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 EI-Farol Problem

Require: the number of states in the strategies $h \in N$
Require: Mutation rate $chi \in (0,1)$
Require: Population size $\lambda \in N$
Require: the number of weeks to simulate per generation $weeks \in N$
Require: the number of generations to run the simulation $max\_t \in N$

```
Initialize population P_0
    for i=0 to λ-1 do
        initialize each individual S with a strategy
            randomly initialize a current state m (m ∈ (0, h) )
            randomly initialize a vector z= (z_0 ,⋯,z_{h-1})of“attendance” probabilities
            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)
            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)
        Return individual S=(m,z,A,B)
    end for
Return P_0 =(S_0,⋯,S_{λ-1})
Initialize everyone's payoff with value 0, store them in a payoff list with the length of population size
for tg=0 . . . until t=max_t do
    Evaluate the population according to the payoff
        for tw=0 to weeks do
            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
                for i=0 to λ-1 do
                    Get current state m= P_tg[i][0]
                    Get“attendance” probabilities vector z= P_tg[i][1]
                    randomly generate a probability
                    if (probability ⩽ z_m ) do
                        Add element 1 in decision list d
                    else do
                        Add element 0 in decision list d
                    end if
                end for
                total_people=sum(d)
            Return decision list d, total_people
            if (total_people < 60%*λ) do
                crowded=0    # (i.e., 0 means the bar is uncrowded)
            else do
                crowded=1    # (i.e., 1 means the bar is crowded)
            end if
            for c=0 to λ-1 do
                if (decision list d[c]≠ crowded) do
                    payoff list[c]= payoff list[c]+1
                end if
            end for
            for e=0 to λ-1 do
                A= P_tg[e][2]
                B= P_tg[e][3]
                Get current state m= P_tg[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,⋯,n-1 },where distribution is represented by a vector of
n probabilities.
47:                         Get the length of probability distribution n
48:                         Initialize an array with the integers { 0,⋯,n-1 }
49:                         Sum the probabilities sum_prob according to the probability distribution
47:                         Cumsum the probability distribution to get a cumsum_prob array
48:                         randomly generate a probability (i.e., probability∈(0, sum_prob ))
49:                         for i=0 to n-1 do
50:                             if（ probability ≤ cumsum_prob[i] ) do
51:                                 Return array[i]
52:                             end if
53:                         end for
54:                 Get the next_state according to the method from line 46 to line 53
55:                 Replace state P_tg[e][0]= next_state
56:             end for
57:             output(tw， tg， total_people， crowded， decision list d)
57:         end for
58:     Return payoff list
59:     for j =0 to λ/2-1 do
60:         Roulette Selection
61:             Sum_payoff = sum(payoff list)
62:             payoff_probability_distribution= payoff list / Sum_payoff
63:             Get the index of high payoff individual call the method from line 46 to line 53
64:             parent= P_tg [index]
