UNIVERSITY OF BIRMINGHAM

School of Computer Science

Nature-Inspired Search and Optimisation

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

Table of Contents

Chapter 1

Exercise 4 Pseudo-code of My Algorithm

Algorithm MAXSAT Problem

```
Require: File on WDIMACS format    wdimacs ∈ ' XXX.wcnf '
Require: Time Budget   time_budget ∈ N and repetitions ∈ N
/* The necessary functions of the algorithm */
Read_wdimacs_file(input:file_path)
    f=open (' file_path ', ' read')
    for i=1 to length(f) do
        if f[i]= ' P' then do
            variable_number = f[i][2]
            clause_number =f[i][3]
            clause_list=f[i+1: length(f)]
            return variable_number, clause_number, clause_list
        end if
    end for
Check_assignment's_satisafiability(input:clause,assignment) /*check whether the assignment is satisfied with the clause or not*/
    for i=1 to length(clause) do
        if clause[i]>0 then do
            assignment_index= clause[i]
                if assignment[assignment_index]= ' 1 ' then do
                    return 1
                else
                    return 0
                end if
        else
            assignment_index= abs(clause[i])
                if assignment[assignment_index]= ' 0 ' then do
                    return 1
                else
                    return 0
                end if
        end if
    end for
Traverse_clause_list(input:clause_list,assignment)
    Number_of_satisfied_clause=0
    for i=1 to length(clause_list) do
        check_result=call Check assignment's satisafiability(input:clause_list[i],assignment)
        if check_result = 1   then do
            Number_of_satisfied_clause= Number_of_satisfied_clause+1
        end if
    end for
    return Number_of_satisfied_clause
Initial_population(Input:population_size,number_of_bitstring)
    for i=1 to population_size do
        P_0(i) ~ Unif({0, 1}^number_of_bitstring)
    end for
    return P_0
Tournament_selection(input:population,k)
    tournament_list ~ randomly select k individuals from population
    return tournament_list
Fitness_Function(input: tournament_list)
```

```
    for i=1 to length(tournament_list) do
        fitness_value_list~call traverse_clause_list(input:clause_list,i) /*calculate the number of satisfied clauses for each one */
    end for
    the_best_individual~ the one who has the max fitness value
    return the_best_individual
Mutation(input:bitstring,mutation_rate)
    for i=1 to length(bitstring) do
        if probability≤mutation_rate then do
            bitstring[i]≠ bitstring[i]
        else
            bitstring[i]= bitstring[i]
        end if
    end for
    return bitstring
Crossover(input:father,mother)
    for i=1 to length(father) do
        if father[i]≠mother[i] then do
