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*

## *Problem Restatement*

The runtime of an evolutionary algorithm is defined as the number of function evaluations (i.e., number of generations multiplied with the population size $\lambda$) before the algorithm discovers the optimal solution for the first time. Run experiments with the genetic algorithm on OneMax where you investigate the following relationships

- runtime vs mutation rate *chi/n*
- runtime vs problem size $n$
- runtime vs population size $\lambda$
- runtime vs tournament size $k$

Repeat each experiment 100 times. Show the results as boxplots (e.g., using the statistical software R), and discuss the results. Note that some of the experiments may not terminate within reasonable time. You therefore need to specify a time-out value, and stop the algorithm after the specified number of generations.

# *Chapter 2*

## *Experiment 1-Runtime VS Mutation Rate*

### 2.1 Experiment Parameter

In order to investigate the relationship between runtime (i.e., number of generations multiplied with the population size $\lambda$) and mutation rate, we should control variables. The experiment parameters were used as shown below.

#### 2.1.1 Constant

##### 2.1.1.1 Bit-string Length

Bit-string length is defined as the individual genetic digits, it is represented by *n* in the source code. In this experiment, two hundred is chose as the value of bit-string length **(i.e., Bit-string Length (*n*)=200)**.

##### 2.1.1.2 Population Size

Population size (i.e., it is represented by $\lambda$) is the number of individuals in a population. The value of population size is defined as one hundred **(i.e., Population Size ($\boldsymbol{\lambda}$)=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)**.

#### 2.1.2 Variable

Mutation rate is represented by *chi/n* in the source code, and *chi* is the specification of mutation rate. The range of *chi* is defined from zero to three **(i.e., Mutation Rate (*chi*) $\in$ ( 0 , 3) )**.

### 2.2 Experiment Results and Analysis

In order to display the overall trends and each individual case, we divide the experiment into overall experiment and step by step experiment because the mutation rate has a big influence on runtime.

### 2.2.1 Boxplot of Results

Some of the experiments may not terminate within reasonable time. Therefore, a time-out value need to be specified, and stop the algorithm after the specified number of generations. In overall experiment, **time-out value is 10000**. In step by step experiment, **time-out value isn't set**. The boxplots of results are shown below.

#### 2.2.1.1 Overall Experiment

Each experiment has been repeated for 100 times. The results are shown as boxplots. In order to display the overall trend, the increments of 0.56 for *chi* was tried in this experiment **(i.e., Mutation Rate (*chi*) $\in$ Set {0.2 ,0.76 ,1.32 ,1.88 ,2.44 ,3.0 } )**.

