



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 Exercise 4 Pseudo-code of My Algorithm</i>	<i>1</i>
<i>Chapter 2 Exercise 5-Number of Satisfied Clauses VS Mutation Rate</i>	<i>3</i>
<i>Chapter 3 Exercise 5-Number of Satisfied Clauses VS Population Size</i>	<i>6</i>
<i>Chapter 4 Exercise 5-Number of Satisfied Clauses VS Tournament Size</i>	<i>6</i>
<i>Appendix A Number of Satisfied Clauses vs Mutation Rate</i>	<i>12</i>
<i>Appendix B Number of Satisfied Clauses vs Population Size</i>	<i>16</i>
<i>Appendix C Number of Satisfied Clauses vs Tournament Size</i>	<i>20</i>

Chapter 1

Exercise 4 Pseudo-code of My Algorithm

Algorithm MAXSAT Problem

```
Require: File on WDIMACS format   wdimacs  $\in$  'XXX.wcnf '  
Require: Time Budget   time_budget  $\in$  N and repetitions  $\in$  N  
/* The necessary functions of the algorithm */  
1: Read_wdimacs_file(input:file_path)  
2:   f=open (' file_path ', ' read')  
3:   for i=1 to length(f) do  
4:     if f[i]= ' P' then do  
5:       variable_number = f[i][2]  
6:       clause_number =f[i][3]  
7:       clause_list=f[i+1: length(f)]  
8:       return variable_number, clause_number, clause_list  
9:     end if  
10:  end for  
11: Check_assignment's_satisfiability(input:clause,assignment) /*check whether the assignment is satisfied with the clause or not*/  
12:  for i=1 to length(clause) do  
13:    if clause[i]>0 then do  
14:      assignment_index= clause[i]  
15:      if assignment[assignment_index]= ' 1 ' then do  
16:        return 1  
17:      else  
18:        return 0  
19:      end if  
20:    else  
21:      assignment_index= abs(clause[i])  
22:      if assignment[assignment_index]= ' 0 ' then do  
23:        return 1  
24:      else  
25:        return 0  
26:      end if  
27:    end if  
28:  end for  
29: Traverse_clause_list(input:clause_list,assignment)  
30:   Number_of_satisfied_clause=0  
31:   for i=1 to length(clause_list) do  
32:     check_result=call Check assignment's satisfiability(input:clause_list[i],assignment)  
33:     if check_result = 1 then do  
34:       Number_of_satisfied_clause= Number_of_satisfied_clause+1  
35:     end if  
36:   end for  
37:   return Number_of_satisfied_clause  
38: Initial_population(Input:population_size,number_of_bitstring)  
39:   for i=1 to population_size do  
40:      $P_0(i) \sim \text{Unif}(\{0, 1\}^{\text{number\_of\_bitstring}})$   
41:   end for  
42:   return  $P_0$   
43: Tournament_selection(input:population,k)  
44:   tournament_list  $\sim$  randomly select k individuals from population  
45:   return tournament_list  
46: Fitness_Function(input: tournament_list)
```

```

47:   for i=1 to length(tournament_list) do
48:       fitness_value_list~call traverse_clause_list(input:clause_list,i) /*calculate the number of satisfied clauses for each one */
49:   end for
50:   the_best_individual~ the one who has the max fitness value
51:   return the_best_individual
52: Mutation(input:bitstring,mutation_rate)
53:   for i=1 to length(bitstring) do
54:       if probability≤mutation_rate then do
55:           bitstring[i] ≠ bitstring[i]
56:       else
57:           bitstring[i]= bitstring[i]
58:       end if
59:   end for
60:   return bitstring
61: Crossover(input:father,mother)
62:   for i=1 to length(father) do
63:       if father[i] ≠ mother[i] then do
64:           if probability≤50% then do
65:               children[i]= ' 0 '
66:           else
67:               children[i]= ' 1 '
68:           end if
69:       else
70:           children[i]=father[i]
71:       end if
72:   end for
73:   return children

/* main function to start the program */
74: Program MAXSAT problem
75: number_of_bitstring, clause_number, clause_list=call Read_wdimacs_file (input: wdimacs)
76: mutation_rate=0.0001
77: k=3 /* tournament size */
78: population_size=70
79: for i=1 to repetitions do
80:   P0=call Initial_population(Input:population_size,number_of_bitstring)
81:   for t=0...until runtime > time_budget or nsat= clause_number do /* nsat is number of satisfied clause */
82:       for i = 1 to population_size do
83:           father =call Fitness_function(call tournament_selection(Pt))
84:           mother=call Fitness_function(call tournament_selection(Pt))
85:           Pt+1(i)=call Crossover(call mutation(father), call mutation(mother))
86:       end for
87:       best_assignment=call Fitness_function(Pt+1)
88:       nsat= call traverse_clause_list(clause_list, best_assignment)
89:   end for
90:   output(runtime,nsat, best_assignment)
91: end for