            if probability≤50% then do
                children[i]= ' 0 '
            else
                children[i]= ' 1 '
            end if
        else
            children[i]=father[i]
        end if
    end for
    return children

/* main function to start the program */
Program MAXSAT problem
number_of_bitstring, clause_number, clause_list=call Read wdimacs file (input: wdimacs)
mutation_rate=0.0001
k=3    /* tournament size */
population_size=70
for i=1 to repetitions do
    P_0=call Initial population(Input:population_size,number_of_bitstring)
    for t=0...until runtime > time_budget or nsat= clause_number do    /* nsat is number of satisfied clause */
        for i = 1 to population_size do
            father =call Fitness_function(call tournament_selection(Pt))
            mother=call Fitness_function(call tournament_selection(Pt))
            P_{t+1}(i)=call Crossover(call mutation(father), call mutation(mother))
        end for
        best_assignment=call Fitness_function(P_{t+1})
        nsat= call traverse_clause_list(clause_list, best_assignment)
    end for
    output(runtime,nsat, best_assignment)
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).**

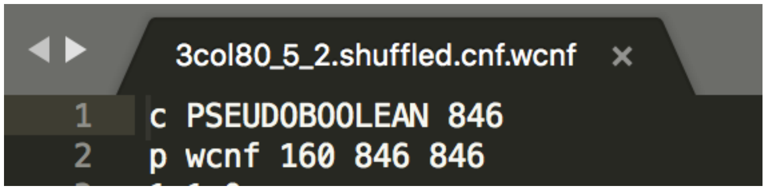

figure 1: The Part of 3col80_5_2.shuffled.cnf.wcnf

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** $\in$ **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

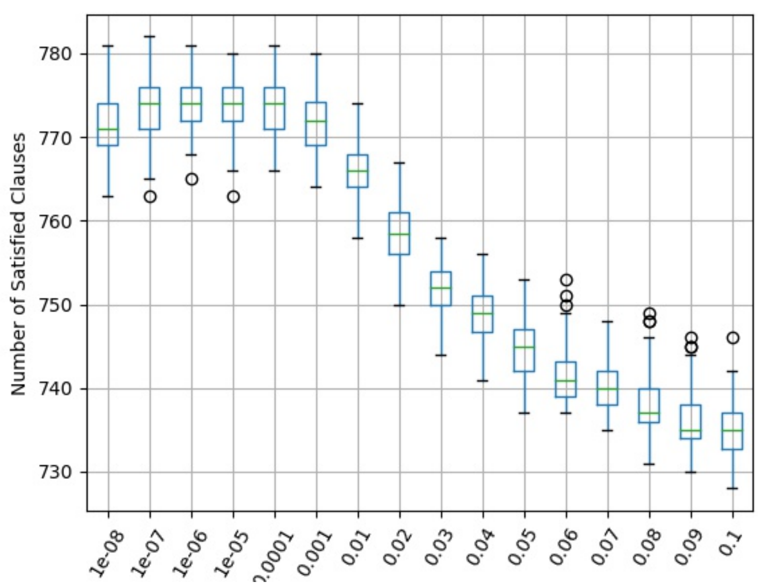

figure 2: Number of Satisfied clauses VS Mutation Rate

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).**

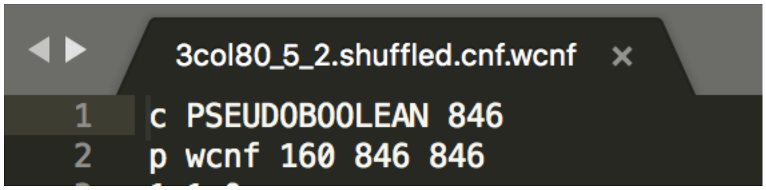

figure 3: The Part of 3col80_5_2.shuffled.cnf.wcnf

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

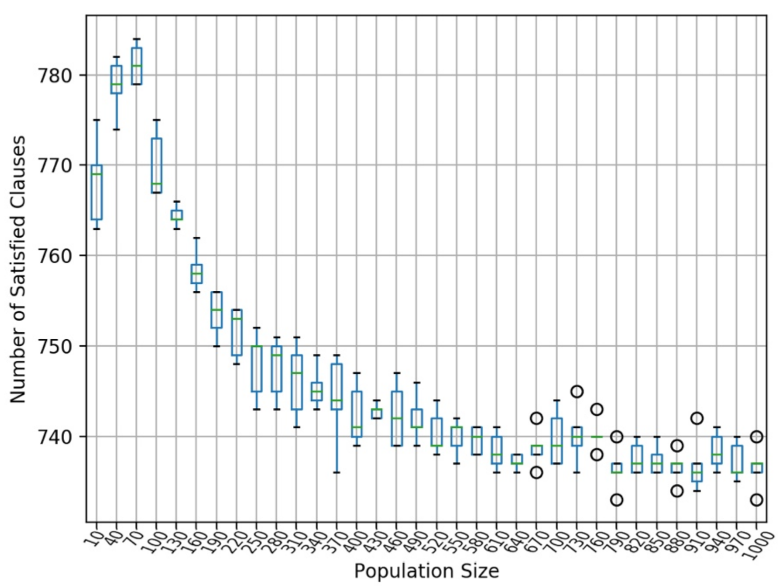

figure 4: Number of Satisfied clauses VS Population Size

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).**

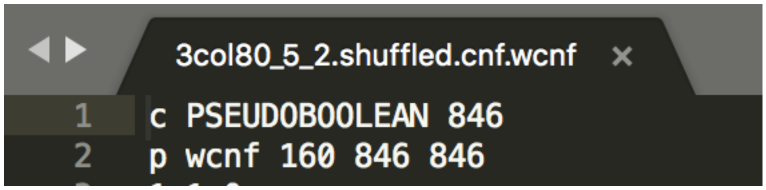

figure 5: The Part of 3col80_5_2.shuffled.cnf.wcnf

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*) ∈ [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*) ∈ Set {2, 3, 4, 5,6,7,8,9 })**.

4.2.1 Boxplot of Results

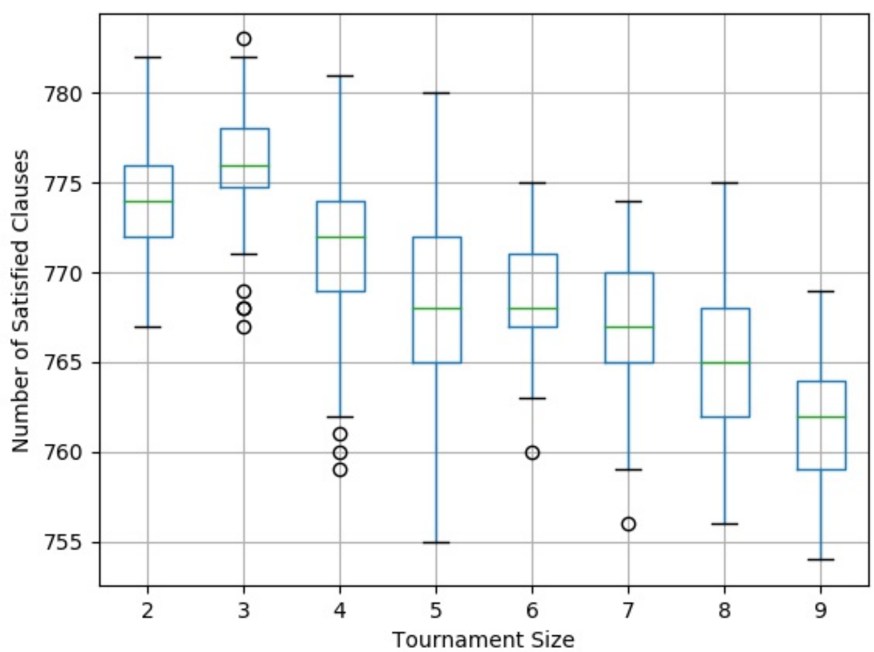

figure 6: Number of Satisfied clauses VS Tournament Size

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

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: Janet Chou
import time
import random
import math
import argparse
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

#read WDIMACS file
def read_file(filename):
    with open(filename,'r') as f:
        context=list()
        next(f)
        for i in f:
            context.append(i)
        arguments = context[0].strip('\n').split(' ')
        variable_number = int(arguments[2])
        clause_number = int(arguments[3])
