Beacon of Life -- Python Simulation of Best Evacuation Routes

Overview:

Group memebers:

陈旻昊 **518120910134**

梁献丹 **716120220014**

方逸舟 **518120910159**

1.Problem

We have seen a lot of cases where an unexpected disaster claimed a huge number of people's lives. For instance, in 2008, the Wenchuan earthquake in Sichuan province, China, killed nearly seventy thousand people, which was truly a calamity. In order to reduce disaster casualty, it is important to develop an efficient evacuation system. Inspired by this, our team is going to write a python project which can **simulate** the evacuation process when an earthquake or a fire occurs during our python class. Furthermore, we want the program we write to be able to tell us the best evacuation routes given the layout of our classroom.

Before we move to the model part, several things we need to think about that pedestrians will take into consideration when evacuation:
1. Distance to the closest exit
2. Obstacles in the way
3. Congestion caused by other evacuees
4. Status of the exits (the width, whether it's new or not, etc.).
We will give full consideration of these elements during modeling.

2. Principle of Simulation

We will use cellular automaton to solve this problem. In our model, we will transform the classroom into a two-dimension cellular space whose length and width are proportionate to that of the real classroom. The cellular space will consist of little squares whose side is 0.4 meters long. Using a square as a position unit, we will simulate the evacuation process based on the rules of the Moore neighborhood. Every square is a cell and each cell has two kinds of state, occupied and available. A cell can be either occupied by an evacuee or a barrier. After one time step, all the evacuees will move into the next cell.

2.1 Cellular Automation [3][4][5][6][7][8]

A cellular automaton is a discrete dynamic system consisting of finite state variables (or cells) on a uniform grid. The operating rules are mainly: the state of all cells changes simultaneously, and at the same time, The state of the ith cell at time $t+1$ is determined by the state of the ith cell at time t, and the state of the adjacent 2r cells whose distance does not exceed r.

In this simulation, we're using the Moore neighborhood rule to define the model. Moore neighborhood is defined on a two-dimensional square lattice and is composed of a central cell and the eight cells which surround it.

2.2 Static Floor Field

We will import the concept of static floor field to further construct our model. It determines which cell a certain evacuee will move into.

Combined with the concept of the Moore neighborhood, the state of each cell is determined by the eight cells around it. The calculation rule is, assuming that the center cell has a static floor field of N, the field value of the four cells adjacent in the lateral direction and the longitudinal direction is $N+1$, and the field value of the four cells in the diagonal direction is $N+1.4$. Reversely, the static floor field of each cell is also influenced by the 8 cells around, so there are different field values for each cell due to different influences from the cells nearby. For some certain purposes, we take each cell's minimum value out of all as the initial static floor strength of it in our model.

However, the static floor strengths we now get are not what we finally want. In reality, people usually tend to choose the closest exit to get out of the classroom during an evacuation. Meanwhile, they are also very likely to choose the shortest route. To embody this common tendency, we aim to use static floor field to quantify the distance between evacuees and exits. **The larger the static floor field strength of a cell is, the closer the cell is to the exits.** The current static floor strengths do not indicate this information. Therefore, we must find out the cell with the largest static floor field strength and deduct its strength value from

all the static field strengths of all the cells. Then we take the absolute values as final values. After this modification, the field values increase with decreasing distance to an exit and is zero for the cell farthest away from the door.

Obviously, people cannot move into a cell that is occupied by any obstacles. This problem can be solved by setting the occupied cell's **static floor field strength for zero**. Evacuees will never move into this kind of cell for its static floor field is always the lowest.

In short, the steps to calculate every cell's static floor field strength are:

1. Define the values of all the cells representing stairs or exits on a certain floor as

zero.

2. Calculate every cell's value of static floor field based on the cells whose values are known. Assume the value of static field of cell A is N, then the calculation principle is that the values of its linear cells are $N+1$ while the values of its diagonal cells are $N+1.4$. If the value of cell A's adjacent cell is also affected by other cells, choose the smallest one as its value.

3. After figure out all the cells' values q_n, we get the maximum value of static field

in the two-dimensional cellular space q_{max}. Replace q_n with ($q_{max} - q_n$).

4. Replace the values of the cells which are occupied by barriers with zero.

Note that our classroom has two exits, which means that we need to calculate two different static floor fields generated by these two different exits.

2.3 Rules of Simulation

Based on the model we built, we can now determine the rules of evacuees' movements.

2.3.1 Various Factors

Each classroom has 2 doors. When choosing the exit, people often face trade-offs. On the one hand, they take the distance between him or her and a certain door into consideration. If the door is too far, people certainly would not choose it. On the other hand, they also tend to heed the objective conditions of the exits. The objective conditions of an exit include its congestion level, width and so on. People always prefer exits that are wide and vacant. In other words, evacuees' decisions are not solely depended on the distance, but also other objective factors.

However, in fact, people's evacuation behaviors are not only affected by objective

factors, but also subjective factors and random factors. Subjective factors determine the utility of different objective variables toward evacuees. Random factors are actually produced by different cognition levels of different people and some latent variables. Taking all these factors into account, using Utility Theory to develop our advanced model is seemingly a good choice.

2.3.2 Utility Equation Describing People's Evacuation Behaviors

All the people's purposes of choosing an exit are to evacuate from the dangerous building. Therefore, we can assume that the profits different people get after they get out of the building are the same. This allows us to write down the utility equation by only considering the cost of different routes. According to discrete choice models based on random Utility Theory, we have

$$U_{in} = V_{in} + \epsilon_{in}$$

where U_{in} is the utility of exit i to evacuee n (define the cell that evacuee n is in as cell n). It consists of V_{in} and $\in_{in}$. $\in_{in}$ represents the changeable part of utility caused by random variables, which we ignore in our model. On the other hand, V_{in} refers to the definite part of utility of exit i to evacuee n determined by all the k observable variables x_{in}^k. In order to simplify the calculation, we use a linear equation to define V_{in} as

$$V_{in} = \sum_k V_{in}^k = \sum_k \beta_{in}^k g_k(x_{in}^k)$$

β_{in}^k is the corresponding sensitivity coefficients of variable x_{in}^k. It describes how V_{in} would be influenced by small changes in the estimates x_{in}^k. g_k is the utility function of variable x_{in}^k, which can figure out the corresponding utility V_{in}^k.