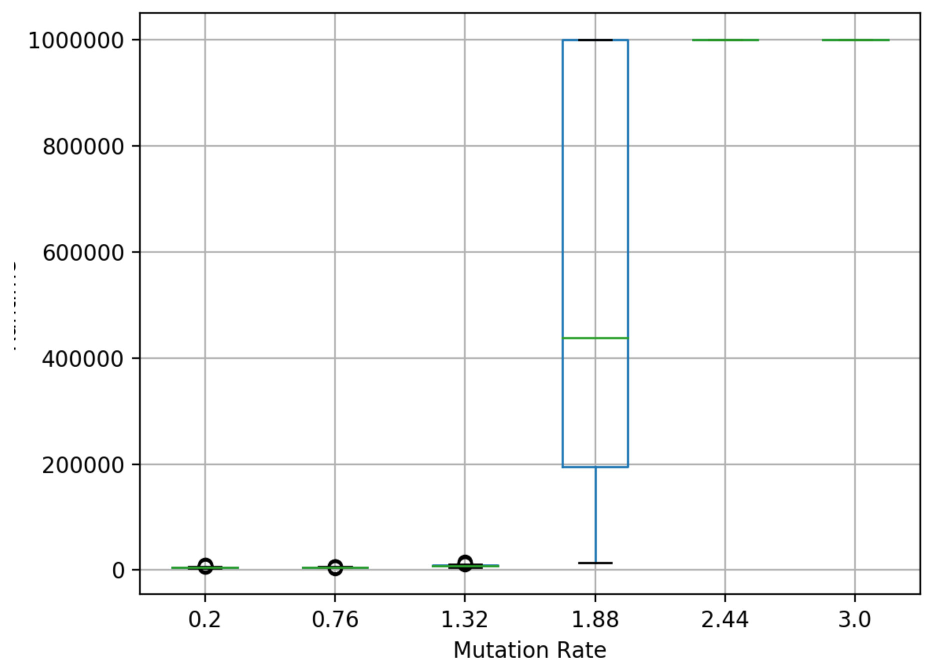


figure 1: Result of Overall Experiment

#### 2.2.1.2 Step by Step Experiment

Each experiment has been repeated for 100 times. The results are shown as boxplots. In order to display each individual case, the increments of 0.1 for *chi* was tried in this experiment **(i.e., Mutation Rate (*chi*) ∈ Set {0.2 ,0.3 ,0.4 ,0.5 ,0.6 ,0.7 ,0.8 ,0.9 ,1.0 ,1.1 ,1.2, 1.3,1.4,1.5,1.6,1.7,1.8,1.9,2.0 } )**.

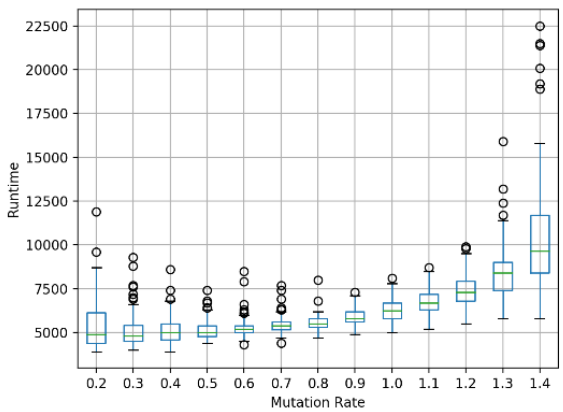


figure 2: chi varies from 0.2 to 1.4

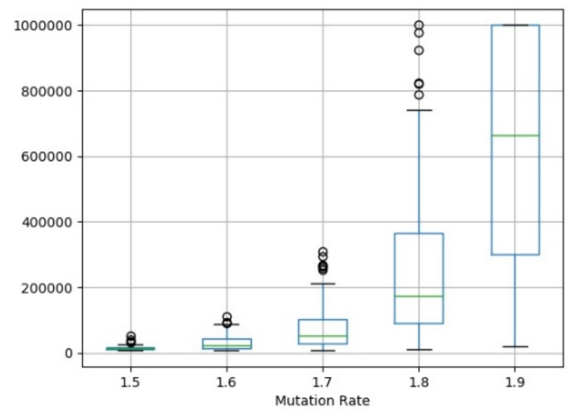


figure 3: chi varies from 1.5 to 1.9

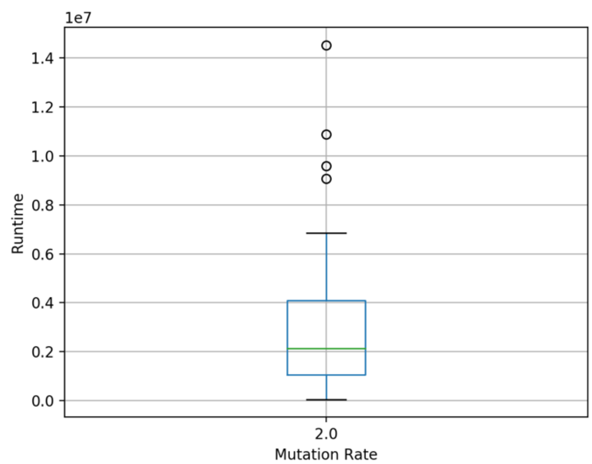


figure 4: chi value is 2.0

### 2.2.2 Analysis of Boxplot

The boxplots summarize the relationship between runtime and problem size.

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 have an approximate exponential growth trend with the increment of problem size. More specifically, when chi value is above 2.0, the runtime increased dramatically, even reached to ten million. Therefore, it is better to choose the value of chi under 2.0 to implement the simple genetic algorithms.

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.

