For office use only
T1 _____
T2 _____
T3 _____
T4 _____

Team Control Number

**1901516**

Problem Chosen

**D**

For office use only
F1 _____
F2 _____
F3 _____
F4 _____

---

**2019 Mathematical Contest in Modeling (MCM) Summary Sheet**
(Attach a copy of this page to each copy of your solution paper.)


## Summary

This paper establishes a static model and then improves it to be a dynamic model to simulate the evacuation and provide optimal routes for visitors located everywhere to get out and for emergency personnel to get in.

Firstly, we study the simplest situation and construct a static model. We simplify each floor of the Louvre to a network topology. The crowds on the floor are simplified as nodes and evacuation routes between two spots are simplified as links in a network topology. The problem of finding the shortest evacuation routes is transformed to finding shortest path between two nodes. Utilizing the theory of weighted graph, we assign weights to links of the network topology. Applying Dijkstra Algorithm to realize the calculation, we can get the shortest evacuation routes between any crowd and any exit. (Figure 9, 10 and 11.)

Secondly, we take the influence of changes in crowd density on walking speed into consideration and build a dynamic model. We establish the relationships of walking speed with crowd density, rate of crowd flow with walking speed respectively. We also identify the recurrence relation between crowd densities at time t and t+$\Delta t$. The above established relationships are used to calculate the time-varying weights. Iterative algorithm is used to simulate the process of reassigning weights to links and planning the optimal routes every $\Delta t$. Finally we can get more practical time-varying evacuation routes. (Figure12,13 and 14).

We identified bottlenecks in both static and dynamic models. Two additional exits are added to let the emergency personnel get in. Using the same Iterative algorithm, we can get the optimal routes for personnel.(Figure15,16,and 17 )

Thirdly, we study how would our model's results change with different guest numbers(Figure 18 ),the unit time $\Delta t$(Figure 19), and diversity of visitors(Figure 20). Fortunately, our results are not sensitive to any factor.

Finally, we provide policy and procedural suggestions to the museum leaders.



**Keywords:Network Topology; Weighed Graghs;Dijkstra Algorithm; Dynamic Model;Iterative Algorithm**

# Contents

# 1 Probem Restatement

## 1.1 An Overview of the Problem

As the terrorists become increasingly rampant in France, French society has laid great emphasis on evacuation plans at some frequently visited places. As a famous and popular museum in the world, the Louvre can receive millions of visitors in a year. It is very urgent to develop an effective evacuation plan which is able to make every occupant get out of the building and allow the emergency personnel to get into the building as quickly and safely as possible. Due to the huge number of tourists and complex structure of the Louvre, making a feasible evacuation plan is challenging.

There are some factors making the evacuation more complex, which leaves a lot for plan makers to consider. First, the number of tourists changes throughout the day and year, and visitors are also various. Thus, the plan makers need to find an adaptable method which can be applied to various numbers of guests and diverse visitors. Second, the utilization of some technology such as an online application which provides current waiting time at each entrance can be used to facilitate the evacuation plan. Third, the employment of additional exits which are mostly known by the emergency personnel and museum officials need to be considered carefully. Fourth, some places where people are slowed and even get stuck may exist, and these places might make some evacuation routes unavailable. Thus, the evacuation plan may vary as the evacuation goes on.

## 1.2 Our Tasks

- Construct an emergency evacuation model which provides the museum leaders with an array of choices to evacuate tourists and get emergency personnel into the building quickly.
- Take various factors which can affect the evacuation into consideration, then improve the model to make it more practical.
- Identify the places where movement is obviously slowed.
- On the basis of the model, give some policy and procedural suggestions to emergency management of the Louvre.

## 1.3 Literature review

Many researches on emergency evacuation have been carried out, and different scholars have employed different methods to study this problem. For the evacuation problem, the aim of most scholars is to get the minimal total evacuation time for all people. For the problems of evacuation with several exits, some people build an evacuation model combining heuristic algorithm and network flow control. Updating the evacuation network constantly is used to find the optimal routes. Some people studied the dynamic model of emergency evacuation and established the relationship of walking speed and visitors flow rate. Based on the previous wisdom and some necessary knowledge like Dijkstra Algorithm, Probability and Statistics, we expect to construct a model which can properly simulate the evacuation process and be adaptable to complex situations.

# 2 Model Assumptions and Notations

## 2.1 Assumptions and Justifications

To simplify the problem and make it easier to develop the model, we make the following assumptions:

- **Visitors keep moving forwards along their routes all the time, except when they come across the bottlenecks.** We assume nobody stops voluntarily and that tourists would not go back.
- **Visitors are clear about which route they should choose and they evacuate along their routes directly.** We assume visitors utilize the upgraded Affluences to know which routes they should choose and they evacuate strictly according to the instructions on the application.
- **Any floor of the Louvre can be regarded as a connected plane, except when bottlenecks exist.** There is no obstacle on the way that connects any two spots on the same floor, except when bottlenecks exist.
- **People are successfully evacuated as soon as they arrive at the entrance.** We assume the entrances are wide enough so people will not crowd at the entrances. Thus, we do not have to take the influence of population density at the exits into consideration.
- **The time spent on the stairs can be ignored.** We assume the exhibition hall is so big that the time spent on stairs is small enough to be ignored compared to time spent moving across the exhibition hall.

We may make other related assumptions in the following sections.

## 2.2 Notations

We list the symbols and notations used in this paper, as shown in Table 1. Some of them will be defined later in the following sections.

# 3 Structure of the Louvre

We get detailed maps of every floor from the official website of the Louvre and make some marks on them. The following figure are the maps.

Table 1: Notations[H]

| Symbol | Description |
|---|---|
| $\rho_i$ | the crowd density of node i |
| $\rho_0$ | the initial crowd density |
| $\rho_a$ | the average number of crowd density in a day |
| $C_i(t)$ | the time-varying number of people in the crowd represented by node i |
| $C_j(t)$ | the time-varying number of people in the crowd represented by node j |
| $v_{ij}(t)$ | the time-varying walking speed of the crowd |
| $u_{ij}(t)$ | the number of people moving from node i to node j in one unit of time |
| $u_{ji}(t)$ | the number of people moving from node j to node i in one unit of time |
| $L_{ew}$ | the length of the link in the east-west direction |
| $L_{sn}$ | the length of the link in the south-north direction |
| L | the distance between two people |
| $w_{ij}(t)$ | the weight of the edge conneting node i and node j |
| $\Delta t$ | the unit time |
| $G_k(N, E)$ | N is the set of nodes, E is the set of links, k is the floor number |
| $\rho_(avg)$ | the average crowd density at node i and j |
| $m_t$ | the number of visitors in the museum |
| $T_{total}$ | the total evacuation time |