65:         Return parent
66:         father= Roulette Selection(P_tg)
67:         mother= Roulette Selection(P_tg)
68:         Mutation
69:             for e=0 to 3 do
70:                 randomly generate a probability
71:                 if（ probability ≤ mutation_rate chi ) do
72:                     Reconstruct individual S [i] call method from line 2 to line 7
73:                     parent [i]= S [i]
74:                 end if
75:             end for
76:         Return parent
77:         Crossover
78:             randomly generate a cross_locus ∈(h/2-2, h/2+2)
79:             offspring_1=father
80:             offspring_2=mother
81:             for c=1 to 3 do
82:                 offspring_1[c][ cross_locus:h]= mother[c][ cross_locus:h]
83:                 offspring_2[c][ cross_locus:h]= father [c][ cross_locus:h]
84:             end for
85:         Return offspring_1, offspring_2
86:         P_tg+1(j), P_tg+1(j+λ/2) = Crossover(Mutation(father),Mutation(mother))
87:     end for
88:     Return P_tg
89: 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 λ) is the number of individuals in a population. The value of population size is defined as one hundred **(i.e., Population Size (λ)=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 ∈ Set {** 0.001, 0.01, 0.05, 0.1, 0.15, 0.2, 0.5 }**).**

2.2.1 Boxplot of Results

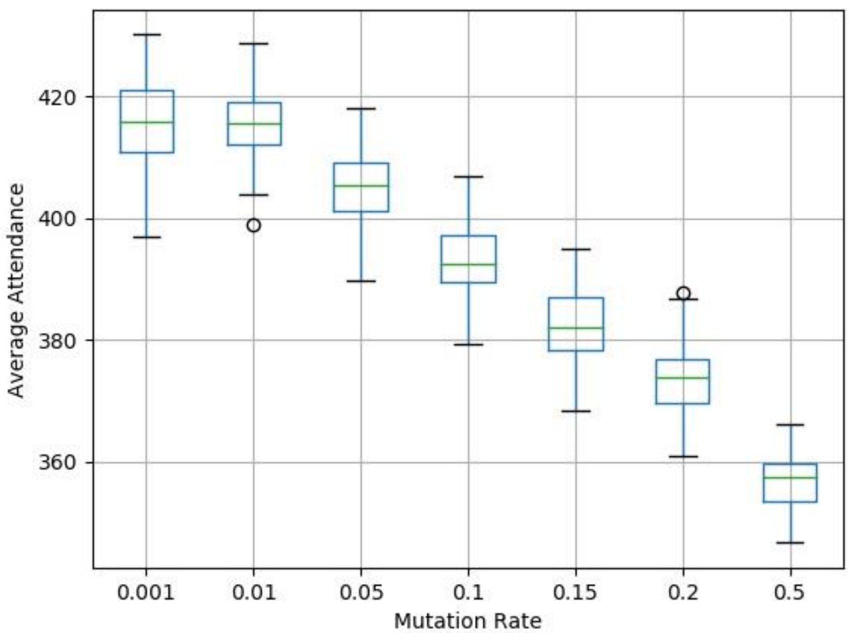

figure 1: Average Attendance VS Mutation Rate

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 λ) is the number of individuals in a population. The value of population size is defined as seven hundred **(i.e., Population Size (λ)=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

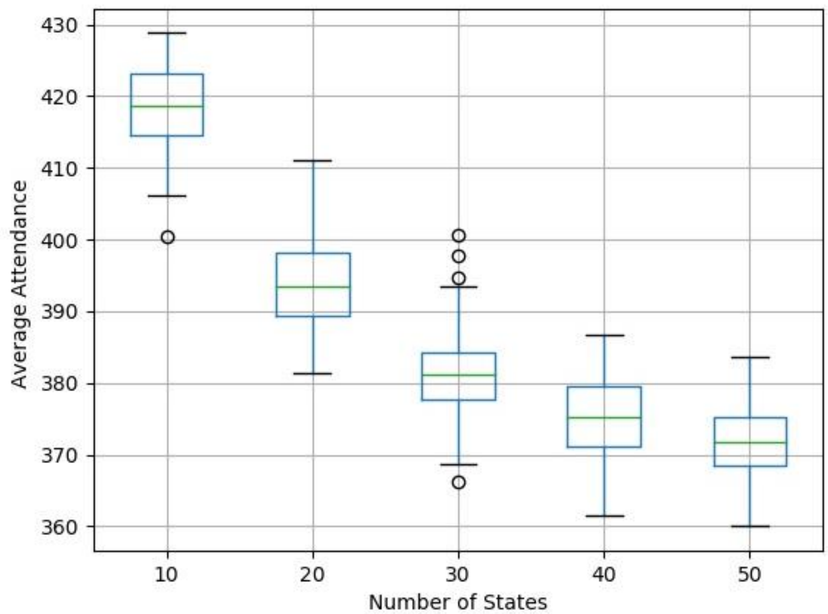

figure 2: Average Attendance VS Number of States

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 λ) 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 (λ) $\in$ [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 (λ) $\in$ Set {10, 110, 210, 310, 410, 510, 610, 710, 810, 910, 1010, 1110, 1210 , 1310, 1410 })**.

4.2.1 Boxplot of Results

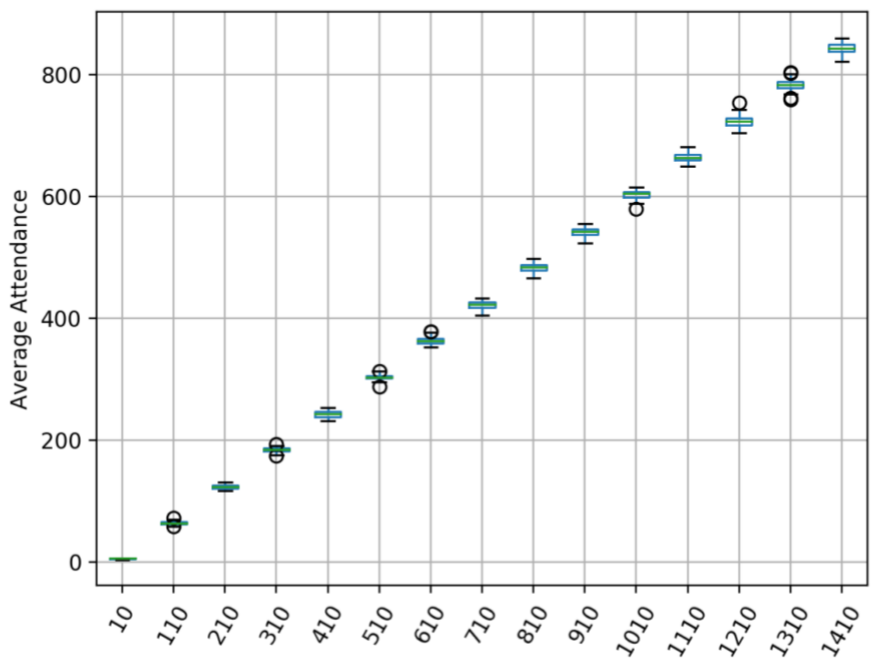

figure 3: Average Attendance VS Population Size

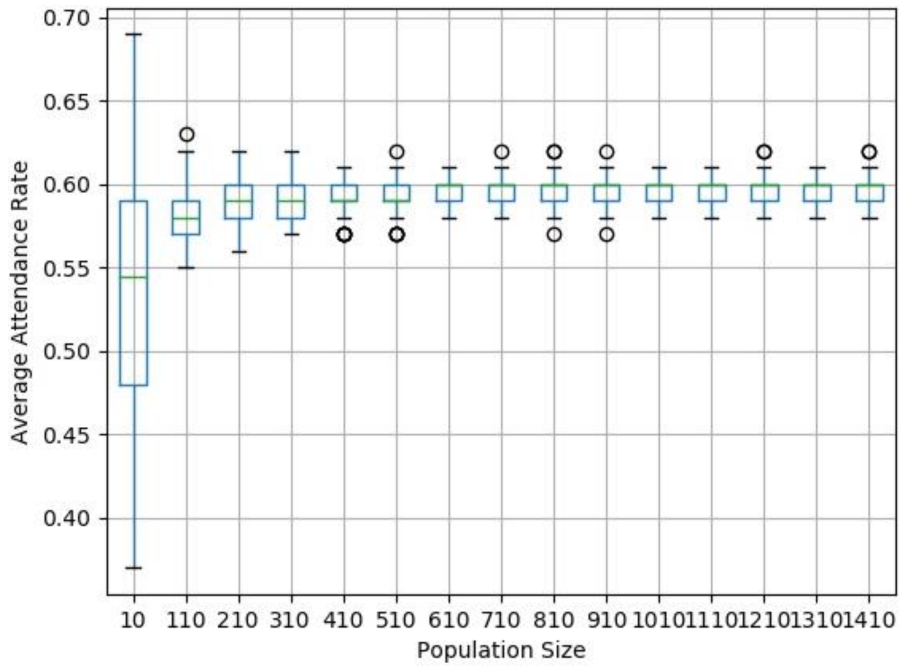

figure 4: Average Attendance Rate VS Population Size

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 λ) is the number of individuals in a population. The value of population size is defined as one thousand **(i.e., Population Size (λ)=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*) ∈ [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*) ∈ Set {1, 4, 7, 10, 13, 16,19 22 })**.

5.2.1 Boxplot of Results

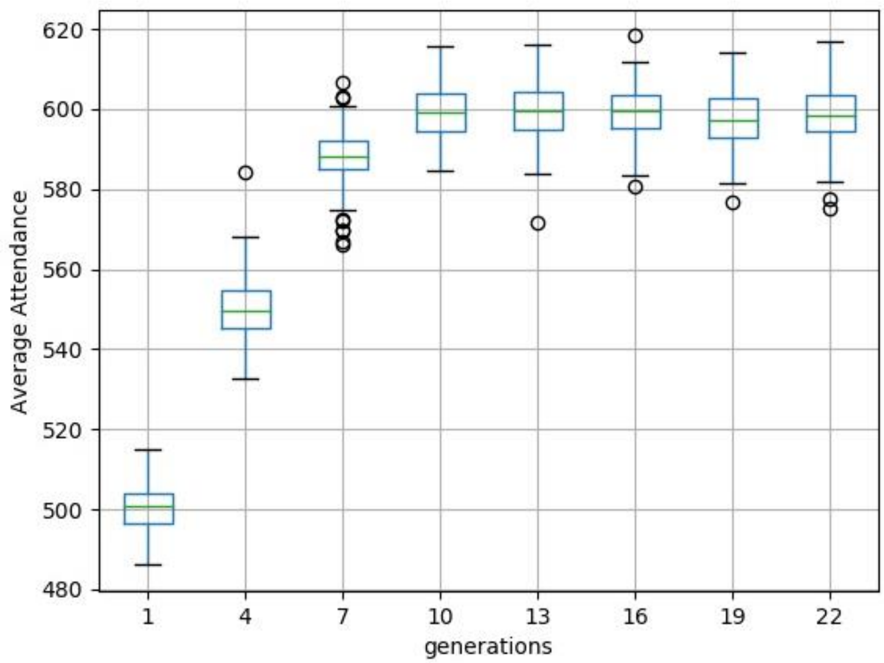

figure 5: Average Attendance VS Generations

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

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


def Sample(prob,times):
    n=len(prob)
    integers=np.arange(n)
    dice=stats.rv_discrete(values=(integers,prob))
    sample_result=dice.rvs(size=times)
    return sample_result


def strategy_handle(strategy):
    attendance_prob = np.array([])
    A = np.array([])
    B = np.array([])
    for i in range(h):
        attendance_prob = np.append(attendance_prob, strategy[i *
(2 * h + 1)])
        A = np.concatenate((A, strategy[1 + i * (2 * h + 1):h + 1 +
        i * (2 * h + 1)]))
        B = np.concatenate((B, strategy[h + 1 + i * (2 * h + 1):2 *
h + 1 +
        i * (2 * h + 1)]))
    A = A.reshape(h, h)
    B = B.reshape(h, h)
    return attendance_prob,A,B


def one_step_desicion(state,crowded,attendance_prob,A,B):
