

An Optimization Method for Campus Routing Problem

Jiasheng Ni, Liyuan Geng

May 2023

1 Introduction

1.1 Background

As NYU Shanghai has built its beautiful New Bund Campus, there are many visitors coming to the new campus, and having a campus tour. Since most visitors have a limited amount of time, the campus tour should be efficient. This project aims to optimize visitors' route of campus tours in order to maximize their campus tour experience within a limited amount of time. Therefore, we seek to construct a mathematical model to choose an optimal visiting route that maximizes the total weights of each part of the campus they have visited, with a time limit as a general constraint.

Routing problems, also known as path-finding problems, refer to the task of finding the optimal path between two points in a graph or network. The objective is typically to find the path that minimizes a given cost or distance metric while adhering to any constraints or limitations that may exist in the network. There are many algorithms to find the optimal path, some use the classical algorithm like A^* algorithm and genetic algorithms [CLY22]. But they all took some heuristic method in settling the path and in certain fields like robotics these algorithms work very well. But in our project, we don't want to include randomness to complicate our model. Thus, we won't use these algorithms. Instead, the backbone theory behind our mathematical optimization model [CE14] is the traditional routing finding algorithm. The modeling process basically follows the optimization setting under the framework of capacitated vehicle routing problems (CVRP in short, see Appendix A).

1.2 Motivations

In this project, we would like to develop an optimization model that given the multi-layered floor plans for our campus, will deliver the best routing strategy for an individual visitor so that following this optimal route, the visitor can maximize their visiting experience within the time limit. Then, the optimal routing can be compared with the current visiting routes that are adopted for visitors every Friday. In practice, such an optimal finding algorithm can be used to reduce travel time, making it more convenient for students, faculty, and staff to move between classes, meetings, and events.

Efficient routing can lead to better time management and increased productivity. Also during peak visiting hours, walkways can become crowded, leading to congestion and delays, and by optimizing the routing, congestion can be reduced, making the campus more accessible and comfortable for everyone.

1.3 Previous Work

The most difficult part of our project is trying to find the proper graph representation of the floor plan. Previous work includes using combinatorial maps and their duals [YW15] to generate an embedded graph that represents the connectivity and topology of the floor plan structure, using a multilayered space-event model [BNK09] for indoor navigation, which involves converting floor plans into a graph model with nodes and edges representing locations and connections in the building and introducing the “door-to-door” routing method [LZ11] to support the natural movement in buildings, typically for emergencies. However, most of these methods are in essence multi-source path-finding, which means the routing starts at an arbitrary node and ends somewhere else. This is different from our goal where the path starts and ends at the same position. But these algorithms do give us inspiration on how the floor plan can be converted into the corresponding graph models.

Once we have the graphical representation of the floor plan, the next part of our project is to run an optimization algorithm to find the optimal routing. Pedro et.al [MDS17] offers one possible algorithm, which was initially developed to solve capacitated vehicle routing problem(CVRP) in polynomial time complexity. We will follow the optimization scheme from this paper, which will be detailed on below.

2 Model Formulation

2.1 Model Explanations

2.1.1 Notations

We denote the graph representation of each floor as $G = \{V, \mathcal{E}\}$, in which $V = \mathcal{P} \cup \{0, n + 1\}$ is the set of nodes that represent the places that visitors will potentially visit. Here we use two nodes 0 and $n + 1$ to represent the same depot for the sake of notation simplicity. The set \mathcal{E} contains the directed edge (i, j) for each pair of nodes $i, j \in V$ (We assume a directed graph but used in an undirected way. i.e. For each edge from node i to node j , there is an edge from j to i , which implies our model formulation).

2.1.2 Model Parameters

1. **Edge Crossing Time, t_{ij} :** The time needed to cross an edge $(i, j) \in \mathcal{E}$ is denoted by t_{ij} (In matrix notation we have \mathbf{T}).

2. **Node Importance, w_i :** Each node has a weight w_i , such that $w_i > 0$ for each $i \in \mathcal{P}$ and $w_0 = w_{n+1} = 0$.
3. **Node Adjacency, a_{ij} :** We use a binary variable $a_{ij} \in \{0, 1\}$ to denote whether node i and node j are directly connected by an edge. (In matrix notation it is \mathbf{A}).

The parameter tables are appendix B.

Parameter	Meaning
t_{ij} \mathbf{T}	Time used to cross the edge between any two nodes, in seconds.
a_{ij} , \mathbf{A}	Binary number 1 or 0 signifying whether there is an edge between any two nodes
w_i	The node weight, represented in a vector \vec{w}
c	The time it takes to travel between the depots of different floors
T	Total time limit of the campus tour

2.1.3 Decision Variables

1. **Edge Choice Flag Variable, x_{ij} :** In two-index vehicle flow formulation [MDS17], we denote x_{ij} to be a binary variable $x_{ij} \in \{0, 1\}$ that represents whether there is an edge from node $i \in V$ to node $j \in V$.
2. **Cumulative Cost, u_i :** We use a continuous variable u_i for each vertex to represent the time that has passed since we started from the depot until we visited this node i to create a concept of the sequence of visited nodes. These decision variables are used to eliminate the sub-tour problems in polynomial time.

2.1.4 Model Assumptions

1. **No Isolated Vertices(Mandatory):** We assume that all the places are somehow reachable, i.e. there exists a walk between any pair of vertices.
2. **No Antenna Vertices(Optional):** For simplicity consideration, we assume that every vertex should be adjacent to at least two other vertices. i.e.

$$\sum_{j=1}^n x_{ij} \geq 2 \quad \forall i = 1, 2, \dots, n$$

, but this should not act as a constraint. We will use this assumption to simplify our optimization model. To visually interpret this, Fig 21 is unsatisfactory because node 3 and node 5 are both antenna nodes. Fig 22, obtained by adding an edge between node 3 and node 5, is valid.

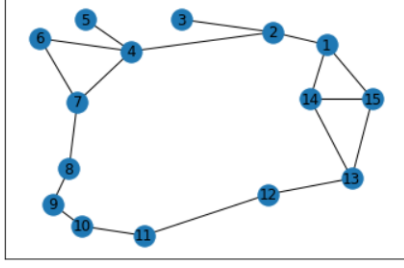


Figure 1: Unsatisfactory Graph with antenna vertices

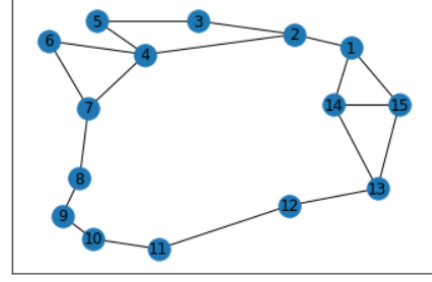


Figure 2: Valid Graph

We have to admit that such simplification is a bit away from reality, but it saves our modeling process by a huge amount.