# *Chapter 3*

## *Experiment 2-Runtime VS Problem Size*

### 3.1 Experiment Parameter

In order to investigate the relationship between runtime (i.e., number of generations multiplied with the population size λ) and problem size (i.e., Bit-string Length), 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 represented by *chi/n* in the source code, and *chi* is the specification of mutation rate. In this experiment, 0.6 is chose as the value of *chi* **(i.e., *chi*=0.6)**.

##### 3.1.1.2 Population Size

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

##### 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)**.

#### 3.1.2 Variable

Bit-string length is defined as the individual genetic digits, it is represented by *n* in the source code. The range of Bit-string length is defined from ten to two hundred **(i.e., Bit-string Length (*n*) $\in$ ( 10 , 200) )**.

### 3.2 Experiment Results and Analysis

Each experiment has been repeated for 100 times. **Time-out value is 10000**. The results are shown as boxplots. In order to display the overall trend and each individual case, the increments of 11 for bit-string length was tried in this experiment **(i.e., Bit-string Length (*n*) ∈ Set {11, 22, 33, 44, 55, 66, 77, 88, 99, 110, 121, 132, 143, 154, 165, 176,187,198} )**.

### 3.2.1 Boxplot of Results

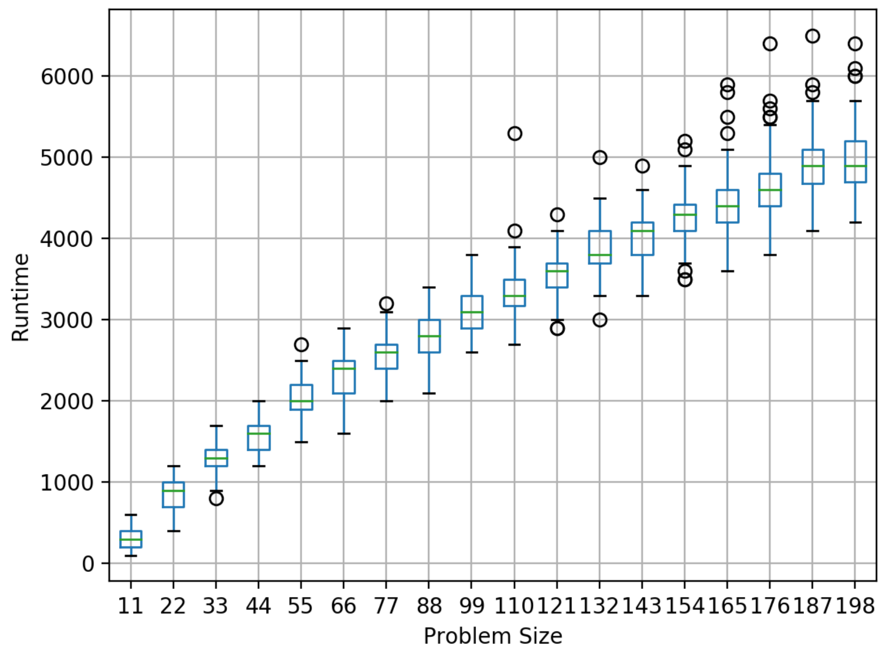


figure 5: Runtime VS Problem Size

### 3.2.2 Analysis of Boxplot

The boxplot summarizes the relationship between runtime and problem size.

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 have an approximate linear growth trend with the increment of problem size. It means the runtime may have a linear positive correlation with problem size.

It is clear that the height of boxplot increased with the increase of problem size. It reveals that when the problem size increases, the distribution of normal values becomes more and more dispersive.

Moreover, with the increment of problem size, the quantity of outliers becomes more and more. The quality of the initial population will affect the probability of outliers occurring. Therefore, it is more likely to get the worst or the best initial population when the problem size increases.

# *Chapter 4*

## *Experiment 3-Runtime VS Population Size*

### 4.1 Experiment Parameter

In order to investigate the relationship between runtime (i.e., number of generations multiplied with the population size $\lambda$) 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 represented by *chi/n* in the source code, and *chi* is the specification of mutation rate. In this experiment, 0.6 is chose as the value of *chi* **(i.e., *chi*=0.6)**.

##### 4.1.1.2 Bit-string Length

Bit-string length is defined as the individual genetic digits, it is represented by *n* in the source code. In this experiment, two hundred is chose as the value of bit-string length **(i.e., Bit-string Length (*n*)=200)**.