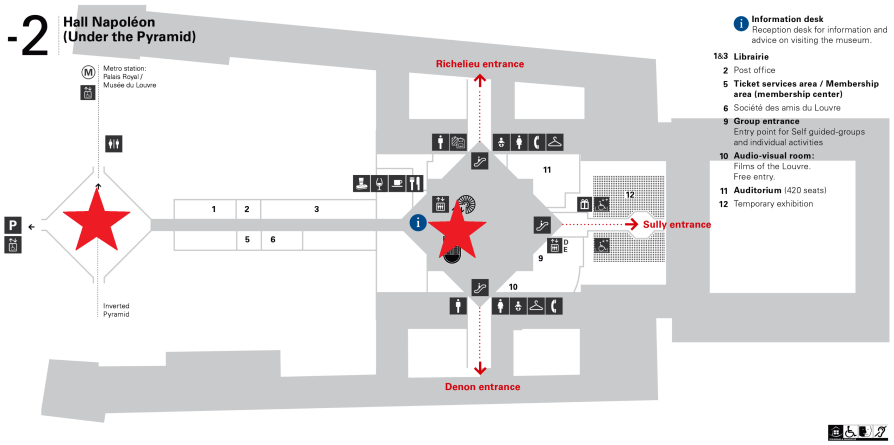


Figure 1: Map of Negative Second Floor

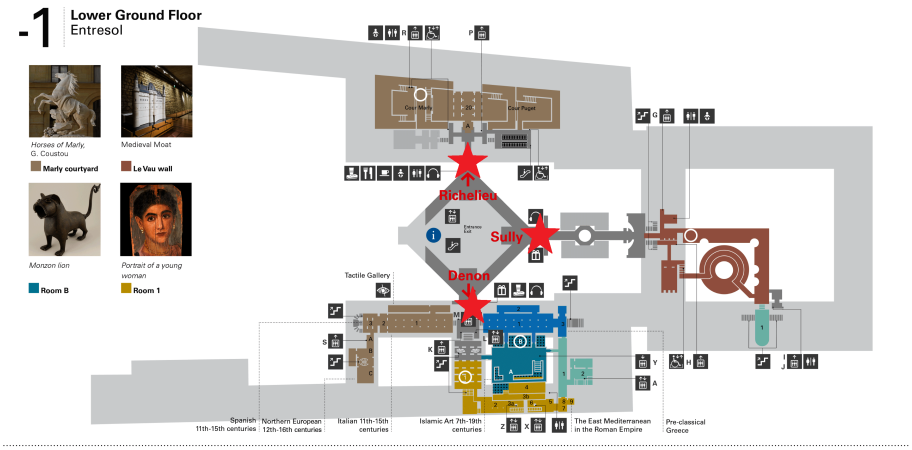


Figure 2: Map of Negative First Floor

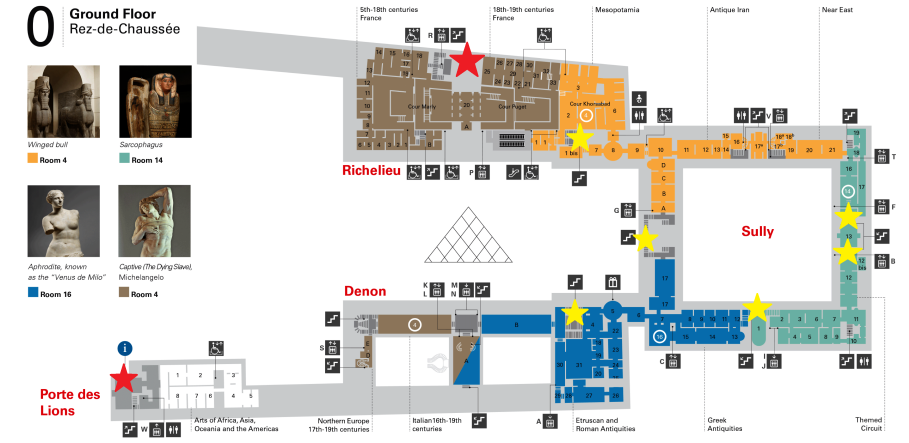


Figure 3: Map of Ground Floor

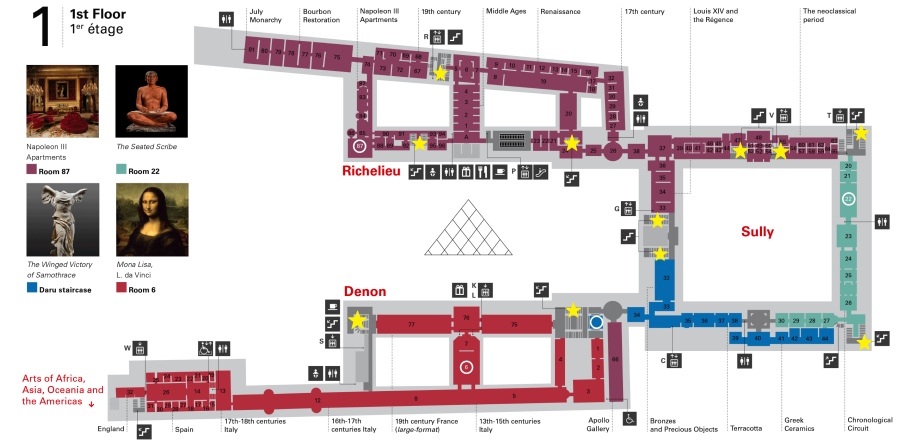


Figure 4: Map of the First Floor

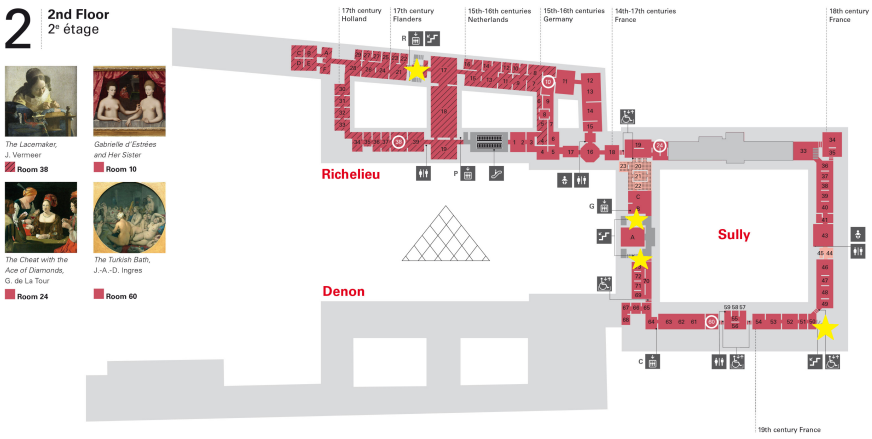


Figure 5: Map of the Second Floor

According to the maps( Maps of other floors are included in the appendices), there are five floors in the Louvre and all of the four entrances are located on the ground floor. The pyramid entrance leads to the negative first floor, so tourists on all the floors have to come to negative first floor if they want to get out through the pyramid entrance. The Passage Richelieu entrance and Portes Des Lions entrance lead to the ground floor, and the Carrousel du Louvre entrance leads to the negative second floor. People who go out through the Passage Richelieu entrance and the Portes Des Lions entrance ought to come to the ground floor and people getting out through Carrousel du Louvre should go to negative second floor. There are one-way and two-way stairs on each floor. For people who want to go downstairs, two-way and one-way downward stairs are usable stairs. For people who want to go upstairs, two-way and one-way upward stairs are usable stairs.In the figure, red stars represent exits and yellow stars represent stairs.

## 4 Outline of Evacuation Routes

Our aim is to find the shortest path from the location of a person to one of the four exits. The shortest path consists of two parts: the shortest path from the person's location to a useable stair and the shortest path from the useable stair to one of the exits.

All the stairs leading to the negative second floor are located in the pyramid. Given the assumption that people are successfully evacuated as soon as they arrive at the entrance, people located above the negative second floor will get out of the building through the pyramid entrance before they arrive at the negative second floor. Therefore, the Carrousel du Louvre entrance is utilized only by people on the negative second floor.

People located on the negative first floor are very close to the pyramid entrance, and it is quite obvious that they will cover a shorter distance when they get out through the pyramid entrance rather than other entrances. Therefore, for the people on the negative first floor, we let the visitors in the three exhibition halls exit through its exhibition hall's own gate, then get out through the pyramids entrance.

We will discuss evacuation routes for other floors which are more complex in the following sections .

# 5 The Basic Model of Evacuation

## 5.1 Local Assumptions

In this section, we only explore the simplest situation. We will discuss other factors that influence the evacuation in the following sections.

- **The walking speed of the visitors is a constant.** We do not consider the influence of crowd density on walking speed or change in the speed over time in this section.
- **Tourists know and use the following exits to evacuate: The pyramid entrance , the Passage Richelieu entrance, the Carrousel du Louvre entrance, and the Portes Des Lions entrance.** We do not utilize additional exits in this section.
- **Visitors distribute uniformly at every spot on the same floor and on every floor in the Louvre.** We do not discuss the variety of guests number or distribution in this section.
- **There is no difference between different individuals.** The diversity of visitors is not taken into consideration at this time.

## 5.2 Network Topology Model

### 5.2.1 Principles of the model

We use Network Topology to simulate the crowd distribution and evacuation routes on every floor. Topology is an abstract method that does not consider physical properties such as size and shape of objects, but only uses points or lines to describe the actual position and relationship of multiple objects. Topology does not care about the details of things, nor does it care about the ratio of one to another, but only expresses the relationship between multiple objects in the form of a graph. Two adjacent nodes are connected by a link.

### 5.2.2 Model Ideas

Through scaling and approximation, the plan of each floor is simplified to the following network topologies.

Since there is no difference between individuals and visitors are uniformly distributed, we equally divide the visitors on a floor into different crowds that are all the same except their locations. We simplify each crowd to a node on the network topology. According to the area and scale of each floor, we set up a node every 40 meters in the east-west direction and every 30 meters in the north-south direction. The link connecting two adjacent nodes represents the evacuation route between two spots.We use $G_k = (N, E)$(k= 0,1,2)to represent the network toplogy. k stands for the number of the floor. N is the set of nodes, and E is the set of links.

Red dots on the figure are nodes. Green stars represent exits. Yellow stars stand for stairs which allow people to go downstairs and red circles represent stairs that allow visitors to go upstairs.

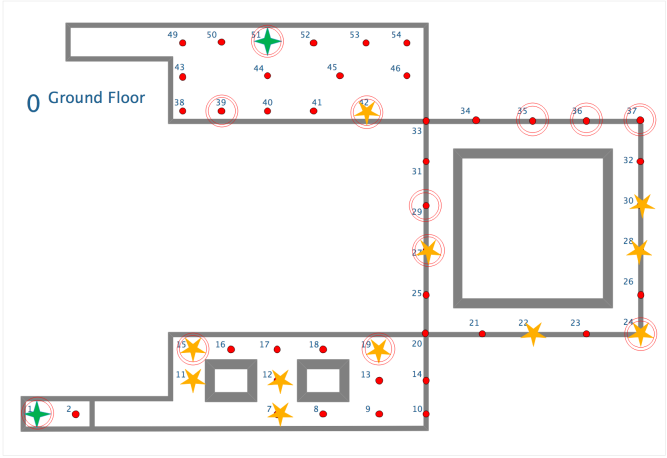


Figure 6: Plan of Ground floor

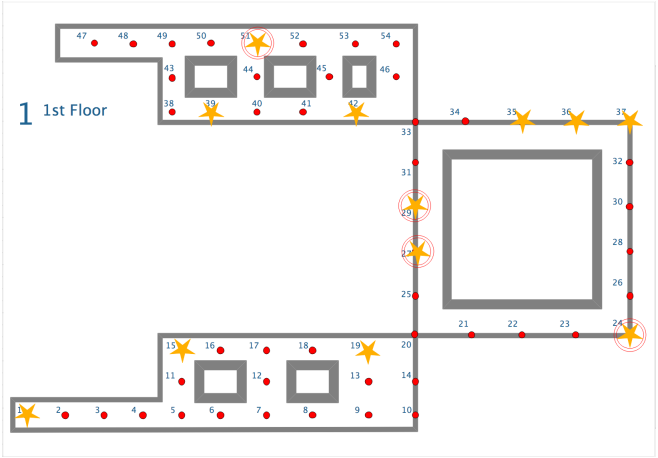


Figure 7: Plan of the First Floor

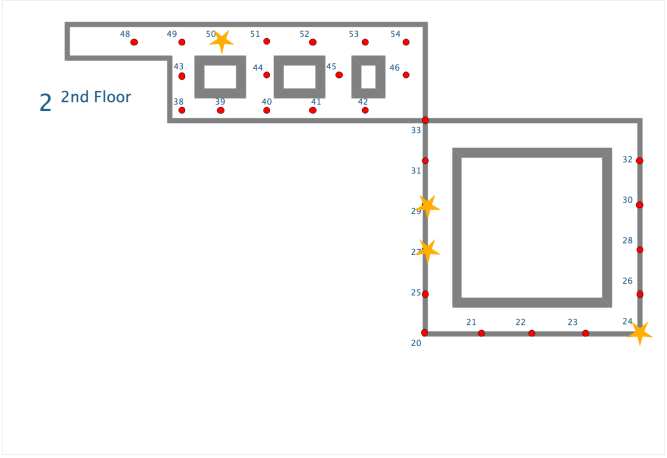


Figure 8: Plan of the Second Floor

## 5.3 Weighed Graphs Model

### 5.3.1 Principles of the model

Graphs that have a number assigned to each edge are called weighted graphs. People use weighted graphs to model computer networks. The sum of the weights of the edges of this path becomes the length of a path in a weighted graph. In our model, we try to determine a path of least length between two vertices in a network.