3. **Constant touring speed(Mandatory):** We assume that walking speed is a constant c across all edges and should not be affected by the importance of the places to be visited and should only be dependent on the distance between the vertices that the edge is incident to. Mathematically, there is a pure linear relationship between the distance and the traveling time.
4. **Floor by Floor(Mandatory):** We assume that the tourist starts to visit the campus from the depot of 3F, and ends at the depot of 5F. Naturally, the tourist will not go back to the floor after visiting it. Based on this point, we assume that the tourist visits the campus floor by floor.
5. **Same depot across floor:** We assume that the locations of the depots for each floor are the same. More preciously, we will use one node to represent multiple depots across the floor. See Fig 9 for details.

2.1.5 Constraints

1. **Visited less than Once:** We require that each node(except for the depot) should be visited once or never, so we have:

$$\sum_{\substack{j=1 \\ j \neq i}}^{n+1} x_{ij} \leq 1, \quad i = 1, \dots, n$$

. This is different from the TSP problem or CVRP where each vertex should be visited exactly once. **This constraint should still be valid in multi-floor cases.**

2. **Must depart from the depot:** We require that the optimal route must begin with the depot. Mathematically, the out-degree of the node 0(node $n + 1$) must be exactly one, meaning that there is exactly one edge that is incident to the depot:

$$\sum_{j \in V \setminus \{0\}} x_{0j} = 1$$

In the multi-floor case, this constraint should be slightly modified. According to our model assumption, we regard the depot nodes from different floors to be overlapped. This means we have to depart multiple times from the same depot node, once for each floor. Thus the constraint should be changed to:

$$\sum_{j \in V \setminus \{0\}} x_{0j} \geq 1$$

3. **Degree Match:** We require that if the visitors arrive at a certain node, then it has to depart from that node. In other words, the in-degree and the out-degree of each node must be the same, so we have the following constraint:

$$\sum_{\substack{i=0 \\ i \neq h}}^n x_{ih} - \sum_{\substack{j=1 \\ j \neq h}}^{n+1} x_{hj} = 0, \quad h = 1, \dots, n$$

This constraint should still be valid in multi-floor cases.

4. **No Shuttle Run (optional):** We require that if the visitors arrive at a certain node, then it has to depart from that node and never go back to its previous node again:

$$x_{ij} \neq x_{ji} \quad \forall i, j$$

. This constraint, together with constraint number 2, controls that there must be an edge that starts from the depot and ends in the depot.

5. **Time Limit:** The total time limit is denoted by T , saying that the time consumption over the chosen subset of edges $\mathcal{C} \subseteq \mathcal{E}$ should be less or equal to T , thus we have:

$$\sum_{i \in V} \sum_{j \in V \setminus \{0\}, i \neq j} t_{ij} x_{ij} \leq T$$

This constraint should still be valid in multi-floor cases.

6. **Path Validity Control:** The edge that we choose must be an edge that is contained in the graph $G = \{V, \mathcal{E}\}$:

$$x_{ij} \leq a_{ij}, \quad \forall i = 1, 2, \dots, n, j = 1, 2, \dots, n$$

, without this constraint, the subset of edges that the solver chooses will contain some edges that don't exist in our initial graph. Fig. 3 shows this scenario.

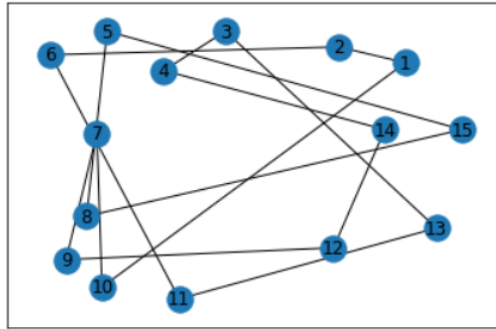


Figure 3: Non-existing edges

This constraint should still be valid in multi-floor cases.

7. **Binary Constraint:** The decision variables must be binary:

$$x_{ij} \in \{0, 1\}, \quad \forall (i, j) \in \mathcal{E}$$

This constraint should still be valid in multi-floor cases.

8. Subtour Elimination Constraint:

$$u_i - u_j + t_{ij}x_{ij} \leq T(1 - x_{ij})$$

for some sufficient large constant T . This serves to eliminate the subtour problems that may occur. Fig. 4 shows the subtour scenario that we want to eliminate. The edge colored in red means that both x_{ij} and x_{ji} are selected, which is indeed a cycle.

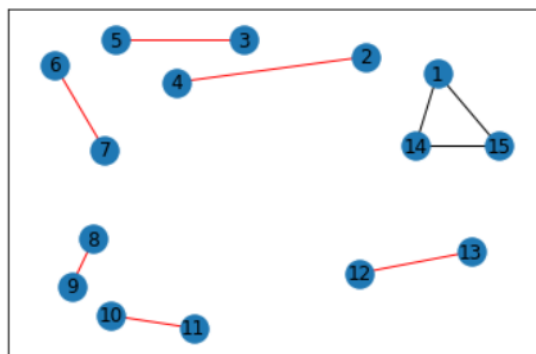


Figure 4: Subtour Problems

This constraint should still be valid in multi-floor cases and the algorithmic details will be introduced in later sections.

2.2 Subtour Elimination

Subtour problem, which is quite famous in the Traveling Salesman Problem(TSP) and is defined to be a subset of edges that form a cycle but do not visit all nodes. In our project, subtour is defined to be a cycle that does not start from the depot and ends at the depot(The cycle in a middle way). One naive algorithm that we come up with is to limit the number of edges to be chosen to be one less than the number of vertices that form a cycle. For each **subset of nodes** with at least two nodes, we limit the maximum number of edges selected:

$$\sum_{i \in S, j \in S, i \neq j} x_{ij} \leq |S| - 1 \quad \forall S \subsetneq V, |S| \geq 2$$