##### 4.1.1.3 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)**.

#### 4.1.2 Variable

Population size (i.e., it is represented by $\lambda$) is the number of individuals in a population. The range of population size is defined from ten to one thousand **(i.e., Population Size ($\lambda$) $\in$ [ 10 , 1000] )**.

### 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 ( $\lambda$ ) ∈ 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 } ). Time-out value is 10000**.

### 4.2.1 Boxplot of Results

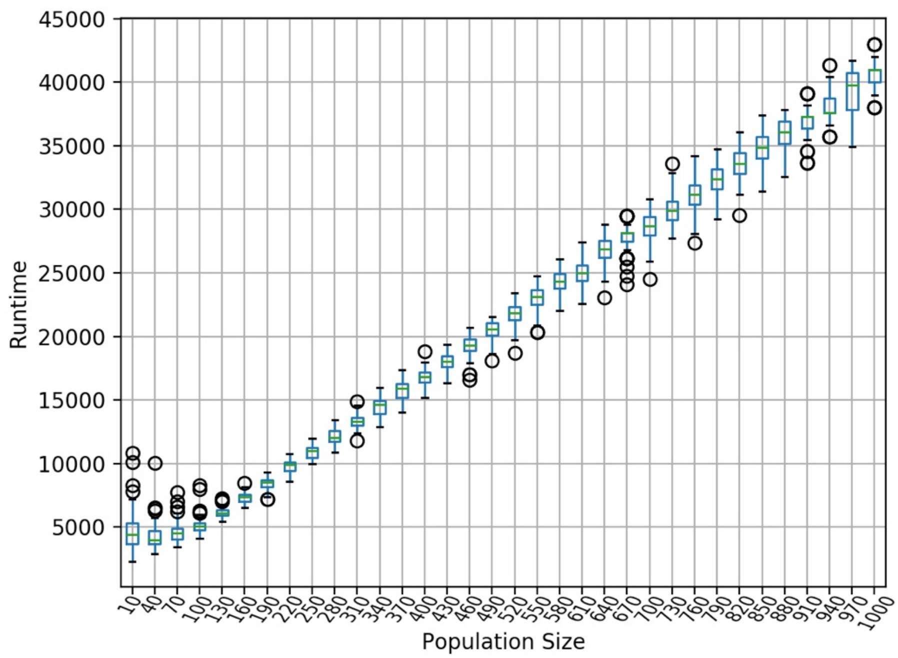


figure 6: Runtime VS Population Size

### 4.2.2 Analysis of Boxplot

The boxplot summarizes the relationship between runtime and population size.

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 have a linear upward trend with the increment of population size. It means the runtime may have a linear positive correlation with population size.

It is clear that before the height of boxplot increases with the increase of population size, the height of boxplot becomes smaller when population size varies from 10 to 220.It reveals that when the population size increases, the distribution of normal values becomes more and more concentrated before they are more and more dispersive.

Moreover, when population size is under 160, the outliers appear above the upper edge (i.e., the maximum value). After that, the outliers appear above the upper edge (i.e., the maximum value) or under the lower edge (i.e., the minimum value) randomly. It reveals that when the population size is small, the quality of the initial population will seriously affect the efficiency of finding the optimal solution.

# *Chapter 5*

# *Experiment 4-Runtime VS Tournament Size*

## 5.1 Experiment Parameter

In order to investigate the relationship between runtime (i.e., number of generations multiplied with the population size $\lambda$) and tournament size, 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 represented by *chi/n* in the source code, and *chi* is the specification of mutation rate. In this experiment, 0.6 is chose as the value of *chi* **(i.e., *chi*=0.6)**.

#### 5.1.1.2 Bit-string Length

Bit-string length is defined as the individual genetic digits, it is represented by *n* in the source code. In this experiment, two hundred is chose as the value of bit-string length **(i.e., Bit-string Length (*n*)=200)**.

#### 5.1.1.3 Population Size

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

### 5.1.2 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 , 5] )**.

## 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 1 for tournament size was tried in this experiment **(i.e., Tournament Size (*k*) $\in$ Set {2, 3, 4, 5} ). Time-out value is 10000**.