```

Chapter 2

Exercise 5-Number of Satisfied clauses VS Mutation Rate

2.1 Experiment Parameter

In order to investigate the relationship between the impact of mutation rate on the quality (i.e., number of satisfied clauses) of the solutions obtained, we should control variables. The experiment parameters were used as shown below.

2.1.1 Constant

2.1.1.1 Time Budget

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

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 one hundred (i.e., **Population Size (λ)=100**).

2.1.1.3 Tournament Size

Tournament size is defined as k in the source code. Two is the value of Tournament Size in the experiment (i.e., **Tournament Size (k)=2**).

2.1.1.4 Wdimacs File

Wdimacs File is a MAXSAT instance on the WDIMACS file format from the MAXSAT competition (MSE17 complete unweighted benchmarks). **'3col80_5_2.shuffled.cnf.wcnf'**(in 'maxone' folder) is the instance to test code on (The figure showed below).

```

3col80_5_2.shuffled.cnf.wcnf
1 c PSEUDOBOOLEAN 846
2 p wcnf 160 846 846

```

figure 1: The Part of 3col80_5_2.shuffled.cnf.wcnf

2.1.2 Variable

Mutation rate is essential for the behavior of the algorithm. The range of mutation rate is defined from 0.00000001 to 0.1 (**i.e., Mutation Rate $\in (0.00000001, 0.1)$**).

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 two parts in one figure. The increment of first part is 10 times than previous one and the increment of second part is 0.01 (**i.e., Mutation Rate $\in \text{Set } \{0.00000001, 0.0000001, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1\}$**).

2.2.1 Boxplot of Results

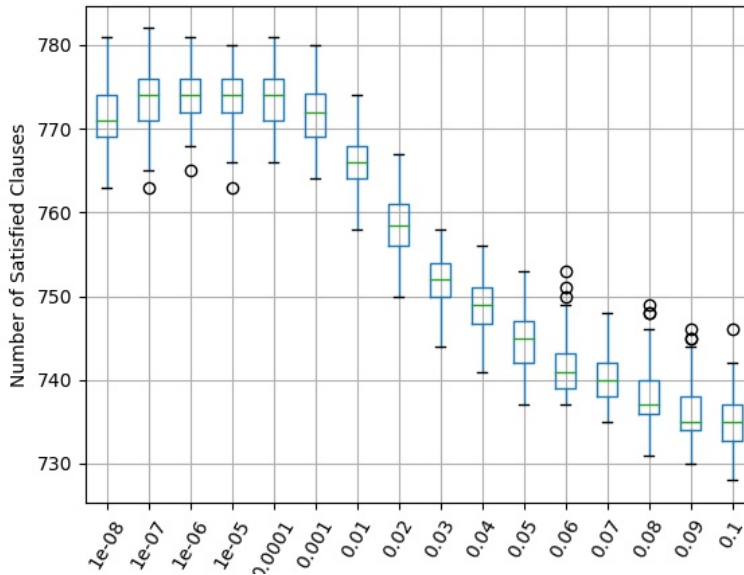


figure 2: Number of Satisfied clauses VS Mutation Rate

2.2.2 Analysis of Boxplot

The boxplots summarize the relationship between the impact of mutation rate on the quality (i.e., number of satisfied clauses) of the solutions obtained.

Overall, when mutation rate is below 0.001, 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. However, when the mutation rate is above 0.001, they have an approximate linear decreasing trend with the decrement of mutation rate.

It is clear that the outliers almost appear above the upper edge (i.e., the maximum value) in the boxplots when the mutation rate is above 0.05. It may show that the mutation rate will affect the probability of outliers occurring.

