



# UNIVERSITY OF BIRMINGHAM

**School of Computer Science**

**Nature-Inspired Search and Optimisation**

**PG Aff Computer Science**

**Author : Wenyi Zhou**

**Student ID : 1887939**

## ***Table of Contents***

<i>Chapter 1 Exercise4 Pseudo-code of My Algorithm .....</i>	<i>1</i>
<i>Chapter 2 Exercise5-Average Attendance VS Mutation Rate.....</i>	<i>3</i>
<i>Chapter 3 Exercise5-Average Attendance VS Number of States .....</i>	<i>6</i>
<i>Chapter 4 Exercise5-Average Attendance VS Population Size .....</i>	<i>9</i>
<i>Chapter 5 Exercise5-Average Attendance VS Generations .....</i>	<i>13</i>
<i>Appendix A Average Attendance vs Mutation Rate .....</i>	<i>16</i>
<i>Appendix B Average Attendance vs Number of States.....</i>	<i>21</i>
<i>Appendix C Average Attendance vs Generations .....</i>	<i>27</i>
<i>Appendix D Average Attendance vs Population Size .....</i>	<i>32</i>

# Chapter 1

## Exercise 4 Pseudo-code of My Algorithm

### Algorithm EI-Farol Problem

**Require:** the number of states in the strategies  $h \in \mathbb{N}$

**Require:** Mutation rate  $chi \in (0,1)$

**Require:** Population size  $\lambda \in \mathbb{N}$

**Require:** the number of weeks to simulate per generation  $weeks \in \mathbb{N}$

**Require:** the number of generations to run the simulation  $max\_t \in \mathbb{N}$

```
0: Initialize population  $P_0$ 
1:   for  $i=0$  to  $\lambda-1$  do
2:     initialize each individual  $S$  with a strategy
3:     randomly initialize a current state  $m$  ( $m \in (0, h)$ )
4:     randomly initialize a vector  $z = (z_0, \dots, z_{h-1})$  of "attendance" probabilities
5:     randomly initialize a state transition matrix  $A = (a_{ij})$  in case the bar is crowded (i.e.,  $\sum(A_i) = 1$  for  $i=0$  to  $h-1$ )
6:     randomly initialize a state transition matrix  $B = (b_{ij})$  in case the bar is uncrowded (i.e.,  $\sum(B_i) = 1$  for  $i=0$  to  $h-1$ )
7:   Return individual  $S = (m, z, A, B)$ 
8:   end for
9: Return  $P_0 = (S_0, \dots, S_{\lambda-1})$ 
10: Initialize everyone's payoff with value 0, store them in a payoff list with the length of population size
11: for  $tg=0$  . . . until  $t = max\_t$  do
12:   Evaluate the population according to the payoff
13:   for  $tw=0$  to  $weeks$  do
14:     Calculate the decision list  $d$  of individual whether to go to the bar or not and the number total_people of
    individuals who attending the bar now
15:     for  $i=0$  to  $\lambda-1$  do
16:       Get current state  $m = P_w[i][0]$ 
17:       Get "attendance" probabilities vector  $z = P_w[i][1]$ 
18:       randomly generate a probability
19:       if (probability  $\leq z_m$ ) do
20:         Add element 1 in decision list  $d$ 
21:       else do
22:         Add element 0 in decision list  $d$ 
23:       end if
24:     end for
25:     total_people =  $\sum(d)$ 
26:   Return decision list  $d$ , total_people
27:   if (total_people  $< 60\% \times \lambda$ ) do
28:     crowded = 0 # (i.e., 0 means the bar is uncrowded)
29:   else do
30:     crowded = 1 # (i.e., 1 means the bar is crowded)
31:   end if
32:   for  $c=0$  to  $\lambda-1$  do
33:     if (decision list  $d[c] \neq$  crowded) do
34:       payoff list $[c] =$  payoff list $[c] + 1$ 
35:     end if
36:   end for
37:   for  $e=0$  to  $\lambda-1$  do
38:      $A = P_w[e][2]$ 
39:      $B = P_w[e][3]$ 
40:     Get current state  $m = P_w[e][0]$ 
```

```

41:         if (crowded) do
42:             Choose  $A[m]$  as transition probability distribution
43:         else do
44:             Choose  $B[m]$  as transition probability distribution
45:         end if
46:         Sample from a distribution over the integers  $\{0, \dots, n-1\}$ , where distribution is represented by a vector of
47:          $n$  probabilities.
48:         Get the length of probability distribution  $n$ 
49:         Initialize an array with the integers  $\{0, \dots, n-1\}$ 
50:         Sum the probabilities sum_prob according to the probability distribution
51:         Cumsum the probability distribution to get a cumsum_prob array
52:         randomly generate a probability (i.e., probability  $\in (0, \text{sum\_prob})$ )
53:         for  $i=0$  to  $n-1$  do
54:             if ( probability  $\leq$  cumsum_prob[ $i$ ] ) do
55:                 Return array[ $i$ ]
56:             end if
57:         end for
58:         Get the next_state according to the method from line 46 to line 53
59:         Replace state  $P_{tg}[e][0] = \text{next\_state}$ 
60:     end for
61:     output(tw, tg, total_people, crowded, decision list d)
62: end for
63: Return payoff list
64: for  $j=0$  to  $\lambda/2-1$  do
65:     Roulette Selection
66:     Sum_payoff = sum(payoff list)
67:     payoff_probability_distribution = payoff list / Sum_payoff
68:     Get the index of high payoff individual call the method from line 46 to line 53
69:     parent =  $P_{tg}[\text{index}]$ 
70:     Return parent
71:     father = Roulette Selection( $P_{tg}$ )
72:     mother = Roulette Selection( $P_{tg}$ )
73:     Mutation
74:     for  $e=0$  to 3 do
75:         randomly generate a probability
76:         if ( probability  $\leq$  mutation_rate chi ) do
77:             Reconstruct individual  $S[i]$  call method from line 2 to line 7
78:             parent [ $i$ ] =  $S[i]$ 
79:         end if
80:     end for
81:     Return parent
82:     Crossover
83:     randomly generate a cross_locus  $\in (h/2-2, h/2+2)$ 
84:     offspring_1 = father
85:     offspring_2 = mother
86:     for  $c=1$  to 3 do
87:         offspring_1[ $c$ ][cross_locus: $h$ ] = mother[ $c$ ][cross_locus: $h$ ]
88:         offspring_2[ $c$ ][cross_locus: $h$ ] = father [ $c$ ][cross_locus: $h$ ]
89:     end for
90:     Return offspring_1, offspring_2
91:      $P_{tg+1}(j), P_{tg+1}(j+\lambda/2) = \text{Crossover}(\text{Mutation}(\text{father}), \text{Mutation}(\text{mother}))$ 
92: end for
93: Return  $P_{tg}$ 
94: end for

```

## Chapter 2

### ***Exercise 5-Average Attendance VS Mutation Rate***

#### 2.1 Experiment Parameter

In order to investigate the relationship between average attendance over several weeks and mutation rate, we should control variables. The experiment parameters were used as shown below.

##### 2.1.1 Constant

###### 2.1.1.1 Number of States

Number of states is defined as the length of individual genes, it is represented by  $h$  in the source code. In this experiment, ten is chose as the value of states (**i.e., Number of States ( $h$ )=10**).