### 5.2.1 Boxplot of Results

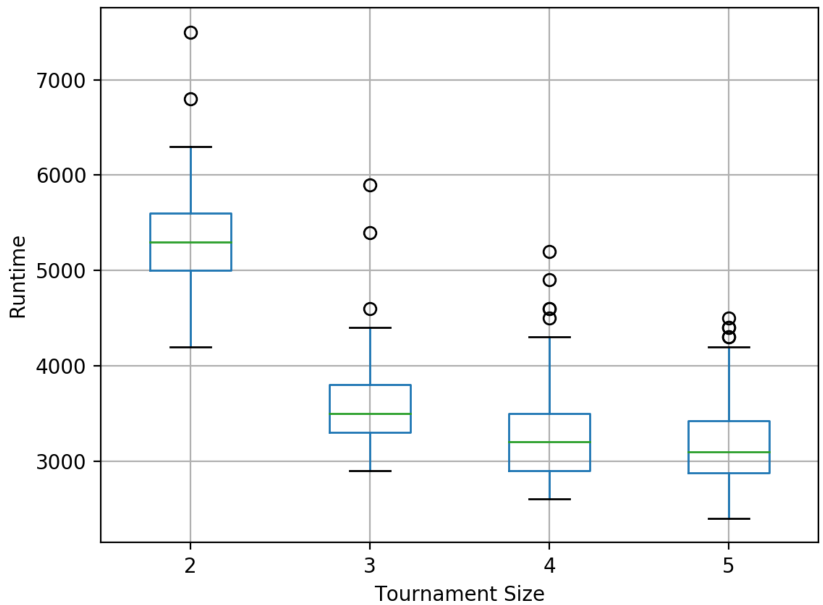


figure7: Runtime VS Tournament Size

### 5.2.2 Analysis of Boxplot

The boxplot summarizes the relationship between runtime and tournament size.

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 drop off rapidly with the increment of tournament size. It seems like that the runtime has an exponential distribution with tournament size.

It is clear that the height of 4 boxplots is a little bit large. It reveals that the distribution of normal values is dispersive.

Moreover, the outliers all appear above the upper edge (i.e., the maximum value) in the four boxplots. Because the quality of the initial population will affect the probability of outliers occurring. Therefore, the quality of initial population may be worse.

# *Appendix A*

# *Runtime vs Mutation Rate*

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

def fitness_function(tournament_list):
    sum_list = list()
    for m in range(k):
        a = tournament_list[m]
        a = str(a)
        a = list(map(int, a))
        fitness_value = sum(a)
        sum_list.append(fitness_value)
    output_z = tournament_list[sum_list.index(max(sum_list))]
    # output_z=str(output_z)
    return output_z



def mutation(bits_x,chi,n):
    bits_x = list(bits_x)
    mutation_rate = chi / n
    output_z = bits_x[:]
    # print(bits_x)
    for j in range(n):
        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'
                # print(bits_x[j])
```

```
        # print(''.join(output_z))
        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)
        # print(z)
        return z


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


def encoding(population_size,n):
        origin_population=list()
        upper_bound=math.pow(2,n)
        for i in range(population_size):
                bitstring_number=int(random.randint(0,upper_bound))
                bitstring='{0:b}'.format(bitstring_number)
                bitstring_number=bitstring.zfill(n)
```

```
            origin_population.append(bitstring_number)
        # print(origin_population)
        return origin_population


if __name__=='__main__':


    n=200
    k=2
    population_size=100
    repetitions=80
    chi_array=np.linspace(0,3,6)


    fbest=n
    upper_bounder = int(math.pow(2, n)) - 1
    xbest = '{0:b}'.format(upper_bounder)

    origin_population=encoding(population_size,n)
    population=origin_population[:]
    origin_population_result=origin_population[:]
    # print(origin_population)
    dic_runtime = dict()


    for c in chi_array:
        chi=c

        list_runtime = list()
    #最后生成 100 个结果
        for j in range(repetitions):
            population= origin_population_result[:]
            # print(population)
            t = 0
            z = str()
            flag=True
            while flag:
                origin_population=population[:]
```