We define two variables in our model. First, we use static field strength to represent every evacuee's location x_{in}^1. Second, due to the different locations of different evacuees, their perceptions of congestion level are different, mainly determined by the number of people who get to the exit ahead of them. Let the number of human-occupied cells whose values of static floor field strengths imposed by exit i are larger than that of cell n's be N_{in}. Then, N_{in} can be used to represent the number of people getting to the exit i ahead of evacuee n.

The equations showed below are the utility equations of these two variables. In these equations, relative value is used to settle the problem of dimensional heterogeneity.

$$V_{in}^1 = g_1(x_{in}^1) = \frac{S_{in}}{\sum_{j \in C_n} S_{jn}}$$

$$V_{in}^2 = g_2(x_{in}^2) = 1 - \frac{(I-1)N_{in}}{\sum_{j \in C_n} N_{jn}}$$

C_n is the set of exit i in the cellular space. I is the number of the exits. S_n is the static field strength that exit i imposes on the cell n and L_i is the number cells that exit i occupies.

Besides two variables, β_{in}^k is also important. However, without trial and error, we do not know what are the best coefficients for different variables. Indeed, we have to collect necessary data through the simulation process. The results will be discussed later.

2.3.3 Multinomial Logistic Regression [1][2]

Evacuees always choose the exit that has the largest utility to them, so the probability of evacuee n choosing exit i is

$$P_{in} = prob[U_{in} > U_{jn}, \forall j \neq i \in C_n]$$

We calculate P_{in} according to the method of multinomial logistic regression. In this case, as a log-linear model, the formulation can be directly extended to multi-way regression. Therefore, we have

$$\begin{aligned} \ln Pr(Y_i = 1) &= \alpha_1 \cdot X_i - \ln Z \\ \ln Pr(Y_i = 2) &= \alpha_2 \cdot X_i - \ln Z \\ &\cdots\cdots \\ \ln Pr(Y_i = 3) &= \alpha_K \cdot X_i - \ln Z \end{aligned}$$

As in the binary case, we need an extra term $ln\,Z$ to ensure that the whole set of probabilities forms a probability distribution, so $\sum_{1}^{K} Pr(Y_i = k) = 1$,then we get $Z = \sum_{k=1}^{K} e^{\alpha_k \cdot X_i}$ .Based on these derivations, consequently we know that

$$Pr(Y_i = c) = \frac{e^{\alpha_c \cdot X_i}}{\sum_{k=1}^{K} e^{\alpha_k \cdot X_i}}$$

In our model we transform this equation into

$$P_{in} = \frac{exp\,(\lambda V_{in})}{\sum\limits_{j \in C_n} exp\,(\lambda V_{jn})}$$

2.3.4 Principles of Choosing Exit and Moving towards Exits

To make the evacuation process closer to the real life facts, we introduce P as an evacuee's probability of keeping his or her selection of exit. **The larger the P is, the more rational the evacuee is**. Based on both P and P_{in} calculated before, we come up with our principles of choosing exits. That is, for each time step, the evacuee keeps the target exit with the probability P , otherwise it is based on the the probability P_{in} to determine the next target exit. After having chosen the target exit, evacuees then choose the cell that he or she wants to move into based on the static floor field generated by the **target exit**. For each step, the evacuee will move into the cell which has the highest field value out of the 8 cells around. When one cell becomes a target cell for different evacuees simultaneously, there is a conflict. Solution is that the person moving into the target cell will be randomly picked, while unchosen one stands still until the next unit of time.

Notice that in section 2.2, we mentioned that we calculated two different static floor field. This means that different exits generate different static floor field in the cellular space.

3. Preparation of Simulation

Before the simulation begins, we need to convert the classroom into a two-dimensional planar grid.

We measured the classroom using the number of cells as the unit of measurement. A typical classroom in SJTU lower hall is 12 cells wide, and 26 cells long. It has a front door and a back door, each of which allows one person passing per time, so it is measured as a cell's width.

The model of classroom is shown below, where each square represents a 0.4×0.4 size cell space. The two red cells are the front and back exits. The green and black cells are both obstacles, representing desks and the podium.

4. Simulation of Evacuation Process

In this section, we will first introduce our codes.

Our project needs three libraries.

```
from random import randrange, random
from math import e
from graphics import *
```

A good program always has good interaction with users. Therefore, we first define two new classes.

- **One is Button. An object of Button can be displayed on windows. Each Button has two states: activated and deactivated. Being activated means that it can be clicked. The size of the Button depends on the augments input.**

```
class Button:
    def __init__(self,win,center,width,height,label):
        w,h = width/2.0, height/2.0
        x,y = center.getX(), center.getY()
        self.xmax, self.xmin = x+w, x-w
        self.ymax, self.ymin = y+h, y-h
        p1 = Point(self.xmin,self.ymin)
        p2 = Point(self.xmax,self.ymax)
        self.rect = Rectangle(p1,p2)
        self.rect.setFill('lightgrey')
        self.rect.draw(win)
        self.label = Text(center,label)
        self.label.draw(win)
        self.deactivate()

    def clicked(self,p):
        return (self.active and self.xmin <= p.getX() <= self.xmax and
                self.ymin <= p.getY() <= self.ymax)
```

```
    def getLabel(self):
        return self.label.getText()

    def activate(self):
        self.label.setFill('black')
        self.rect.setWidth(2)
        self.active = True

    def deactivate(self):
        self.label.setFill('darkgrey')
        self.rect.setWidth(1)
        self.active = False
```