###### 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 ( $\lambda$ )=700**).

###### 2.1.1.3 Generations

Generations are defined as  $max\_t$  in the source code. Ten is the value of generations in the experiment (**i.e., Generations ( $max\_t$ )=10**).

###### 2.1.1.4 Weeks

Weeks are the number of weeks to simulate per generation. Twenty is the value of weeks in the experiment (**i.e., Weeks =20**).

#### 2.1.2 Variable

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

## 2.2 Experiment Results and Analysis

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

### 2.2.1 Boxplot of Results

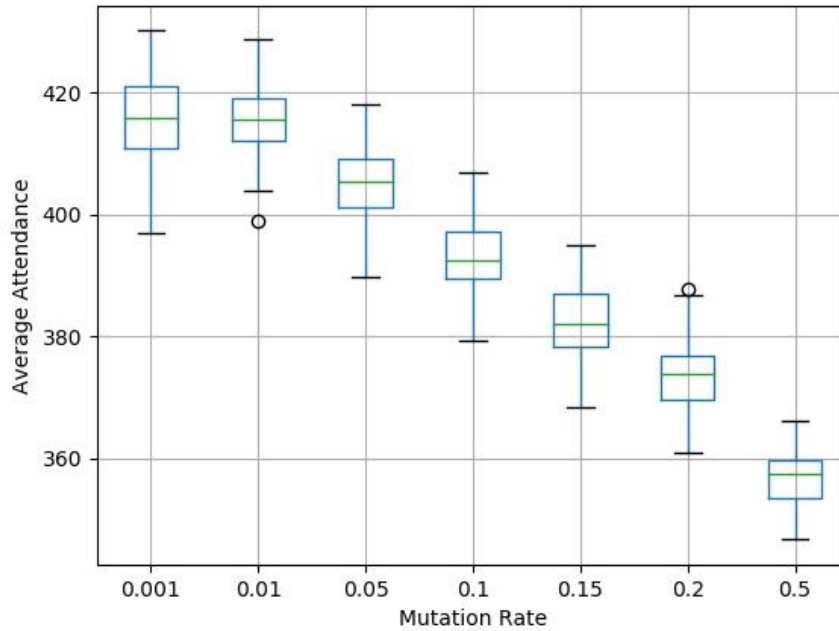


figure 1: Average Attendance VS Mutation Rate

### 2.2.2 Analysis of Boxplot

The boxplots summarize the relationship between the impact of mutation rate on the quality (i.e., average attendance over several weeks) of the solutions obtained.

Overall, when mutation rate is below 0.01, 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. The average attendance rate is close to 60% (i.e., divide the average attendance by population size). It seems that my co-evolutionary algorithm leads the population to as efficient utilization of the bar as possible. However, when the mutation rate is above 0.01, they have an approximate linear decreasing trend with the decrement of mutation rate. It means the average attendance may have a linear negative correlation with mutation rate.

It is clear that the outliers hardly appear in the boxplots during this experiment. It may show that the experiment is very stable.

In conclusion, setting mutation rate to 0.01 is the best choice. It can not only get the efficient utilization of the bar, but also can keep the genes diverse.

## Chapter 3

### ***Experiment 2-Average Attendance VS Number of States***

#### 3.1 Experiment Parameter

In order to investigate the relationship between average attendance over several weeks and number of states (i.e.,  $h$ ), 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, 0.01 is chose as the value of *mutation rate* (i.e., ***Mutation Rate=0.01***).

###### 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 seven hundred (i.e., **Population Size ( $\lambda$ )=700**).

###### 3.1.1.3 Generations

Generations is defined as *max\_t* in the source code. Ten is the value of generations in the experiment (i.e., **Generations (*max\_t*)=10**).

###### 3.1.1.4 Weeks

Weeks are the number of weeks to simulate per generation. Twenty is the value of weeks in the experiment (i.e., **Weeks =20**).

#### 3.1.2 Variable

Number of states is defined as the length of individual genes, it is represented by  $h$  in the source code. The range of states is defined from ten to fifty (i.e., **Number of States ( $h$ )  $\in$  ( 10 , 50 )**).



## 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 10 for number of states was tried in this experiment (i.e., **Number of States ( $h$ )**  $\in$  **Set {10, 20, 30, 40, 50}** ).

### 3.2.1 Boxplot of Results

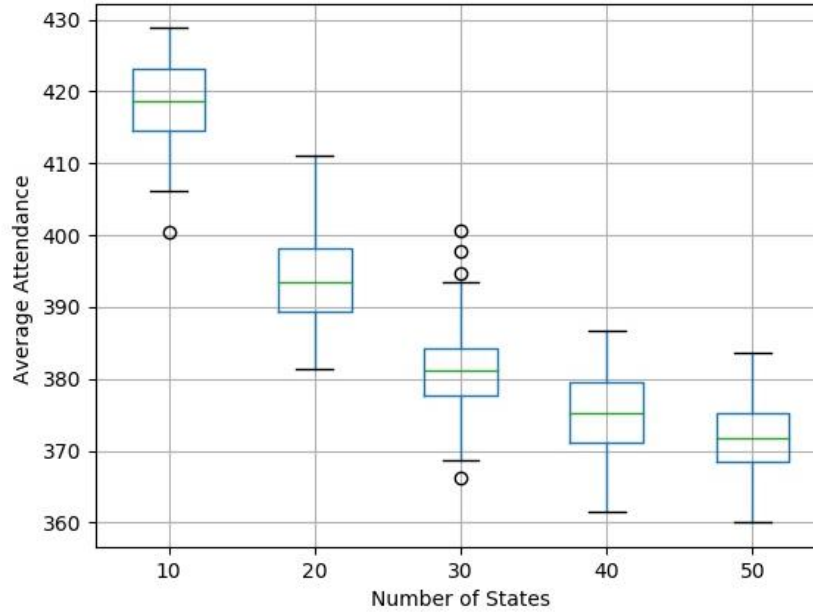


figure 2: Average Attendance VS Number of States

### 3.2.2 Analysis of Boxplot

The boxplots summarize the relationship between the impact of states on the quality (i.e., average attendance over several weeks) 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 drop off a little bit rapidly with the increment of number of states. It seems like that the average attendance has an exponential distribution with number of states. It may reveal that the algorithm needs to evolve

more generations to get the best solution when the genes become more.

It is clear that the outliers hardly appear in the boxplots during this experiment. It may show that the experiment is very stable.

In conclusion, when the states value is 10, the average attendance rate is close to 60% (i.e., divide the average attendance by population size). It seems that in this experiment, 10 states will let my co-evolutionary algorithm leads the population to as efficient utilization of the bar as possible with other parameters set above.

## Chapter 4

### ***Exercise 5-Average Attendance VS Population Size***

#### 4.1 Experiment Parameter

In order to investigate the relationship between average attendance over several weeks and population size, we should control variables. The experiment parameters were used as shown below.

##### 4.1.1 Constant

###### 4.1.1.1 Mutation Rate

Mutation rate is essential for the behavior of the algorithm. In this experiment, 0.01 is chose as the value of *mutation rate* (i.e., ***Mutation Rate=0.01***).

###### 4.1.1.2 Number of States

Number of states is defined as the length of individual genes, it is represented by  $h$  in the source code. In this experiment, ten is chose as the value of states (i.e., **Number of States ( $h$ )=10**).

