
45
46 class node:
47     def __init__(self, type, children, funwrap, var=None,
48 const=None):
49         self.type = type
```

```

50     self.children = children
51     self.funwrap = funwrap
52     self.variable = var
53     self.const = const
54     self.depth = self.refreshdepth()
55     self.value = 0
56     self.fitness = 0
57
58     def eval(self,x):
59         if self.type == "variable":
60             return self.variable.value
61         elif self.type == "constant":
62             return self.const.value
63         else:
64             for c in self.children:
65                 result = [c.eval(x) for c in self.children]
66                 return self.funwrap.function(result,x)
67
68     def getfitness(self, checkdata):#checkdata like
69     {"x":1,"result":3}
70     diff = 0
71     #set variable value
72     for data in checkdata:
73         self.setvariablevalue(data)
74         diff += (self.eval(data['x']) - data["result"])**2
75     self.fitness = diff/len(checkdata)
76
77     def setvariablevalue(self, value):
78         if self.type == "variable":
79             if self.variable.var in value:
80                 self.variable.setvar(value[self.variable.var])
81             else:
82                 print("There is no value for variable:",
83 self.variable.var)
84             return
85         if self.type == "constant":
86             pass
87         if self.children:#function node
88             for child in self.children:
89                 child.setvariablevalue(value)
90
91     def refreshdepth(self):
92         if self.type == "constant" or self.type == "variable":
93             return 0
94         else:
95             depth = []
96             for c in self.children:
97                 depth.append(c.refreshdepth())
98             return max(depth) + 1
99
100
101     def display(self, indent=0):
102         if self.type == "function":
103             print ((' '*indent) + self.funwrap.name)

```

```

104     elif self.type == "variable":
105         print ((' '*indent) + self.variable.name)
106     elif self.type == "constant":
107         print ((' '*indent) + self.const.name)
108     if self.children:
109         for c in self.children:
110             c.display(indent + 1)
111     ##for draw node
112     def getwidth(self):
113         if self.type == "variable" or self.type == "constant":
114             return 1
115         else:
116             result = 0
117             for i in range(0, len(self.children)):
118                 result += self.children[i].getwidth()
119             return result
120     def drawnode(self, draw, x, y):
121         if self.type == "function":
122             allwidth = 0
123             for c in self.children:
124                 allwidth += c.getwidth()*100
125             left = x - allwidth / 2
126             #draw the function name
127             draw.text((x - 10, y - 10), self.funwrap.name, (0, 0, 0))
128             #draw the children
129             for c in self.children:
130                 wide = c.getwidth()*100
131                 draw.line((x, y, left + wide / 2, y + 100), fill=(255, 0,
132 0))
133                 c.drawnode(draw, left + wide / 2, y + 100)
134                 left = left + wide
135             elif self.type == "variable":
136                 draw.text((x - 5, y), self.variable.name, (0, 0, 0))
137             elif self.type == "constant":
138                 draw.text((x - 5, y), self.const.name, (0, 0, 0))
139
140     def drawtree(self, jpeg="tree.png"):
141         w = self.getwidth()*100
142         h = self.depth * 100 + 120
143
144         img = Image.new('RGB', (w, h), (255, 255, 255))
145         draw = ImageDraw.Draw(img)
146         self.drawnode(draw, w / 2, 20)
147         img.save(jpeg, 'PNG')
148
149     class enviroment:
150         def __init__(self, crossrate, funwraplist, variablelist,
151 constantlist, checkdata,
152             minimaxtype="min", population=None,
153 size=10, mutationrate=0.1,
154             maxdepth=10,
155             maxgen=500, newbirthrate=1):
156             self.funwraplist = funwraplist
157             self.variablelist = variablelist

```