When we have n nodes, we will have to put constraints on every subset of nodes that contain $2n - 1$ nodes, so that we will have $2^n - n - 2$ constraints since there will be 2^n ways to choose a subset, n ways to choose a subset of one node, 2 ways to choose a subset of zero nodes or n nodes. But such a method takes up too much memory for the constraints, which is of exponential order. The method we use is called Miller, Tucker, Zemlin [BG14]. The method, in its simplest form, utilizes the concept of timestamp and triangle inequality. The formulation is given below:

$$u_i - u_j + t_{ij}x_{ij} \leq T(1 - x_{ij})$$

Here u_i represents the total time spent in traveling from the depot to the node i . When we select the edge between node i and node j , we have $u_j \geq u_i + t_{ij}x_{ij} - T(1 - x_{ij})$, meaning that we arrive node i first and then node j . Suppose now there is a cycle formed by three vertices i, j, k and three edges $x_{ij} = 1, x_{jk} = 1, x_{ki} = 1$. By the constraint, we have:

$$u_j > u_i \tag{1}$$

$$u_k > u_j \tag{2}$$

$$u_i > u_k \tag{3}$$

From constraint (1) and (3) we have $u_j > u_k$, which contradicts the constraint (2), so such cycle can never be formed. The T here acts like an upper bound for $u_i, \forall i = 1, 2, \dots, n$, saying that the total time spent from depot to node i should be no greater than the time limit. For example, for arbitrary node i , when $x_{0i} = 1$, then we have $u_i + t_{0i}x_{0i} \leq T$ and thus $u_i < T$, which is an upper bound. The advantage of using this method is that it can limit the number of constraints to linear order(only n constraints in total).

2.3 Objective Function

The objective function is the total weight which means "maximizing the weights of the visited nodes" for the visitor.

$$\max \sum_{(i,j) \in \mathcal{E}} \frac{(w_i + w_j) * x_{ij}}{2} \tag{4}$$

In the multi-floor case, the objective function should look similar to the single-floor case where

Since it is almost impossible for visitors to go to the same floor several times, we assume that visitors take the campus tour floor by floor. Then the mathematical model becomes a combination of n inter-dependent problems (n is the number of floors of the campus). Each problem is an optimization of the route of a floor as shown in (4), and in order to maximize the total weights, those problems are interconnected by a time factor (spent in elevators) since the general limitation is the total amount of time.

For simplicity, we assume that the time that is spent between ending path-finding on floor i and starting path-finding on floor j is a constant c . (By taking a specific elevator).

3 Dataset

3.1 Description

3.1.1 Single Floor

The data we utilize in our project involves both ground truths and inferred information. Ground truths include the distance information between different sections of the floor plan. Inferred information includes the importance of the place to be visited, which should be more robust by doing surveys.

We first convert the floor plan to their topological structures:



Figure 5: 1F Floor Plan



Figure 6: 3F Floor Plan

For simplicity consideration, We construct graphs of 1F and 3F of the campus that look like the following:

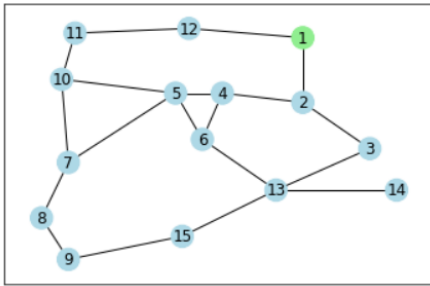


Figure 7: 1F Graph

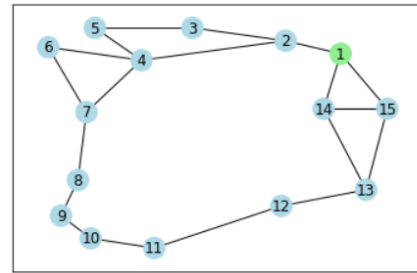


Figure 8: 3F Graph

We focus on the multi-layered routing problem so due to the complexity of the graph model, we don't include large number of nodes for each floor. Then, we construct the total time it takes to complete the tour as the summation of the walking time between any two consecutive sites(represented in edges). The walking time is measured by the walking distance between each site divided by the walking speed. The walking distance is measured directly by optical sensing tools and a Photoshop ruler and the walking speed is assumed to be 0.25 meters per second.

On top of that, we assign the importance (represented in the node weight, a constant) to each site. The weights of each site are constant, which can be obtained from our personal experience and the area of a site.

3.1.2 Multi-Floor

For multi-floor graph construction, we follow the algorithm articulated in Algorithm 4 below, and we use the depot as the common node between the graphs of different floors. Fig 9 shows the construction result

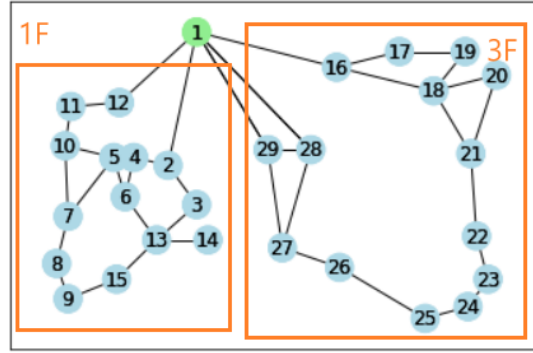


Figure 9: Concatenated Graph

Here, **node 1** is the depot node that connects different floors. On the left is the topology for the first floor and on the right it is the third floor. The corresponding concatenated adjacency matrix takes the form of

$$\begin{pmatrix} w_1 & \dots & w_{29} \end{pmatrix}$$

$\mathbf{A}_{concat} =$

$$\begin{pmatrix} a_{1,1} & \dots & a_{1,15} & a_{1,16} & \dots & a_{1,29} \\ \vdots & \ddots & \vdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ a_{15,1} & \dots & a_{15,15} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \hline a_{16,1} & \mathbf{0} & \mathbf{0} & a_{16,16} & \dots & a_{16,29} \\ \vdots & \mathbf{0} & \mathbf{0} & \vdots & \ddots & \vdots \\ a_{29,1} & \mathbf{0} & \mathbf{0} & a_{29,16} & \dots & a_{29,29} \end{pmatrix}$$

The corresponding concatenated cost matrix takes the form of $\mathbf{T}_{concat} =$

$$\begin{pmatrix} c_{1,1} & \dots & c_{1,15} & c_{1,16} & \dots & c_{1,29} \\ \vdots & \ddots & \vdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ c_{15,1} & \dots & c_{15,15} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \hline c_{16,1} & \mathbf{0} & \mathbf{0} & c_{16,16} & \dots & c_{16,29} \\ \vdots & \mathbf{0} & \mathbf{0} & \vdots & \ddots & \vdots \\ c_{29,1} & \mathbf{0} & \mathbf{0} & c_{29,16} & \dots & c_{29,29} \end{pmatrix}$$

Here, notice that, since the **node 1** is assumed to be the connecting vertex between different floors, thus the first row and the first column should be the vectors that show the nodes that are connected to the depot. Thus they should not be as simple as all zero.