###### 4.1.1.3 Generations

Generations are defined as *max\_t* in the source code. Ten is the value of generations in the experiment (i.e., **Generations (*max\_t*)=10**).

###### 4.1.1.4 Weeks

Weeks are the number of weeks to simulate per generation. Twenty is the value of weeks in the experiment (i.e., **Weeks =20**).

#### 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 and five hundred (**i.e., Population Size ( $\lambda$ )  $\in [10, 1410]$**  ).

## 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 100 for population size was tried in this experiment (**i.e., Population Size ( $\lambda$ )  $\in$  Set  $\{10, 110, 210, 310, 410, 510, 610, 710, 810, 910, 1010, 1110, 1210, 1310, 1410\}$**  ).

### 4.2.1 Boxplot of Results

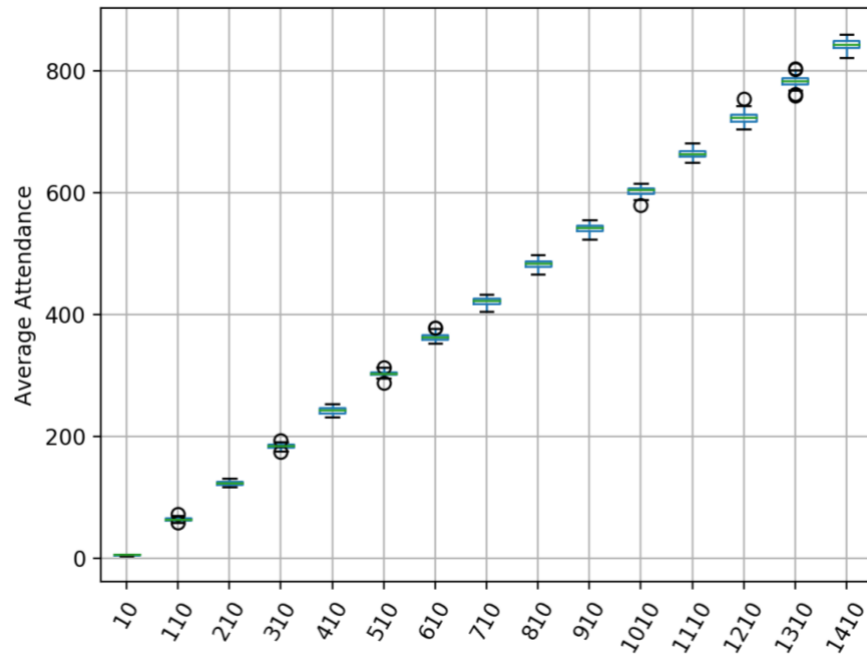


figure 3: Average Attendance VS Population Size

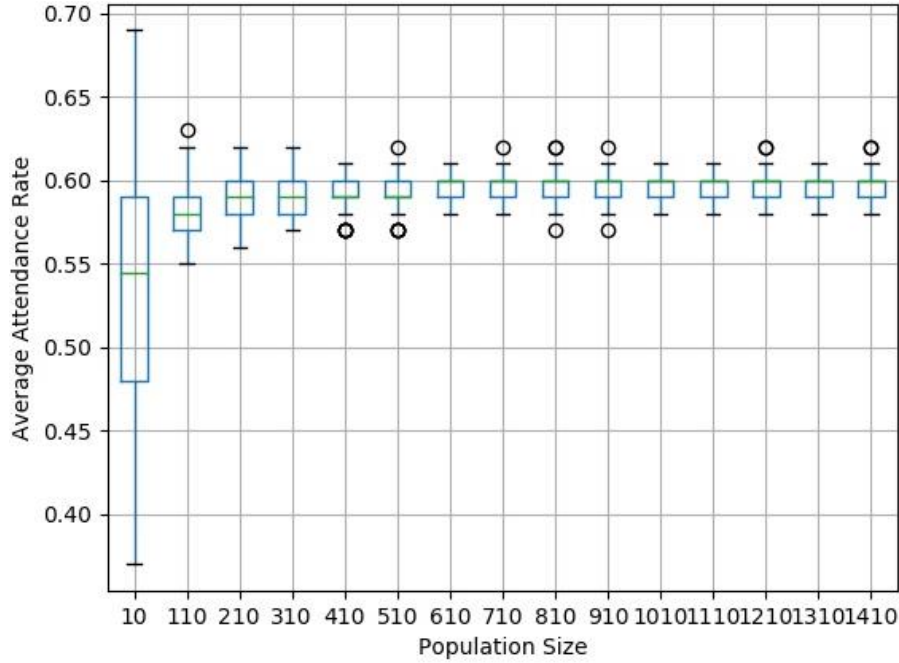


figure 4: Average Attendance Rate VS Population Size

#### 4.2.2 Analysis of Boxplot

The boxplots summarize the relationship between the impact of population size on the quality (i.e., average attendance over several weeks) of the solutions obtained.

In order to show the average attendance and the utilization of the bar, two boxplots were plot in figure 3 and figure 4.

Overall, in figure 3 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 average attendance may have a linear positive correlation with population size. In figure 4, they all fluctuates slightly, the average attendance rate is close to 60%. It seems the algorithm always gets a best solution. It may also reveal that the population size hardly has influence on the efficiency of the algorithm.

It is clear that the outliers hardly appear in the boxplots during this experiment. It may show that the experiment is very stable.

In conclusion, Although the population size become more and more, the average attendance rate is always close to 60% (i.e., divide the average attendance by population size). It seems that in this experiment, the population size doesn't affect the efficiency of the algorithm.

## Chapter 5

### ***Exercise 5-Average Attendance VS Generations***

#### 5.1 Experiment Parameter

In order to investigate the relationship between average attendance over several weeks and generations, we should control variables. The experiment parameters were used as shown below.

##### 5.1.1 Constant

###### 5.1.1.1 Mutation Rate

Mutation rate is essential for the behavior of the algorithm. In this experiment, 0.01 is chose as the value of *mutation rate* (i.e., ***Mutation Rate=0.01***).

###### 5.1.1.2 Number of States

Number of states is defined as the length of individual genes, it is represented by  $h$  in the source code. In this experiment, ten is chose as the value of states (i.e., **Number of States ( $h$ )=10**).

###### 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 thousand (i.e., **Population Size ( $\lambda$ )=1000**).

###### 5.1.1.4 Weeks

Weeks are the number of weeks to simulate per generation. Twenty is the value of weeks in the experiment (i.e., **Weeks =20**).

#### 5.1.2 Variable

Generations are defined as *max\_t* in the source code. The range of generation is defined from two to twenty five (i.e., **Generations**

$(max\_t) \in [1, 22]$  ).

## 5.2 Experiment Results and Analysis

Each experiment has been repeated for 100 times. The results are shown as boxplots. In order to display the overall trends and each individual case, the increments of 3 for generation was tried in this experiment (i.e.,  $Generations(max\_t) \in \text{Set } \{1, 4, 7, 10, 13, 16, 19, 22\}$  ).