    if crowded:
        next_state=Sample(A[state],times=1)
    else:
        next_state=Sample(B[state],times=1)
    if np.random.random()<=attendance_prob[next_state]:
        decision=1
    else:
        decision = 0
    return decision,next_state
```

```
def all_decisions(origin_population):
    decision_list=list()
    for i in origin_population:
        state=i[0]
        P=i[1]
        if np.random.random() <= P[state]:
            decision_list.append(1)
        else:
            decision_list.append(0)
    bar_people=sum(decision_list)
    return bar_people,decision_list


def initial_individual(h):
    state=np.random.randint(0,h)
    P=np.random.random(h)
    A=np.zeros((h,h))
    B=np.zeros((h,h))
    for i in range(h):
        random_list=np.random.randint(0,100,h)
        Sum=sum(random_list)
        A[i]=random_list/Sum
    for j in range(h):
        random_list = np.random.randint(0, 100, h)
        Sum = sum(random_list)
        B[j] = random_list / Sum
    return state,P,A,B


def initial_population(population_size,h):
    origin_population=list()
    for i in range(population_size):
        strategy = list(initial_individual(h))
        origin_population.append(strategy)
    return origin_population


def fitness_function(weeks,origin_population,population_size):
    bar_people_list=list()
    payoff_list=[0]*population_size
    for i in range(1,weeks+1):
        bar_people, decision_list =
all_decisions(origin_population)
        bar_people_list.append(bar_people)
        if bar_people<0.6*population_size:
            crowded=0
        else:
            crowded=1
        for c in range(population_size):
            if decision_list[c]!=crowded:
                payoff_list[c]+=1
        for m in range(population_size):
            # print(origin_population[i][0])
            # print('----------------------------')
```

```
            A=origin_population[m][2]
            B=origin_population[m][3]
            if crowded:
                origin_population[m][0]=Sample(A[origin_population[m][0]],times=1)[0]
            else:
                origin_population[m][0] = Sample(B[origin_population[m][0]], times=1)[0]
            # print(origin_population[i][0])
        # print(str(i)+'\t'+str(t)+'\t'+str(bar_people)+'\t'+str(crowded)+'\t')
    average_attendance=sum(bar_people_list)/weeks

    return payoff_list,average_attendance

def Roulette_Selection(payoff,copy_origin_population):
    Sum_payoff=sum(payoff)
    payoff_array=np.array(payoff)
    payoff_distribution=payoff_array/Sum_payoff
    select_index=Sample(payoff_distribution,times=1)[0]
    # print(select_index)
    output=copy_origin_population[select_index]
    return output

def mutation(x,h,mutation_rate):
    strategy = list(initial_individual(h))
    for i in range(4):
        if np.random.random()<=mutation_rate:
            x[i]=strategy[i]
    return x

def crossover(x,y,h):
    cross_locus=np.random.randint(h/2-2,h/2+2)
    z1=x[:].copy()
    z2=y[:].copy()
    for i in range(1, 4):
        z1[i]=x[i].copy()
        z2[i]=y[i].copy()
    # print(cross_locus)
    z1[1][cross_locus:h] = y[1][cross_locus:h].copy()
    z2[1][cross_locus:h] = x[1][cross_locus:h].copy()
    for i in range(2,4):
        z1[i][cross_locus:h,:]=y[i][cross_locus:h,:].copy()
        z2[i][cross_locus:h,:]=x[i][cross_locus:h,:].copy()
    return z1,z2


if __name__ == '__main__':
    parser=argparse.ArgumentParser(description='manual to this script')
```

```
    parser.add_argument('-question',type=int,default=3)