- **Another one is InputDialog. It is an graphics window containing input boxes and several Buttons.**

```
class InputDialog:
    def __init__(self,number,weigh):
        self.win = win = GraphWin('Evacuee Number', 200, 300)
        win.setCoords(0,4.5,4,.5)

        Text(Point(2, 1), 'Disaster is coming!').draw(win)

        Text(Point(1,2),'Number').draw(win)
        self.number = Entry(Point(3,2),5)
        self.number.draw(win)
        self.number.setText(str(number))

        Text(Point(1,3),'Weigh').draw(win)
        self.weigh = Entry(Point(3,3), 5)
        self.weigh.draw(win)
        self.weigh.setText(str(weigh))

        self.simulate = Button(win,Point(1,4),1.4,.5,"Simulate!")
        self.simulate.activate()

        self.quit = Button(win,Point(3,4),1.4,.5,'Quit')
        self.quit.activate()
```

- **An InputDialog has three functions: interact(), getValues() and close(). Function interact() is responsible for the interation with users. Function getValues() can record the values that users input into the boxes.**

```
def interact(self):
    while True:
        pt = self.win.getMouse()
        if self.quit.clicked(pt):
            return 'Quit'
        if self.simulate.clicked(pt):
```

```
            return 'Simulate!'

def getValues(self):
    n = float(self.number.getText())
    wg = float(self.weigh.getText())
    return n,wg

def close(self):
    self.win.close()
```

After finishing the interaction part, we come to the core of our program.

- **We define a new class called Classroom. In essence, it is also a graohics window.**

```
class Classroom:
    def __init__(self):
        self.win = GraphWin('Classroom 105', 320, 520, autoflush=False)
        self.win.setCoords(11.5,-0.5,-0.5,25.5)
```

- **We draw the layout of our claasroom on this graphics window. Desks and podium are green. Available paths are white. Doors are drawn red.**

```
def draw_cell_vertical_line(self):
    for i in range(12):
        line = Line(Point(i-0.5,-0.5),Point(i-0.5,25.5))
        line.draw(self.win)
    line = Line(Point(11.5,-0.5),Point(11.5,25.5))
    line.draw(self.win)

def draw_cell_horizontal_line(self):
    for i in range(26):
        line = Line(Point(-0.5,i-0.5),Point(11.5,i-0.5))
        line.draw(self.win)
    line = Line(Point(-0.5,25.5),Point(11.5,25.5))
    line.draw(self.win)

def fill_green(self):
    for y in range(5,22,2):
        for x in range(1,3):
            rec = Rectangle(Point(x+0.5,y-0.5),Point(x-0.5,y+0.5))
            rec.setFill('green')
            rec.draw(self.win)

    for y in range(3,22,2):
        for x in range(4,8):
            rec = Rectangle(Point(x+0.5,y-0.5),Point(x-0.5,y+0.5))
            rec.setFill('green')
            rec.draw(self.win)
```

```
    for y in range(1,22,2):
        for x in range(9,12):
            rec = Rectangle(Point(x+0.5,y-0.5),Point(x-0.5,y+0.5))
            rec.setFill('green')
            rec.draw(self.win)

    for y in range(23,25):
        for x in range(4,8):
            rec = Rectangle(Point(x+0.5,y-0.5),Point(x-0.5,y+0.5))
            rec.setFill('green')
            rec.draw(self.win)

def draw_doors(self):
    door1 = Rectangle(Point(0.5,-0.5),Point(-0.5,0.5))
    door1.setFill('red')
    door1.draw(self.win)
    door2 = Rectangle(Point(0.5, 24.5), Point(-0.5, 25.5))
    door2.setFill('red')
    door2.draw(self.win)
```

- **On the graphics window, orange rectangles represent evacuees. They can be either drawn onto or undrawn from the graphics window.**

```
def turn_orange(self,x,y):
    rec = Rectangle(Point(x+0.5,y-0.5),Point(x-0.5,y+0.5))
    rec.setFill('orange')
    rec.draw(self.win)

def turn_off_orange(self,x,y):
    rec = Rectangle(Point(x + 0.5, y - 0.5), Point(x - 0.5, y + 0.5))
    rec.setFill('white')
    rec.draw(self.win)

def draw_evacuee(self,evacuee):
    self.turn_orange(evacuee.get_X(),evacuee.get_Y())

def undraw_evacuees(self,evacuee):
    self.turn_off_orange(evacuee.get_X(),evacuee.get_Y())
```

The next core of our programs is the newly defined class cell. Cells are the basic units of the cellular space. We use it to calculate the two static floor fields. Each cell has three main characteristics: coordinate, static floor field strength and current state (occupied or not occupied).

```
class Cell:

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
        self.static = None
        self.occupied = False

    def get_X(self):
        return self.x

    def get_Y(self):
        return self.y

    def get_static(self):
        return self.static

    def is_occupied(self):
        self.occupied = True

    def not_occupied(self):
        self.occupied = False
```

Calculate the first static floor field. Create a list of Cells representing the cellular space and set the static floor field strength of the door at 0.

```
def create_space1():
    cellular_space1 = [[Cell(x, y) for x in range(12)] for y in range(26)]
    cellular_space1[0][0].static = 0
```

Starting at the door, we use for loop to calculate the whole field. The crux of this step is that not every cell has eight cells around it. This means that the list index can generate IndexError. Therefore, we use try/except syntax to hold the error.

```
for line in cellular_space1:
    for cell in line:

        for dx, dy in [(0, 1), (1, 0), (1, 1)]:
            try:
                if cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static == None:
                    if dx == dy == 1:
                        cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static =
round(cell.static + 1.4, 1)
                    else:
                        cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static =
round(cell.static + 1, 1)
                else:
                    if dx == dy == 1:
                        cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static = round(
                            min(cell.static + 1.4, cellular_space1[cell.get_Y() + dy][cell.get_X() +
dx].static), 1)
                    else:
                        cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static = round(
                            min(cell.static + 1, cellular_space1[cell.get_Y() + dy][cell.get_X() +
```

```
dx].static), 1)
            except IndexError:
                pass

        if cell.get_X() - 1 > 0:
            for dx, dy in [(-1, 0), (-1, 1)]:
                try:
                    if cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static == None:
                        if dy == 1:
                            cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static =
round(cell.static + 1.4,
                                                                                        1)
                        else:
                            cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static =
round(cell.static + 1, 1)
                    else:
                        if dy == 1:
                            cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static = round(
                                min(cell.static + 1.4,
                                    cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static),
1)
                        else:
                            cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static = round(
                                min(cell.static + 1, cellular_space1[cell.get_Y() + dy][cell.get_X()
+ dx].static),
                                1)
                except IndexError:
                    pass

        if cell.get_Y() - 1 > 0:
            for dx, dy in [(0, -1), (1, -1)]:
                try:
                    if cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static == None:
                        if dx == 1:
                            cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static =
round(cell.static + 1.4,
                                                                                        1)
                        else:
                            cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static =
round(cell.static + 1, 1)
                    else:
                        if dx == 1:
                            cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static = round(
                                min(cell.static + 1.4,
                                    cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static),
1)
                        else:
                            cellular_space1[cell.get_Y() + dy][cell.get_X() + dx].static = round(
                                min(cell.static + 1, cellular_space1[cell.get_Y() + dy][cell.get_X()
+ dx].static),
                                1)
```

```
                except IndexError:
                    pass
```