```
                # print(origin_population)
                population=list()
                t=t+1
                if t==10000:
                    break
                for h in range(population_size):
                    x=tournament_selection(origin_population,k)
                    y=tournament_selection(origin_population,k)
                    z=crossover(mutation(x,chi,n), mutation(y,chi,n),n)
                    if z==xbest :
                        # print(z)
                        flag=False
                    population.append(z)
            list_runtime.append(population_size*t)
            print(t)
        dic_runtime[chi]=list_runtime

    data=pd.DataFrame(dic_runtime)
    data.boxplot()
    plt.ylabel('Runtime')
    plt.xlabel('Mutation Rate')
    plt.savefig('figure3.jpg')
```

# *Appendix B*

# *Runtime vs Problem Size*

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

def fitness_function(tournament_list):
    sum_list = list()
    for m in range(k):
        a = tournament_list[m]
        a = str(a)
        a = list(map(int, a))
        fitness_value = sum(a)
        sum_list.append(fitness_value)
    output_z = tournament_list[sum_list.index(max(sum_list))]
    # output_z=str(output_z)
    return output_z



def mutation(bits_x,chi,n):
    bits_x = list(bits_x)
    mutation_rate = chi / n
    output_z = bits_x[:]
    # print(bits_x)
    for j in range(n):
        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'
                # print(bits_x[j])
    # print(''.join(output_z))
    return output_z
```

```python
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)
    # print(z)
    return z


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


def encoding(population_size,n):
    origin_population=list()
    upper_bound=math.pow(2,n)
    for i in range(population_size):
        bitstring_number=int(random.randint(0,upper_bound))
        bitstring='{0:b}'.format(bitstring_number)
        bitstring_number=bitstring.zfill(n)
        origin_population.append(bitstring_number)
    # print(origin_population)
    return origin_population


if __name__=='__main__':
```

```
chi=0.6
k=2
population_size=100
repetitions=100
# chi_array=np.linspace(0,3,6)






# print(origin_population)
dic_runtime = dict()


for c in range(11,201,11):
    n=c
    fbest = n
    upper_bounder = int(math.pow(2, n)) - 1
    xbest = '{0:b}'.format(upper_bounder)
    list_runtime = list()

    origin_population = encoding(population_size, n)
    population = origin_population[:]
    origin_population_result = origin_population[:]
#最后生成 100 个结果
    for j in range(repetitions):
        population= origin_population_result[:]
        # print(population)
        t = 0
        z = str()
        flag=True
        while flag:
            origin_population=population[:]
            # print(origin_population)
            population=list()
            t=t+1
            if t==10000:
                break
            for h in range(population_size):
                x=tournament_selection(origin_population,k)
                y=tournament_selection(origin_population,k)
                z=crossover(mutation(x,chi,n), mutation(y,chi,n),n)
                if z==xbest :
                    # print(z)
                    flag=False
```

```
                    population.append(z)
                list_runtime.append(population_size*t)
                print(t)
            dic_runtime[c]=list_runtime

    data=pd.DataFrame(dic_runtime)
    data.boxplot()
    plt.ylabel('Runtime')
    plt.xlabel('Problem Size')
    plt.show()
```

# *Appendix C*

# *Runtime vs Tournament Size*

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author：Janet Chou

import random
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def fitness_function(tournament_list):
    sum_list = list()
    for m in range(k):
        a = tournament_list[m]
        a = str(a)
        a = list(map(int, a))
        fitness_value = sum(a)
        sum_list.append(fitness_value)
    output_z = tournament_list[sum_list.index(max(sum_list))]
    # output_z=str(output_z)
    return output_z



def mutation(bits_x,chi,n):
    bits_x = list(bits_x)
    mutation_rate = chi / n
    output_z = bits_x[:]
    # print(bits_x)
    for j in range(n):
        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'
                # print(bits_x[j])
    # print(''.join(output_z))
    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)
    # print(z)
    return z


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


def encoding(population_size,n):
    origin_population=list()
    upper_bound=math.pow(2,n)
    for i in range(population_size):
        bitstring_number=int(random.randint(0,upper_bound))
```