    parser.add_argument('-repetitions', type=int, default=5)
    parser.add_argument('-prob', type=str, default='0 0 1 0')
    parser.add_argument('-strategy', type=str, default='2 0.1 0.0
1.0 1.0 0.0
    1.0 0.9 0.1 0.9 0.1')
    parser.add_argument('-state',type=int,default=0)
    parser.add_argument('-crowded', type=int, default=1)
    parser.add_argument('-lamda','-lambda', type=int, default=150)
    parser.add_argument('-ha',type=int,default=10)
    parser.add_argument('-weeks', type=int, default=10)
    parser.add_argument('-time_budget', type=int, default=20)
    args= parser.parse_args()
    question_number=args.question

    if question_number==1:
        prob = list(args.prob.split(' '))
        prob = tuple(map(float, prob))
        for i in range(args.repetitions):
            output_1=Sample(prob,times=1)
            print(output_1[0])

    elif question_number == 2:
        strategy=list(map(eval,args.strategy.split(' ')))
        state=args.state
        crowded=args.crowded
        h=int(strategy[0])
        strategy=strategy[1:len(strategy)]
        attendance_prob, A, B=strategy_handle(strategy)
        for i in range(args.repetitions):

d,s=one_step_desicion(state,crowded,attendance_prob,A,B)
            print(str(d)+'\t'+str(s[0]))


    else:
        try:
            population_size = args.lamda
            h=args.ha
            weeks=args.weeks
            time_budget=args.time_budget

            mutation_list = [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]

            t=0
            dic_average_attend=dict()
            for mutation_rate in mutation_list:
                list_average_attend=list()

                for ex_times in range(100):

origin_population=initial_population(population_size,h)
```

```
                    start = time.clock()
                    fbest=0.6*population_size
                    while True:
                        copy_origin_population = origin_population[:]
                        for i in range(population_size):
                            copy_origin_population[i] =
origin_population[i].copy()

                        population=list()
                        t=t+1
                        payoff, average_attendance =
fitness_function(weeks,
                        origin_population, population_size)
                        end = time.clock()
                        runtime = end - start

                        if runtime > time_budget:
                            break

                        for j in range(int(population_size/2)):

father=Roulette_Selection(payoff,copy_origin_population)

mother=Roulette_Selection(payoff,copy_origin_population)
                            x=mutation(father,h,mutation_rate)
                            y=mutation(mother,h,mutation_rate)
                            z1,z2=crossover(x,y,h)
                            population.append(z1)
                            population.append(z2)
                            middle=time.clock()
                        origin_population=population[:]
                    list_average_attend.append(average_attendance)
                    print(average_attendance)
                  #
list_average_attend=list(map(float,list_average_attend))

dic_average_attend[mutation_rate]=list_average_attend
            data = pd.DataFrame(dic_average_attend)
            data.plot(kind='box',rot=60,grid=True)
            plt.ylabel('Average Attendance')
            plt.xlabel('Mutation Rate')
            plt.savefig('figure5.jpg')


        except Exception as e:
            print('time over!')
```