### 5.2.1 Boxplot of Results

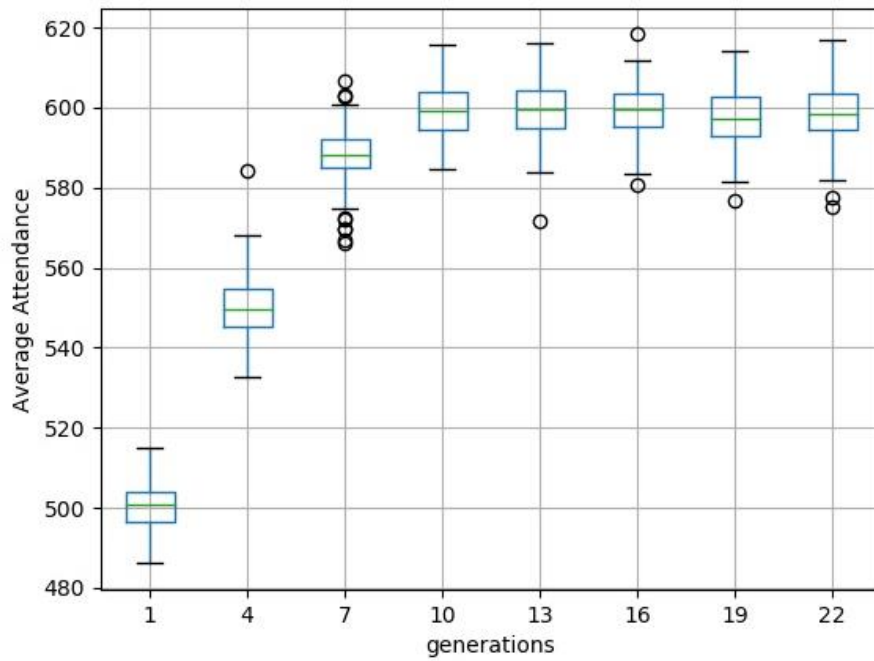


figure 5: Average Attendance VS Generations

### 5.2.2 Analysis of Boxplot

The boxplots summarize the relationship between the impact of generations on the quality (i.e., average attendance over several weeks) 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) all fluctuates slightly with the generations varying from 10 to 22 (the average attendance is about 600 and the average attendance rate is about 60%), they have an approximate linear growth trend with the increment of generations. It means this algorithm not only can improve the quality of population with the increment of generations, but also can let the attendance converge to the bar capacity.

It is clear that the outliers hardly appear in the boxplots during this experiment. It may show that the experiment is very stable.

In conclusion, before the average attendance rate is close to 60%, the average attendance will increase with the increment of generations. Therefore, in order to save runtime, an appropriate number of generations should be chosen. And in this experiment, setting generations to 10 is the most efficient choice to get the best solution.

## Appendix A

### *Average Attendance vs Mutation Rate*

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # Author: Janet Chou
4  import numpy as np
5  import time
6  import argparse
7  from scipy import stats
8  import pandas as pd
9  import matplotlib.pyplot as plt
10
11
12  def Sample(prob, times):
13      n=len(prob)
14      integers=np.arange(n)
15      dice=stats.rv_discrete(values=(integers,prob))
16      sample_result=dice.rvs(size=times)
17      return sample_result
18
19
20  def strategy_handle(strategy):
21      attendance_prob = np.array([])
22      A = np.array([])
23      B = np.array([])
24      for i in range(h):
25          attendance_prob = np.append(attendance_prob, strategy[i *
26 (2 * h + 1)])
27          A = np.concatenate((A, strategy[1 + i * (2 * h + 1):h + 1 +
28 i * (2 * h + 1)]))
29          B = np.concatenate((B, strategy[h + 1 + i * (2 * h + 1):2 *
30 h + 1 +
31 i * (2 * h + 1)]))
32      A = A.reshape(h, h)
33      B = B.reshape(h, h)
34      return attendance_prob,A,B
35
36
37  def one_step_desicion(state,crowded,attendance_prob,A,B):
38      if crowded:
39          next_state=Sample(A[state],times=1)
40      else:
41          next_state=Sample(B[state],times=1)
42      if np.random.random()<=attendance_prob[next_state]:
43          decision=1
44      else:
45          decision = 0
46      return decision,next_state
47
48
```

```

49 def all_decisions(origin_population):
50     decision_list=list()
51     for i in origin_population:
52         state=i[0]
53         P=i[1]
54         if np.random.random() <= P[state]:
55             decision_list.append(1)
56         else:
57             decision_list.append(0)
58     bar_people=sum(decision_list)
59     return bar_people,decision_list
60
61
62 def initial_individual(h):
63     state=np.random.randint(0,h)
64     P=np.random.random(h)
65     A=np.zeros((h,h))
66     B=np.zeros((h,h))
67     for i in range(h):
68         random_list=np.random.randint(0,100,h)
69         Sum=sum(random_list)
70         A[i]=random_list/Sum
71     for j in range(h):
72         random_list = np.random.randint(0, 100, h)
73         Sum = sum(random_list)
74         B[j] = random_list / Sum
75     return state,P,A,B
76
77
78 def initial_population(population_size,h):
79     origin_population=list()
80     for i in range(population_size):
81         strategy = list(initial_individual(h))
82         origin_population.append(strategy)
83     return origin_population
84
85
86 def fitness_function(weeks,origin_population,population_size):
87     bar_people_list=list()
88     payoff_list=[0]*population_size
89     for i in range(1,weeks+1):
90         bar_people, decision_list =
91 all_decisions(origin_population)
92         bar_people_list.append(bar_people)
93         if bar_people<0.6*population_size:
94             crowded=0
95         else:
96             crowded=1
97         for c in range(population_size):
98             if decision_list[c]!=crowded:
99                 payoff_list[c]+=1
100         for m in range(population_size):
101             # print(origin_population[i][0])
102             # print('-----')

```

```

1         A=origin_population[m][2]
10        B=origin_population[m][3]
2         if crowded:
10
3         origin_population[m][0]=Sample(A[origin_population[m][0]],times=1
10        )[0]
4         else:
10             origin_population[m][0] =
5         Sample(B[origin_population[m][0]], times=1)[0]
10             # print(origin_population[i][0])
6             #
10         print(str(i)+'\t'+str(t)+'\t'+str(bar people)+'\t'+str(crowded)+'
7         \t')
10         average attendance=sum(bar people list)/weeks
8
10         return payoff list,average attendance
9
11
0         def Roulette Selection(payoff,copy origin population):
11             Sum payoff=sum(payoff)
1             payoff array=np.array(payoff)
11             payoff distribution=payoff array/Sum payoff
2             select index=Sample(payoff distribution,times=1)[0]
11             # print(select index)
3             output=copy origin population[select index]
11             return output
4
11         def mutation(x,h,mutation rate):
5             strategy = list(initial individual(h))
11             for i in range(4):
6                 if np.random.random()<=mutation rate:
11                     x[i]=strategy[i]
7             return x
11
8         def crossover(x,y,h):
11             cross locus=np.random.randint(h/2-2,h/2+2)
9             z1=x[:].copy()
12             z2=y[:].copy()
0             for i in range(1, 4):
12                 z1[i]=x[i].copy()
1                 z2[i]=y[i].copy()
12             # print(cross locus)
2             z1[1][cross locus:h] = y[1][cross locus:h].copy()
12             z2[1][cross locus:h] = x[1][cross locus:h].copy()
3             for i in range(2,4):
12                 z1[i][cross_locus:h,:]=y[i][cross_locus:h,:].copy()
4                 z2[i][cross_locus:h,:]=x[i][cross_locus:h,:].copy()
12             return z1,z2
5
12
6
12         if __name__ == '__main__':
7             parser=argparse.ArgumentParser(description='manual to this
12             script')

```