Note that we have to find out the cell that has the largest static floor field strength.

```
maximum = max(cell.static for line in cellular_space1 for cell in line)
for line in cellular_space1:
    for cell in line:
        cell.static = round(maximum - cell.static, 1)
```

Then we set the static floor filed strength of those cells which are occupied by desks and podium at 0.

```
for x, y in [(a, b) for a in range(1,3) for b in range(5, 22, 2)]:
    cellular_space1[y][x].static = 0

for x, y in [(a, b) for a in range(4, 8) for b in range(3, 22, 2)]:
    cellular_space1[y][x].static = 0

for x, y in [(a, b) for a in range(9, 12) for b in range(1, 22, 2)]:
    cellular_space1[y][x].static = 0

for x, y in [(a, b) for a in range(4, 8) for b in range(23, 25)]:
    cellular_space1[y][x].static = 0

return cellular_space1
```

The calculation of the second static floor field is similar to that of the first field. But notice that the starting point's coordinate is no longer (0,0). Here we use some techniques.

```
for line in cellular_space2:
    for cell in line:
        cell.y = 25 - cell.y

cellular_space2.reverse()
```

The third core of our program is the class called Evacuee. An Evacuee object records its coordinate and the static floor field strength of the cell that it is in. It also remembers the last target door it chooses and has several utility values for different factors.

```
class Evacuee:
    def __init__(self, x, y, cellular_space1, cellular_space2, coe = 0.5):
        self.coe = coe
        self.x = x
        self.y = y
```

```
        self.statics = [cellular_space1[self.y][self.x].static,
                        cellular_space2[self.y][self.x].static]

        self.exit = False
        self.utility_distances = [0, 0]
        self.utility_crowd = [0, 0]
        self.utility_doors = [0, 0]
        self.probabilities = [e ** self.utility_doors[0] / (e ** self.utility_doors[0] + e **
self.utility_doors[1]),
                              e ** self.utility_doors[0] / (e ** self.utility_doors[0] + e **
self.utility_doors[1])]
        self.last_door = ''
```

It has a few simple functions.

```
def get_X(self):  # get the x-coordinate
    return self.x

def get_Y(self):  # get the y-coordinate
    return self.y

def get_Static1(self):  # get the static field where the evacuee now stays
    return self.statics[0]

def get_Static2(self):
    return self.statics[1]
```

It also has some critical functions. An Evacuee moves through the move() function. It updates its utility for different factors through several update functions. It can also check whether it has successfully evacuated from the classroom.

```
def move(self, dx, dy):  # movement into the next cell
    self.x += dx
    self.y += dy

def update_statics(self, cellular_space1, cellular_space2):
    self.statics = [cellular_space1[self.y][self.x].static, cellular_space2[self.y][self.x].static]

def update_utility_distances(self):
    self.utility_distances = [self.coe * self.statics[0] / sum(self.statics), self.coe *
self.statics[1] / sum(self.statics)]

def update_utility_doors(self):
    self.utility_doors = self.utility_distances + self.utility_crowd

def update_probabilities(self):
    self.probabilities = [e ** self.utility_doors[0] / (e ** self.utility_doors[0] + e **
self.utility_doors[1]),
                          e ** self.utility_doors[0] / (e ** self.utility_doors[0] + e **
self.utility_doors[1])]
```

```
def exit_judge(self, x_door, y_door):
    if self.get_X() == x_door and self.get_Y() == y_door:
        self.exit = True
```

The last core is the Space class. After randomly create evacuees in the classroom, it records the overall conditions of the cellular space including the number of remained evacuees, each evacuee's position and the two static floor field.

```
class Space:
    def __init__(self, cellular_space1, cellular_space2):
        self.evacuees = []
        self.cellular_space1 = cellular_space1
        self.cellular_space2 = cellular_space2

    def create_evacuees(self, n, coefficient):
        number_evacuee = 0
        while number_evacuee < n:
            new = Evacuee(randrange(0, 12), randrange(0, 26), self.cellular_space1,
self.cellular_space2, coe = coefficient)
            if new.statics[0] == 0 and new.statics[1] == 0 or new in self.evacuees:
                continue
            else:
                self.evacuees.append(new)
                number_evacuee += 1
        for evacuee in self.evacuees:
            evacuee.exit_judge(0, 0)
            evacuee.exit_judge(0, 25)
```