### 5.3.2 Model Ideas

We utilize weighed connected digraph to simulate the evacuation process. According to the above model, there is a node every 40 meters in the east-west direction and every 30 meters in the north-south direction. In our model, weights are shown in terms of distance, so weight of the link in the east-west direction is 40 and weight of the link in the south-north direction is 30. Given that the walking speed is a constant, the distance is proportional to time.Thus, the weight is proportional to time. In order to make an evacuation plan which is able to get all the visitors out of the building as quickly as possible, we try to figure out the routes along which visitors can use the least of the time to get out. Since the weight is proportional to time, our task can be transformed into finding a path of which the sum of the weights of its edges is least.

## 5.4 The Results of the Model

### 5.4.1 Optimal Evacuation Routes

We make use of Dijkstra Algorithm to figure out the shortest path for each node. C++ is used to realize the model programmatically and we summarize the results in the following figures.

The crowds represented by nodes of the same color are supposed to go to the stairs pointed by the arrow of the same color.

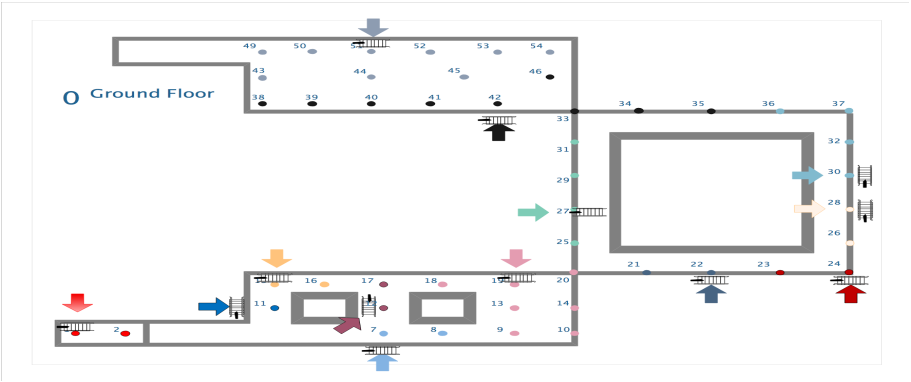


Figure 9: Evacuation Routes for the Ground Floor

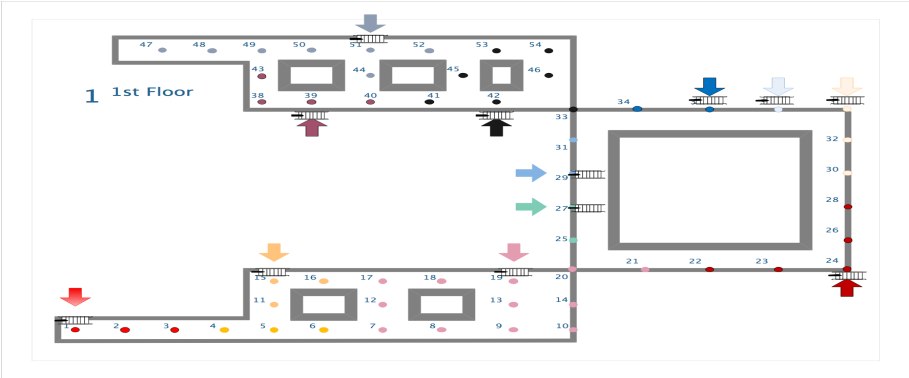


Figure 10: Evacuation Arrange for the First Floor

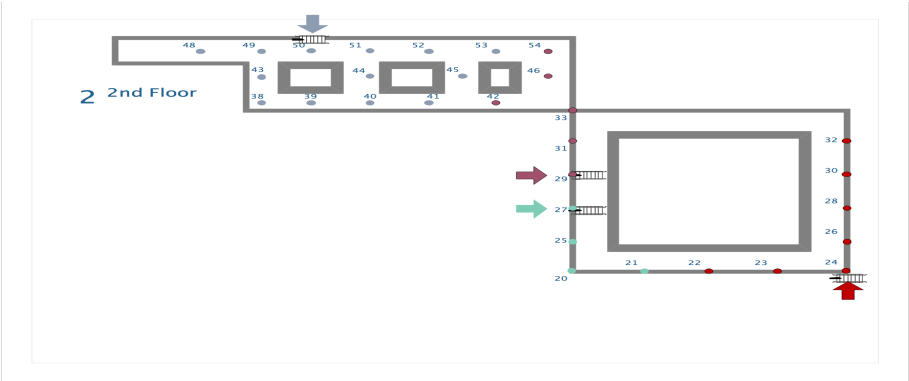


Figure 11: Evacuation Routes for the Second Floor

### 5.4.2 Identify the Bottlenecks

From the above graphs, we can clearly see that there are several stairs that should be obviously more crowded than others, because many crowds are supposed to go to these stairs simultaneously. On the ground floor, stairs 19, 42 and 51 are very crowded. On the first ground, stairs 18, 41 and 51 are very crowded. On the second floor, stairs 50 is very crowded. Therefore, the stairs mentioned above can be the bottlenecks which may limit the movements to the exits.

# 6 Improved Model of the Evacuation

## 6.1 Model Ideas

In this section, we take the influence of change in crowd density on the walking speed into consideration. Let $\Delta t$ be the unit time. Firstly we get the optimal evacuation routes by using the above static model. We assume that visitors distribute uniformly at the beginning. As people begin to move, crowd densities of different parts are no longer the same. Considering the influence of crowd density on walking speed, we construct a dynamic model which gives the change of crowd density over time and then gives the change of the walking speed over time. Given the distance between two adjacent nodes(the length of the link), we are able to calculate the time spent in covering the link. In this section, weights of the edges are shown in terms of time spent covering the edge. Then we use the time-varying weights to find the

shortest evacuation routes. Repeat the above process every $\Delta t$ until the number of people at each node reaches zero.

In this dynamic model, weights change every $Deltat$. Thus, we replan the evacuation routes every $\Delta t$.This model provides time-varying evacuation routes which is more practical.

The following equations are used for iteration:

$$\begin{cases} \rho_{avg} = \frac{C_i(t)+C_j(t)}{2S} \\ v_{ij}(t) = -0.4ln(C_i(t)+C_j(t)) - 2\times 10^{-5}(C_i(t)+C_j(t)) + 4.26 \\ u_{ij}(t) = v_{ij}(t)\sqrt{\rho_{avg}} \\ C_i(t+\Delta t) = C_i(t) + \sum_{j\in\delta^-(i)} u_{ij}(t) - \Delta t \sum_{j\in\delta^+(i)} u_{ji}(t) \end{cases} \tag{1}$$

where:
$\Delta t$ is the unit time;
$\rho_{avg}$ is the average crowd density of node 1 and node j;
$C_i(t)$ is the time-varying number of people in the crowd repre- sented by node i;
$C_j(t)$ is the time-varying number of people in the crowd repre- sented by node j;
$v_{ij}(t)$ is the time-varying walking speed of the crowd;
$u_{ij}(t)$ is the flow of people from node i to node j per unit time;
S is the area of a crowd;
$\delta^-(i)$ isthe set of predecessor nodes linked to the node i;
$\delta^+(i)$ is the set of successor nodes linked to the node i.

The following section shows derivation of the above equations.

### 6.2 Local Assumptions

- The number of people visiting the Louvre everyday is equal to the average numebr of visitors in a day shown on the official website of the Louvre.
- We set the unit time $\Delta t$ as 1s.
- The initial walking speed of people is the normal walking speed without obstacles.
- People represented by the same node always move simultaneously.

### 6.3 Calculations of the Dynamic Model

**Step one: Calculate the initial crowd density.**
According to the official website of the Louvre, the average number of visitors in a day is 32600. The average open time of the Louvre is ten hours a day. We can easily get the hourly visitor volume which is 3260. We assume that visitors stay in the Louvre for approximate 3.5 hours. Then we can get the equation for the number of visitors in the building over time:

$$m(t) = \begin{cases} 3260t & 0 \leqslant t \leqslant 3.5 \\ 11410 & 3.5 < t \leqslant 10 \end{cases} \tag{2}$$

where:
m(t) describes the total number of visitors in the Louvre.
The equation for the average number of people in the Louvre $m_{avg}$is as below:

$$m_{avg} = \frac{1}{10}\int_0^{10} m(t)dt \tag{3}$$

Put the equation (1) into the equation (2), then we get $m_{avg}$ = 9413.
Then we estimate the total area of the Louvre: 30*40*165 = 19800$m^2$
Finally we can get the average crowd density: $\rho_a = \frac{9413}{19800} = 0.48$
Since we assume that people distribute uniformly at the beginning, crowd density of each node is the same. We set the initial crowd density of every node as $\rho_a$. Thus, $\rho_a = \rho_0$.
We suppose that i and j are two adjacent nodes in $G_k = (N, E)$, and that the crowd represented by node i or j covers an area of S. From the above section, the link in the east-west direction is 40 and the link in the south-north direction is 30. Therefore, S=40*30=1200. The initial number of visitors is $C_0$=1200*0.48=576.
**Step two: Figure out the expression of the crowd density of the link over time.**
The crowd densities of i and j can be calculated as followings:

$$\rho_i(t) = \frac{C_I(T)}{S} \tag{4}$$

$$\rho_j(t) = \frac{C_j(T)}{S} \tag{5}$$

where:
$C_i(t)$ is the time-varying number of people in the crowd represented by node i. $C_i(0)$=$\rho_0$= 0.48.
$C_j(t)$ is the time-varying number of people in the crowd represented by node j. $C_j(0)$=$\rho_0$= 0.48.
We set the average of $\rho_i(t)$ and $\rho_j(t)$ as the crowd density of the link connecting i and j.

$$\rho_{avg} = \frac{C_i(t) + C_j(t)}{2S} \tag{6}$$

where:
$\rho_{avg}$ is the crowd density of the link connecting i and j.
**Step three: Find out the equation of walking speed on the link.**
Lu et al.(2002)put forward the relationship between walking speed and crowd density during evacuation(p.69).

$$v_{ij}(\rho_{avg}) = u_m(\alpha A + \beta B + \gamma) \tag{7}$$

$$A = 1.32 - 0.82ln(\rho_{avg})B = 3 - 0.76\rho_{avg} \tag{8}$$

where:
$\alpha$ is between 0.25-0.44;
$\beta$ id between 0.014-0.08;
$\gamma$ is between 0.15-0.26;
$u_m$ is the normal walking speed;
$\rho_{avg}$ is the average crowd density of the link connecting node i and j.
Bohannon (1997) states that the normol walking speed without obstacles is approximately 1.2m/s. We assume that the initial walking speed $v_0$ is the normol walking speed without obstacle. We set $\alpha, \beta \ and \ \gamma$ at their averages and simplify the equations to the following expression.

$$v_{ij}(\rho_{avg}) = 1.15 - 0.4ln(\rho_{avg} - 0.05\rho_{avg} \tag{9}$$

Put equations (5) into (8), we can get the expression of walking speed on the link over time.

$$v_{ij}(t) = -0.4ln(C_i(t) + C_j(t)) - 2 \times 10^{-5}(C_i(t) + C_j(t)) + 4.26 \tag{10}$$

**Step four: Figure out the equation of the flow of crowd density .**
Since we assume that visitors distribute uniformly, the distance between any two people is the

same.Let L represent the distance between two people. The following expressions can be get according to math knowledge:

$$\rho = \frac{1}{L^2} \tag{11}$$

$$u_{ij}(t) = \frac{v_{ij}(t)}{L} \tag{12}$$

where:
$u_{ij}(t)$ represents the flow of crowd density.
$v_{ij}(t)$ represents the walking speed of people.