In conclusion, setting mutation rate to 0.0001 is the best choice to get the biggest number of satisfied clauses and not to get outliers.

Chapter 3

Exercise 5-Number of Satisfied clauses vs Population Size

3.1 Experiment Parameter

In order to investigate the relationship between the impact of population size on the quality (i.e., number of satisfied clauses) of the solutions obtained, 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, 0.001 is chose as the value of *mutation rate* (i.e., ***Mutation Rate=0.001***).

3.1.1.2 Time Budget

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

3.1.1.3 Tournament Size

Tournament size is defined as k in the source code. Two is the value of Tournament Size in the experiment (i.e., ***Tournament Size (k)=2***).

3.1.1.4 Wdimacs File

Wdimacs File is a MAXSAT instance on the WDIMACS file format from the MAXSAT competition (MSE17 complete unweighted benchmarks). ***'3col80_5_2.shuffled.cnf.wcnf'***(in 'maxone' folder) is the instance to test code on (The figure showed below).


```

3col80_5_2.shuffled.cnf.wcnf x
1 c PSEUDOBOOLEAN 846
2 p wcnf 160 846 846

```

figure 3: The Part of 3col80_5_2.shuffled.cnf.wcnf

3.1.2 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 one thousand (i.e., **Population Size (λ) \in [10 , 1000]**).

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 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, 430, 460, 490, 520, 550, 580, 610, 640, 670, 700, 730, 760, 790, 820, 850, 880, 910, 940, 970, 1000 }**).

3.2.1 Boxplot of Results

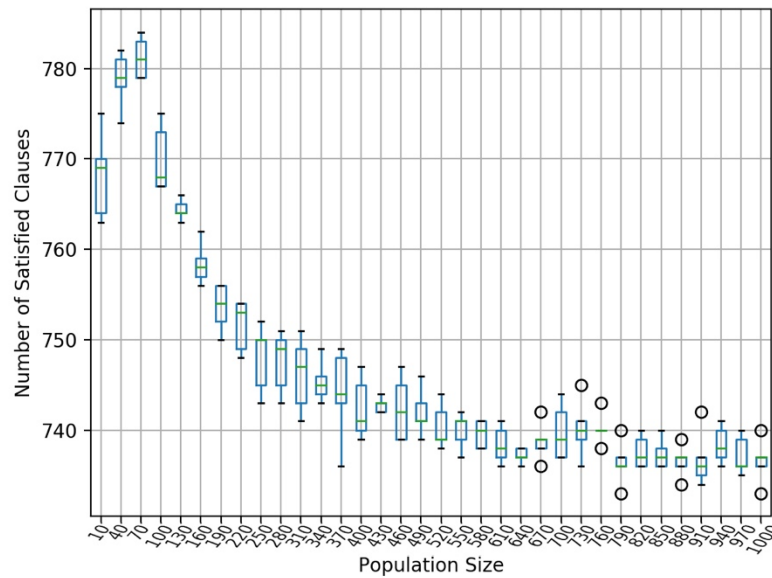


figure 4: Number of Satisfied clauses VS Population Size

3.2.2 Analysis of Boxplot

The boxplot summarizes the relationship between the impact of population size on the quality (i.e., number of satisfied clauses) 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 reach to the peak (the number of satisfied clauses reach to about 780 when population size is 70) before they have an exponential downward trend with the increment of population size (when population size varies from 70 to 640). After that, they fluctuate slight with the population size varying from 640 to 1000.

Moreover, when population size is over 160, the outliers appear above the upper edge (i.e., the maximum value) and below the lower edge (i.e., the minimum value). It may show that if the population size gets much bigger, the probability of the appearing outliers will be high.

It is clear that setting population size to 70 is the best choice to get the biggest number of satisfied clauses and not to get outliers. More obviously, at that point, the distribution of normal values becomes more concentrated than others.

Chapter 4

Exercise 5-Number of Satisfied clauses vs Tournament Size

4.1 Experiment Parameter

In order to investigate the relationship between the impact of tournament size on the quality (i.e., number of satisfied clauses) of the solutions obtained, 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, 0.001 is chose as the value of *mutation rate* (i.e., ***Mutation Rate=0.001***).

4.1.1.2 Time Budget

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

4.1.1.3 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***).