        clause_list = context[1:len(context)]
    return variable_number, clause_number, clause_list


def check_satisafiability(clause,assignment):
    end_index = len(clause) - 1
    clause=clause[1:end_index]
    for i in clause:
        i=int(i)
        if i>0:
            x_index = i - 1
            if assignment[x_index]=='1':
                return 1

        else:
            x_index = abs(i)- 1
            if assignment[x_index]=='0':
                return 1
    else:
        return 0

def traverse_clause_list(clause_list,assignment):
    satisfied_number = 0
    for i in clause_list:
        clause = list(i.strip('\n').split(' '))
        output = check_satisafiability(clause, assignment)
        if output == 1:
```

```
            satisfied_number += 1
    return satisfied_number

def initial_population(population_size,variable_number):
    origin_population=list()
    upper_bound=2**variable_number
    for i in range(population_size):
        bitstring_number=int(random.randint(0,upper_bound-1))
        bitstring='{0:b}'.format(bitstring_number)
        bitstring_number=bitstring.zfill(variable_number)
        origin_population.append(bitstring_number)
    # print(origin_population)
    return origin_population

def tournament_selection(origin_population,k):
    tournament_list = list()
    random_index = list(range(population_size))
    random_list = random.sample(random_index, k)
    for l in random_list:
        tournament_list.append(origin_population[l])
    return fitness_function(tournament_list)

def fitness_function(tournament_list):
    sum_list=list()
    for m in tournament_list:
        fitness_value=traverse_clause_list(clause_list,m)
        sum_list.append(fitness_value)
    number_sat=max(sum_list)
    output_z = tournament_list[sum_list.index(number_sat)]
    return output_z

def mutation(bits_x,mutation_rate):
    bits_x = list(bits_x)
    output_z = bits_x[:]
    for j in range(variable_number):
        random_mutation_probability = random.random()
        if random_mutation_probability <= mutation_rate:
            if output_z[j] == '0':
                output_z[j] = '1'
            else:
                output_z[j] = '0'
    return output_z




def crossover(bits_x,bits_y,n):
    output_z = list()
    for j in range(n):
        if bits_x[j] != bits_y[j]:
            if random.random() <= 0.5:
                output_z.append('0')
            else:
                output_z.append('1')
```

```
        else:
            output_z.append(bits_x[j])
    z=''.join(output_z)
    return z




if __name__ == '__main__':
    parser=argparse.ArgumentParser(description='manual
    to this script')
    parser.add_argument('-question',type=int,default=3)
    parser.add_argument('-clause', type=str, default='0.5
    1 2 -3 -4 0')