Appendix B

Average Attendance vs Number of States

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


def Sample(prob):
    n = len(prob)
    integers = np.arange(n)
    all_prob = sum(prob)
    total_prob = np.cumsum(prob)
    dics = np.random.uniform(0, all_prob)
    for i in range(n):
        if dics <= total_prob[i]:
            return integers[i]


def strategy_handle(strategy):
    attendance_prob = np.array([])
    A = np.array([])
    B = np.array([])
    for i in range(h):
        attendance_prob = np.append(attendance_prob,
strategy[i * (2 * h + 1)])
        A = np.concatenate((A, strategy[1 + i * (2 * h + 1):h
+
        1 + i * (2 * h + 1)]))
        B = np.concatenate((B, strategy[h + 1 + i * (2 * h +
1):2 * h +
        1 + i * (2 * h + 1)]))
    A = A.reshape(h, h)
    B = B.reshape(h, h)
    return attendance_prob,A,B


def one_step_desicion(state,crowded,attendance_prob,A,B):
    if crowded:
        next_state=Sample(A[state])
    else:
        next_state=Sample(B[state])
    if np.random.random()<=attendance_prob[next_state]:
        decision=1
    else:
```

```
        decision = 0
    return decision,next_state


def all_decisions(origin_population):
    decision_list=list()
    for i in origin_population:
        state=i[0]
        P=i[1]
        if np.random.random() <= P[state]:
            decision_list.append(1)
        else:
            decision_list.append(0)
    bar_people=sum(decision_list)
    return bar_people,decision_list


def initial_individual(h):
    state=np.random.randint(0,h)
    P=np.random.random(h)
    A=np.zeros((h,h))
    B=np.zeros((h,h))
    for i in range(h):
        random_list=np.random.randint(0,100,h)
        Sum=sum(random_list)
        A[i]=random_list/Sum
    for j in range(h):
        random_list = np.random.randint(0, 100, h)
        Sum = sum(random_list)
        B[j] = random_list / Sum
    return state,P,A,B


def initial_population(population_size,h):
    origin_population=list()
    for i in range(population_size):
        strategy = list(initial_individual(h))
        origin_population.append(strategy)
    return origin_population


def
fitness_function(weeks,origin_population,population_size):
    bar_people_list = list()
    payoff_list=[0]*population_size
    for i in range(0,weeks):
        bar_people, decision_list =
all_decisions(origin_population)
        bar_people_list.append(bar_people)
        if bar_people<0.6*population_size:
            crowded=0
        else:
            crowded=1
        for c in range(population_size):
```

```
            if decision_list[c]!=crowded:
                payoff_list[c]+=1
        for m in range(population_size):
            # print(origin_population[i][0])
            # print('----------------------------')
            A=origin_population[m][2]
            B=origin_population[m][3]
            if crowded:

origin_population[m][0]=Sample(A[origin_population[m][0]])
            else:
                origin_population[m][0] =
Sample(B[origin_population[m][0]])
            # print(origin_population[i][0])
        decision_list = list(map(str, decision_list))
        #
print(str(i)+'\t'+str(t)+'\t'+str(bar_people)+'\t'+str(crowde
d)+
        '\t'+('\t'.join(decision_list)))
    average_attendance = sum(bar_people_list) / weeks
    return payoff_list,origin_population,average_attendance


def Roulette_Selection(payoff,copy_origin_population):
    Sum_payoff=sum(payoff)
    payoff_array=np.array(payoff)
    payoff_distribution=payoff_array/Sum_payoff
    select_index=Sample(payoff_distribution)
    # print(select_index)
    output=copy_origin_population[select_index]
    return output

def mutation(x,h,mutation_rate):
    strategy = list(initial_individual(h))
    for i in range(4):
        if np.random.random()<=mutation_rate:
            x[i]=strategy[i]
    return x

def crossover(x,y,h):
    cross_locus=np.random.randint(h/2-2,h/2+2)
    z1=x[:]
    z2=y[:]
    for i in range(1, 4):
        z1[i]=x[i].copy()
        z2[i]=y[i].copy()
    # print(cross_locus)
    z1[1][cross_locus:h] = y[1][cross_locus:h].copy()
    z2[1][cross_locus:h] = x[1][cross_locus:h].copy()
    for i in range(2,4):
        z1[i][cross_locus:h,:]=y[i][cross_locus:h,:].copy()
        z2[i][cross_locus:h,:]=x[i][cross_locus:h,:].copy()
    return z1,z2

```

```


if __name__ == '__main__':
    parser=argparse.ArgumentParser(description='manual to this
script',add_help=False)
    parser.add_argument('-question',type=int,default=3)
    parser.add_argument('-repetitions', type=int, default=5)
    parser.add_argument('-prob', type=str, default='0 0 1 0')
    parser.add_argument('-strategy', type=str, default='2 0.1
0.0 1.0
    1.0 0.0 1.0 0.9 0.1 0.9 0.1')
    parser.add_argument('-state',type=int,default=1)
    parser.add_argument('-crowded', type=int, default=0)
    parser.add_argument('-lamda','-lambda', type=int,
default=700)
    # parser.add_argument('-h',type=int,default=10)
    parser.add_argument('-weeks', type=int, default=10)
    parser.add_argument('-max_t', type=int, default=10)
    args= parser.parse_args()
    question_number=args.question

    if question_number==1:
        prob = list(args.prob.split(' '))
        prob = list(map(float, prob))
        for i in range(args.repetitions):
            output_1 = Sample(prob)
            print(output_1)

    elif question_number == 2:
        strategy=list(map(eval,args.strategy.split(' ')))
        state=args.state
        crowded=args.crowded
        h=int(strategy[0])
        strategy=strategy[1:len(strategy)]
        attendance_prob, A, B=strategy_handle(strategy)
        for i in range(args.repetitions):

d,s=one step desicion(state,crowded,attendance prob,A,B)
            print(str(d)+'\t'+str(s))


    else:

        population_size = args.lamda
        # h=args.h
        weeks=args.weeks
        time_budget=args.max_t


        mutation_rate=0.0001

        dic_average_attend = dict()

        for h in range(10,60,10):
```

```
        list_average_attend = list()