4.1.1.4 Wdimacs File

Wdimacs File is a MAXSAT instance on the WDIMACS file format from the MAXSAT competition (MSE17 complete unweighted benchmarks). ***'3col80_5_2.shuffled.cnf.wcnf'***(in 'maxone' folder) **is the instance to test code on** (The figure showed below).

```
3col80_5_2.shuffled.cnf.wcnf x
1 c PSEUDOBOOLEAN 846
2 p wcnf 160 846 846
```

figure 5: The Part of 3col80_5_2.shuffled.cnf.wcnf

4.1.2 Variable

Tournament size is defined as k in the source code. The range of Tournament size is defined from two to five (i.e., **Tournament Size** (k) $\in [2, 9]$).

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 trends and each individual case, the increments of 1 for tournament size was tried in this experiment (i.e., **Tournament Size** (k) $\in \text{Set } \{2, 3, 4, 5, 6, 7, 8, 9\}$).

4.2.1 Boxplot of Results

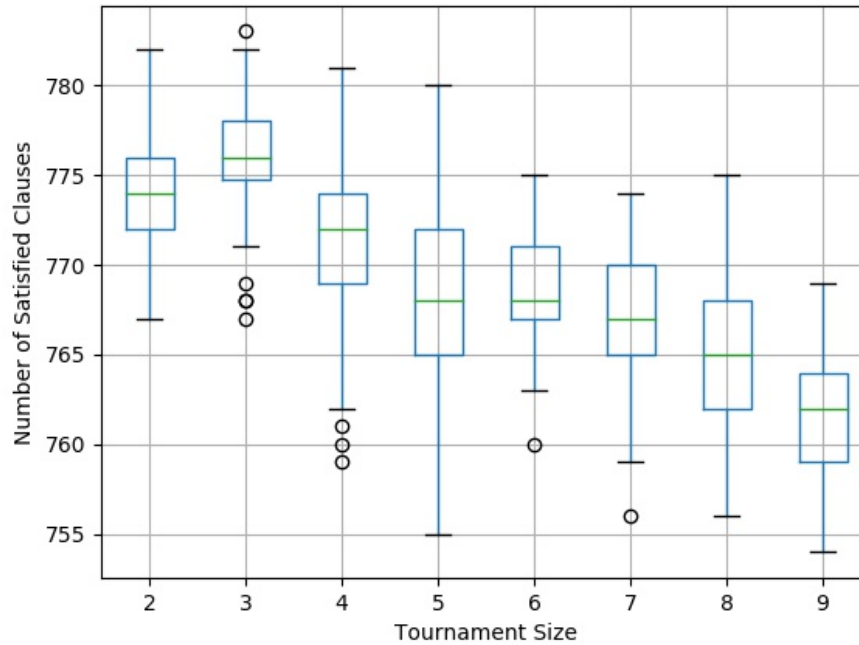


figure 6: Number of Satisfied clauses VS Tournament Size

4.2.2 Analysis of Boxplot

The boxplot summarizes the relationship between the impact of tournament size on the quality (i.e., number of satisfied clauses) of the solutions obtained.

Overall, before 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) have a downward trend with the increment of tournament size, they reach to the peak (the number of satisfied clauses is about 775 when the tournament size is 3).

In conclusion, setting tournament size to 3 is the best choice to get the biggest number of satisfied clauses. More obviously, at that point, the distribution of normal values becomes more concentrated than others.

Appendix A

Number of Satisfied clauses VS Mutation Rate

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # Author: Janet Chou
4  import time
5  import random
6  import math
7  import argparse
8  import pandas as pd
9  import matplotlib.pyplot as plt
10 import numpy as np
11
12 #read WDIMACS file
13 def read_file(filename):
14     with open(filename,'r') as f:
15         context=list()
16         next(f)
17         for i in f:
18             context.append(i)
19         arguments = context[0].strip('\n').split(' ')
20         variable_number = int(arguments[2])
21         clause_number = int(arguments[3])
22         clause_list = context[1:len(context)]
23     return variable number, clause number, clause list
24
25
26 def check_satisafiability(clause,assignment):
27     end_index = len(clause) - 1
28     clause=clause[1:end index]
29     for i in clause:
30         i=int(i)
31         if i>0:
32             x_index = i - 1
33             if assignment[x_index]=='1':
34                 return 1
35
36         else:
37             x_index = abs(i)- 1
38             if assignment[x_index]=='0':
39                 return 1
40     else:
41         return 0
42
43 def traverse_clause_list(clause_list,assignment):
44     satisfied_number = 0
45     for i in clause list:
46         clause = list(i.strip('\n').split(' '))
47         output = check satisafiability(clause, assignment)
48         if output == 1:
```