From the equatios above ,we can get the following equation:

$$u_{ij}(t) = v_{ij}(t)\sqrt{\rho} \tag{13}$$

**Step five: Figure out the iterative formula of the number of people**

The change of the number of people on the node is equal to the sum of influx from all the adjacent nodes minus the sum of outflow to all the adjacent nodes. We assume that the time interval is $\Delta t$. Based on this thought, we can get the following iterative formula:

$$C_i(t + \Delta t) = C_i(t) + \sum_{j \in \delta^-(i)} u_{ij}(t) - \Delta t \sum_{j \in \delta^+(i)} u_{ji}(t) \tag{14}$$

where:
$\delta^-(i)$ represents the set of predecessor nodes linked to the node i.
$\delta^+(i)$ represents the set of successor nodes linked to the node i.
**Step six: Figure out the equation of weights.**

Since we assume that visitors distribute uniformly, the lengths of any two links of the same direction are the same. According to our network topologies established in the last section, links in the east-west direction are 40 meters long and links in the south-north direction are 30 meters long. Therefore,$L_{ew} = 40m$, and $L_{sn} = 30m$. Then we are able to get the following equations of weights:

$$w_{ij}(t) = \begin{cases} \frac{L_{ew}}{v_{ij}(t)} & \text{if the link connecting i and j are in the east-west direction} \\ \frac{L_{sn}}{v_{ij}(t)} & \text{if the link connecting i and j are in the south-north direction} \end{cases} \tag{15}$$

**Step seven: Based on the above equations, begin the iteration process.**

We start from the initial crowd density and use Dijkstra Algorithm to find the shortest path for every node. After one unit of time($\Delta t$), we recalculate the time-varying weights of edges through above equations and then use the same algorithm to get the shortest path.This iteration process should keep going until $C_i$ reaches zero.

### 6.4 Results of the Dynamic model

C++ is used to realize the calculation. We use Iterative Algorithm and Dijkstra Algorithm to get the results. The dynamic evacuation routes are shown in the following figure.

Lines represent evacuation routes, and yellow stars stand for the stairs. Green stars represent exits.

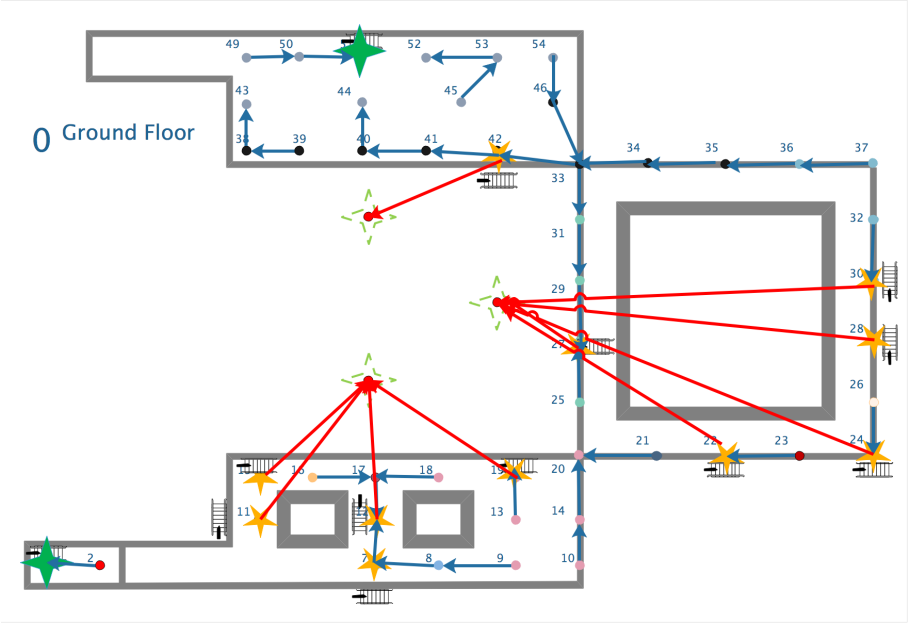


Figure 12: Evacuation Routes for Ground Floor

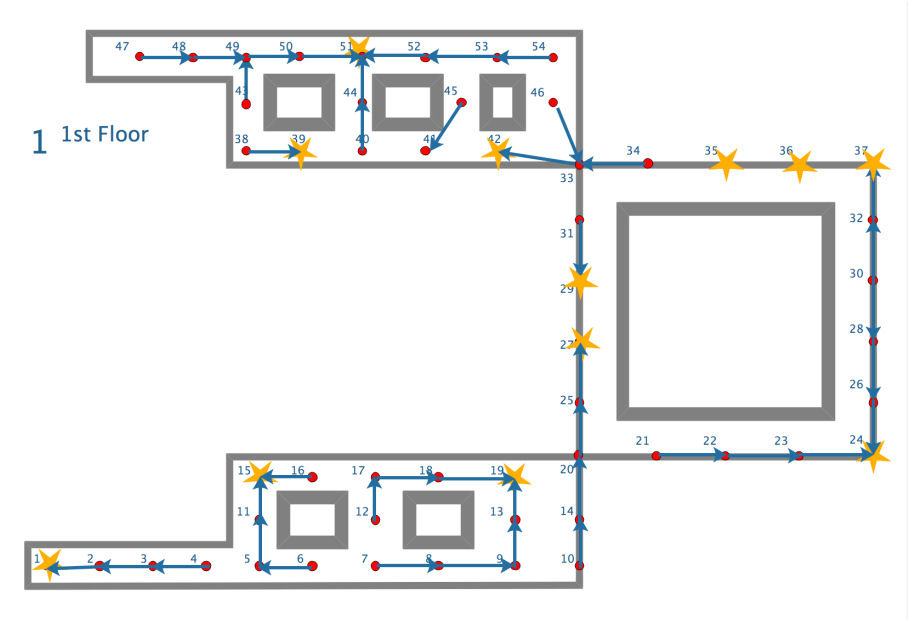


Figure 13: Evacuation Routes for the First Floor

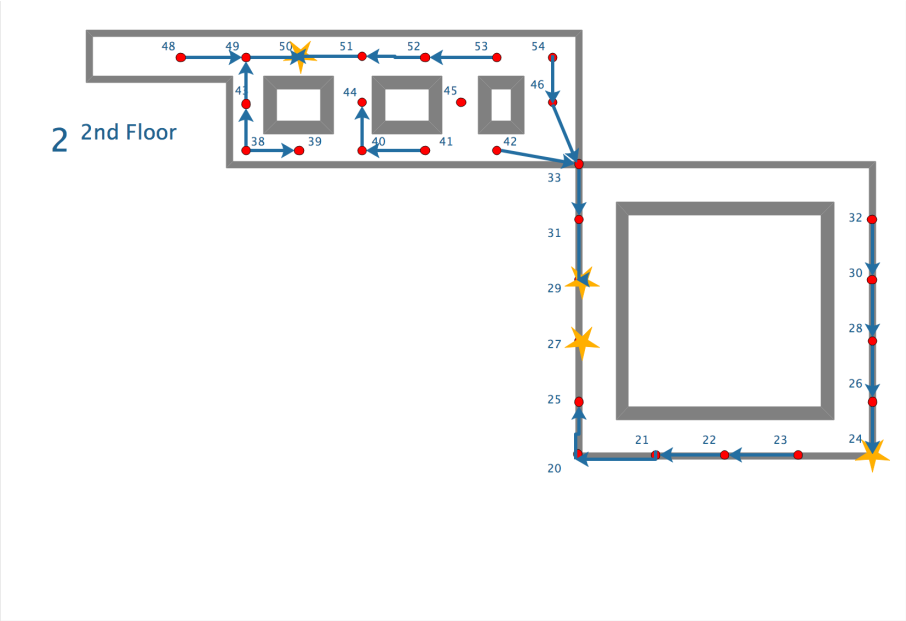


Figure 14: Evacuation Routes for the Second Floor

## 6.5 Identify the Bottlenecks

From the above graphs, we can clearly see that there are some nodes that are more crowded that others.Crowds represented by about 7 to 8 nodes will finally come to these nodes. On the ground floor, node 42 is the bottleneck.On the first floor, node 24 and 51 is the bottlenck. There is no bottleneck on the second floor.

## 6.6 Addition of Exits

Considering that the four main entrances may be crowded with visitors during evacuation, we let all the evacuation personnel enter the building through some additional exits and leave the main entrances for visitors evacuation. Given the lower or limited security postures at these addtional exits, we only allow the emergency personnel to use the additional exits so that no more security concern is caused.(We assume that no visitor knows or uses additional exits.)

We will open two additional exits which are located at node 7 and node 28 on the ground floor. We open additional exits which are at the least crowded stairs so that personnel can easily get to other floors.We use the same method(network topology, weighted graphs and Dijkstra Algorithm) to get the shortest routes from each additional exit to each node. The shortest routes for personnel are shown in the following figure.