```
            bitstring='{0:b}'.format(bitstring_number)
            bitstring_number=bitstring.zfill(n)
            origin_population.append(bitstring_number)
    # print(origin_population)
    return origin_population


if __name__=='__main__':
    # try:
    #       n,chi,k,population_size,repetitions=input('please input bitstring
length,mutation rate,tournament size, population size'
    #                                                                         ' and
repetitions respectively（use commas to separate them）：').split(',')
    # except Exception as e:
    #       print('you enter the wrong information,please start again')


    n=200
    chi=0.6
    population_size=100
    repetitions=100
    # chi_array=np.linspace(0.2,3,6)


    fbest=n
    upper_bounder = int(math.pow(2, n)) - 1
    xbest = '{0:b}'.format(upper_bounder)

    origin_population=encoding(population_size,n)
    population=origin_population[:]
    origin_population_result=origin_population[:]
    # print(origin_population)
    dic_runtime = dict()


    for c in range(2,6):
        k=c

        list_runtime = list()
```

```
    #最后生成 100 个结果
        for j in range(repetitions):
            population= origin_population_result[:]
            # print(population)
            t = 0
            z = str()
            flag=True
            while flag:
                origin_population=population[:]
                # print(origin_population)
                population=list()
                t=t+1
                if t==10000:
                    break
                for h in range(population_size):
                    x=tournament_selection(origin_population,k)
                    y=tournament_selection(origin_population,k)
                    z=crossover(mutation(x,chi,n), mutation(y,chi,n),n)
                    if z==xbest :
                        # print(z)
                        flag=False
                    population.append(z)
            list_runtime.append(population_size*t)
            print(t)
        dic_runtime[k]=list_runtime

    data=pd.DataFrame(dic_runtime)
    data.boxplot()
    plt.ylabel('Runtime')
    plt.xlabel('Tournament Size')
    plt.show()
```

# *Appendix D*

# *Runtime vs Population Size*

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

def fitness_function(tournament_list):
    sum_list = list()
    for m in range(k):
        a = tournament_list[m]
        a = str(a)
        a = list(map(int, a))
        fitness_value = sum(a)
        sum_list.append(fitness_value)
    output_z = tournament_list[sum_list.index(max(sum_list))]
    # output_z=str(output_z)
    return output_z



def mutation(bits_x,chi,n):
    bits_x = list(bits_x)
    mutation_rate = chi / n
    output_z = bits_x[:]
    # print(bits_x)
    for j in range(n):
        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'
```

```
                # print(bits_x[j])
    # print(''.join(output_z))
    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)
    # print(z)
    return z


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


def encoding(population_size,n):
    origin_population=list()
    upper_bound=math.pow(2,n)
    for i in range(population_size):
        bitstring_number=int(random.randint(0,upper_bound))
        bitstring='{0:b}'.format(bitstring_number)
```

```
            bitstring_number=bitstring.zfill(n)
            origin_population.append(bitstring_number)
        # print(origin_population)
        return origin_population


if __name__=='__main__':


        n=200
        k=2
        chi=0.6
        repetitions=100
        # chi_array=np.linspace(0.2,3,6)


        fbest=n
        upper_bounder = int(math.pow(2, n)) - 1
        xbest = '{0:b}'.format(upper_bounder)


        # print(origin_population)
        dic_runtime = dict()


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

            origin_population = encoding(population_size, n)
            population = origin_population[:]
            origin_population_result = origin_population[:]

        #最后生成 100 个结果
            for j in range(repetitions):
                population= origin_population_result[:]
                # print(population)
                t = 0
                z = str()
```

```
            flag=True
            while flag:
                origin_population=population[:]
                # print(origin_population)
                population=list()
                t=t+1
                if t==10000:
                    break
                for h in range(population_size):
                    x=tournament_selection(origin_population,k)
                    y=tournament_selection(origin_population,k)
                    z=crossover(mutation(x,chi,n), mutation(y,chi,n),n)
                    if z==xbest :
                        # print(z)
                        flag=False
                    population.append(z)
            list_runtime.append(population_size*t)
            print(t)
        dic_runtime[c]=list_runtime

    data=pd.DataFrame(dic_runtime)
    data.boxplot()
    plt.ylabel('Runtime')
    plt.xlabel('Population Size')
    plt.show()
```