3.2 Data Files

We collect the required information to construct a graph from floor plan. The dataset looks like the following:

1. 1st Floor:

	edge_from	edge_to	weight
1	1	12	19.476
2	1	2	11.329
3	2	3	19.064
4	3	13	13.556
5	2	4	14.192
6	4	6	8.721
7	4	5	6.416
8	5	6	9.17
9	6	13	12.241
10	5	10	14.992
11	10	11	8.806
12	11	12	13.645
13	10	7	15.57
14	5	7	18.763
15	7	8	9.736
16	8	9	9.17
17	9	15	14.928
18	15	13	13.129
19	13	14	15.269

	node	weight
1	1	1
2	2	5
3	3	2
4	4	1
5	5	2
6	6	1
7	7	1
8	8	4
9	9	2
10	10	1
11	11	3
12	12	3
13	13	1
14	14	2
15	15	2

	node	coordinate_x	coordinate_y
1	1	5	6
2	2	5	4.6
3	3	6	3.6
4	4	3.8	4.8
5	5	3.1	4.8
6	6	3.5	3.8
7	7	1.5	3.3
8	8	1.1	2.1
9	9	1.5	1.2
10	10	1.4	5.1
11	11	1.6	6.1
12	12	3.3	6.2
13	13	4.6	2.7
14	14	6.4	2.7
15	15	3.2	1.7

Figure 10: Edge Weights

Figure 11: Node Weights

Figure 12: Node Coordinates

2. 3st Floor

	edge_from	edge_to	weight
1	1	2	8.559
2	2	4	22.26
3	2	3	14.505
4	3	5	15.000
5	4	5	7.952
6	4	6	13.893
7	4	7	13.917
8	6	7	14.232
9	7	8	13.581
10	8	9	7.772
11	9	10	7.136
12	10	11	10.154
13	11	12	20.943
14	12	13	13.529
15	13	14	16.614
16	13	15	16.124
17	14	15	9.51
18	14	1	11.166
19	15	1	13.126

	node	weight
1	1	1
2	2	4
3	3	1
4	4	1
5	5	2
6	6	2
7	7	2
8	8	1
9	9	4
10	10	1
11	11	2
12	12	1
13	13	1
14	14	3
15	15	2

	node	coordinate_x	coordinate_y
1	1	7.4	6.7
2	2	6.1	7.1
3	3	3.9	7.5
4	4	2.7	6.5
5	5	1.6	7.5
6	6	0.5	6.9
7	7	1.4	4.9
8	8	1.2	2.8
9	9	0.8	1.7
10	10	1.5	1
11	11	3	0.7
12	12	6	2
13	13	8	2.5
14	14	7	5
15	15	8.5	5

Figure 13: Edge Weights

Figure 14: Node Weights

Figure 15: Node Coordinates

For each floor, we have edge weights, node weights and node coordinates.csv file for later construction of graph object that is described later.

3.3 Dataset Preprocessing

3.3.1 Single Floor

We use the python [networkx](#) package to construct and visualize the graph structure. We first assign each node a xy coordinate system to better reflect the relative locations of each node and avoid the default behavior of [networkx](#) when drawing the graph structure and the tool that we use is the Photoshop grid view. We then construct an adjacency matrix and a cost matrix from the edge weights file that follows from the algorithm:

Algorithm 1 constructMatrix(G)

Require: A graph object G containing Edge Information \mathcal{E} and Num of Nodes $|V|$ Create an zero adjacency matrix \mathbf{A} of shape $|V| \times |V|$ Create an zero cost matrix \mathbf{T} of shape $|V| \times |V|$

for e in \mathcal{E} **do**

$\mathbf{A}[\text{edgeFrom}(e)][\text{edgeTo}(e)] = 1$

$\mathbf{A}[\text{edgeTo}(e)][\text{edgeFrom}(e)] = 1$

$\mathbf{T}[\text{edgeFrom}(e)][\text{edgeTo}(e)] = \text{edgeWeight}(e)$

$\mathbf{T}[\text{edgeTo}(e)][\text{edgeFrom}(e)] = \text{edgeWeight}(e)$

end for

return \mathbf{A} or \mathbf{T} , depending on the request.

For graph construction, we have the following algorithm:

Algorithm 2 constructGraph(\mathcal{E}, \mathcal{N})

Require: Edge Information \mathcal{E} , Node Information \mathcal{N} Create an empty graph object G

for n in \mathcal{N} **do**

$G.\text{addNode}(n, \text{nodeWeight}(n), \text{pos}=(\text{xCoordinate}(n), \text{yCoordinate}(n)))$

end for

for e in \mathcal{E} **do**

$G.\text{addEdge}(e, \text{edgeWeight}(e))$

end for

return G

These algorithms serve as the building blocks for the more complicated multi-floor graph construction, which will be detailed below.

3.3.2 Multi Floor

For the multi-floor routing, the first thing we did is to construct a concatenated graph from the individual graphs that represent the individual floor plan. The algorithm breaks apart into two components: concatenating cost/adjacency matrix and concatenating graph structure.

Algorithm 3 concatenateGraph(G_1, G_2)

Require: First Graph G_1 , Second Graph G_2 , option Create an empty graph object NG , used to store the new information for the newly constructed concatenated graph.

```
for node in  $G_1$  do
    if it is the first node in the graph then
        Skip the current loop
    else
        Add the node into the new graph  $NG$  with all the node information
    end if
end for
for node in  $G_2$  do
    if it is the first node in the graph then
        Skip the current loop
    else
        Add the node into the new graph  $NG$  with all the node information, where the index for the
        nodes in graph 2 are all incremented by  $|V_1| - 1$  where  $v_1$  stands for the number of nodes in  $G_1$ 
    end if
end for
return  $NG$ 
```

Algorithm 4 concatenateMatrix($\{G\}$)

Require: A list of graph object

```
 $A \leftarrow$  zero matrix of  $1 \times 1$ 
 $B \leftarrow$  zero matrix of  $1 \times 1$ 
 $C \leftarrow$  zero matrix of  $1 \times 1$ 
 $D \leftarrow$  zero matrix of  $1 \times 1$ 
for  $G$  in  $\{G\}$  do
     $M \leftarrow$  constructMatrix( $G$ )
     $B \leftarrow$  hstack( $B, M[0, 1 :]$ )
     $C \leftarrow$  vstack( $C, M[1 :, 0]$ )
     $D \leftarrow$  hstack( $D, M[1 :, 1 :]$ )
end for
 $B \leftarrow B[0, 1 :]$ 
 $C \leftarrow C[1 :, 0]$ 
 $D \leftarrow D[1 :, 1 :]$ 
 $Res \leftarrow$  vstack([hstack([ $A, B$ ]), hstack( $C, D$ )])
return  $Res$ 
```