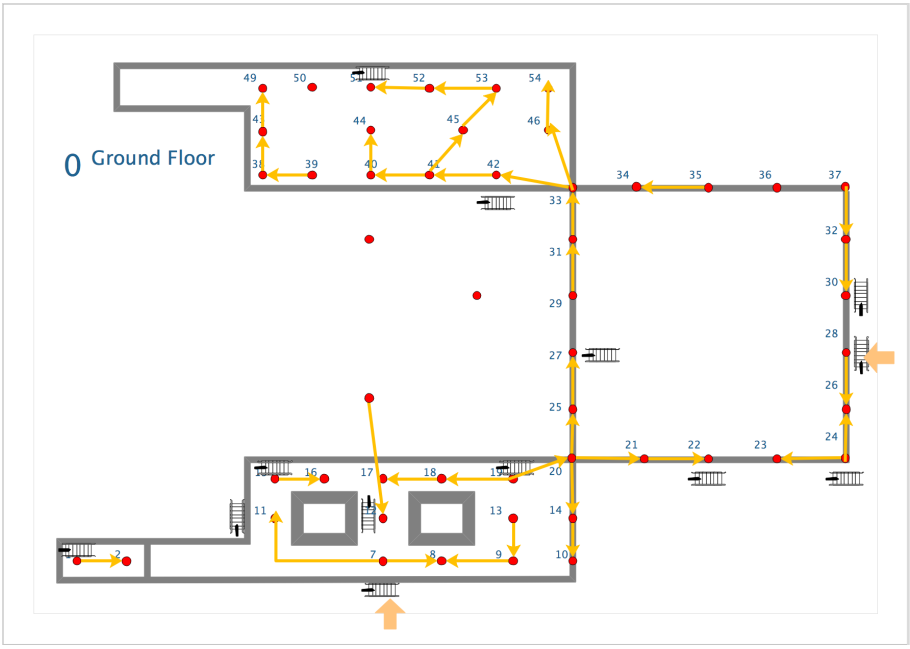


Figure 15: Evacuation Routes for Emergency Personnel

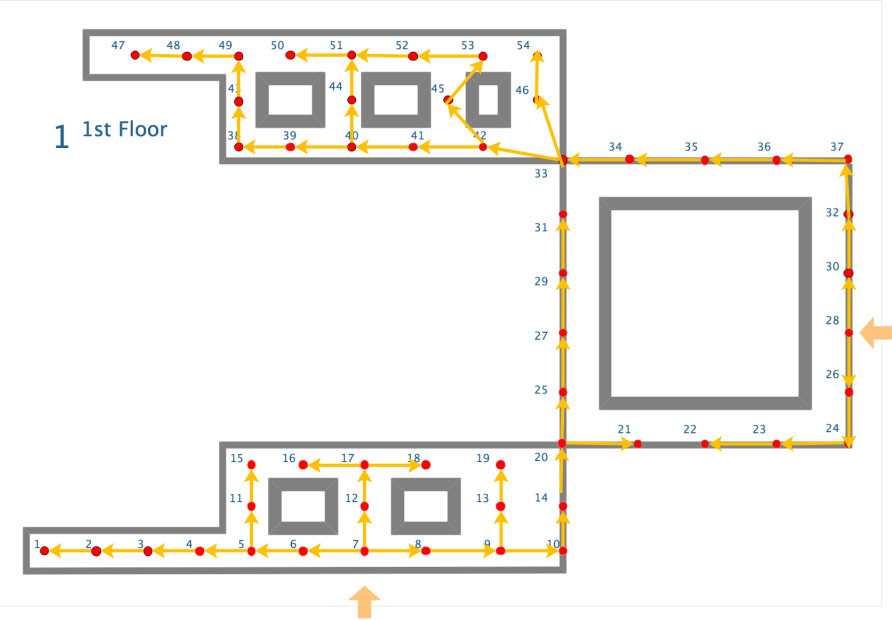


Figure 16: Evacuation Routes for Emergency Personnel

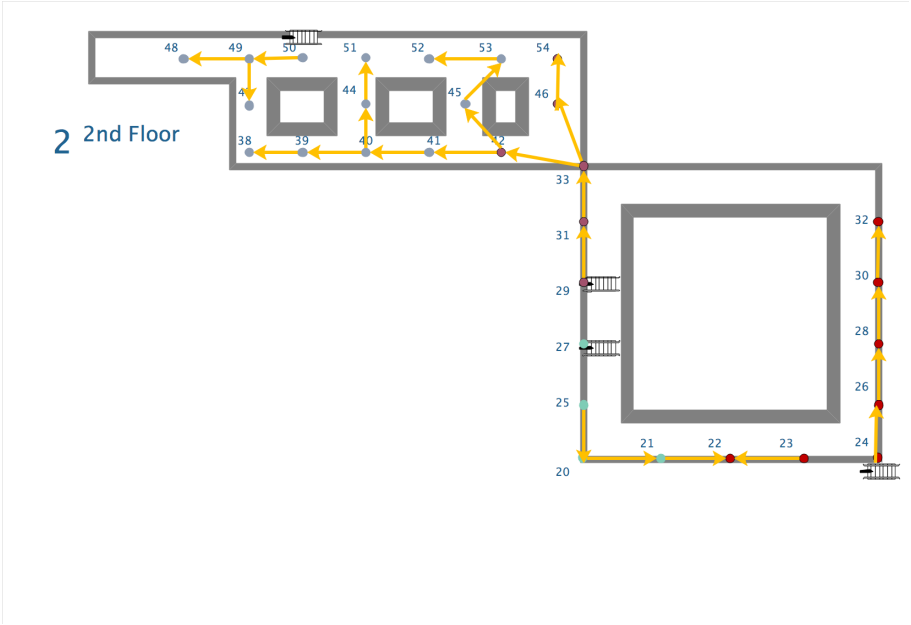


Figure 17: Evacuation Routes for Emergency Personnel

# 7 Model Evaluation and Sensitivity Analysis

In our model, we find dynamic evacuation routes for peopel located everywhere in the Louvre and get the shortest evacuation time. We also figure out the routes for emergency personnel to enter the building as quickly as possible. In this section,we discuss the influence of changes in the following factors on our results.

## 7.1 The Influence of Changes in the Number of Guests Thoughtout the Day

The number of tourists in the museum varies throughout the day and year, so the initial number of visitors may be different from the value we calculate by using the average numebr of visitors in a day shown on the official website of the Louvre. Here we show the change of total evacuation time with different initial number of people.

From the figure above, we can tell that the total evacuation time does not change a lot with various initial numbers. Thus ,our model is adapatable to the changes in the number of guests.

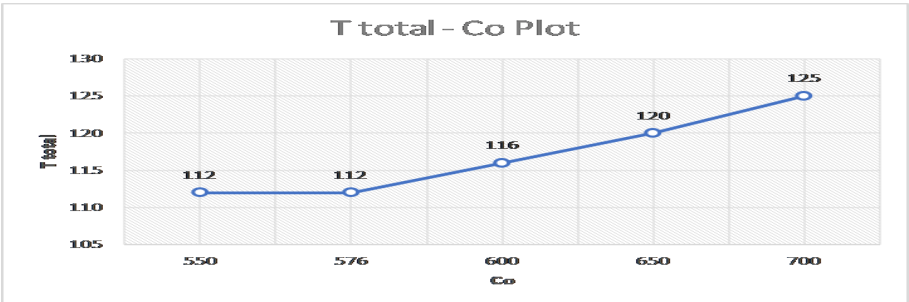


Figure 18: distribution

### 7.2 The Influence of $\Delta t$

As the time interval $\Delta t$increases, we get more practical evacuation routes since we plan the routes for more times. As it decreases, we get more rough routes.The f shows the change of total evacuation time with various $\Delta t$. From the figure, we can see that the total time does not change a lot,which means our results are not sensitive to $\Delta t$. We still can get almost the same results with various $\Delta t$.

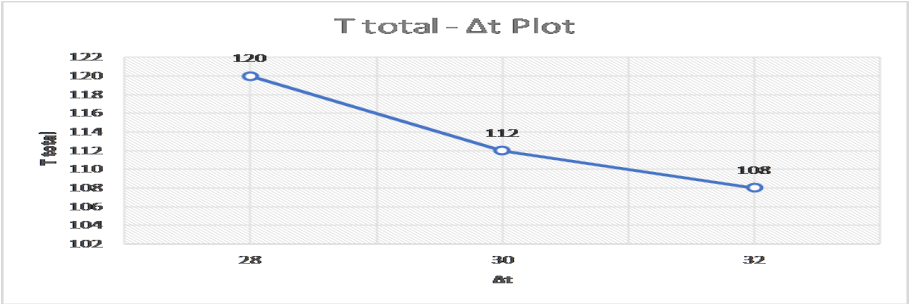


Figure 19: distribution

### 7.3 The Influence of the Diversity of Visitors

Visitors whose native language is not French may need more time to understand the insructions. When a group of people traveling together, they may waste some time waiting for one another. Disabled visitors move slowly. These are all factors that make the initial waiking time various. shows the change of total evacuation time with different walking speed. It is obvious that the total evacuation time does not change a lot with walking speed. Our model can be applied to diverse visitors.

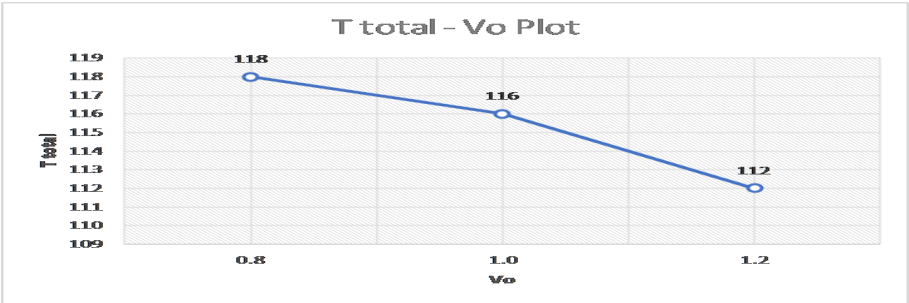


Figure 20: distribution

## 8 Conclusions

### 8.1 Strengths and Weaknesses

Like any model,our model has its strengths and weaknesses. Some of the major points are presented below.

#### 8.1.1 Strengths

- Our model applies widely. The evacuation routes we get can be applied to diverse visitors, and they are also applicable when the number of visitors varies.
- Our model is able to get the shortest evacuation routes for people located anywhere in the Louvre.
- Our model provides time-varying routes which is more practical.

#### 8.1.2 Weaknesses

- Our model is not adapatable to unevenly distributed visitors. We assume people distribute uniformly both in the static and dynamic models.In the real world, people may crowd around some famous displays.
- We simplify the crowd to a node. In fact, the crowd has its own size so people represented by the same node may not walk along the exactly same routes.