What's more, it can calculate each evacuee's perception of congestion level through counting the number of evacuees who are closer to the target door than this evacuee.

```
def larger_static1(self, remain_evacuees):
    larger_static1 = []
    all_statics = [evacuee.get_Static1() for evacuee in remain_evacuees]
    for this_evacuee_static in all_statics:
        larger = [other_evacuee_static for other_evacuee_static in all_statics if
                  other_evacuee_static > this_evacuee_static]
        larger_static1.append(len(larger))
    return larger_static1

def larger_static2(self, remain_evacuees):
    larger_static2 = []
    all_statics = [evacuee.get_Static2() for evacuee in remain_evacuees]
    for this_evacuee_static in all_statics:
        larger = [other_evacuee_static for other_evacuee_static in all_statics if
                  other_evacuee_static > this_evacuee_static]
        larger_static2.append(len(larger))
    return larger_static2
```

```
def utility_crowd_assignment(self, larger_static1, larger_static2, remain_evacuees):
    for evacuee in remain_evacuees:
        if larger_static1[remain_evacuees.index(evacuee)] +
larger_static2[remain_evacuees.index(evacuee)] != 0:
            evacuee.utility_crowd = [(1 - larger_static1[remain_evacuees.index(evacuee)] /
                                      (larger_static1[remain_evacuees.index(evacuee)] +
                                       larger_static2[remain_evacuees.index(evacuee)])) * (1-
evacuee.coe),
                                     (1 - larger_static2[remain_evacuees.index(evacuee)] /
                                      (larger_static1[remain_evacuees.index(evacuee)] +
                                       larger_static2[remain_evacuees.index(evacuee)])) * (1-
evacuee.coe)]
        else:
            evacuee.utility_crowd = [1, 1]
```

Last but not least, it is able to conduct simulation based on the principles we put forward earlier and visualize the simulation process on the Classroom window. Since the code of function simulate() is too long, we do not put it here.

With all the things above, we can finally write the execution part to finish our program. We first create an interaction window and wait the user to input something.

```
inputwindow = InputDialog(0,0.0)
interaction = inputwindow.interact()
```

Then if the user clicks the simulation Button, the simulation process begins. The input will be used during the process.

```
if interaction == 'Simulate!':
    people, coefficient = inputwindow.getValues()
    inputwindow.close()
```

We create Classroom graphics window and calculate the two static floor field.

```
classroom105 = Classroom()
classroom105.draw_cell_horizontal_line()
classroom105.draw_cell_vertical_line()
classroom105.fill_green()
classroom105.draw_doors()

cellular_space1 = create_space1()
cellular_space2 = create_space2()
```

All things prepared, we finally create the Space based on the two static floor field and randomly create n evacuees according to the input of the user. Start counting the time.

```
classroom = Space(cellular_space1, cellular_space2)
classroom.create_evacuees(people, coefficient)
remained_evacuees = [evacuee for evacuee in classroom.evacuees if evacuee.exit == False]
time_steps = 0
```

We use for loop to conduct the simulation. The loop goes through every evacuee and updates their position after each time step.

```
for evacuee in remained_evacuees:
    classroom105.draw_evacuee(evacuee)
    cellular_space1[evacuee.get_Y()][evacuee.get_X()].is_occupied()
    cellular_space2[evacuee.get_Y()][evacuee.get_X()].is_occupied()
    evacuee.update_statics(cellular_space1, cellular_space2)
    evacuee.update_utility_distances()
    classroom.utility_crowd_assignment(larger_static1=classroom.larger_static1(remained_evacuees),
                                       larger_static2=classroom.larger_static2(remained_evacuees),
                                       remain_evacuees=remained_evacuees)
    evacuee.update_utility_doors()
    evacuee.update_probabilities()
classroom.simulate_once(cellular_space1, cellular_space2, remained_evacuees, classroom105)
# print([[evacuee.get_X(), evacuee.get_Y()] for evacuee in remained_evacuees])
remained_evacuees = [evacuee for evacuee in remained_evacuees if evacuee.exit == False]
while remained_evacuees:
    time_steps += 1
    for evacuee in remained_evacuees:
        cellular_space1[evacuee.get_Y()][evacuee.get_X()].is_occupied()
        cellular_space2[evacuee.get_Y()][evacuee.get_X()].is_occupied()
        evacuee.update_statics(cellular_space1, cellular_space2)
        evacuee.update_utility_distances()

classroom.utility_crowd_assignment(larger_static1=classroom.larger_static1(remained_evacuees),

larger_static2=classroom.larger_static2(remained_evacuees),
                                       remain_evacuees=remained_evacuees)
        evacuee.update_utility_doors()
        evacuee.update_probabilities()
    classroom.simulate(cellular_space1, cellular_space2, remained_evacuees, classroom105)
    # print([[evacuee.get_X(), evacuee.get_Y()] for evacuee in remained_evacuees])
    remained_evacuees = [evacuee for evacuee in remained_evacuees if evacuee.exit == False]
print(time_steps)
```

After the simulation finishes, the user can close the simulation window.

```
a = input('')
    classroom105.win.close()
```

```
else:
    inputwindow.close()
```

After we run our program using different value of P (an evacuee's probability of keeping his or her selection of exit), different value of weight and different number of evacuees to cross check the results, we get a table of data in the attached excel file and visualized in the histograms below.

For figure 1-4, the horizontal line represents the number of evacuees and the vertical line represents the average time for each individual to escape. For each number of evacuees (40, 50, 60, 70, 80) there are 5 bars with 5 different color representing the weight (0.5, 0.6, 0.7, 0.8, 0.9, 1).

For figure 5-9, the horizontal line represents the weight and the vertical line represents the average time for each individual to escape. For each of the weight there are 4 bars with different color representing the P (0.75, 0.8, 0.85, 0.9).

We can see that

1. When there are a large number of people, change the probability of keeping the target exit (the smaller the P), we can achieve a shorter average escape time.