4 Results and Discussion

4.1 Single Floor Routing

1. 1st Floor:

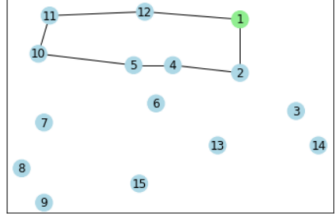
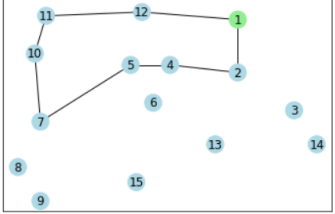
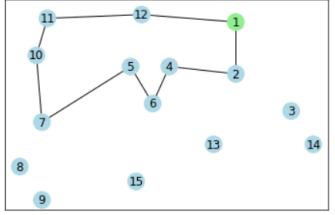
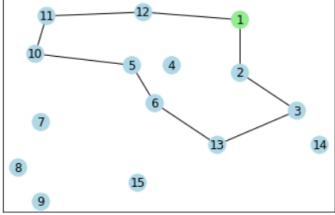
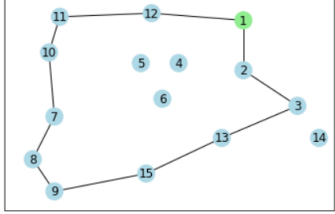
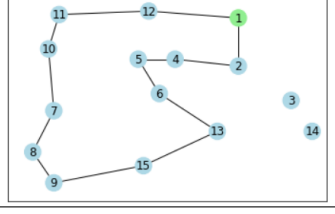
Time Limit(T)	Objective Value	Selected Path
90	16	 A graph with 15 nodes (1-15) and 15 edges. Node 1 is green. The selected path is 1-12-11-10-5-4-2. Nodes 3, 6, 7, 8, 9, 13, 14, and 15 are not connected to the path.
110	17	 A graph with 15 nodes (1-15) and 15 edges. Node 1 is green. The selected path is 1-12-11-10-7-5-4-2. Nodes 3, 6, 8, 9, 13, 14, and 15 are not connected to the path.
120	18	 A graph with 15 nodes (1-15) and 15 edges. Node 1 is green. The selected path is 1-12-11-10-7-5-6-4-2. Nodes 3, 8, 9, 13, 14, and 15 are not connected to the path.
125	19	 A graph with 15 nodes (1-15) and 15 edges. Node 1 is green. The selected path is 1-12-11-10-5-4-2-3-13-6. Nodes 7, 8, 9, and 15 are not connected to the path.
150	25	 A graph with 15 nodes (1-15) and 15 edges. Node 1 is green. The selected path is 1-12-11-10-7-8-9-15-13-2-3-14. Nodes 4, 5, and 6 are not connected to the path.
160	27	 A graph with 15 nodes (1-15) and 15 edges. Node 1 is green. The selected path is 1-12-11-10-7-8-9-15-13-6-2. Nodes 3, 4, and 14 are not connected to the path.

Table 1: Results for 1st Floor

2. 3rd Floor:

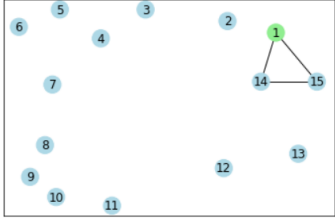
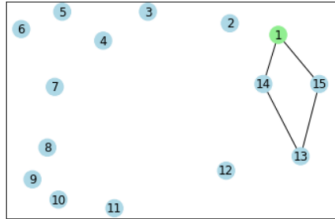
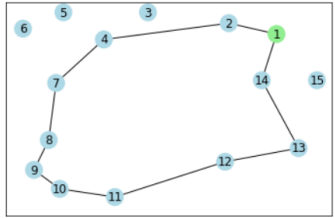
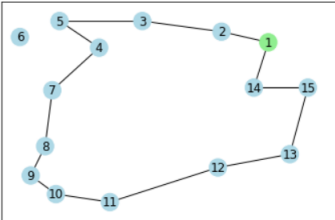
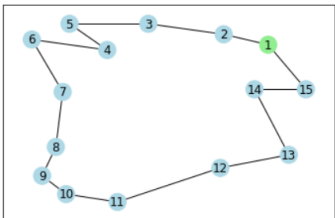
Time Limit(T)	Objective Value	Selected Path
50	5	
75	6	
150	24	
170	30	
200	31	

Table 2: Results for 3rd Floor

The code that generates these results is in appendix [C.3](#).

We find that the bigger the time limit, the bigger the total node weights to be maximized. When time limit reaches 200, all the nodes in the graph can be covered by the selected path.

4.2 Multi-floor routing

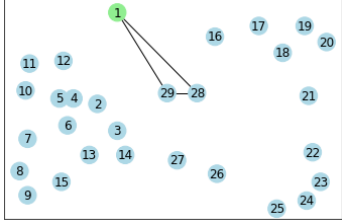
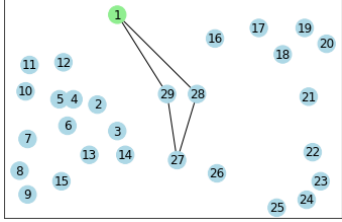
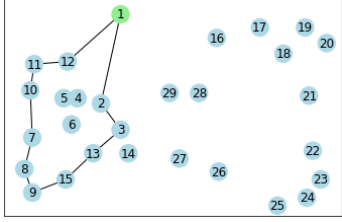
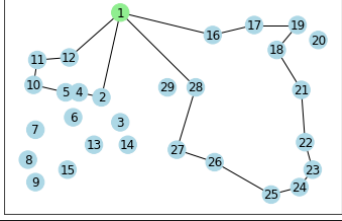
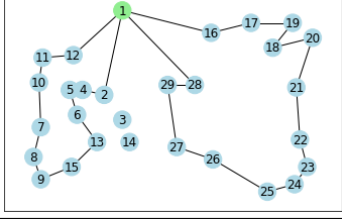
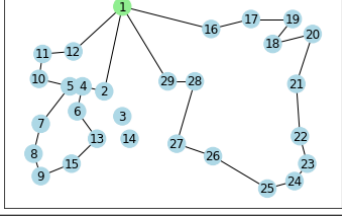
Time Limit(T)	Objective Value	Selected Path
50	6	
75	7	
150	25	
250	40	
350	55	
600	55	

Table 3: Results for Multi-floor Planning

The code that generates these results is in appendix [C.4](#).

In the graph of selected paths given multi-floor situation, we regard the starting point of both 1F and 3F as **node 1** , which is always highlighted in green color. Besides, nodes 2 - 15 on the left side of the graph represent the corresponding nodes on the 1st floor, and nodes 16 - 29 on the right side of the graph represent the corresponding nodes on the 3rd floor.