```

8     parser.add_argument('-question',type=int,default=3)
12    parser.add_argument('-repetitions', type=int, default=5)
9     parser.add_argument('-prob', type=str, default='0 0 1 0')
13    parser.add_argument('-strategy', type=str, default='2 0.1 0.0
0 1.0 1.0 0.0
13    1.0 0.9 0.1 0.9 0.1')
1     parser.add_argument('-state',type=int,default=0)
13    parser.add_argument('-crowded', type=int, default=1)
2     parser.add_argument('-lamda','-lambda', type=int, default=150)
13    parser.add_argument('-ha',type=int,default=10)
3     parser.add_argument('-weeks', type=int, default=10)
13    parser.add_argument('-time budget', type=int, default=20)
4     args= parser.parse args()
13    question number=args.question
5
13    if question number==1:
6        prob = list(args.prob.split(' '))
13        prob = tuple(map(float, prob))
7        for i in range(args.repetitions):
13            output 1=Sample(prob,times=1)
8            print(output 1[0])
13
9        elif question number == 2:
14            strategy=list(map(eval,args.strategy.split(' ')))
0            state=args.state
14            crowded=args.crowded
1            h=int(strategy[0])
14            strategy=strategy[1:len(strategy)]
2            attendance prob, A, B=strategy handle(strategy)
14            for i in range(args.repetitions):
3
14            d,s=one step desicion(state,crowded,attendance prob,A,B)
4            print(str(d)+'\t'+str(s[0]))
14
5
14        else:
6            try:
14                population size = args.lamda
7                h=args.ha
14                weeks=args.weeks
8                time budget=args.time budget
14
9                mutation list = [0.00001, 0.0001, 0.001, 0.01, 0.02,
15    0.03, 0.04,
0                0.05,0.06, 0.07, 0.08, 0.09, 0.1]
15
1            t=0
15            dic_average_attend=dict()
2            for mutation rate in mutation_list:
15                list average attend=list()
3
15                for ex_times in range(100):
4
15    origin_population=initial_population(population_size,h)

```

```

5         start = time.clock()
15        fbest=0.6*population_size
6        while True:
15            copy_origin_population = origin_population[:]
7            for i in range(population_size):
15                copy_origin_population[i] =
8 origin_population[i].copy()
15
9                population=list()
16                t=t+1
0                payoff, average attendance =
16 fitness function(weeks,
1                origin population, population size)
16                end = time.clock()
2                runtime = end - start
16
3                if runtime > time budget:
16                    break
4
16                for j in range(int(population size/2)):
5
16 father=Roulette Selection(payoff,copy origin population)
6
16 mother=Roulette Selection(payoff,copy origin population)
7                x=mutation(father,h,mutation rate)
16                y=mutation(mother,h,mutation rate)
8                z1,z2=crossover(x,y,h)
16                population.append(z1)
9                population.append(z2)
17                middle=time.clock()
0                origin population=population[:]
17                list average attend.append(average attendance)
1                print(average attendance)
17                #
2 list average attend=list(map(float,list average attend))
17
3 dic average attend[mutation rate]=list average attend
17        data = pd.DataFrame(dic average attend)
4        data.plot(kind='box',rot=60,grid=True)
17        plt.ylabel('Average Attendance')
5        plt.xlabel('Mutation Rate')
17        plt.savefig('figure5.jpg')
6
17
7        except Exception as e:
17            print('time over!')
8

```

## Appendix B

### *Average Attendance vs Number of States*

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # Author: Janet Chou
4  import numpy as np
5  import time
6  import argparse
7  import pandas as pd
8  import matplotlib.pyplot as plt
9  from scipy import stats
10
11
12  def Sample(prob):
13      n = len(prob)
14      integers = np.arange(n)
15      all_prob = sum(prob)
16      total_prob = np.cumsum(prob)
17      dics = np.random.uniform(0, all_prob)
18      for i in range(n):
19          if dics <= total_prob[i]:
20              return integers[i]
21
22
23  def strategy_handle(strategy):
24      attendance_prob = np.array([])
25      A = np.array([])
26      B = np.array([])
27      for i in range(h):
28          attendance_prob = np.append(attendance_prob,
29 strategy[i * (2 * h + 1)])
30          A = np.concatenate((A, strategy[1 + i * (2 * h + 1):h
31 +
32 1 + i * (2 * h + 1)]))
33          B = np.concatenate((B, strategy[h + 1 + i * (2 * h +
34 1):2 * h +
35 1 + i * (2 * h + 1)]))
36          A = A.reshape(h, h)
37          B = B.reshape(h, h)
38          return attendance_prob, A, B
39
40
41  def one_step_desicion(state, crowded, attendance_prob, A, B):
42      if crowded:
43          next_state = Sample(A[state])
44      else:
45          next_state = Sample(B[state])
46      if np.random.random() <= attendance_prob[next_state]:
47          decision = 1
48      else:
```

```

49         decision = 0
50     return decision,next_state
51
52
53 def all_decisions(origin_population):
54     decision_list=list()
55     for i in origin_population:
56         state=i[0]
57         P=i[1]
58         if np.random.random() <= P[state]:
59             decision list.append(1)
60         else:
61             decision list.append(0)
62     bar people=sum(decision list)
63     return bar people,decision list
64
65
66 def initial_individual(h):
67     state=np.random.randint(0,h)
68     P=np.random.random(h)
69     A=np.zeros((h,h))
70     B=np.zeros((h,h))
71     for i in range(h):
72         random list=np.random.randint(0,100,h)
73         Sum=sum(random list)
74         A[i]=random list/Sum
75     for j in range(h):
76         random list = np.random.randint(0, 100, h)
77         Sum = sum(random list)
78         B[j] = random list / Sum
79     return state,P,A,B
80
81
82 def initial_population(population size,h):
83     origin population=list()
84     for i in range(population size):
85         strategy = list(initial individual(h))
86         origin population.append(strategy)
87     return origin population
88
89
90 def
91 fitness function(weeks,origin population,population size):
92     bar_people_list = list()
93     payoff_list=[0]*population_size
94     for i in range(0,weeks):
95         bar_people, decision_list =
96     all_decisions(origin population)
97         bar_people_list.append(bar_people)
98         if bar people<0.6*population size:
99             crowded=0
100        else:
101            crowded=1
102        for c in range(population_size):