    parser.add_argument('-assignment', type=str, default='0000')
    parser.add_argument('-wdimacs',
default='3col80_5_2.shuffled.cnf.wcnf')
    parser.add_argument('-repetitions', type=int, default=100)
    parser.add_argument('-time_budget', type=int, default=20)


    args= parser.parse_args()
    question_number=args.question
    if question_number==1:
        clause = list(args.clause.split(' '))
        assignment=args.assignment
        output=check_satisafiability(clause,assignment)
        print(output)


    elif question_number == 2:
        assignment = args.assignment
        filename=args.wdimacs
        variable_number, clause_number,
clause_list=read_file(filename)
        output=traverse_clause_list(clause_list,assignment)
        print(output)


    else:
        time_budget = args.time_budget
        repetitions = args.repetitions
        filename = args.wdimacs
        variable_number, clause_number, clause_list =
read_file(filename)
        mutation_rate=0.001
        #
mutation_list=[0.001,0.002,0.003,0.004,0.005,0.006,0.007,
0.008,0.009]
        # k=2
        population_size=100
        nsat=clause_number
        dic_nsat = dict()
```

```
        for k in range(2,10):
            list_nsat = list()

            for j in range(repetitions):
                start = time.clock()
                origin_population = initial_population(population_size, variable_number)
                z_value =0
                t = 0
                z = str()
                flag = True
                while flag:
                    population = list()
                    population_value=list()

                    end = time.clock()
                    runtime = end - start
                    if runtime>time_budget or z_value==nsat :
                        break
                    t = t + 1
                    for h in range(population_size):
                        x = tournament_selection(origin_population, k)
                        y = tournament_selection(origin_population, k)
                        z = crossover(mutation(x,mutation_rate ), mutation(y,
                        mutation_rate), variable_number)
                        z_value=traverse_clause_list(clause_list,z)
                        population.append(z)
                        population_value.append(z_value)
                    z_value=max(population_value)
                    z = population[population_value.index(z_value)]
                    origin_population = population[:]
                list_nsat.append(z_value)
                print(z_value)
            dic_nsat[k] = list_nsat
        ax = pd.DataFrame(dic_nsat)
        ax.plot(kind='box', grid=True)
        plt.ylabel('Number of Satisfied Clauses')
        plt.xlabel('Tournament Size')
        plt.savefig('figure4.jpg')
```