As you can see in the table, the trend is generally similar to that in the single floor case. We find that the bigger the time limit, the bigger the total node weights to be maximized, until the time limit reaches 350. After the time limit reaches 350, if you further increase the total time limit, the objective value will not increase a lot, and almost all the sites can be covered in the graph of selected paths.

5 Conclusion and Future Works

In this project, we develop an optimization model to solve a practical problem, the campus tour routing problem. We finally get the routing plans which try to maximize the tourists' campus tour experience given different time limits, and in the scenario of single floor and multiple floor, respectively.

As for the future improvement of the optimization model, we can improve the accuracy of the parameters in the model. As you can see, the node importance/weight is formulated based on our own subjective viewpoints. We can improve its accuracy by distributing some survey among students and parents to get a public viewpoint of the importance/weight of each site, which can make the parameter node importance much more generalized and appropriate so that we can formulate a better optimization model.

Appendix A CVRP

$$\min \sum_{i=0}^{n+1} \sum_{j=0}^{n+1} c_{ij} x_{ij} \quad (2.1)$$

$$\text{s.t.} \quad \sum_{\substack{j=1 \\ j \neq i}}^{n+1} x_{ij} = 1, \quad i = 1, \dots, n, \quad (2.2)$$

$$\sum_{\substack{i=0 \\ i \neq h}}^n x_{ih} - \sum_{\substack{j=1 \\ j \neq h}}^{n+1} x_{hj} = 0, \quad h = 1, \dots, n, \quad (2.3)$$

$$\sum_{j=1}^n x_{0j} \leq K, \quad (2.4)$$

$$y_j \geq y_i + q_j x_{ij} - Q(1 - x_{ij}), \quad i, j = 0, \dots, n+1, \quad (2.5)$$

$$d_i \leq y_i \leq Q, \quad i = 0, \dots, n+1, \quad (2.6)$$

$$x_{ij} \in \{0, 1\}, \quad i, j = 0, \dots, n+1. \quad (2.7)$$

Figure 16: CVPR formulation

Appendix B Parameter Value Table

B.1 1st floor

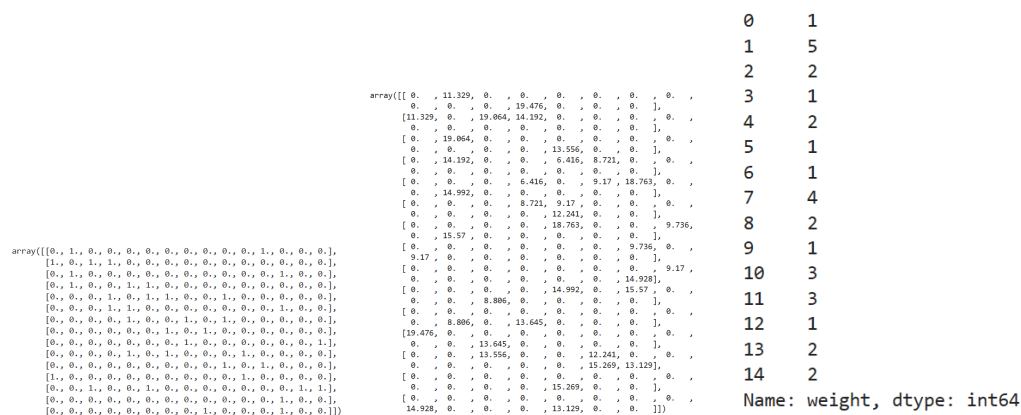


Figure 17: Edge Weights

Figure 18: Node Weights

Figure 19: Node Coordinates

B.2 3rd floor

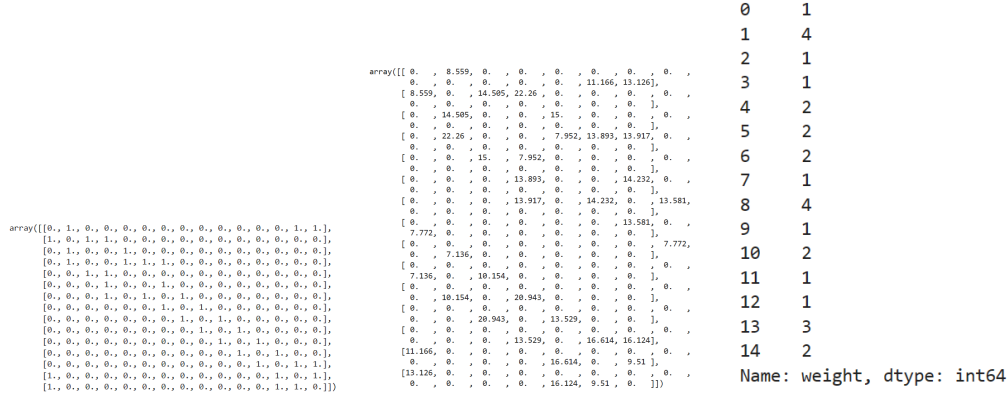


Figure 20: Edge Weights

Figure 21: Node Weights

Figure 22: Node Coordinates

Appendix C Algorithm Implementation Codes

Complete codes can be found at the following github repository with dataset: [Complete Codes with Dataset](#)

C.1 Environment

```
1 import numpy as np
2 import pulp
3 import itertools
4 # import seaborn as sns
5 import matplotlib.pyplot as plt
6 import pandas as pd
7 import networkx as nx
```

C.2 Dataset Construction

```
1 graph_3f_edges = pd.read_csv(
2     './3F/DATA-SHU_240_Data_Representation_--Sheet1.csv')
3 graph_3f_weights = pd.read_csv(
4     './3F/DATA-SHU_240_Data_Representation_--Sheet2.csv')
5 graph_3f_node_grid = pd.read_csv(
6     './3F/DATA-SHU_240_Data_Representation_--Sheet3.csv')
7
```

```

8 graph_1f_edges = pd.read_csv(
9     "./1F/DATA-SHU_240_Data_Representation_--Sheet4.csv")
10 graph_1f_weights = pd.read_csv(
11     "./1F/DATA-SHU_240_Data_Representation_--Sheet5.csv")
12 graph_1f_node_grid = pd.read_csv(
13     "./1F/DATA-SHU_240_Data_Representation_--Sheet6.csv")
14
15 G1 = nx.Graph()
16 G3 = nx.Graph()
17 for i in graph_1f_weights.index:
18     G1.add_node(graph_1f_weights['node'][i],
19                 weight=graph_1f_weights['weight'][i],
20                 pos=(graph_1f_node_grid['coordinate_x'][i]
21                     ,graph_1f_node_grid['coordinate_y'][i]))
22 for i in graph_3f_weights.index:
23     G3.add_node(graph_3f_weights['node'][i],
24                 weight=graph_3f_weights['weight'][i],
25                 pos=(graph_3f_node_grid['coordinate_x'][i]
26                     ,graph_3f_node_grid['coordinate_y'][i]))
27
28 for i in graph_1f_edges.index:
29     G1.add_edge(graph_1f_edges['edge_from'][i]
30                 graph_1f_edges['edge_to'][i]
31                 , weight=graph_1f_edges['weight'][i])
32
33 for i in graph_3f_edges.index:
34     G3.add_edge(graph_3f_edges['edge_from'][i],
35                 graph_3f_edges['edge_to'][i]
36                 , weight=graph_3f_edges['weight'][i])