```



```

103         if decision_list[c] != crowded:
104             payoff_list[c] += 1
105         for m in range(population_size):
106             # print(origin_population[i][0])
107             # print('-----')
108             A = origin_population[m][2]
109             B = origin_population[m][3]
110             if crowded:
111
112                 origin_population[m][0] = Sample(A[origin_population[m][0]])
113             else:
114                 origin_population[m][0] =
115                 Sample(B[origin_population[m][0]])
116             # print(origin_population[i][0])
117             decision_list = list(map(str, decision_list))
118             #
119             print(str(i) + '\t' + str(t) + '\t' + str(bar_people) + '\t' + str(crowde
120             d) +
121                   '\t' + (' '.join(decision_list)))
122             average_attendance = sum(bar_people_list) / weeks
123             return payoff_list, origin_population, average_attendance
124
125
126 def Roulette_Selection(payoff, copy_origin_population):
127     Sum_payoff = sum(payoff)
128     payoff_array = np.array(payoff)
129     payoff_distribution = payoff_array / Sum_payoff
130     select_index = Sample(payoff_distribution)
131     # print(select_index)
132     output = copy_origin_population[select_index]
133     return output
134
135 def mutation(x, h, mutation_rate):
136     strategy = list(initial_individual(h))
137     for i in range(4):
138         if np.random.random() <= mutation_rate:
139             x[i] = strategy[i]
140     return x
141
142 def crossover(x, y, h):
143     cross_locus = np.random.randint(h/2-2, h/2+2)
144     z1 = x[:]
145     z2 = y[:]
146     for i in range(1, 4):
147         z1[i] = x[i].copy()
148         z2[i] = y[i].copy()
149     # print(cross_locus)
150     z1[1][cross_locus:h] = y[1][cross_locus:h].copy()
151     z2[1][cross_locus:h] = x[1][cross_locus:h].copy()
152     for i in range(2, 4):
153         z1[i][cross_locus:h] = y[i][cross_locus:h].copy()
154         z2[i][cross_locus:h] = x[i][cross_locus:h].copy()
155     return z1, z2
156

```

```

157
158
159 if __name__ == '__main__':
160     parser=argparse.ArgumentParser(description='manual to this
161 script',add_help=False)
162     parser.add_argument('-question',type=int,default=3)
163     parser.add_argument('-repetitions', type=int, default=5)
164     parser.add_argument('-prob', type=str, default='0 0 1 0')
165     parser.add_argument('-strategy', type=str, default='2 0.1
166 0.0 1.0
167 1.0 0.0 1.0 0.9 0.1 0.9 0.1')
168     parser.add_argument('-state',type=int,default=1)
169     parser.add_argument('-crowded', type=int, default=0)
170     parser.add_argument('-lamda','-lambda', type=int,
171 default=700)
172     # parser.add_argument('-h',type=int,default=10)
173     parser.add_argument('-weeks', type=int, default=10)
174     parser.add_argument('-max t', type=int, default=10)
175     args= parser.parse_args()
176     question number=args.question
177
178     if question number==1:
179         prob = list(args.prob.split(' '))
180         prob = list(map(float, prob))
181         for i in range(args.repetitions):
182             output 1 = Sample(prob)
183             print(output 1)
184
185     elif question number == 2:
186         strategy=list(map(eval,args.strategy.split(' ')))
187         state=args.state
188         crowded=args.crowded
189         h=int(strategy[0])
190         strategy=strategy[1:len(strategy)]
191         attendance prob, A, B=strategy handle(strategy)
192         for i in range(args.repetitions):
193
194 d,s=one step desicion(state,crowded,attendance prob,A,B)
195         print(str(d)+'\t'+str(s))
196
197
198     else:
199
200         population_size = args.lamda
201         # h=args.h
202         weeks=args.weeks
203         time_budget=args.max_t
204
205
206         mutation rate=0.0001
207
208         dic_average_attend = dict()
209
210         for h in range(10,60,10):

```

```

211         list_average_attend = list()
212
213         for ex_times in range(80):
214             t = 0
215
216             origin_population=initial_population(population_size,h)
217             start = time.clock()
218             while True:
219                 copy_origin_population =
220                 origin_population[:]
221                 for i in range(population_size):
222                     copy_origin_population[i] =
223                     origin_population[i][:]
224                     end = time.clock()
225                     runtime = end - start
226                     # print(runtime)
227                     if t== time_budget-1:
228                         # print(runtime)
229                         break
230                     population=list()
231
232                     payoff, origin_population,average_attendance
233 = fitness_function(weeks,
234                     origin_population, population_size)
235                     #
236                     list_average_attend.append(average_attendance)
237                     # print(average_attendance)
238                     # print('*****')
239                     # print(bar_people)
240                     # Sum_payoff=sum(payoff)
241                     # print(Sum_payoff)
242                     t = t + 1
243                     for j in range(int(population_size/2)):
244
245                     father=Roulette_Selection(payoff,copy_origin_population)
246
247                     mother=Roulette_Selection(payoff,copy_origin_population)
248                     x=mutation(father,h,mutation_rate)
249                     y=mutation(mother,h,mutation_rate)
250                     z1,z2=crossover(x,y,h)
251                     population.append(z1)
252                     population.append(z2)
253                     middle=time.clock()
254                     # if (middle-start)>time_budget:
255                     #     print(runtime)
256                     #     break
257                     origin_population=population[:]
258                     list_average_attend.append(average_attendance)
259                     print(average_attendance)
260                     # list_average_attend = list(map(float,
261                     list_average_attend))
262                     print('-----')
263
264

```

```
265         #
266     list_average_attend=list(map(float,list_average_attend))
267     dic_average_attend[h] = list_average_attend
268     data = pd.DataFrame(dic_average_attend)
269     data.plot(kind='box', grid=True)
270     plt.ylabel('Average Attendance ')
271     plt.xlabel('Number of States')
272     plt.savefig('figure33.jpg')
273
274
275
```

## Appendix C

### *Average Attendance vs Generations*

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # Author: Janet Chou
4  import numpy as np
5  import time
6  import argparse
7  import pandas as pd
8  import matplotlib.pyplot as plt
9  from scipy import stats
10
11
12  def Sample(prob):
13      n = len(prob)
14      integers = np.arange(n)
15      all_prob = sum(prob)
16      total_prob = np.cumsum(prob)
17      dics = np.random.uniform(0, all_prob)
18      for i in range(n):
19          if dics <= total_prob[i]:
20              return integers[i]
21
22
23  def strategy_handle(strategy):
24      attendance_prob = np.array([])
25      A = np.array([])
26      B = np.array([])
27      for i in range(h):
28          attendance_prob = np.append(attendance_prob, strategy[i *
29 (2 * h + 1)])
30          A = np.concatenate((A, strategy[1 + i * (2 * h + 1):h + 1
31 +
32 i * (2 * h + 1)]))
33          B = np.concatenate((B, strategy[h + 1 + i * (2 * h + 1):2
34 * h +
35 1 + i * (2 * h + 1)]))
36      A = A.reshape(h, h)
37      B = B.reshape(h, h)
38      return attendance_prob, A, B
39
40
41  def one_step_desicion(state, crowded, attendance_prob, A, B):
42      if crowded:
43          next_state = Sample(A[state])
44      else:
45          next_state = Sample(B[state])
46      if np.random.random() <= attendance_prob[next_state]:
47          decision = 1
48      else:
```

```

49         decision = 0
50     return decision,next_state
51
52
53 def all_decisions(origin_population):
54     decision_list=list()
55     for i in origin_population:
56         state=i[0]
57         P=i[1]
58         if np.random.random() <= P[state]:
59             decision list.append(1)
60         else:
61             decision list.append(0)
62     bar people=sum(decision list)
63     return bar people,decision list
64
65
66 def initial_individual(h):
67     state=np.random.randint(0,h)
68     P=np.random.random(h)
69     A=np.zeros((h,h))
70     B=np.zeros((h,h))
71     for i in range(h):
72         random list=np.random.randint(0,100,h)
73         Sum=sum(random list)
74         A[i]=random list/Sum
75     for j in range(h):
76         random list = np.random.randint(0, 100, h)
77         Sum = sum(random list)
78         B[j] = random list / Sum
79     return state,P,A,B
80
81
82 def initial_population(population size,h):
83     origin population=list()
84     for i in range(population size):
85         strategy = list(initial individual(h))
86         origin population.append(strategy)
87     return origin population
88
89
90 def fitness_function(weeks,origin population,population size):
91     bar people list = list()
92     payoff_list=[0]*population_size
93     for i in range(0,weeks):
94         bar_people, decision_list =
95     all_decisions(origin_population)
96         bar_people_list.append(bar_people)
97         if bar_people<0.6*population_size:
98             crowded=0
99         else:
100             crowded=1
101         for c in range(population_size):
102             if decision_list[c]!=crowded:

```