```

49         satisfied_number += 1
50     return satisfied_number
51
52 def initial_population(population_size, variable_number):
53     origin_population = list()
54     upper_bound = 2 ** variable_number
55     for i in range(population_size):
56         bitstring_number = int(random.randint(0, upper_bound - 1))
57         bitstring = '{0:b}'.format(bitstring_number)
58         bitstring_number = bitstring.zfill(variable_number)
59         origin_population.append(bitstring_number)
60     # print(origin_population)
61     return origin_population
62
63 def tournament_selection(origin_population, k):
64     tournament_list = list()
65     random_index = list(range(population_size))
66     random_list = random.sample(random_index, k)
67     for l in random_list:
68         tournament_list.append(origin_population[l])
69     return fitness_function(tournament_list)
70
71 def fitness_function(tournament_list):
72     sum_list = list()
73     for m in tournament_list:
74         fitness_value = traverse_clause_list(clause_list, m)
75         sum_list.append(fitness_value)
76     number_sat = max(sum_list)
77     output_z = tournament_list[sum_list.index(number_sat)]
78     return output_z
79
80 def mutation(bits_x, mutation_rate):
81     bits_x = list(bits_x)
82     output_z = bits_x[:]
83     for j in range(variable_number):
84         random_mutation_probability = random.random()
85         if random_mutation_probability <= mutation_rate:
86             if output_z[j] == '0':
87                 output_z[j] = '1'
88             else:
89                 output_z[j] = '0'
90     return output_z
91
92
93
94
95 def crossover(bits_x, bits_y, n):
96     output_z = list()
97     for j in range(n):
98         if bits_x[j] != bits_y[j]:
99             if random.random() <= 0.5:
100                 output_z.append('0')
101             else:
102                 output_z.append('1')

```

```

103         else:
104             output_z.append(bits_x[j])
105     z=' '.join(output_z)
106     return z
107
108
109
110
111
112 if __name__ == '__main__':
113     parser=argparse.ArgumentParser(description='manual
114     to this script')
115     parser.add_argument('-question',type=int,default=3)
116     parser.add_argument('-clause', type=str, default='0.5
117     1 2 -3 -4 0')
118     parser.add_argument('-assignment', type=str, default='0000')
119     parser.add_argument('-wdimacs',
120     default='3col80_5_2.shuffled.cnf.wcnf')
121     parser.add_argument('-repetitions', type=int, default=100)
122     parser.add_argument('-time_budget', type=int, default=20)
123
124
125     args= parser.parse_args()
126     question_number=args.question
127     if question_number==1:
128         clause = list(args.clause.split(' '))
129         assignment=args.assignment
130         output=check_satisfiability(clause,assignment)
131         print(output)
132
133
134     elif question_number == 2:
135         assignment = args.assignment
136         filename=args.wdimacs
137         variable_number, clause_number,
138     clause_list=read_file(filename)
139         output=traverse_clause_list(clause_list,assignment)
140         print(output)
141
142
143     else:
144         time_budget = args.time_budget
145         repetitions = args.repetitions
146         filename = args.wdimacs
147         variable_number, clause_number, clause_list =
148     read_file(filename)
149         mutation_rate=0.001
150         #
151     mutation_list=[0.001,0.002,0.003,0.004,0.005,0.006,0.007,
152     0.008,0.009]
153         # k=2
154         population_size=100
155         nsat=clause_number
156         dic_nsat = dict()

```