### 8.2 Recommdations

- Set a policy: All the visitors should download the upgraded Affluences which can show the real-time evacuation routes.Input the evacuation routes into the application according to our results of dynamic model. according to the location of the visitor.
- Open two additional exits for emergency personnel to get in during evacuation.
- Set exit signs in proper locations especially locations that are not close to stairs or entrance to show the evacuation routes in the stactic model(according to figure 9,10,11) in case that visitors' phone are out of power.
- Museum staffs in different exhibition areas should help lead visitors and prevent people from crowding in certain area.

## References

[1] Chang, L., Zhanli, M., & Zhimin, F., (2016). Emergency evacuation model and algorithm in the building with several exits. *Procedia Engineering*, 135, 12-18

[2] https://wenku.baidu.com/view/652cda3617fc700abb68a98271fe910ef12dae1a.html

[3] Junan, L., Zheng, F., Zhaoming, Lu.,&Chunmei, Z. (2002),Mathematical model of building evacuation velocity,*Engineering Journal of Wuhan University*, 35(2), 66-70

[4] https://www.louvre.fr/sites/default/files/medias/medias_fichiers/fichiers/pdf/louvre-plan-visitors-mobility-impairments

[5] Richard, W.B., (1997). Comfortable and maximum walking speed of adults aged 20-79 years: reference values and determinants. *Age and Ageing*, 26, 15-19

[6] in, P., Lo, S. M., Huang, H. C., & Yuen, K. K. (2008). On the use of multi-stage time-varying quickest time approach for optimization of evacuation planning. Fire Safety Journal, 43(4), 282–290.

[7] UJI YOSHIMURA et al. An analysis of visitors' behavior in The Louvre Museum: a study using Bluetooth data. Environment & Planning B: Planning & Design, [s. l.], v. 41, n. 6, p. 1113–1131, 2014.

[8] ttps://www.louvre.fr/en/plan

[9] ttps://www.louvre.fr/sites/default/files/dp_pyramide

[10] ttp://presse.louvre.fr/10-millions-de-visiteurs-au-louvre-en-2018/

# Appendices

## Appendix A First appendix

Here are algorithm programmes we used in our static model.

**Input C++ source:**

```
//============================================================================
// Name: SSSP
// Author: Pujin Ning
// Version
// Copyright
// Description : shortest path in evacuation
//============================================================================
#include <iostream>
#include <map>
#include <vector>
#include <queue>

#define inf INT_MAX

using namespace std;

int n;

int  dijkstra(int src, int dst, int **wmap, map<int, vector<int>> list)
{
//    cout << src << " go to " << dst << endl;
    if (list[src].empty())
        return inf;
    map<int, bool> visited;
    map<int, int> predecessor;

    int weight[n]; // the list of total weight of every vertex

    //initialize
    for (int i = 0; i < n; i ++)
    {
        visited.insert(pair<int, bool>(i, false));
        weight[i] = inf;
        predecessor.insert(pair<int, int>(i, -1));
//        cout << weight[i] << endl;
    }


    weight[src] = 0;
```

```
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    // the first element in the pair denotes the cumulative weight
    // the second element in the pair denotes the order of the node
    pq.push(pair<int, int>(0, src));


    while (!pq.empty())
    {
        pair<int, int> c = pq.top();
        pq.pop();
        visited[c.second] = true;

        // for every unvisited successor of c in list[c.second], calculate the shortest path
        for(int i = 0; i < list[c.second].size(); i ++)
        {
            int s = list[c.second][i]; // s is the successor of c
//            cout << "successor of " << c.second << " is " << s << endl;

            if (!visited[s])
            {

//                /* cout << "the weight of " << c.second << " is " << c.first << endl;
//                 cout << "the weight on the edge is " << wmap[c.second][s] << endl;
//                 cout << "the current weight of the successor is " << weight[s] << endl;*/
                // if the total weight if the successor can be reduced
                if (c.first + wmap[c.second][s] <= weight[s]) {
                    weight[s] = c.first + wmap[c.second][s];
                    predecessor[s] = c.second;

//                      cout << "push " << weight[s] << " " << s << endl;
                    pq.push(pair<int, int>(weight[s], s));
                }
            }
        }
    }

    vector<int> sssp; //sssp stores the inverted order of the path
    int stat = dst;
    sssp.push_back(dst);
    while (stat != src && predecessor[stat] != -1)
    {
//        cout << "the predecessor of " << stat << " is " << predecessor[stat] << endl;
        stat = predecessor[stat];
        sssp.push_back(stat);
    }

    if (stat == src)
    {
//        cout << "The route is " << endl;
        for (int i = sssp.size() - 1; i >= 0; i--)
            cout << sssp[i] << " ";
        cout << endl;
        return weight[dst];
    } else return inf;
}


int main() {
    freopen("data.txt","r",stdin);
    freopen("personnel.txt","w",stdout);
    cin >> n;
    map<int, vector<int>> list;
    vector<int> exit;
```

```
    int **wmap;
    wmap = new int *[n];
    for (int i = 0; i < n; i ++)
        wmap[i] = new int[n];



    for (int i = 0; i < n; i ++)
    {
        list.insert(pair<int, vector<int>>{i, NULL});
        for (int j = 0; j < n; j ++)
        {
            cin >> wmap[i][j];
            if (wmap[i][j] != -1 && i != j)
                list[i].push_back(j);
        }
    }

//    for (int i = 0; i < list.size(); i ++) {
//        for (int j = 0; j < list[i].size(); j ++) {
//            cout << list [i][j] << " ";
//        }
//        cout << endl;
//    }
//    cout << endl;

    int temp;
    while (cin >> temp)
        exit.push_back(temp);

    int max = 0;

//        cout << "start from " << i << endl;
        int res = inf, index = 0;
        for (int j = 0; j < exit.size(); j ++)
        {
            int path1 = dijkstra(6,exit[j],wmap,list);
            int path2 = dijkstra(27,exit[j],wmap,list);
//            cout << "return " << path << endl;
            if (path1 <= res) {
                res = path1;
                index = 6;
            }
            if (path2 <= res)
            {
                res = path2;
                index = 27;
            }

            if (res >= max)
                max = res;

        cout << "The nearest entrance to " << j << " is " << index << endl;
        }
        cout << endl;


        /*cout << "The nearest exit from " << 0 << " is " << 14 << " length is "
        << dijkstra(0, 14, wmap, list) << endl;*/

    cout << "The longest time is " << max << endl;

    return 0;
```

```
}v.begin(), v.end(), x) != v.end())
        return true;
    else
        return false;
}

bool isEnd(vector<int> exit, map<int, double> quantity)
{
    for (int i = 0; i < n; i ++) {
        cout << quantity[i] << " ";
    }
    cout << endl;
    for (const auto &pair : quantity)
    {
        if (contains(exit, pair.first))
            continue;
        else if (pair.second != 0)
            return false;
    }
    return true;
}

int main() {

    freopen("data.txt","r",stdin);
    freopen("result.txt","w",stdout);

    int initquan;
    double initvel, deltat;
    cin >> n >> area >> initquan;
    cin >> initvel >> deltat; // initial velocity and time gap

    double initdens = (double)initquan / area;
//    cout << "initdens is" << initdens << endl;endl
    double initu = initvel * sqrt(initdens);
    map<int, set<int>> predecessors, successors;
    map<int, vector<int>> list;
    int **wmap;
    double **velocity, **u;

    wmap = new int *[n];
    velocity = new double *[n];
    u = new double *[n];
    for (int i = 0; i < n; i ++)
    {
        wmap[i] = new int[n];
        velocity[i] = new double[n];
        u[i] = new double[n];
    }

    vector<int> exit;     // exit stores the staircase that lead to lower floor

    map<int, double> density;
    map<int, double> quantity;

    for (int i = 0; i < n; i ++)
    {
        set<int> emp;
        predecessors.insert(pair<int, set<int>>(i, emp));
        successors.insert(pair<int, set<int>>(i, emp));
        list.insert(pair<int, vector<int>>(i, NULL));
        for (int j = 0; j < n; j ++)
        {
```

```
            cin >> wmap[i][j];
            if (wmap[i][j] != -1 && i != j)
                list[i].push_back(j);
            velocity[i][j] = initvel;
            u[i][j] = initu;
        }
    }

    int temp;
    while (cin >> temp)
        exit.push_back(temp);


    for (int i = 0; i < n; i ++)
    {
        density.insert(pair<int,double>(i,initdens));
        quantity.insert(pair<int,double>(i,initquan));
    }

    //-------------------------------------------------------------------------------------------------

    int overall = 0;
    while (!isEnd(exit, quantity))
    {

//        cout << "u[i][j] " << endl;
//        for (int i = 0; i < n; i ++) {
//            for (int j = 0; j < n; j++) {
//                cout << u[i][j] << " ";
//            }
//            cout << endl;
//        }
        overall += refresh(wmap, list, predecessors, successors, exit, quantity);
//        cout << "after update" << endl;
        cout << "overall time is " << overall << endl;
//        for (int i = 0; i < n; i ++) {
//            cout << quantity[i] << " ";
//        }
//        cout << endl;
        // update quantity
        for (int i = 0; i < n; i ++)
        {
            double sum1 = 0, sum2 = 0;
            if (predecessors[i].size() > 0) {
                for (int j = 0; j < predecessors[i].size(); j++) {
//                    cout << u[j][i] << endl;
//                    cout <<"delta t is " << deltat << endl;
                    sum1 += (u[j][i] * deltat);
//                    cout << "sum1 now is " << sum1 << endl;
                }
            }
            if (successors[i].size() > 0) {
                for (int j = 0; j < successors[i].size(); j++){
//                    cout << u[j][i] << endl;
                    sum2 += (u[i][j] * deltat);
//                    cout << "sum2 now is " << sum2 << endl;
                }
            }
//            cout << i <<  ": sum1 = " << sum1 << " sum2 = " << sum2 << endl;
            if (sum1 < sum2)
                quantity[i] += (sum1 - sum2);
            else
                quantity[i] -= sum2;
```

```
            if (quantity[i] < 0)
                quantity[i] = 0;
        }

        // update density
        for (int i = 0; i < n; i ++)
            density[i] = (double)quantity[i] / area;

        // update velocity
        for (int i = 0; i < n; i ++) {
            for (int j = 0; j < n; j++) {
                velocity[i][j] = 0.4 * log((double) (quantity[i] + quantity[j])) - 2e-5 *
                        ((double) (quantity[i] + quantity[j])) + 4.26;
                if (velocity[i][j] < 0.0)
                    velocity[i][j] = 0.0;
                if(velocity[i][j] > 1.2)
                    velocity[i][j] = 1.2;
            }
        }

//        cout << "velocity" << endl;
//        for (int i = 0; i < n; i ++) {
//            for (int j = 0; j < n; j++) {
//                cout << velocity[i][j] << " ";
//            }
//            cout << endl;
//        }
//
//        cout << "density" << endl;
//        for (int i = 0; i < n; i ++) {
//            cout << density[i] << " ";
//        }
//        cout << endl;



        // update u
        for (int i = 0; i < n; i ++) {
            for (int j = 0; j < n; j++) {
                u[i][j] = velocity[i][j] * sqrt((density[i] + density[j]) / 2);
//                cout << u[i][j] << " ";
            }
//            cout << endl;
        }
//        for (int i = 0; i < n; i ++) {
//            for (int j = 0; j < n; j++) {
//                cout << u[i][j] << " ";
//            }
//            cout << endl;
//        }

    }
//    for (int i = 0; i < list.size(); i ++) {
//        for (int j = 0; j < list[i].size(); j ++) {
//            cout << list [i][j] << " ";
//        }
//        cout << endl;
//    }
//    cout << endl;
```

```
    return 0;
}
```