```

103         payoff_list[c]+=1
104     for m in range(population_size):
105         A=origin_population[m][2]
106         B=origin_population[m][3]
107         if crowded:
108
109 origin_population[m][0]=Sample(A[origin_population[m][0]])
110         else:
111             origin_population[m][0] =
112 Sample(B[origin_population[m][0]])
113             # print(origin_population[i][0])
114             decision_list = list(map(str, decision_list))
115             #
116 print(str(i)+'\t'+str(t)+'\t'+str(bar_people)+'\t'+str(crowded)+
117       '\t'+('\t'.join(decision_list)))
118         average_attendance = sum(bar_people_list) / weeks
119     return payoff_list,origin_population,average_attendance
120
121
122 def Roulette_Selection(payoff,copy_origin_population):
123     Sum_payoff=sum(payoff)
124     payoff_array=np.array(payoff)
125     payoff_distribution=payoff_array/Sum_payoff
126     select_index=Sample(payoff_distribution)
127     # print(select_index)
128     output=copy_origin_population[select_index]
129     return output
130
131 def mutation(x,h,mutation_rate):
132     strategy = list(initial_individual(h))
133     for i in range(4):
134         if np.random.random()<=mutation_rate:
135             x[i]=strategy[i]
136     return x
137
138 def crossover(x,y,h):
139     cross_locus=np.random.randint(h/2-2,h/2+2)
140     z1=x[:]
141     z2=y[:]
142     for i in range(1, 4):
143         z1[i]=x[i].copy()
144         z2[i]=y[i].copy()
145     # print(cross_locus)
146     z1[1][cross_locus:h] = y[1][cross_locus:h].copy()
147     z2[1][cross_locus:h] = x[1][cross_locus:h].copy()
148     for i in range(2,4):
149         z1[i][cross_locus:h,:]=y[i][cross_locus:h,:].copy()
150         z2[i][cross_locus:h,:]=x[i][cross_locus:h,:].copy()
151     return z1,z2
152
153 if __name__ == '__main__':
154     parser=argparse.ArgumentParser(description='manual to this
155 script',add_help=False)
156     parser.add_argument('-question',type=int,default=3)

```

```

157     parser.add_argument('-repetitions', type=int, default=5)
158     parser.add_argument('-prob', type=str, default='0 0 1 0')
159     parser.add_argument('-strategy', type=str, default='2 0.1
160 0.0 1.0
161 1.0 0.0 1.0 0.9 0.1 0.9 0.1')
162     parser.add_argument('-state', type=int, default=1)
163     parser.add_argument('-crowded', type=int, default=0)
164     parser.add_argument('-lamda', '-lambda', type=int,
165 default=1000)
166     parser.add_argument('-h', type=int, default=10)
167     parser.add_argument('-weeks', type=int, default=10)
168     # parser.add_argument('-max t', type=int, default=20)
169     args = parser.parse_args()
170     question number=args.question
171
172     if question number==1:
173         prob = list(args.prob.split(' '))
174         prob = list(map(float, prob))
175         for i in range(args.repetitions):
176             output 1 = Sample(prob)
177             print(output 1)
178
179     elif question number == 2:
180         strategy=list(map(eval,args.strategy.split(' ')))
181         state=args.state
182         crowded=args.crowded
183         h=int(strategy[0])
184         strategy=strategy[1:len(strategy)]
185         attendance prob, A, B=strategy handle(strategy)
186         for i in range(args.repetitions):
187
188 d,s=one step desicion(state,crowded,attendance prob,A,B)
189         print(str(d)+'\t'+str(s))
190
191
192     else:
193
194         population size = args.lamda
195         h=args.h
196         weeks=args.weeks
197         # time budget=args.max t
198
199
200         mutation_rate=0.0001
201
202         dic_average_attend = dict()
203
204         for time_budget in range(2,25,3):
205             list_average_attend = list()
206
207             for ex_times in range(100):
208                 t = 0
209
210 origin_population=initial_population(population_size,h)

```



```

211         start = time.clock()
212         while True:
213             copy_origin_population = origin_population[:]
214             for i in range(population_size):
215                 copy_origin_population[i] =
216 origin_population[i][:]
217             end = time.clock()
218             runtime = end - start
219             # print(runtime)
220             if t== time_budget-1:
221                 # print(runtime)
222                 break
223             population=list()
224
225             payoff, origin population,average attendance =
226 fitness function(weeks,
227                 origin population, population size)
228             #
229 list average attend.append(average attendance)
230             # print(average attendance)
231             # print('*****')
232             # print(bar people)
233             # Sum payoff=sum(payoff)
234             # print(Sum payoff)
235             t = t + 1
236             for j in range(int(population size/2)):
237
238 father=Roulette Selection(payoff,copy origin population)
239
240 mother=Roulette Selection(payoff,copy origin population)
241             x=mutation(father,h,mutation rate)
242             y=mutation(mother,h,mutation rate)
243             z1,z2=crossover(x,y,h)
244             population.append(z1)
245             population.append(z2)
246             middle=time.clock()
247             origin population=population[:]
248             list average attend.append(average attendance)
249             print(average attendance)
250             print('-----')
251
252
253 list average attend=list(map(float,list average attend))
254             dic_average_attend[time_budget-1] =
255 list_average_attend
256             data = pd.DataFrame(dic_average_attend)
257             data.plot(kind='box', grid=True)
258             plt.ylabel('Average Attendance')
259             plt.xlabel('generations')
260             plt.savefig('figure11.jpg')

```

## Appendix D

### *Average Attendance vs Population Size*

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # Author: Janet Chou
4  import numpy as np
5  import time
6  import argparse
7  import pandas as pd
8  import matplotlib.pyplot as plt
9  from scipy import stats
10
11
12  def Sample(prob):
13      n = len(prob)
14      integers = np.arange(n)
15      all_prob = sum(prob)
16      total_prob = np.cumsum(prob)
17      dics = np.random.uniform(0, all_prob)
18      for i in range(n):
19          if dics <= total_prob[i]:
20              return integers[i]
21
22
23  def strategy_handle(strategy):
24      attendance_prob = np.array([])
25      A = np.array([])
26      B = np.array([])
27      for i in range(h):
28          attendance_prob = np.append(attendance_prob,
29 strategy[i * (2 * h + 1)])
30          A = np.concatenate((A, strategy[1 + i * (2 * h +
31 1):h + 1 + i * (2 * h + 1)]))
32          B = np.concatenate((B, strategy[h + 1 + i * (2 * h +
33 1):2 * h +
34 1 + i * (2 * h + 1)]))
35      A = A.reshape(h, h)
36      B = B.reshape(h, h)
37      return attendance_prob,A,B
38
39
40  def one_step_desicion(state,crowded,attendance_prob,A,B):
41      if crowded:
42          next_state=Sample(A[state])
43      else:
44          next state=Sample(B[state])
45      if np.random.random()<=attendance_prob[next_state]:
46          decision=1
47      else:
48          decision = 0
```

```

49     return decision,next_state
50
51
52 def all_decisions(origin_population):
53     decision_list=list()
54     for i in origin_population:
55         state=i[0]
56         P=i[1]
57         if np.random.random() <= P[state]:
58             decision_list.append(1)
59         else:
60             decision list.append(0)
61     bar people=sum(decision list)
62     return bar people,decision list
63
64
65 def initial_individual(h):
66     state=np.random.randint(0,h)
67     P=np.random.random(h)
68     A=np.zeros((h,h))
69     B=np.zeros((h,h))
70     for i in range(h):
71         random list=np.random.randint(0,100,h)
72         Sum=sum(random list)
73         A[i]=random list/Sum
74     for j in range(h):
75         random list = np.random.randint(0, 100, h)
76         Sum = sum(random list)
77         B[j] = random list / Sum
78     return state,P,A,B
79
80
81 def initial_population(population size,h):
82     origin population=list()
83     for i in range(population size):
84         strategy = list(initial individual(h))
85         origin population.append(strategy)
86     return origin population
87
88
89 def
90 fitness function(weeks,origin population,population size):
91     bar people list = list()
92     payoff_list=[0]*population_size
93     for i in range(0,weeks):
94         bar_people, decision_list =
95 all_decisions(origin_population)
96         bar_people_list.append(bar_people)
97         if bar_people<0.6*population_size:
98             crowded=0
99         else:
100             crowded=1
101     for c in range(population_size):
102         if decision_list[c]!=crowded:

```

```

103         payoff_list[c]+=1
104     for m in range(population_size):
105         # print(origin_population[i][0])
106         # print('-----')
107         A=origin_population[m][2]
108         B=origin_population[m][3]
109         if crowded:
110
111     origin_population[m][0]=Sample(A[origin_population[m][0]])
112     else:
113         origin_population[m][0] =
114     Sample(B[origin_population[m][0]])
115     # print(origin_population[i][0])
116     decision_list = list(map(str, decision_list))
117     # print(str(i)+'\t'+str(t)+'\t'+str(bar_people)+'\t'+
118     str(crowded)+'\t'+('\t'.join(decision_list)))
119     average_attendance = sum(bar_people_list) / weeks
120     return payoff_list,origin_population,average_attendance
121
122
123 def Roulette_Selection(payoff,copy_origin_population):
124     Sum_payoff=sum(payoff)
125     payoff_array=np.array(payoff)
126     payoff_distribution=payoff_array/Sum_payoff
127     select_index=Sample(payoff_distribution)
128     # print(select_index)
129     output=copy_origin_population[select_index]
130     return output
131
132 def mutation(x,h,mutation_rate):
133     strategy = list(initial_individual(h))
134     for i in range(4):
135         if np.random.random()<=mutation_rate:
136             x[i]=strategy[i]
137     return x
138
139 def crossover(x,y,h):
140     cross_locus=np.random.randint(h/2-2,h/2+2)
141     z1=x[:]
142     z2=y[:]
143     for i in range(1, 4):
144         z1[i]=x[i].copy()
145         z2[i]=y[i].copy()
146     # print(cross_locus)
147     z1[1][cross_locus:h] = y[1][cross_locus:h].copy()
148     z2[1][cross_locus:h] = x[1][cross_locus:h].copy()
149     for i in range(2,4):
150         z1[i][cross_locus:h,:]=y[i][cross_locus:h,:].copy()
151         z2[i][cross_locus:h,:]=x[i][cross_locus:h,:].copy()
152     return z1,z2
153
154 if __name__ == '__main__':
155     parser=argparse.ArgumentParser(description='manual to this
156     script',add_help=False)

```

```

157     parser.add_argument('-question',type=int,default=3)
158     parser.add_argument('-repetitions', type=int, default=5)
159     parser.add_argument('-prob', type=str, default='0 0 1 0')
160     parser.add_argument('-strategy', type=str, default='2 0.1
161     0.0 1.0 1.0 0.0 1.0 0.9 0.1 0.9 0.1')
162     parser.add_argument('-state',type=int,default=1)
163     parser.add_argument('-crowded', type=int, default=0)
164     # parser.add_argument('-lamda','-lambda', type=int,
165     default=1000)
166     parser.add_argument('-h',type=int,default=10)
167     parser.add_argument('-weeks', type=int, default=10)
168     parser.add_argument('-max t', type=int, default=10)
169     args= parser.parse_args()
170     question number=args.question
171
172     if question number==1:
173         prob = list(args.prob.split(' '))
174         prob = list(map(float, prob))
175         for i in range(args.repetitions):
176             output 1 = Sample(prob)
177             print(output 1)
178
179     elif question number == 2:
180         strategy=list(map(eval,args.strategy.split(' ')))
181         state=args.state
182         crowded=args.crowded
183         h=int(strategy[0])
184         strategy=strategy[1:len(strategy)]
185         attendance prob, A, B=strategy handle(strategy)
186         for i in range(args.repetitions):
187
188     d,s=one step desicion(state,crowded,attendance prob,A,B)
189         print(str(d)+'\t'+str(s))
190
191     else:
192
193         # population size = args.lamda
194         h=args.h
195         weeks=args.weeks
196         time budget=args.max t
197
198
199         mutation rate=0.0001
200
201         dic_average_attend = dict()
202
203         for population_size in range(10,1500,100):
204             list_average_attend = list()
205
206             for ex times in range(80):
207                 t = 0
208
209         origin_population=initial_population(population_size,h)
210         start = time.clock()

```

```

211         while True:
212             copy_origin_population =
213             origin_population[:]
214             for i in range(population_size):
215                 copy_origin_population[i] =
216                 origin_population[i][:]
217                 end = time.clock()
218                 runtime = end - start
219                 # print(runtime)
220                 if t== time_budget-1:
221                     # print(runtime)
222                     break
223                 population=list()
224
225                 payoff, origin population,average attendance
226 =
227                 fitness function(weeks, origin population,
228 population size)
229                 #
230                 t = t + 1
231                 for j in range(int(population size/2)):
232
233                 father=Roulette Selection(payoff,copy origin population)
234
235                 mother=Roulette Selection(payoff,copy origin population)
236                 x=mutation(father,h,mutation rate)
237                 y=mutation(mother,h,mutation rate)
238                 z1,z2=crossover(x,y,h)
239                 population.append(z1)
240                 population.append(z2)
241                 middle=time.clock()
242                 origin population=population[:]
243
244                 list average attend.append('%.2f'%(average attendance/populat
245 ion size))
246
247                 print('%.2f'%(average attendance/population size))
248                 list average attend = list(map(float,
249 list average attend))
250                 print('-----')
251
252                 dic average attend[population size] =
253 list average attend
254                 data = pd.DataFrame(dic_average_attend)
255                 data.plot(kind='box', grid=True)
256                 plt.ylabel('Average Attendance Rate')
257                 plt.xlabel('Population Size')
258                 plt.savefig('figure0.jpg')
259
260

```