        for ex_times in range(80):
            t = 0

origin_population=initial_population(population_size,h)
            start = time.clock()
            while True:
                copy_origin_population =
origin_population[:]
                for i in range(population_size):
                    copy_origin_population[i] =
origin_population[i][:]
                end = time.clock()
                runtime = end - start
                # print(runtime)
                if t== time_budget-1:
                    # print(runtime)
                    break
                population=list()

                payoff, origin_population,average_attendance
= fitness_function(weeks,
                origin_population, population_size)
                #
list_average_attend.append(average_attendance)
                # print(average_attendance)
                # print('********')
                # print(bar_people)
                # Sum_payoff=sum(payoff)
                # print(Sum_payoff)
                t = t + 1
                for j in range(int(population_size/2)):

father=Roulette_Selection(payoff,copy_origin_population)

mother=Roulette_Selection(payoff,copy_origin_population)
                    x=mutation(father,h,mutation_rate)
                    y=mutation(mother,h,mutation_rate)
                    z1,z2=crossover(x,y,h)
                    population.append(z1)
                    population.append(z2)
                    middle=time.clock()
                    # if (middle-start)>time_budget:
                    #     print(runtime)
                    #     break
                origin_population=population[:]
            list_average_attend.append(average_attendance)
            print(average_attendance)
        # list_average_attend = list(map(float,
list_average_attend))
        print('------')
```

```
                  #
list_average_attend=list(map(float,list_average_attend))
          dic_average_attend[h] = list_average_attend
       data = pd.DataFrame(dic_average_attend)
       data.plot(kind='box', grid=True)
       plt.ylabel('Average Attendance ')
       plt.xlabel('Number of States')
       plt.savefig('figure33.jpg')
```

Appendix C

Average Attendance vs Generations

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


def Sample(prob):
    n = len(prob)
    integers = np.arange(n)
    all_prob = sum(prob)
    total_prob = np.cumsum(prob)
    dics = np.random.uniform(0, all_prob)
    for i in range(n):
        if dics <= total_prob[i]:
            return integers[i]


def strategy_handle(strategy):
    attendance_prob = np.array([])
    A = np.array([])
    B = np.array([])
    for i in range(h):
        attendance_prob = np.append(attendance_prob, strategy[i *
(2 * h + 1)])
        A = np.concatenate((A, strategy[1 + i * (2 * h + 1):h + 1
+
        i * (2 * h + 1)]))
        B = np.concatenate((B, strategy[h + 1 + i * (2 * h + 1):2
* h +
        1 + i * (2 * h + 1)]))
    A = A.reshape(h, h)
    B = B.reshape(h, h)
    return attendance_prob,A,B


def one_step_desicion(state,crowded,attendance_prob,A,B):
    if crowded:
        next_state=Sample(A[state])
    else:
        next_state=Sample(B[state])
    if np.random.random()<=attendance_prob[next_state]:
        decision=1
    else:
```

```
        decision = 0
    return decision,next_state


def all_decisions(origin_population):
    decision_list=list()
    for i in origin_population:
        state=i[0]
        P=i[1]
        if np.random.random() <= P[state]:
            decision_list.append(1)
        else:
            decision_list.append(0)
    bar_people=sum(decision_list)
    return bar_people,decision_list


def initial_individual(h):
    state=np.random.randint(0,h)
    P=np.random.random(h)
    A=np.zeros((h,h))
    B=np.zeros((h,h))
    for i in range(h):
        random_list=np.random.randint(0,100,h)
        Sum=sum(random_list)
        A[i]=random_list/Sum
    for j in range(h):
        random_list = np.random.randint(0, 100, h)
        Sum = sum(random_list)
        B[j] = random_list / Sum
    return state,P,A,B


def initial_population(population_size,h):
    origin_population=list()
    for i in range(population_size):
        strategy = list(initial_individual(h))
        origin_population.append(strategy)
    return origin_population


def fitness_function(weeks,origin_population,population_size):
    bar_people_list = list()
    payoff_list=[0]*population_size
    for i in range(0,weeks):
        bar_people, decision_list =
all_decisions(origin_population)
        bar_people_list.append(bar_people)
        if bar_people<0.6*population_size:
            crowded=0
        else:
            crowded=1
        for c in range(population_size):
            if decision_list[c]!=crowded:
```

```
                payoff_list[c]+=1
        for m in range(population_size):
            A=origin_population[m][2]
            B=origin_population[m][3]
            if crowded:

origin_population[m][0]=Sample(A[origin_population[m][0]])
            else:
                origin_population[m][0] =
Sample(B[origin_population[m][0]])
            # print(origin_population[i][0])
        decision_list = list(map(str, decision_list))
        #
print(str(i)+'\t'+str(t)+'\t'+str(bar_people)+'\t'+str(crowded)+
        '\t'+('\t'.join(decision_list)))
    average_attendance = sum(bar_people_list) / weeks
    return payoff_list,origin_population,average_attendance


def Roulette_Selection(payoff,copy_origin_population):
    Sum_payoff=sum(payoff)
    payoff_array=np.array(payoff)
    payoff_distribution=payoff_array/Sum_payoff
    select_index=Sample(payoff_distribution)
    # print(select_index)
    output=copy_origin_population[select_index]
    return output

def mutation(x,h,mutation_rate):
    strategy = list(initial_individual(h))
    for i in range(4):
        if np.random.random()<=mutation_rate:
            x[i]=strategy[i]
    return x

def crossover(x,y,h):
    cross_locus=np.random.randint(h/2-2,h/2+2)
    z1=x[:]
    z2=y[:]
    for i in range(1, 4):
        z1[i]=x[i].copy()
        z2[i]=y[i].copy()
    # print(cross_locus)
    z1[1][cross_locus:h] = y[1][cross_locus:h].copy()
    z2[1][cross_locus:h] = x[1][cross_locus:h].copy()
    for i in range(2,4):
        z1[i][cross_locus:h,:]=y[i][cross_locus:h,:].copy()
        z2[i][cross_locus:h,:]=x[i][cross_locus:h,:].copy()
    return z1,z2

if __name__ == '__main__':
    parser=argparse.ArgumentParser(description='manual to this
script',add_help=False)
    parser.add_argument('-question',type=int,default=3)
```

```
    parser.add_argument('-repetitions', type=int, default=5)
    parser.add_argument('-prob', type=str, default='0 0 1 0')
    parser.add_argument('-strategy', type=str, default='2 0.1
0.0 1.0
    1.0 0.0 1.0 0.9 0.1 0.9 0.1')
    parser.add_argument('-state',type=int,default=1)
    parser.add_argument('-crowded', type=int, default=0)
    parser.add_argument('-lamda','-lambda', type=int,
default=1000)
    parser.add_argument('-h',type=int,default=10)
    parser.add_argument('-weeks', type=int, default=10)
    # parser.add_argument('-max_t', type=int, default=20)
    args= parser.parse_args()
    question_number=args.question

    if question_number==1:
        prob = list(args.prob.split(' '))
        prob = list(map(float, prob))
        for i in range(args.repetitions):
            output_1 = Sample(prob)
            print(output_1)

    elif question_number == 2:
        strategy=list(map(eval,args.strategy.split(' ')))
        state=args.state
        crowded=args.crowded
        h=int(strategy[0])
        strategy=strategy[1:len(strategy)]
        attendance_prob, A, B=strategy_handle(strategy)
        for i in range(args.repetitions):

d,s=one_step_desicion(state,crowded,attendance_prob,A,B)
            print(str(d)+'\t'+str(s))


    else:

        population_size = args.lamda
        h=args.h
        weeks=args.weeks
        # time_budget=args.max_t


        mutation_rate=0.0001

        dic_average_attend = dict()

        for time_budget in range(2,25,3):
            list_average_attend = list()

            for ex_times in range(100):
                t = 0

origin_population=initial_population(population_size,h)
```

```
                start = time.clock()
                while True:
                    copy_origin_population = origin_population[:]
                    for i in range(population_size):
                        copy_origin_population[i] =
origin_population[i][:]
                    end = time.clock()
                    runtime = end - start
                    # print(runtime)
                    if t== time_budget-1:
                        # print(runtime)
                        break
                    population=list()

                    payoff, origin_population,average_attendance =
fitness_function(weeks,
                    origin_population, population_size)
                    #
list_average_attend.append(average_attendance)
                    # print(average_attendance)
                    # print('********')
                    # print(bar_people)
                    # Sum_payoff=sum(payoff)
                    # print(Sum_payoff)
                    t = t + 1
                    for j in range(int(population_size/2)):

father=Roulette_Selection(payoff,copy_origin_population)

mother=Roulette_Selection(payoff,copy_origin_population)
                        x=mutation(father,h,mutation_rate)
                        y=mutation(mother,h,mutation_rate)
                        z1,z2=crossover(x,y,h)
                        population.append(z1)
                        population.append(z2)
                        middle=time.clock()
                    origin_population=population[:]
                list_average_attend.append(average_attendance)
                print(average_attendance)
            print('------')


list_average_attend=list(map(float,list_average_attend))
            dic_average_attend[time_budget-1] =
list_average_attend
        data = pd.DataFrame(dic_average_attend)
        data.plot(kind='box', grid=True)
        plt.ylabel('Average Attendance')
        plt.xlabel('generations')
        plt.savefig('figure11.jpg')
```