2. When there are less people, a better P value is showed as 0.8.

3. At any given P, the smaller the weight, the shorter the average escape time.

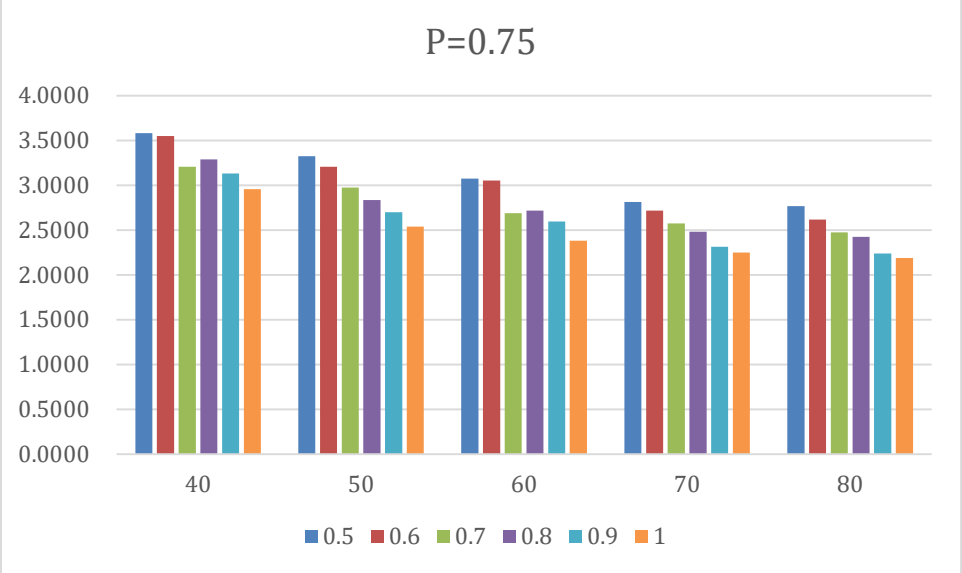

Figure 1

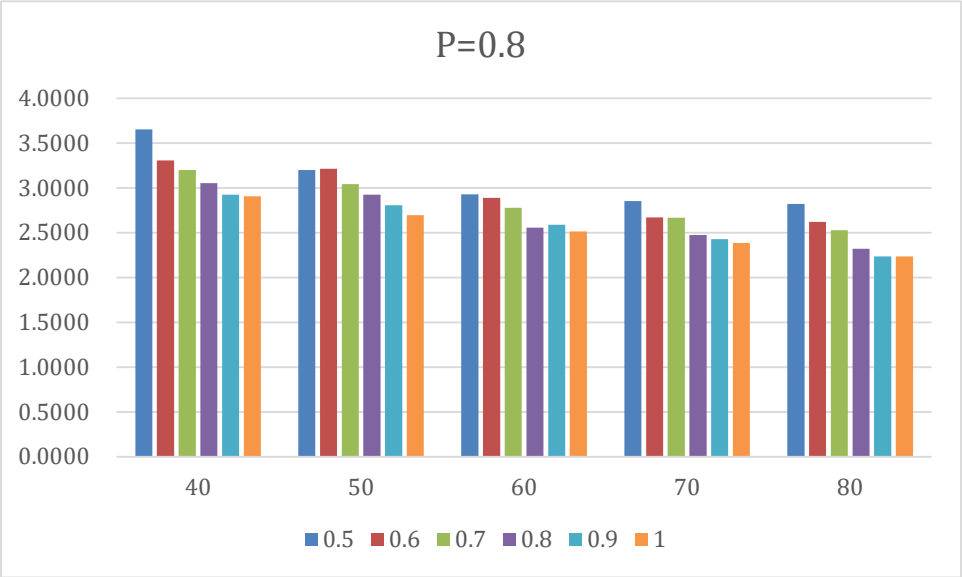

Figure2

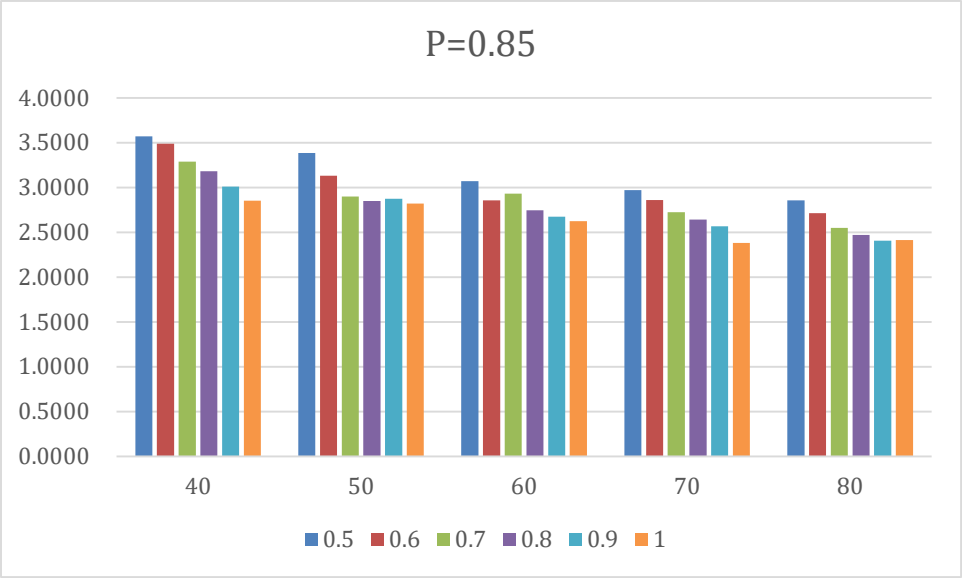

Figure3

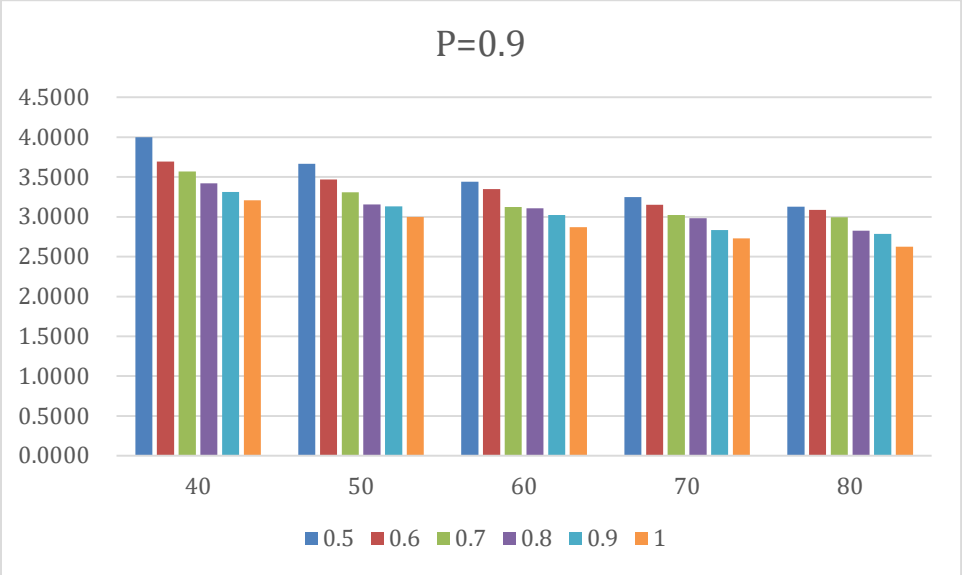
P=0.9
4.5000
4.0000
3.5000
3.0000
2.5000
2.0000
1.5000
1.0000
0.5000
0.0000
40
50
60
70
80
0.5
0.6
0.7
0.8
0.9
1