# Appendix B Second appendix

Here are algorithm programmes we used in our dynamic model. **Input C++ source:**

```
//============================================================================
// Name: SSSP
// Author: Pujin Ning
// Version
// Copyright
// Description : shortest path in evacuation
//============================================================================
#include <iostream>
#include <map>
#include <vector>
#include <queue>

#define inf INT_MAX

using namespace std;

int n;

int  dijkstra(int src, int dst, int **wmap, map<int, vector<int>> list)
{
//    cout << src << " go to " << dst << endl;
    if (list[src].empty())
        return inf;
    map<int, bool> visited;
    map<int, int> predecessor;

    int weight[n]; // the list of total weight of every vertex

    //initialize
    for (int i = 0; i < n; i ++)
    {
        visited.insert(pair<int, bool>(i, false));
        weight[i] = inf;
        predecessor.insert(pair<int, int>(i, -1));
//        cout << weight[i] << endl;
    }


    weight[src] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    // the first element in the pair denotes the cumulative weight
    // the second element in the pair denotes the order of the node
    pq.push(pair<int, int>(0, src));


    while (!pq.empty())
    {
        pair<int, int> c = pq.top();
        pq.pop();
        visited[c.second] = true;

        // for every unvisited successor of c in list[c.second], calculate the shortest path
        for(int i = 0; i < list[c.second].size(); i ++)
```

```
        {
            int s = list[c.second][i]; // s is the successor of c
//            cout << "successor of " << c.second << " is " << s << endl;

            if (!visited[s])
            {

//                /* cout << "the weight of " << c.second << " is " << c.first << endl;
//                 cout << "the weight on the edge is " << wmap[c.second][s] << endl;
//                 cout << "the current weight of the successor is " << weight[s] << endl;*/
                // if the total weight if the successor can be reduced
                if (c.first + wmap[c.second][s] <= weight[s]) {
                    weight[s] = c.first + wmap[c.second][s];
                    predecessor[s] = c.second;

//                    cout << "push " << weight[s] << " " << s << endl;
                    pq.push(pair<int, int>(weight[s], s));
                }
            }
        }
    }

    vector<int> sssp; //sssp stores the inverted order of the path
    int stat = dst;
    sssp.push_back(dst);
    while (stat != src && predecessor[stat] != -1)
    {
//        cout << "the predecessor of " << stat << " is " << predecessor[stat] << endl;
        stat = predecessor[stat];
        sssp.push_back(stat);
    }

    if (stat == src)
    {
//        cout << "The route is " << endl;
        for (int i = sssp.size() - 1; i >= 0; i--)
            cout << sssp[i] << " ";
        cout << endl;
        return weight[dst];
    } else return inf;
}


int main() {
    freopen("data.txt","r",stdin);
    freopen("personnel.txt","w",stdout);
    cin >> n;
    map<int, vector<int>> list;
    vector<int> exit;
    int **wmap;
    wmap = new int *[n];
    for (int i = 0; i < n; i ++)
        wmap[i] = new int[n];



    for (int i = 0; i < n; i ++)
    {
        list.insert(pair<int, vector<int>>{i, NULL});
        for (int j = 0; j < n; j ++)
        {
            cin >> wmap[i][j];
            if (wmap[i][j] != -1 && i != j)
```

```
                list[i].push_back(j);
        }
    }

//    for (int i = 0; i < list.size(); i ++) {
//        for (int j = 0; j < list[i].size(); j ++) {
//            cout << list [i][j] << " ";
//        }
//        cout << endl;
//    }
//    cout << endl;

    int temp;
    while (cin >> temp)
        exit.push_back(temp);

    int max = 0;

//        cout << "start from " << i << endl;
        int res = inf, index = 0;
        for (int j = 0; j < exit.size(); j ++)
        {
            int path1 = dijkstra(6,exit[j],wmap,list);
            int path2 = dijkstra(27,exit[j],wmap,list);
//            cout << "return " << path << endl;
            if (path1 <= res) {
                res = path1;
                index = 6;
            }
            if (path2 <= res)
            {
                res = path2;
                index = 27;
            }

            if (res >= max)
                max = res;

        cout << "The nearest entrance to " << j << " is " << index << endl;
        }
        cout << endl;


        /*cout << "The nearest exit from " << 0 << " is " << 14 << " length is "
        << dijkstra(0, 14, wmap, list) << endl;*/

    cout << "The longest time is " << max << endl;

    return 0;
}v.begin(), v.end(), x) != v.end())
        return true;
    else
        return false;
}

bool isEnd(vector<int> exit, map<int, double> quantity)
{
    for (int i = 0; i < n; i ++) {
        cout << quantity[i] << " ";
    }
    cout << endl;
    for (const auto &pair : quantity)
    {
```

```
        if (contains(exit, pair.first))
            continue;
        else if (pair.second != 0)
            return false;
    }
    return true;
}

int main() {

    freopen("data.txt","r",stdin);
    freopen("result.txt","w",stdout);

    int initquan;
    double initvel, deltat;
    cin >> n >> area >> initquan;
    cin >> initvel >> deltat; // initial velocity and time gap

    double initdens = (double)initquan / area;
//    cout << "initdens is" << initdens << endl;endl
    double initu = initvel * sqrt(initdens);
    map<int, set<int>> predecessors, successors;
    map<int, vector<int>> list;
    int **wmap;
    double **velocity, **u;

    wmap = new int *[n];
    velocity = new double *[n];
    u = new double *[n];
    for (int i = 0; i < n; i ++)
    {
        wmap[i] = new int[n];
        velocity[i] = new double[n];
        u[i] = new double[n];
    }

    vector<int> exit;     // exit stores the staircase that lead to lower floor

    map<int, double> density;
    map<int, double> quantity;

    for (int i = 0; i < n; i ++)
    {
        set<int> emp;
        predecessors.insert(pair<int, set<int>>(i, emp));
        successors.insert(pair<int, set<int>>(i, emp));
        list.insert(pair<int, vector<int>>(i, NULL));
        for (int j = 0; j < n; j ++)
        {
            cin >> wmap[i][j];
            if (wmap[i][j] != -1 && i != j)
                list[i].push_back(j);
            velocity[i][j] = initvel;
            u[i][j] = initu;
        }
    }

    int temp;
    while (cin >> temp)
        exit.push_back(temp);


    for (int i = 0; i < n; i ++)
```

```
    {
        density.insert(pair<int,double>(i,initdens));
        quantity.insert(pair<int,double>(i,initquan));
    }

    //-------------------------------------------------------------------------------------------

    int overall = 0;
    while (!isEnd(exit, quantity))
    {

//        cout << "u[i][j] " << endl;
//        for (int i = 0; i < n; i ++) {
//            for (int j = 0; j < n; j++) {
//                cout << u[i][j] << " ";
//            }
//            cout << endl;
//        }
        overall += refresh(wmap, list, predecessors, successors, exit, quantity);
//        cout << "after update" << endl;
        cout << "overall time is " << overall << endl;
//        for (int i = 0; i < n; i ++) {
//            cout << quantity[i] << " ";
//        }
//        cout << endl;
        // update quantity
        for (int i = 0; i < n; i ++)
        {
            double sum1 = 0, sum2 = 0;
            if (predecessors[i].size() > 0) {
                for (int j = 0; j < predecessors[i].size(); j++) {
//                    cout << u[j][i] << endl;
//                    cout <<"delta t is " << deltat << endl;
                    sum1 += (u[j][i] * deltat);
//                    cout << "sum1 now is " << sum1 << endl;
                }
            }
            if (successors[i].size() > 0) {
                for (int j = 0; j < successors[i].size(); j++){
//                    cout << u[j][i] << endl;
                    sum2 += (u[i][j] * deltat);
//                    cout << "sum2 now is " << sum2 << endl;
                }
            }
//            cout << i <<  ": sum1 = " << sum1 << " sum2 = " << sum2 << endl;
            if (sum1 < sum2)
                quantity[i] += (sum1 - sum2);
            else
                quantity[i] -= sum2;

            if (quantity[i] < 0)
                quantity[i] = 0;
        }

        // update density
        for (int i = 0; i < n; i ++)
            density[i] = (double)quantity[i] / area;

        // update velocity
        for (int i = 0; i < n; i ++) {
            for (int j = 0; j < n; j++) {
                velocity[i][j] = 0.4 * log((double) (quantity[i] + quantity[j])) - 2e-5 *
                        ((double) (quantity[i] + quantity[j])) + 4.26;
```

```
                if (velocity[i][j] < 0.0)
                    velocity[i][j] = 0.0;
                if(velocity[i][j] > 1.2)
                    velocity[i][j] = 1.2;
            }
        }

//        cout << "velocity" << endl;
//        for (int i = 0; i < n; i ++) {
//            for (int j = 0; j < n; j++) {
//                cout << velocity[i][j] << " ";
//            }
//            cout << endl;
//        }
//
//        cout << "density" << endl;
//        for (int i = 0; i < n; i ++) {
//            cout << density[i] << " ";
//        }
//        cout << endl;



        // update u
        for (int i = 0; i < n; i ++) {
            for (int j = 0; j < n; j++) {
                u[i][j] = velocity[i][j] * sqrt((density[i] + density[j]) / 2);
//                cout << u[i][j] << " ";
            }
//            cout << endl;
        }
//        for (int i = 0; i < n; i ++) {
//            for (int j = 0; j < n; j++) {
//                cout << u[i][j] << " ";
//            }
//            cout << endl;
//        }

    }
//    for (int i = 0; i < list.size(); i ++) {
//        for (int j = 0; j < list[i].size(); j ++) {
//            cout << list [i][j] << " ";
//        }
//        cout << endl;
//    }
//    cout << endl;



    return 0;
}
```