Appendix D

Average Attendance vs Population Size

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


def Sample(prob):
    n = len(prob)
    integers = np.arange(n)
    all_prob = sum(prob)
    total_prob = np.cumsum(prob)
    dics = np.random.uniform(0, all_prob)
    for i in range(n):
        if dics <= total_prob[i]:
            return integers[i]


def strategy_handle(strategy):
    attendance_prob = np.array([])
    A = np.array([])
    B = np.array([])
    for i in range(h):
        attendance_prob = np.append(attendance_prob,
strategy[i * (2 * h + 1)])
        A = np.concatenate((A, strategy[1 + i * (2 * h +
        1):h + 1 + i * (2 * h + 1)]))
        B = np.concatenate((B, strategy[h + 1 + i * (2 * h +
1):2 * h +
        1 + i * (2 * h + 1)]))
    A = A.reshape(h, h)
    B = B.reshape(h, h)
    return attendance_prob,A,B


def one_step_desicion(state,crowded,attendance_prob,A,B):
    if crowded:
        next_state=Sample(A[state])
    else:
        next_state=Sample(B[state])
    if np.random.random()<=attendance_prob[next_state]:
        decision=1
    else:
        decision = 0
```

```
    return decision,next_state


def all_decisions(origin_population):
    decision_list=list()
    for i in origin_population:
        state=i[0]
        P=i[1]
        if np.random.random() <= P[state]:
            decision_list.append(1)
        else:
            decision_list.append(0)
    bar_people=sum(decision_list)
    return bar_people,decision_list


def initial_individual(h):
    state=np.random.randint(0,h)
    P=np.random.random(h)
    A=np.zeros((h,h))
    B=np.zeros((h,h))
    for i in range(h):
        random_list=np.random.randint(0,100,h)
        Sum=sum(random_list)
        A[i]=random_list/Sum
    for j in range(h):
        random_list = np.random.randint(0, 100, h)
        Sum = sum(random_list)
        B[j] = random_list / Sum
    return state,P,A,B


def initial_population(population_size,h):
    origin_population=list()
    for i in range(population_size):
        strategy = list(initial_individual(h))
        origin_population.append(strategy)
    return origin_population


def
fitness_function(weeks,origin_population,population_size):
    bar_people_list = list()
    payoff_list=[0]*population_size
    for i in range(0,weeks):
        bar_people, decision_list =
all_decisions(origin_population)
        bar_people_list.append(bar_people)
        if bar_people<0.6*population_size:
            crowded=0
        else:
            crowded=1
        for c in range(population_size):
            if decision_list[c]!=crowded:
```