Figure4

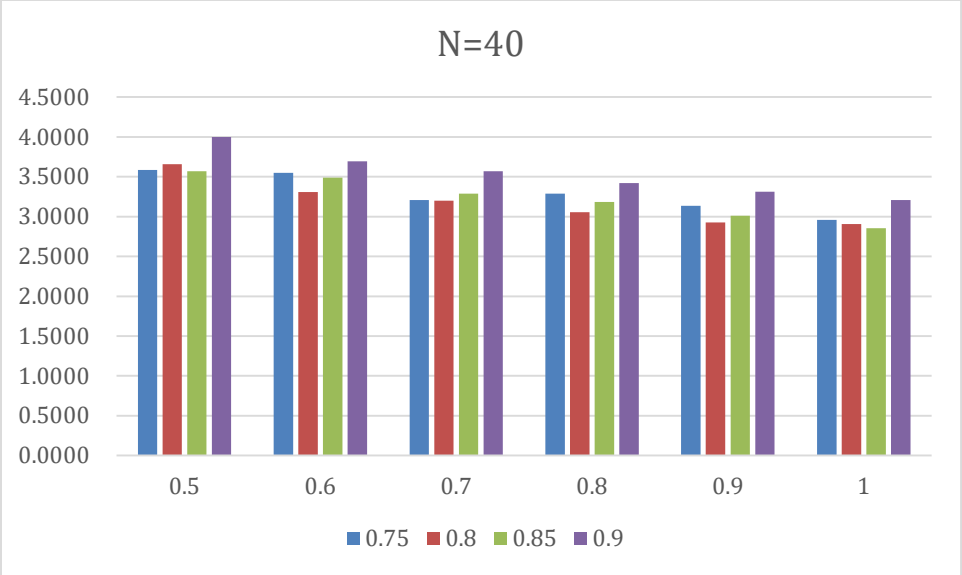

Figure5

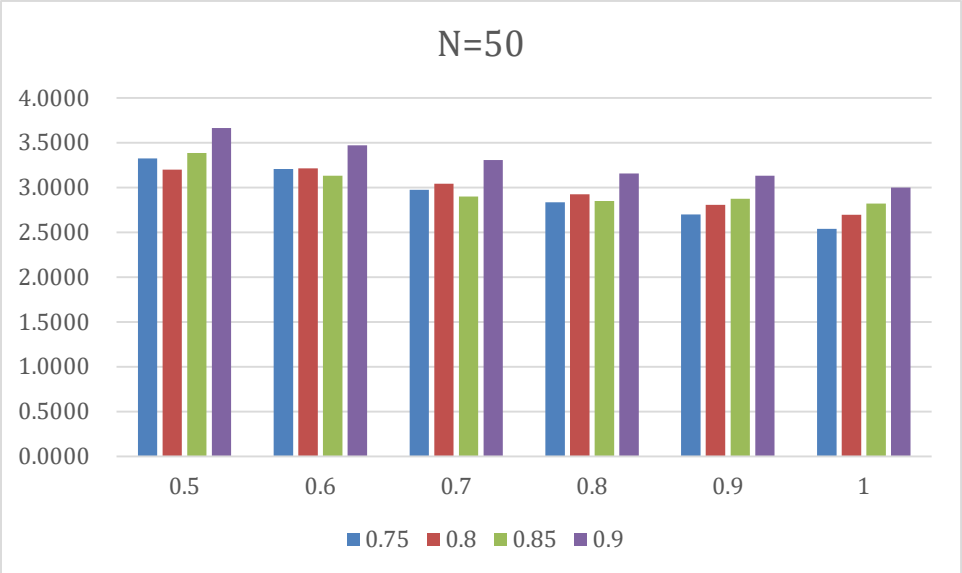

Figure6

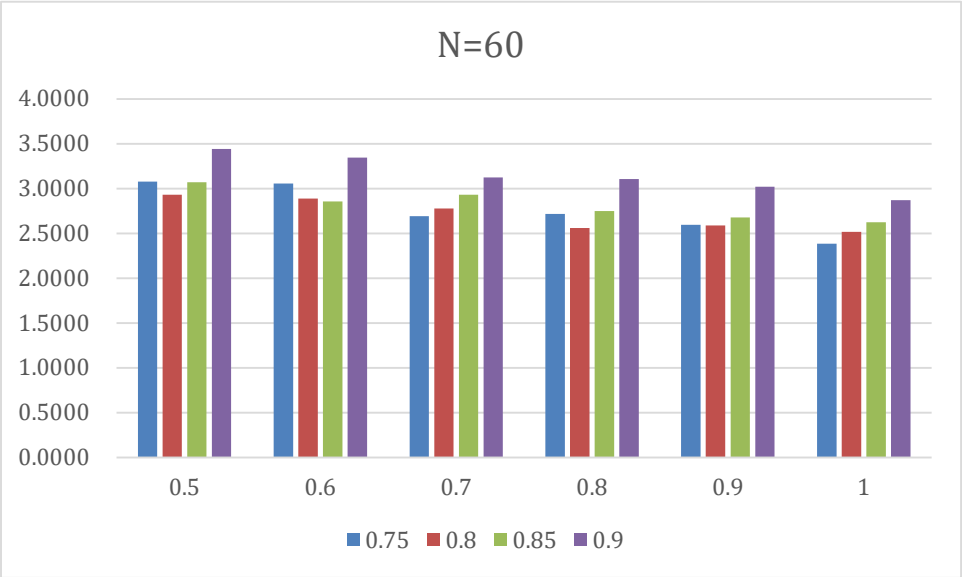
N=60
4.0000
3.5000
3.0000
2.5000
2.0000
1.5000
1.0000
0.5000
0.0000
0.5
0.6
0.7
0.8
0.9
1
0.75
0.8
0.85
0.9