```

157
158     for k in range(2,10):
159         list_nsat = list()
160
161         for j in range(repetitions):
162             start = time.clock()
163             origin_population =
164 initial_population(population_size, variable_number)
165             z_value = 0
166             t = 0
167             z = str()
168             flag = True
169             while flag:
170                 population = list()
171                 population_value=list()
172
173                 end = time.clock()
174                 runtime = end - start
175                 if runtime>time_budget or z_value==nsat :
176                     break
177                 t = t + 1
178                 for h in range(population_size):
179                     x = tournament_selection(origin_population,
180 k)
181                     y = tournament_selection(origin_population,
182 k)
183                     z = crossover(mutation(x,mutation_rate ),
184 mutation(y,
185                     mutation_rate), variable_number)
186                     z_value=traverse_clause_list(clause_list,z)
187                     population.append(z)
188                     population_value.append(z_value)
189                     z_value=max(population_value)
190                     z = population[population_value.index(z_value)]
191                     origin_population = population[:]
192                     list_nsat.append(z_value)
193                     print(z_value)
194                 dic_nsat[k] = list_nsat
195             ax = pd.DataFrame(dic_nsat)
196             ax.plot(kind='box', grid=True)
197             plt.ylabel('Number of Satisfied Clauses')
198             plt.xlabel('Tournament Size')
199             plt.savefig('figure4.jpg')
200
201
202
203
204

```

Appendix B

Number of Satisfied clauses vs Population Size

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # Author: Janet Chou
4  import time
5  import random
6  import math
7  import argparse
8  import pandas as pd
9  import matplotlib.pyplot as plt
10
11  #read WDIMACS file
12  def read_file(filename):
13      with open(filename,'r') as f:
14          context=list()
15          next(f)
16          for i in f:
17              context.append(i)
18          arguments = context[0].strip('\n').split(' ')
19          variable_number = int(arguments[2])
20          clause_number = int(arguments[3])
21          clause_list = context[1:len(context)]
22      return variable_number, clause_number, clause_list
23
24
25  def check_satisfiability(clause,assignment):
26      end_index = len(clause) - 1
27      clause=clause[1:end_index]
28      for i in clause:
29          i=int(i)
30          if i>0:
31              x_index = i - 1
32              if assignment[x_index]=='1':
33                  return 1
34
35          else:
36              x_index = abs(i)- 1
37              if assignment[x_index]=='0':
38                  return 1
39      else:
40          return 0
41
42  def traverse_clause_list(clause_list,assignment):
43      satisfied_number = 0
44      for i in clause_list:
45          clause = list(i.strip('\n').split(' '))
46          output = check_satisfiability(clause, assignment)
47          if output == 1:
48              satisfied_number += 1
```

```

49     return satisfied_number
50
51 def initial_population(population_size,variable_number):
52     origin_population=list()
53     upper_bound=2**variable_number
54     for i in range(population_size):
55         bitstring_number=int(random.randint(0,upper_bound-1))
56         bitstring='{0:b}'.format(bitstring_number)
57         bitstring_number=bitstring.zfill(variable_number)
58         origin_population.append(bitstring_number)
59     # print(origin_population)
60     return origin_population
61
62 def tournament_selection(origin_population,k):
63     tournament_list = list()
64     random_index = list(range(population_size))
65     random_list = random.sample(random_index, k)
66     for l in random_list:
67         tournament_list.append(origin_population[l])
68     return fitness_function(tournament_list)
69
70 def fitness_function(tournament_list):
71     sum_list=list()
72     for m in tournament_list:
73         fitness_value=traverse_clause_list(clause_list,m)
74         sum_list.append(fitness_value)
75     number_sat=max(sum_list)
76     output_z = tournament_list[sum_list.index(number_sat)]
77     return output_z
78
79 def mutation(bits_x,mutation_rate):
80     bits_x = list(bits_x)
81     output_z = bits_x[:]
82     for j in range(variable_number):
83         random_mutation_probability = random.random()
84         if random_mutation_probability <= mutation_rate:
85             if output_z[j] == '0':
86                 output_z[j] = '1'
87             else:
88                 output_z[j] = '0'
89     return output_z
90
91
92
93
94 def crossover(bits_x,bits_y,n):
95     output_z = list()
96     for j in range(n):
97         if bits_x[j] != bits_y[j]:
98             if random.random() <= 0.5:
99                 output_z.append('0')
100             else:
101                 output_z.append('1')
102     else:

```

```

103         output_z.append(bits_x[j])
104     z=''.join(output_z)
105     return z
106
107
108
109
110
111 if __name__ == '__main__':
112     parser=argparse.ArgumentParser(description='manual to this
113 script')
114     parser.add_argument('-question',type=int,default=3)
115     parser.add_argument('-clause', type=str, default='0.5 1 2 -3 -
116 4 0')
117     parser.add_argument('-assignment', type=str, default='0000')
118     parser.add_argument('-wdimacs',
119 default='3col80_5_2.shuffled.cnf.wcnf')
120     parser.add_argument('-repetitions', type=int, default=5)
121     parser.add_argument('-time_budget', type=int, default=20)
122
123
124     args= parser.parse_args()
125     question_number=args.question
126     if question_number==1:
127         clause = list(args.clause.split(' '))
128         assignment=args.assignment
129         output=check_satisfiability(clause,assignment)
130         print(output)
131
132
133     elif question_number == 2:
134         assignment = args.assignment
135         filename=args.wdimacs
136         variable_number, clause_number,
137 clause_list=read_file(filename)
138         output=traverse_clause_list(clause_list,assignment)
139         print(output)
140
141
142     else:
143         time_budget = args.time_budget
144         repetitions = args.repetitions
145         filename = args.wdimacs
146         variable_number, clause_number, clause_list =
147 read_file(filename)
148         mutation_rate=0.001
149         k=2
150         # population_size=100
151         nsat=clause_number
152         dic_nsat=dict()
153
154         for population_size in range(10, 1001, 30):
155             list_nsat=list()
156

```

```

157         for j in range(repetitions):
158             start = time.clock()
159             origin_population =
160 initial_population(population_size, variable_number)
161             z_value = 0
162             t = 0
163             z = str()
164             flag = True
165             while flag:
166                 population = list()
167                 population_value=list()
168
169                 end = time.clock()
170                 runtime = end - start
171                 if runtime>time_budget or z_value==nsat :
172                     break
173                 t = t + 1
174                 for h in range(population_size):
175                     x = tournament_selection(origin_population,
176 k)
177                     y = tournament_selection(origin_population,
178 k)
179                     z = crossover(mutation(x,mutation_rate ),
180 mutation(y, mutation_rate), variable_number)
181                     z_value=traverse_clause_list(clause_list,z)
182                     population.append(z)
183                     population_value.append(z_value)
184                     z_value=max(population_value)
185                     z = population[population_value.index(z_value)]
186                     origin_population = population[:]
187                     list_nsat.append(z_value)
188                     print(z_value)
189                 dic_nsat[population_size]=list_nsat
190                 data = pd.DataFrame(dic_nsat)
191                 data.boxplot()
192                 plt.ylabel('Number of Satisfied Clauses')
193                 plt.xlabel('Population Size')
194                 plt.show()
195
196
197
198
199
200

```

Appendix C

Number of Satisfied clauses vs Tournament Size

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # Author: Janet Chou
4  import time
5  import random
6  import math
7  import argparse
8  import pandas as pd
9  import matplotlib.pyplot as plt
10 import numpy as np
11
12 #read WDIMACS file
13 def read_file(filename):
14     with open(filename,'r') as f:
15         context=list()
16         next(f)
17         for i in f:
18             context.append(i)
19             arguments = context[0].strip('\n').split(' ')
20             variable_number = int(arguments[2])
21             clause_number = int(arguments[3])
22             clause_list = context[1:len(context)]
23     return variable number, clause number, clause list
24
25
26 def check_satisfiability(clause,assignment):
27     end_index = len(clause) - 1
28     clause=clause[1:end_index]
29     for i in clause:
30         i=int(i)
31         if i>0:
32             x_index = i - 1
33             if assignment[x_index]=='1':
34                 return 1
35
36         else:
37             x_index = abs(i)- 1
38             if assignment[x_index]=='0':
39                 return 1
40     else:
41         return 0
42
43 def traverse_clause_list(clause_list,assignment):
44     satisfied_number = 0
45     for i in clause_list:
46         clause = list(i.strip('\n').split(' '))
47         output = check_satisfiability(clause, assignment)
48         if output == 1:
```