```
                payoff_list[c]+=1
        for m in range(population_size):
            # print(origin_population[i][0])
            # print('----------------------------')
            A=origin_population[m][2]
            B=origin_population[m][3]
            if crowded:

origin_population[m][0]=Sample(A[origin_population[m][0]])
            else:
                origin_population[m][0] =
Sample(B[origin_population[m][0]])
            # print(origin_population[i][0])
        decision_list = list(map(str, decision_list))
        # print(str(i)+'\t'+str(t)+'\t'+str(bar_people)+'\t'+
        str(crowded)+'\t'+('\t'.join(decision_list)))
    average_attendance = sum(bar_people_list) / weeks
    return payoff_list,origin_population,average_attendance


def Roulette_Selection(payoff,copy_origin_population):
    Sum_payoff=sum(payoff)
    payoff_array=np.array(payoff)
    payoff_distribution=payoff_array/Sum_payoff
    select_index=Sample(payoff_distribution)
    # print(select_index)
    output=copy_origin_population[select_index]
    return output

def mutation(x,h,mutation_rate):
    strategy = list(initial_individual(h))
    for i in range(4):
        if np.random.random()<=mutation_rate:
            x[i]=strategy[i]
    return x

def crossover(x,y,h):
    cross_locus=np.random.randint(h/2-2,h/2+2)
    z1=x[:]
    z2=y[:]
    for i in range(1, 4):
        z1[i]=x[i].copy()
        z2[i]=y[i].copy()
    # print(cross_locus)
    z1[1][cross_locus:h] = y[1][cross_locus:h].copy()
    z2[1][cross_locus:h] = x[1][cross_locus:h].copy()
    for i in range(2,4):
        z1[i][cross_locus:h,:]=y[i][cross_locus:h,:].copy()
        z2[i][cross_locus:h,:]=x[i][cross_locus:h,:].copy()
    return z1,z2

if __name__ == '__main__':
    parser=argparse.ArgumentParser(description='manual to this
script',add_help=False)
```

```
    parser.add_argument('-question',type=int,default=3)
    parser.add_argument('-repetitions', type=int, default=5)
    parser.add_argument('-prob', type=str, default='0 0 1 0')
    parser.add_argument('-strategy', type=str, default='2 0.1
    0.0 1.0 1.0 0.0 1.0 0.9 0.1 0.9 0.1')
    parser.add_argument('-state',type=int,default=1)
    parser.add_argument('-crowded', type=int, default=0)
    # parser.add_argument('-lamda','-lambda', type=int,
default=1000)
    parser.add_argument('-h',type=int,default=10)
    parser.add_argument('-weeks', type=int, default=10)
    parser.add_argument('-max_t', type=int, default=10)
    args= parser.parse_args()
    question_number=args.question

    if question_number==1:
        prob = list(args.prob.split(' '))
        prob = list(map(float, prob))
        for i in range(args.repetitions):
            output_1 = Sample(prob)
            print(output_1)

    elif question_number == 2:
        strategy=list(map(eval,args.strategy.split(' ')))
        state=args.state
        crowded=args.crowded
        h=int(strategy[0])
        strategy=strategy[1:len(strategy)]
        attendance_prob, A, B=strategy_handle(strategy)
        for i in range(args.repetitions):

d,s=one_step_desicion(state,crowded,attendance_prob,A,B)
            print(str(d)+'\t'+str(s))

    else:

        # population_size = args.lamda
        h=args.h
        weeks=args.weeks
        time_budget=args.max_t


        mutation_rate=0.0001

        dic_average_attend = dict()

        for population_size in range(10,1500,100):
            list_average_attend = list()

            for ex_times in range(80):
                t = 0

origin_population=initial_population(population_size,h)
                start = time.clock()
```

```
                while True:
                    copy_origin_population =
origin_population[:]
                    for i in range(population_size):
                        copy_origin_population[i] =
origin_population[i][:]
                    end = time.clock()
                    runtime = end - start
                    # print(runtime)
                    if t== time_budget-1:
                        # print(runtime)
                        break
                    population=list()

                    payoff, origin_population,average_attendance
=
                    fitness_function(weeks, origin_population,
population_size)
                    #
                          t = t + 1
                    for j in range(int(population_size/2)):
father=Roulette_Selection(payoff,copy_origin_population)
mother=Roulette_Selection(payoff,copy_origin_population)
                        x=mutation(father,h,mutation_rate)
                        y=mutation(mother,h,mutation_rate)
                        z1,z2=crossover(x,y,h)
                        population.append(z1)
                        population.append(z2)
                        middle=time.clock()
                      origin_population=population[:]

list_average_attend.append('%.2f'%(average_attendance/populat
ion_size))

print('%.2f'%(average_attendance/population_size))
            list_average_attend = list(map(float,
list_average_attend))
            print('------')

            dic_average_attend[population_size] =
list_average_attend
        data = pd.DataFrame(dic_average_attend)
        data.plot(kind='box', grid=True)
        plt.ylabel('Average Attendance Rate')
        plt.xlabel('Population Size')
        plt.savefig('figure0.jpg')
```