Appendix B

Number of Satisfied clauses vs Population Size

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: Janet Chou
import time
import random
import math
import argparse
import pandas as pd
import matplotlib.pyplot as plt

#read WDIMACS file
def read_file(filename):
    with open(filename,'r') as f:
        context=list()
        next(f)
        for i in f:
            context.append(i)
        arguments = context[0].strip('\n').split(' ')
        variable_number = int(arguments[2])
        clause_number = int(arguments[3])
        clause_list = context[1:len(context)]
    return variable_number, clause_number, clause_list


def check_satisafiability(clause,assignment):
    end_index = len(clause) - 1
    clause=clause[1:end_index]
    for i in clause:
        i=int(i)
        if i>0:
            x_index = i - 1
            if assignment[x_index]=='1':
                return 1

        else:
            x_index = abs(i)- 1
            if assignment[x_index]=='0':
                return 1
    else:
        return 0

def traverse_clause_list(clause_list,assignment):
    satisfied_number = 0
    for i in clause_list:
        clause = list(i.strip('\n').split(' '))
        output = check_satisafiability(clause, assignment)
        if output == 1:
            satisfied_number += 1
```

```
    return satisfied_number

def initial_population(population_size,variable_number):
    origin_population=list()
    upper_bound=2**variable_number
    for i in range(population_size):
        bitstring_number=int(random.randint(0,upper_bound-1))
        bitstring='{0:b}'.format(bitstring_number)
        bitstring_number=bitstring.zfill(variable_number)
        origin_population.append(bitstring_number)
    # print(origin_population)
    return origin_population

def tournament_selection(origin_population,k):
    tournament_list = list()
    random_index = list(range(population_size))
    random_list = random.sample(random_index, k)
    for l in random_list:
        tournament_list.append(origin_population[l])
    return fitness_function(tournament_list)

def fitness_function(tournament_list):
    sum_list=list()
    for m in tournament_list:
        fitness_value=traverse_clause_list(clause_list,m)
        sum_list.append(fitness_value)
    number_sat=max(sum_list)
    output_z = tournament_list[sum_list.index(number_sat)]
    return output_z

def mutation(bits_x,mutation_rate):
    bits_x = list(bits_x)
    output_z = bits_x[:]
    for j in range(variable_number):
        random_mutation_probability = random.random()
        if random_mutation_probability <= mutation_rate:
            if output_z[j] == '0':
                output_z[j] = '1'
            else:
                output_z[j] = '0'
    return output_z




def crossover(bits_x,bits_y,n):
    output_z = list()
    for j in range(n):
        if bits_x[j] != bits_y[j]:
            if random.random() <= 0.5:
                output_z.append('0')
            else:
                output_z.append('1')
        else:
```

```
            output_z.append(bits_x[j])
    z=''.join(output_z)
    return z





if __name__ == '__main__':
    parser=argparse.ArgumentParser(description='manual to this
script')
    parser.add_argument('-question',type=int,default=3)
    parser.add_argument('-clause', type=str, default='0.5 1 2 -3 -
4 0')
    parser.add_argument('-assignment', type=str, default='0000')
    parser.add_argument('-wdimacs',
default='3col80_5_2.shuffled.cnf.wcnf')
    parser.add_argument('-repetitions', type=int, default=5)
    parser.add_argument('-time_budget', type=int, default=20)


    args= parser.parse_args()
    question_number=args.question
    if question_number==1:
        clause = list(args.clause.split(' '))
        assignment=args.assignment
        output=check_satisafiability(clause,assignment)
        print(output)


    elif question_number == 2:
        assignment = args.assignment
        filename=args.wdimacs
        variable_number, clause_number,
clause_list=read_file(filename)
        output=traverse_clause_list(clause_list,assignment)
        print(output)


    else:
        time_budget = args.time_budget
        repetitions = args.repetitions
        filename = args.wdimacs
        variable_number, clause_number, clause_list =
read_file(filename)
        mutation_rate=0.001
        k=2
        # population_size=100
        nsat=clause_number
        dic_nsat=dict()

        for population_size in range(10, 1001, 30):
            list_nsat=list()

```

```
            for j in range(repetitions):
                start = time.clock()
                origin_population =
initial_population(population_size, variable_number)
                z_value =0
                t = 0
                z = str()
                flag = True
                while flag:
                    population = list()
                    population_value=list()