```

158     self.constantlist = constantlist
159     self.checkdata = checkdata
160     self.minimaxtype = minimaxtype
161     self.maxdepth = maxdepth
162     self.population = population or self._makepopulation(size)
163     self.size = size
164     self.maxgen = maxgen
165     self.crossrate = crossrate
166     self.mutationrate = mutationrate
167     self.newbirthrate = newbirthrate
168
169     self.besttree = self.population[0]
170     for i in range(0, self.size):
171         self.population[i].depth=self.population[i].refreshdepth()
172         self.population[i].getfitness(checkdata)
173         if self.minimaxtype == "min":
174             if self.population[i].fitness < self.besttree.fitness:
175                 self.besttree = self.population[i]
176         elif self.minimaxtype == "max":
177             if self.population[i].fitness > self.besttree.fitness:
178                 self.besttree = self.population[i]
179
180     def makepopulation(self, popsize):
181         return [self.maketree(0) for i in range(0, popsize)]
182
183     def maketree(self, startdepth):
184         if startdepth == 0:
185             #make a new tree
186             nodepattern = 0#function
187         elif startdepth == self.maxdepth:
188             nodepattern = 1#variable or constant
189         else:
190             nodepattern = randint(0, 1)
191         if nodepattern == 0:
192             childlist = []
193             selectedfun = randint(0, len(self.funwraplist) - 1)
194             for i in range(0, self.funwraplist[selectedfun].childcount):
195                 child = self.maketree(startdepth + 1)
196                 childlist.append(child)
197             return node("function", childlist,
198 self.funwraplist[selectedfun])
199         else:
200             # if randint(0, 1) == 0:#variable
201             #     selectedvariable = randint(0, len(self.variablelist) - 1)
202             #     return node("variable", None, None,
203             #         variable(self.variablelist[selectedvariable]),
204             None)
205             # else:
206             selectedconstant = randint(0, len(self.constantlist) - 1)
207             return node("constant", None, None, None,
208                 const(self.constantlist[selectedconstant]))
209
210     def mutate(self, tree, probchange=0.1, startdepth=0):
211         if random() < probchange:

```

```

212     return self._maketree(startdepth)
213 else:
214     result = deepcopy(tree)
215     if result.type == "function":
216         result.children = [self.mutate(c, probchange, startdepth +
217 1) \
218                             for c in tree.children]
219     return result
220
221 def crossover(self, tree1, tree2, probswap, top=1):
222     if random() < probswap and not top:
223         return deepcopy(tree2)
224     else:
225         result = deepcopy(tree1)
226         if tree1.type == "function" and tree2.type == "function":
227             result.children = [self.crossover(c,
228 choice(tree2.children),
229                             probswap, 0) for c in tree1.children]
230     return result
231
232 def evolve(self, crossrate, mutationrate=0.1, maxgen=1000):
233     timebudget=50
234     start=time.clock()
235     while True:
236         # print("generation no.", i)
237         child = []
238         end=time.clock()
239         if timebudget<(end-start):
240             # print(end-start)
241             break
242         for j in range(0, int(self.size * self.newbirthrate / 2)):
243             parent1, p1 = self.roulettewheelssel()
244             parent2, p2 = self.roulettewheelssel()
245             newchild = self.crossover(parent1, parent2, crossrate)
246             child.append(newchild) #generate new tree
247             parent, p3 = self.roulettewheelssel()
248             newchild = self.mutate(parent, mutationrate)
249             child.append(newchild)
250         #refresh all tree's fitness
251         for j in range(0, int(self.size * self.newbirthrate)):
252             replacedtree, replacedindex =
253 self.roulettewheelssel(reverse=True)
254             #replace bad tree with child
255             self.population[replacedindex] = child[j]
256
257         for k in range(0, self.size):
258             self.population[k].getfitness(self.checkdata)
259             self.population[k].depth=self.population[k].refreshdepth()
260             if self.minimaxtype == "min":
261                 if self.population[k].fitness < self.besttree.fitness:
262                     self.besttree = self.population[k]
263             elif self.minimaxtype == "max":
264                 if self.population[k].fitness > self.besttree.fitness:
265                     self.besttree = self.population[k]

```

```

266     # print("best tree's fitness..",self.besttree.fitness)
267     # self.besttree.display()
268     # self.besttree.drawtree()
269     return self.besttree.fitness
270
271
272     def roulettewheelself(self, reverse=False):
273         if reverse == False:
274             allfitness = 0
275             for i in range(0, self.size):
276                 allfitness += self.population[i].fitness
277             randomnum = random()*(self.size - 1)
278             check = 0
279             for i in range(0, self.size):
280                 # print(self.population[i].fitness)
281                 check += (1.0 - self.population[i].fitness / allfitness)
282                 # print('---')
283                 # print(check)
284                 if check >= randomnum:
285                     return self.population[i], i
286         if reverse == True:
287             allfitness = 0
288             for i in range(0, self.size):
289                 allfitness += self.population[i].fitness
290             randomnum = random()
291             check = 0
292             for i in range(0, self.size):
293                 check += self.population[i].fitness * 1.0 / allfitness
294                 if check >= randomnum:
295                     return self.population[i], i
296
297
298     #####
299
300     def add(ValuesList,x):
301         sumtotal = 0
302         for val in ValuesList:
303             sumtotal = sumtotal + val
304         return sumtotal
305
306     def sub(ValuesList,x):
307         return ValuesList[0] - ValuesList[1]
308
309     def mul(ValuesList,x):
310         return ValuesList[0] * ValuesList[1]
311
312     def div(ValuesList,x):
313         if ValuesList[1] == 0:
314             return 1
315         return ValuesList[0] / ValuesList[1]
316
317     def pow(ValuesList,x):
318         if ValuesList[0]==0 and ValuesList[1]==0:
319             return 0

```