```

C.3 Single Floor

```

1 def floor_routing_solver(G, time_limit):
2
3     num_of_sites = compute_num_of_sites(G)
4

```

```

5 cost_matrix= generate_cost_matrix(G)
6
7 adjacency_matrix = generate_adjacency_matrix(G)
8
9 problem = pulp.LpProblem("CVRP", pulp.LpMaximize)
10
11 node_weights = list(nx.get_node_attributes(G1,"weight").values())
12
13 # definition of binary variables
14 x = [[pulp.LpVariable("x%s_%s"%(i,j), cat="Binary")
15         if i != j else None
16         for j in range(num_of_sites)]
17        for i in range(num_of_sites)]
18 y = np.array(x)
19 t = pulp.LpVariable.dicts("t"
20         , (i for i in range(num_of_sites))
21         , lowBound=0
22         , upBound=time_limit
23         , cat='Continuous')
24
25 # add objective function
26 problem += pulp.lpSum((node_weights[i]
27 + node_weights[j]) * y[i][j] / 2 if i != j else 0
28         for j in range(num_of_sites)
29         for i in range(num_of_sites))
30
31 # constraints
32 # Constraint 1: Visit less than Once
33 for j in range(num_of_sites):
34     problem += pulp.lpSum(y[i][j] if i != j else 0
35         for i in range(num_of_sites)) <= 1
36
37 # Constraint 2: Must Depart from the Depot
38 problem += pulp.lpSum(y[0][j]
39         for j in range(1,num_of_sites)) == 1
40 # problem += pulp.lpSum(y[i][0]
41         for i in range(1,num_of_sites)) == 1

```

```

42
43
44 # Constraint 3: Degree Match
45 for j in range(1, num_of_sites):
46     problem += pulp.lpSum(y[i][j] if i != j else 0
47         for i in range(num_of_sites)) \
48     - pulp.lpSum(x[j][i] if i != j else 0
49         for i in range(num_of_sites)) \
50     == 0
51
52 # Constraint 4: No Shuttle Run
53 for i in range(num_of_sites):
54     for j in range(num_of_sites):
55         if i < j:
56             problem += (y[i][j] + y[j][i]) <= 1
57
58 # Constraint 5: time constraint
59 problem += pulp.lpSum(cost_matrix[i][j] * x[i][j]
60     if i != j else 0
61         for i in range(num_of_sites)
62         for j in range (num_of_sites)) <= time_limit
63
64
65 # Constraint 6: pass through edges that exist
66 for i in range(num_of_sites):
67     for j in range(num_of_sites):
68         if i != j:
69             problem += (y[i][j] <= int(adjacency_matrix[i][j]))
70
71
72 # Constraint 7: remove subtours
73 for i in range(1, num_of_sites):
74     for j in range(1, num_of_sites):
75         if i != j and (i != 0 and j != 0):
76             problem += t[j] >= t[i] + cost_matrix[i][j] * y[i][j]
77     - time_limit * (1 - y[i][j])
78

```

```

79
80     if problem.solve() == 1:
81         print('Maximum weights: ', pulp.value(problem.objective))
82
83     varValues = np.zeros((num_of_sites, num_of_sites))
84     for i in range(len(y)):
85         for j in range(len(y[i])):
86             if y[i][j] != None:
87                 varValues[i][j] = y[i][j].varValue
88
89     return varValues

```

C.4 Multiple Floor

```

1         class MultiFloorSolver:
2
3             # Assume to be two graphs, sharing one depot node
4             def __init__(self, G_list, time_limit):
5                 self.G_list = G_list
6                 self.time_limit = time_limit
7                 self.compute_concated_num_of_sites()
8                 self.construct_concated_node_weights()
9                 self.construct_concated_adjacency_matrix()
10                self.construct_concated_cost_matrix()
11
12            def compute_concated_num_of_sites(self):
13                self.concated_num_of_sites = 0
14                for G in self.G_list:
15                    self.concated_num_of_sites += self.compute_num_of_sites(G)
16                self.concated_num_of_sites -= (len(self.G_list) - 1)
17
18            def construct_concated_node_weights(self):
19                temp = np.array([[1]])
20                for G in self.G_list:
21                    temp = np.hstack([temp, self.compute_node_weights(G)])
22                self.node_weights = temp.squeeze()

```

```

23
24 def construct_concated_adjacency_matrix(self):
25     A = np.array([[0]])
26     B = np.array([[0]])
27     C = np.array([[0]])
28     D = np.array([[0]])
29     for G in self.G_list:
30         adj_temp = self.construct_adjacency_matrix(G)
31         B = np.hstack([B, adj_temp[0, 1:].reshape(1, -1)])
32         C = np.vstack([C, adj_temp[1:, 0].reshape(-1, 1)])
33         D = block_diag(D, adj_temp[1:, 1:])
34     B = B[0, 1:].reshape(1, -1)
35     C = C[1:, 0].reshape(-1, 1)
36     D = D[1:, 1:]
37
38     adj_concated = np.vstack([np.hstack([A, B]), np.hstack([C, D])])
39     self.adj_matrix = adj_concated
40
41 def construct_concated_cost_matrix(self):
42     A = np.array([[0]])
43     B = np.array([[0]])
44     C = np.array([[0]])
45     D = np.array([[0]])
46     for G in self.G_list:
47         adj_temp = self.construct_cost_matrix(G)
48         B = np.hstack([B, adj_temp[0, 1:].reshape(1, -1)])
49         C = np.vstack([C, adj_temp[1:, 0].reshape(-1, 1)])
50         D = block_diag(D, adj_temp[1:, 1:])
51     B = B[0, 1:].reshape(1, -1)
52     C = C[1:, 0].reshape(-1, 1)
53     D = D[1:, 1:]
54
55     adj_concated = np.vstack([np.hstack([A, B]), np.hstack([C, D])])
56     self.cost_matrix = adj_concated
57
58 def compute_num_of_sites(self, G):
59     return len(G.nodes())