# Appendix C Third appendix

Here are algorithm programmes we used to find optional routes for emergency personnel.
**Input C++ source:**

```
//============================================================================
// Name: SSSP
// Author: Pujin Ning
```

```
// Version
// Copyright
// Description : shortest path in evacuation
//============================================================================
#include <iostream>
#include <map>
#include <vector>
#include <queue>

#define inf INT_MAX

using namespace std;

int n;

int  dijkstra(int src, int dst, int **wmap, map<int, vector<int>> list)
{
//    cout << src << " go to " << dst << endl;
    if (list[src].empty())
        return inf;
    map<int, bool> visited;
    map<int, int> predecessor;

    int weight[n]; // the list of total weight of every vertex

    //initialize
    for (int i = 0; i < n; i ++)
    {
        visited.insert(pair<int, bool>(i, false));
        weight[i] = inf;
        predecessor.insert(pair<int, int>(i, -1));
//        cout << weight[i] << endl;
    }


    weight[src] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    // the first element in the pair denotes the cumulative weight
    // the second element in the pair denotes the order of the node
    pq.push(pair<int, int>(0, src));


    while (!pq.empty())
    {
        pair<int, int> c = pq.top();
        pq.pop();
        visited[c.second] = true;

        // for every unvisited successor of c in list[c.second], calculate the shortest path
        for(int i = 0; i < list[c.second].size(); i ++)
        {
            int s = list[c.second][i]; // s is the successor of c
//            cout << "successor of " << c.second << " is " << s << endl;

            if (!visited[s])
            {

//                /* cout << "the weight of " << c.second << " is " << c.first << endl;
//                 cout << "the weight on the edge is " << wmap[c.second][s] << endl;
//                 cout << "the current weight of the successor is " << weight[s] << endl;*/
                // if the total weight if the successor can be reduced
                if (c.first + wmap[c.second][s] <= weight[s]) {
                    weight[s] = c.first + wmap[c.second][s];
```

```
                    predecessor[s] = c.second;

//                    cout << "push " << weight[s] << " " << s << endl;
                    pq.push(pair<int, int>(weight[s], s));
                }
            }
        }
    }

    vector<int> sssp; //sssp stores the inverted order of the path
    int stat = dst;
    sssp.push_back(dst);
    while (stat != src && predecessor[stat] != -1)
    {
//        cout << "the predecessor of " << stat << " is " << predecessor[stat] << endl;
        stat = predecessor[stat];
        sssp.push_back(stat);
    }

    if (stat == src)
    {
//        cout << "The route is " << endl;
        for (int i = sssp.size() - 1; i >= 0; i--)
            cout << sssp[i] << " ";
        cout << endl;
        return weight[dst];
    } else return inf;
}


int main() {
    freopen("data.txt","r",stdin);
    freopen("personnel.txt","w",stdout);
    cin >> n;
    map<int, vector<int>> list;
    vector<int> exit;
    int **wmap;
    wmap = new int *[n];
    for (int i = 0; i < n; i ++)
        wmap[i] = new int[n];



    for (int i = 0; i < n; i ++)
    {
        list.insert(pair<int, vector<int>>{i, NULL});
        for (int j = 0; j < n; j ++)
        {
            cin >> wmap[i][j];
            if (wmap[i][j] != -1 && i != j)
                list[i].push_back(j);
        }
    }

//    for (int i = 0; i < list.size(); i ++) {
//        for (int j = 0; j < list[i].size(); j ++) {
//            cout << list [i][j] << " ";
//        }
//        cout << endl;
//    }
//    cout << endl;

    int temp;
```

```
    while (cin >> temp)
        exit.push_back(temp);

    int max = 0;

//        cout << "start from " << i << endl;
        int res = inf, index = 0;
        for (int j = 0; j < exit.size(); j ++)
        {
            int path1 = dijkstra(6,exit[j],wmap,list);
            int path2 = dijkstra(27,exit[j],wmap,list);
//            cout << "return " << path << endl;
            if (path1 <= res) {
                res = path1;
                index = 6;
            }
            if (path2 <= res)
            {
                res = path2;
                index = 27;
            }

            if (res >= max)
                max = res;

        cout << "The nearest entrance to " << j << " is " << index << endl;
        }
        cout << endl;


        /*cout << "The nearest exit from " << 0 << " is " << 14 << " length is "
        << dijkstra(0, 14, wmap, list) << endl;*/

    cout << "The longest time is " << max << endl;

    return 0;
}v.begin(), v.end(), x) != v.end())
        return true;
    else
        return false;
}

bool isEnd(vector<int> exit, map<int, double> quantity)
{
    for (int i = 0; i < n; i ++) {
        cout << quantity[i] << " ";
    }
    cout << endl;
    for (const auto &pair : quantity)
    {
        if (contains(exit, pair.first))
            continue;
        else if (pair.second != 0)
            return false;
    }
    return true;
}

int main() {

    freopen("data.txt","r",stdin);
    freopen("result.txt","w",stdout);
```

```
    int initquan;
    double initvel, deltat;
    cin >> n >> area >> initquan;
    cin >> initvel >> deltat; // initial velocity and time gap

    double initdens = (double)initquan / area;
//    cout << "initdens is" << initdens << endl;endl
    double initu = initvel * sqrt(initdens);
    map<int, set<int>> predecessors, successors;
    map<int, vector<int>> list;
    int **wmap;
    double **velocity, **u;

    wmap = new int *[n];
    velocity = new double *[n];
    u = new double *[n];
    for (int i = 0; i < n; i ++)
    {
        wmap[i] = new int[n];
        velocity[i] = new double[n];
        u[i] = new double[n];
    }

    vector<int> exit;     // exit stores the staircase that lead to lower floor

    map<int, double> density;
    map<int, double> quantity;

    for (int i = 0; i < n; i ++)
    {
        set<int> emp;
        predecessors.insert(pair<int, set<int>>(i, emp));
        successors.insert(pair<int, set<int>>(i, emp));
        list.insert(pair<int, vector<int>>(i, NULL));
        for (int j = 0; j < n; j ++)
        {
            cin >> wmap[i][j];
            if (wmap[i][j] != -1 && i != j)
                list[i].push_back(j);
            velocity[i][j] = initvel;
            u[i][j] = initu;
        }
    }

    int temp;
    while (cin >> temp)
        exit.push_back(temp);


    for (int i = 0; i < n; i ++)
    {
        density.insert(pair<int,double>(i,initdens));
        quantity.insert(pair<int,double>(i,initquan));
    }

    //-------------------------------------------------------------------------------------------------

    int overall = 0;
    while (!isEnd(exit, quantity))
    {

//        cout << "u[i][j] " << endl;
//        for (int i = 0; i < n; i ++) {
```

```
//            for (int j = 0; j < n; j++) {
//                cout << u[i][j] << " ";
//            }
//            cout << endl;
//        }
        overall += refresh(wmap, list, predecessors, successors, exit, quantity);
//        cout << "after update" << endl;
        cout << "overall time is " << overall << endl;
//        for (int i = 0; i < n; i ++) {
//            cout << quantity[i] << " ";
//        }
//        cout << endl;
        // update quantity
        for (int i = 0; i < n; i ++)
        {
            double sum1 = 0, sum2 = 0;
            if (predecessors[i].size() > 0) {
                for (int j = 0; j < predecessors[i].size(); j++) {
//                    cout << u[j][i] << endl;
//                    cout <<"delta t is " << deltat << endl;
                    sum1 += (u[j][i] * deltat);
//                    cout << "sum1 now is " << sum1 << endl;
                }
            }
            if (successors[i].size() > 0) {
                for (int j = 0; j < successors[i].size(); j++){
//                    cout << u[j][i] << endl;
                    sum2 += (u[i][j] * deltat);
//                    cout << "sum2 now is " << sum2 << endl;
                }
            }
//            cout << i <<  ": sum1 = " << sum1 << " sum2 = " << sum2 << endl;
            if (sum1 < sum2)
                quantity[i] += (sum1 - sum2);
            else
                quantity[i] -= sum2;

            if (quantity[i] < 0)
                quantity[i] = 0;
        }

        // update density
        for (int i = 0; i < n; i ++)
            density[i] = (double)quantity[i] / area;

        // update velocity
        for (int i = 0; i < n; i ++) {
            for (int j = 0; j < n; j++) {
                velocity[i][j] = 0.4 * log((double) (quantity[i] + quantity[j])) - 2e-5 *
                        ((double) (quantity[i] + quantity[j])) + 4.26;
                if (velocity[i][j] < 0.0)
                    velocity[i][j] = 0.0;
                if(velocity[i][j] > 1.2)
                    velocity[i][j] = 1.2;
            }
        }

//        cout << "velocity" << endl;
//        for (int i = 0; i < n; i ++) {
//            for (int j = 0; j < n; j++) {
//                cout << velocity[i][j] << " ";
//            }
//            cout << endl;
```

```
//         }
//
//         cout << "density" << endl;
//         for (int i = 0; i < n; i ++) {
//             cout << density[i] << " ";
//         }
//         cout << endl;



        // update u
        for (int i = 0; i < n; i ++) {
            for (int j = 0; j < n; j++) {
                u[i][j] = velocity[i][j] * sqrt((density[i] + density[j]) / 2);
//                 cout << u[i][j] << " ";
            }
//             cout << endl;
        }
//         for (int i = 0; i < n; i ++) {
//             for (int j = 0; j < n; j++) {
//                 cout << u[i][j] << " ";
//             }
//             cout << endl;
//         }

    }
//     for (int i = 0; i < list.size(); i ++) {
//         for (int j = 0; j < list[i].size(); j ++) {
//             cout << list [i][j] << " ";
//         }
//         cout << endl;
//     }
//     cout << endl;



    return 0;
}
```