```

49         satisfied_number += 1
50     return satisfied_number
51
52 def initial_population(population_size, variable_number):
53     origin_population = list()
54     upper_bound = 2 ** variable_number
55     for i in range(population_size):
56         bitstring_number = int(random.randint(0, upper_bound - 1))
57         bitstring = '{0:b}'.format(bitstring_number)
58         bitstring_number = bitstring.zfill(variable_number)
59         origin_population.append(bitstring_number)
60     # print(origin_population)
61     return origin_population
62
63 def tournament_selection(origin_population, k):
64     tournament_list = list()
65     random_index = list(range(population_size))
66     random_list = random.sample(random_index, k)
67     for l in random_list:
68         tournament_list.append(origin_population[l])
69     return fitness_function(tournament_list)
70
71 def fitness_function(tournament_list):
72     sum_list = list()
73     for m in tournament_list:
74         fitness_value = traverse_clause_list(clause_list, m)
75         sum_list.append(fitness_value)
76     number_sat = max(sum_list)
77     output_z = tournament_list[sum_list.index(number_sat)]
78     return output_z
79
80 def mutation(bits_x, mutation_rate):
81     bits_x = list(bits_x)
82     output_z = bits_x[:]
83     for j in range(variable_number):
84         random_mutation_probability = random.random()
85         if random_mutation_probability <= mutation_rate:
86             if output_z[j] == '0':
87                 output_z[j] = '1'
88             else:
89                 output_z[j] = '0'
90     return output_z
91
92
93
94
95 def crossover(bits_x, bits_y, n):
96     output_z = list()
97     for j in range(n):
98         if bits_x[j] != bits_y[j]:
99             if random.random() <= 0.5:
100                 output_z.append('0')
101             else:
102                 output_z.append('1')

```

```

103         else:
104             output_z.append(bits_x[j])
105         z=' '.join(output_z)
106         return z
107
108
109
110
111
112 if __name__ == '__main__':
113     parser=argparse.ArgumentParser(description='manual to this
114 script')
115     parser.add_argument('-question',type=int,default=3)
116     parser.add_argument('-clause', type=str, default='0.5
117 1 2 -3 -4 0')
118     parser.add_argument('-assignment', type=str, default='0000')
119     parser.add_argument('-wdimacs',
120 default='3col80_5_2.shuffled.cnf.wcnf')
121     parser.add_argument('-repetitions', type=int, default=100)
122     parser.add_argument('-time_budget', type=int, default=20)
123
124
125     args= parser.parse_args()
126     question_number=args.question
127     if question_number==1:
128         clause = list(args.clause.split(' '))
129         assignment=args.assignment
130         output=check_satisfiability(clause,assignment)
131         print(output)
132
133
134     elif question_number == 2:
135         assignment = args.assignment
136         filename=args.wdimacs
137         variable_number, clause_number,
138 clause_list=read_file(filename)
139         output=traverse_clause_list(clause_list,assignment)
140         print(output)
141
142
143     else:
144         time_budget = args.time_budget
145         repetitions = args.repetitions
146         filename = args.wdimacs
147         variable_number, clause_number, clause_list =
148 read_file(filename)
149         mutation_rate=0.001
150         #
151 mutation_list=[0.001,0.002,0.003,0.004,0.005,0.006,0.007,
152 0.008,0.009]
153         # k=2
154         population_size=100
155         nsat=clause_number
156         dic_nsat = dict()

```



```

157
158     for k in range(2,10):
159         list_nsatsat = list()
160
161         for j in range(repetitions):
162             start = time.clock()
163             origin_population =
164 initial_population(population_size, variable_number)
165             z_value = 0
166             t = 0
167             z = str()
168             flag = True
169             while flag:
170                 population = list()
171                 population_value=list()
172
173                 end = time.clock()
174                 runtime = end - start
175                 if runtime>time_budget or z_value==nsat :
176                     break
177                 t = t + 1
178                 for h in range(population_size):
179                     x = tournament_selection(origin_population,
180 k)
181                     y = tournament_selection(origin_population,
182 k)
183                     z = crossover(mutation(x,mutation_rate ),
184 mutation(y,
185                     mutation_rate), variable_number)
186                     z_value=traverse_clause_list(clause_list,z)
187                     population.append(z)
188                     population_value.append(z_value)
189                     z_value=max(population_value)
190                     z = population[population_value.index(z_value)]
191                     origin_population = population[:]
192                     list_nsatsat.append(z_value)
193                     print(z_value)
194                 dic_nsatsat[k] = list_nsatsat
195             ax = pd.DataFrame(dic_nsatsat)
196             ax.plot(kind='box', grid=True)
197             plt.ylabel('Number of Satisfied Clauses')
198             plt.xlabel('Tournament Size')
199             plt.savefig('figure4.jpg')
200
201
202
203
204

```