                    end = time.clock()
                    runtime = end - start
                    if runtime>time_budget or z_value==nsat :
                        break
                    t = t + 1
                    for h in range(population_size):
                        x = tournament_selection(origin_population,
k)
                        y = tournament_selection(origin_population,
k)
                        z = crossover(mutation(x,mutation_rate ),
mutation(y, mutation_rate), variable_number)
                        z_value=traverse_clause_list(clause_list,z)
                        population.append(z)
                        population_value.append(z_value)
                    z_value=max(population_value)
                    z = population[population_value.index(z_value)]
                    origin_population = population[:]
                list_nsat.append(z_value)
                print(z_value)
            dic_nsat[population_size]=list_nsat
        data = pd.DataFrame(dic_nsat)
        data.boxplot()
        plt.ylabel('Number of Satisfied Clauses')
        plt.xlabel('Population Size')
        plt.show()
```

Appendix C

Number of Satisfied clauses vs Tournament Size

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: Janet Chou
import time
import random
import math
import argparse
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

#read WDIMACS file
def read_file(filename):
    with open(filename,'r') as f:
        context=list()
        next(f)
        for i in f:
            context.append(i)
        arguments = context[0].strip('\n').split(' ')
        variable_number = int(arguments[2])
        clause_number = int(arguments[3])
        clause_list = context[1:len(context)]
    return variable_number, clause_number, clause_list


def check_satisafiability(clause,assignment):
    end_index = len(clause) - 1
    clause=clause[1:end_index]
    for i in clause:
        i=int(i)
        if i>0:
            x_index = i - 1
            if assignment[x_index]=='1':
                return 1

        else:
            x_index = abs(i)- 1
            if assignment[x_index]=='0':
                return 1
    else:
        return 0

def traverse_clause_list(clause_list,assignment):
    satisfied_number = 0
    for i in clause_list:
        clause = list(i.strip('\n').split(' '))
        output = check_satisafiability(clause, assignment)
        if output == 1:
```

```
            satisfied_number += 1
    return satisfied_number

def initial_population(population_size,variable_number):
    origin_population=list()
    upper_bound=2**variable_number
    for i in range(population_size):
        bitstring_number=int(random.randint(0,upper_bound-1))
        bitstring='{0:b}'.format(bitstring_number)
        bitstring_number=bitstring.zfill(variable_number)
        origin_population.append(bitstring_number)
    # print(origin_population)
    return origin_population

def tournament_selection(origin_population,k):
    tournament_list = list()
    random_index = list(range(population_size))
    random_list = random.sample(random_index, k)
    for l in random_list:
        tournament_list.append(origin_population[l])
    return fitness_function(tournament_list)

def fitness_function(tournament_list):
    sum_list=list()
    for m in tournament_list:
        fitness_value=traverse_clause_list(clause_list,m)
        sum_list.append(fitness_value)
    number_sat=max(sum_list)
    output_z = tournament_list[sum_list.index(number_sat)]
    return output_z

def mutation(bits_x,mutation_rate):
    bits_x = list(bits_x)
    output_z = bits_x[:]
    for j in range(variable_number):
        random_mutation_probability = random.random()
        if random_mutation_probability <= mutation_rate:
            if output_z[j] == '0':
                output_z[j] = '1'
            else:
                output_z[j] = '0'
    return output_z




def crossover(bits_x,bits_y,n):
    output_z = list()
    for j in range(n):
        if bits_x[j] != bits_y[j]:
            if random.random() <= 0.5:
                output_z.append('0')
            else:
                output_z.append('1')
```

```
        else:
            output_z.append(bits_x[j])
    z=''.join(output_z)
    return z




if __name__ == '__main__':
    parser=argparse.ArgumentParser(description='manual to this script')
    parser.add_argument('-question',type=int,default=3)
    parser.add_argument('-clause', type=str, default='0.5 1 2 -3 -4 0')
    parser.add_argument('-assignment', type=str, default='0000')
    parser.add_argument('-wdimacs', default='3col80_5_2.shuffled.cnf.wcnf')
    parser.add_argument('-repetitions', type=int, default=100)
    parser.add_argument('-time_budget', type=int, default=20)


    args= parser.parse_args()
    question_number=args.question
    if question_number==1:
        clause = list(args.clause.split(' '))
        assignment=args.assignment
        output=check_satisafiability(clause,assignment)
        print(output)


    elif question_number == 2:
        assignment = args.assignment
        filename=args.wdimacs
        variable_number, clause_number, clause_list=read_file(filename)
        output=traverse_clause_list(clause_list,assignment)
        print(output)


    else:
        time_budget = args.time_budget
        repetitions = args.repetitions
        filename = args.wdimacs
        variable_number, clause_number, clause_list = read_file(filename)
        mutation_rate=0.001
        # mutation_list=[0.001,0.002,0.003,0.004,0.005,0.006,0.007,0.008,0.009]
        # k=2
        population_size=100
        nsat=clause_number
        dic_nsat = dict()
```

```python

        for k in range(2,10):
            list_nsat = list()

            for j in range(repetitions):
                start = time.clock()
                origin_population =
initial_population(population_size, variable_number)
                z_value =0
                t = 0
                z = str()
                flag = True
                while flag:
                    population = list()
                    population_value=list()

                    end = time.clock()
                    runtime = end - start
                    if runtime>time_budget or z_value==nsat :
                        break
                    t = t + 1
                    for h in range(population_size):
                        x = tournament_selection(origin_population,
k)
                        y = tournament_selection(origin_population,
k)
                        z = crossover(mutation(x,mutation_rate ),
mutation(y,
                        mutation_rate), variable_number)
                        z_value=traverse_clause_list(clause_list,z)
                        population.append(z)
                        population_value.append(z_value)
                    z_value=max(population_value)
                    z = population[population_value.index(z_value)]
                    origin_population = population[:]
                list_nsat.append(z_value)
                print(z_value)
            dic_nsat[k] = list_nsat
        ax = pd.DataFrame(dic_nsat)
        ax.plot(kind='box', grid=True)
        plt.ylabel('Number of Satisfied Clauses')
        plt.xlabel('Tournament Size')
        plt.savefig('figure4.jpg')
```