```

320     else:
321         return float(ValuesList[0] ** ValuesList[1])
322
323 def sqrt(ValuesList,x):
324     if ValuesList[0]<0:
325         return 0
326     else:
327         return np.sqrt(float(ValuesList[0]))
328 def log(ValuesList,x):
329     if ValuesList[0] <=0:
330         return 0
331     else:
332         return np.log2(float(ValuesList[0]))
333 def exp(ValuesList,x):
334     return round(np.exp(float(ValuesList[0])),2)
335
336 def maximum(ValuesList,x):
337     return max(ValuesList)
338
339 def ifleq(ValuesList,x):
340     if ValuesList[0]<=ValuesList[1]:
341         return ValuesList[2]
342     else:
343         return ValuesList[3]
344
345
346 def data(ValuesList,x):
347     index x=np.mod(round(float(ValuesList[0])),n)
348     return x[int(index x)]
349
350 def diff(ValuesList,x):
351     k = np.mod(round(float(ValuesList[0])), n)
352     l = np.mod(round(float(ValuesList[1])), n)
353     result = np.subtract(x[int(k)], x[int(l)])
354     return result
355
356 def avg(ValuesList,x):
357     sum x = 0
358     k = np.mod(round(float(ValuesList[0])), n)
359     l = np.mod(round(float(ValuesList[1])), n)
360     if k == l:
361         return 0
362     else:
363         t = np.minimum(k, l)
364         upper = np.maximum(k, l) - 1
365         for z in range(int(t), int(upper) + 1):
366             sum_x = sum_x + x[z]
367         result = np.divide(sum_x, (k - l))
368         return result
369
370
371
372 addwrapper = funwrapper(add, 2, "add")
373 subwrapper = funwrapper(sub, 2, "sub")

```

```

374 mulwrapper = funwrapper(mul, 2, "mul")
375 divwrapper = funwrapper(div, 2, "div")
376
377 powwrapper = funwrapper(pow, 2, "pow")
378 sqrtwrapper = funwrapper(sqrt, 1, "sqrt")
379 logwrapper = funwrapper(log, 1, "log")
380 expwrapper = funwrapper(exp, 1, "exp")
381 maximumwrapper = funwrapper(maximum, 2, "maximum")
382 ifleqwrapper = funwrapper(ifleq, 4, "ifleq")
383
384 datawrapper = funwrapper(data, 1, "data")
385 diffwrapper = funwrapper(diff, 2, "diff")
386 avgwrapper = funwrapper(avg, 2, "avg")
387
388 def constructcheckdata(filename,m,n):
389     with open(filename,'r') as f:
390         checkdata=list()
391         for i in f:
392             dic=dict()
393             train data = list(map(float,
394 i.strip('\n').strip('\t').split('\t')))
395             x = train data[:n]
396             result= train data[n]
397             dic['x'] = x
398             dic['result'] = result
399             checkdata.append(dic)
400         return checkdata
401
402
403     pass
404
405 # def examplefun(x, y):
406 #     return x * x + x + 2 * y + 1
407 # def constructcheckdata(count=10):
408 #     checkdata = []
409 #     for i in range(0, count):
410 #         dic = {}
411 #         x = randint(0, 10)
412 #         y = randint(0, 10)
413 #         dic['x'] = x
414 #         dic['y'] = y
415 #         dic['result'] = examplefun(x, y)
416 #         checkdata.append(dic)
417 #     return checkdata
418
419 if __name__ == "__main__":
420     filename='data.txt'
421     m=10
422     n=10
423     checkdata = constructcheckdata(filename,m,n)
424     # print(checkdata)
425     dic_nsat=dict()
426     crossrate_list = [ 0,0.2,0.4,0.6,0.8,0.9,1]
427     for crossrate in crossrate_list:

```



```

428     list_nsats=list()
429     for i in range(80):
430         env = environment(crossrate,[addwrapper, subwrapper,
431 mulwrapper,divwrapper,
432          sqrtwrapper,logwrapper,maximumwrapper,
433 ifleqwrapper,datawrapper,diffwrapper,avgwrapper],
434          ["x", "y"],
435          list(range(-100,100)), checkdata)
436         best_fitness=env.enhance(crossrate)
437         list_nsats.append(best_fitness)
438         print(best_fitness)
439     # print('---')
440     dic_nsats[crossrate]=list_nsats
441     data = pd.DataFrame(dic_nsats)
442     data.boxplot()
443     plt.ylabel('Fitness of the Fittest Solution')
444     plt.xlabel('Crossover Rate')
445     plt.savefig('figure5.jpg')
446
447
448
449
450
451
452
453
454
455

```