Figure7

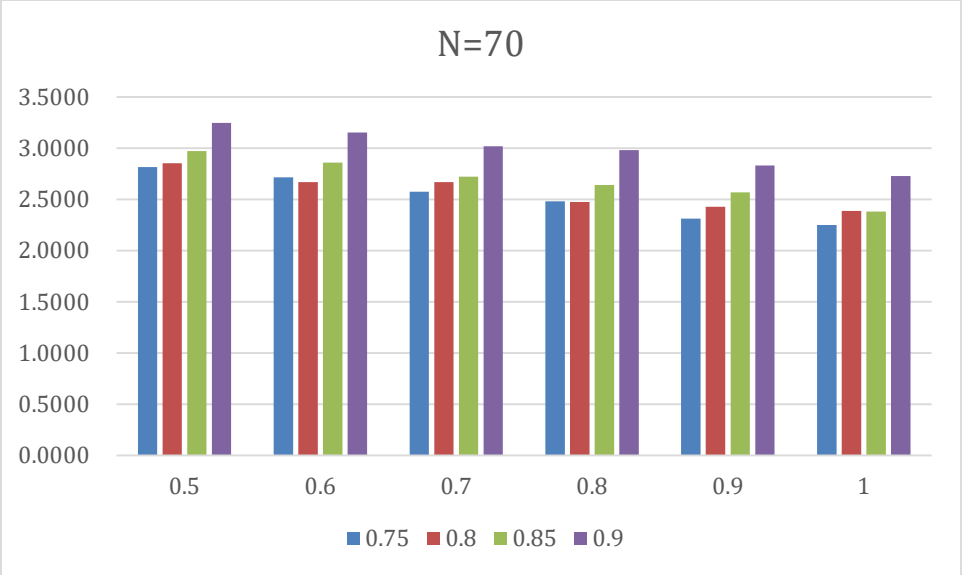

Figure8

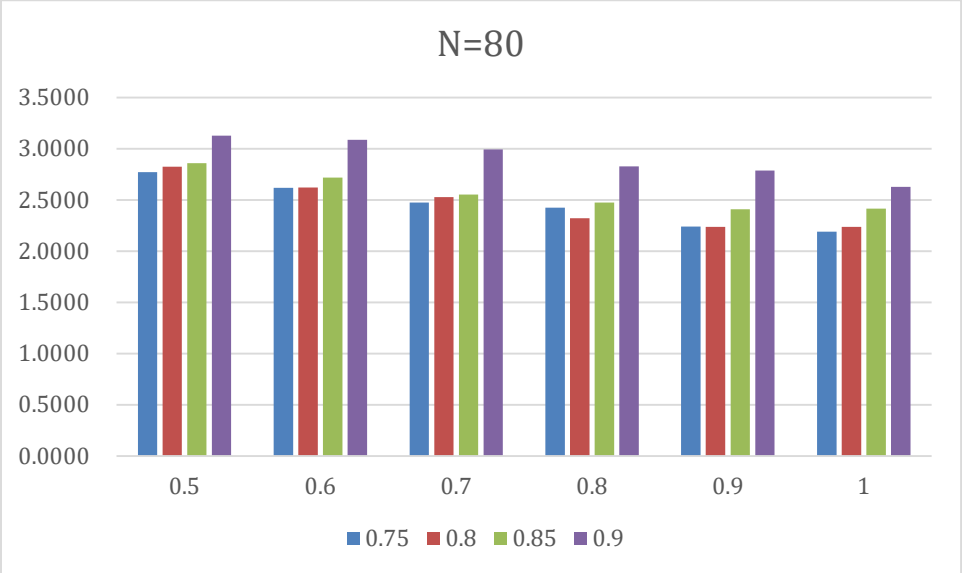

Figure9

5. Conclusion

Our module is designed to simulate the optimized escaping routes of every students in the classroom, capable to handle any given number of students and weight of other variables. Through the module, we are able to estimate average escape time when potential disasters occur to the classroom, providing insights for emergency evacuation plan, classroom design, and risk assessment.

We also provide the exact description of how people interact with each other during evacuation, which can well be applied to any disastrous occasion. Based on the functions we defined, our module can easily be adjusted for simulation in other places like museums and hospitals, helping them prepare for underlying catastrophe.

As the simulation demonstrates, our module works well when there is a relatively large number of remaining students in the room. But when the number decreases, the module fails to describe how people might escape under emergency, as we still use the same deciding function as that when loads of people are around. This limitation, however, does not matter much. When there is a small amount of people and the module goes wrong, it is easy for manual operation to calculate their movements. Thus, shall our module be implemented in real life, we will combine the module with artificial supervision to ensure that the simulation process goes smoothly. If time permits, we will design another module specifically targeted at the situation when there are few people in the room.

6.Reference

[1] Kwak Chanyeong, Clayton-Matthews Alan(2002). Multinomial Logistic Regression

[2] Multinomial Logistic Regression: https://en.wikipedia.org/wiki/Mul

tinomial_logistic_regression

[3] 基于元胞自动机的火灾中人员逃生的模型. https://www.ixueshu.com/document/

316cdc995e9a22ea318947a18e7f9386.html

[4] 基于元胞自动机的火灾场景行人流疏散仿真研究. http://tjxb.cnjournals.cn/ch/reader/

view_abstract.aspx?file_no=17473&flag=1

[5] 特殊人群疏散特性的元胞自动机模拟. https://www.hanspub.org/journal/

PaperInformation.aspx?paperID=23874

[6] 数学建模理论与方法, 沈世云、杨春德、刘勇、张清华、潘显兵、郑继明, 2016.

[7] Boarding at the Speed of Flight，2007.

[8] 元胞自动机理论研究及其仿真应用 科学出版社 段晓东等.