```



```

60
61 def compute_node_weights(self, G):
62     return np.array(list(nx.get_node_attributes(G, "weight")
63                        .values())[1:]).reshape(1, -1)
64
65 def construct_adjacency_matrix(self, G):
66     num_of_sites = compute_num_of_sites(G)
67     adjacency_matrix = np.zeros((num_of_sites, num_of_sites))
68     for (edge_from, edge_to), weight in
69     nx.get_edge_attributes(G, "weight").items():
70         adjacency_matrix[edge_from - 1][edge_to - 1] = 1
71         adjacency_matrix[edge_to - 1][edge_from - 1] = 1
72     return adjacency_matrix
73
74 def construct_cost_matrix(self, G):
75     num_of_sites = compute_num_of_sites(G)
76     cost_matrix = np.zeros((num_of_sites, num_of_sites))
77     for (edge_from, edge_to), weight
78     in nx.get_edge_attributes(G, "weight").items():
79         cost_matrix[edge_from - 1][edge_to - 1] = weight
80         cost_matrix[edge_to - 1][edge_from - 1] = weight
81     return cost_matrix
82
83 # Two Graph Case
84 def construct_concatated_graph(self, option):
85     NG = nx.Graph()
86     if len(self.G_list) <= 2:
87         G1 = self.G_list[0] # 1F Floor
88         G2 = self.G_list[1] # 3F Floor
89
90     # Node Information
91     pos_dict_1 = nx.get_node_attributes(G1, "pos")
92     pos_dict_2 = nx.get_node_attributes(G2, "pos")
93     weights_dict_1 = nx.get_node_attributes(G1, "weight")
94     weights_dict_2 = nx.get_node_attributes(G2, "weight")
95
96     # Edge Information

```

```

97     edge_weights_dict_1 = nx.get_edge_attributes(G1
98     , "weight")
99     edge_weights_dict_2 = nx.get_edge_attributes(G2
100     , "weight")
101
102     flag = True
103     # Constructing nodes
104     for node in G1:
105         if flag:
106             NG.add_node(node, weight=edge_weights_dict_1[node]
107             , pos=pos_dict_1[node])
108             flag = False
109         else:
110             # Add nodes from graph 1
111             NG.add_node(node, weight=edge_weights_dict_1[node]
112             , pos=(pos_dict_1[node][0] - 1
113             , pos_dict_1[node][1] - 2))
114     flag = True
115     for node in G2:
116         if flag:
117             flag = False
118         else:
119             # Add nodes from graph 2, with coordinate_x all drifted(except
120             NG.add_node(node + len(G1.nodes()) - 1
121             , weight=edge_weights_dict_1[node]
122             , pos=(2 * pos_dict_1[1][0]
123             - pos_dict_2[node][0] + 6
124             , pos_dict_2[node][1] - 2))
125
126
127     if option == "full":
128         # Constructing Edges
129         for (edge_from, edge_to), weight in
130         nx.get_edge_attributes(G1, "weight").items():
131             NG.add_edge(edge_from
132             , edge_to
133             , weight=weight)

```

```

134
135
136     for (edge_from, edge_to), weight in
137         nx.get_edge_attributes(G2, "weight").items():
138         if edge_from == 1:
139             NG.add_edge(edge_from, edge_to
140                         + len(G1.nodes)
141                         - 1, weight=weight)
142         elif edge_to == 1:
143             NG.add_edge(edge_from
144                         + len(G1.nodes)
145                         - 1, edge_to, weight=weight)
146         else:
147             NG.add_edge(edge_from
148                         + len(G1.nodes) - 1, edge_to
149                         + len(G1.nodes) - 1, weight=weight)
150
151     return NG
152
153     else:
154         # Future Work
155         pass
156
157 # Two graph case
158 def visualize_graph(self, G):
159     if len(self.G_list) <= 2:
160         color_map = ["lightgreen"]
161         + ["lightblue" for i in range(self.concated_num_of_sites-1)]
162         nx.draw_networkx(G, with_labels=True
163                         , pos = nx.get_node_attributes(G, 'pos')
164                         , node_color = color_map)
165
166     else:
167         # Future Work
168         pass
169
170 def visualize_routing_results(self, varValues):

```

```

171     NG = self.construct_concated_graph(option="else")
172
173     for i in range(len(varValues)):
174         for j in range(len(varValues)):
175             if varValues[i][j]==1:
176                 if varValues[i][j] == varValues[j][i] == 1:
177                     NG.add_edge(i + 1, j + 1, color="r")
178                 else:
179                     NG.add_edge(i + 1, j + 1, color="black")
180
181     colors = [NG[u][v]['color'] for u,v in NG.edges]
182     color_map = ["lightgreen"]
183     + ["lightblue"]
184     for i in range(self.concated_num_of_sites-1)]
185     nx.draw_networkx(NG, with_labels=True
186                     , pos = nx.get_node_attributes(NG, 'pos')
187                     , edge_color=colors
188                     , node_color = color_map)
189
190 def floor_routing_solver(self):
191     print("Time_Limit", self.time_limit)
192
193     problem = pulp.LpProblem("CVRP", pulp.LpMaximize)
194
195     # definition of binary variables
196     x = [[pulp.LpVariable("x%s_%s"%(i,j)
197         , cat="Binary") if i != j else None
198           for j in range(self.concated_num_of_sites)]
199           for i in range(self.concated_num_of_sites)]
200     t = pulp.LpVariable.dicts("t", (i for i in
201     range(self.concated_num_of_sites))
202         , lowBound=0
203         , upBound=time_limit
204         , cat='Continuous')
205
206     # add objective function
207     problem += pulp.lpSum((self.node_weights[i]

```

```

208         + self.node_weights[j]) * x[i][j] / 2
209         if i != j else 0
210         for j in range(self.concated_num_of_sites)
211         for i in range(self.concated_num_of_sites))
212
213     # constraints
214     # Constraint 1: Visit less than Once
215     for j in range(1, self.concated_num_of_sites):
216         problem += pulp.lpSum(x[i][j] if i != j else 0
217                                for i in range(self.concated_num_of_sites)) <= 1
218
219     # Constraint 2: Must Depart from the Depot
220     problem += pulp.lpSum(x[0][j] for j in range(1, self.concated_num_of_sites))
221
222     # Constraint 3: Degree Match
223     for i in range(self.concated_num_of_sites):
224         for j in range(self.concated_num_of_sites):
225             if i < j:
226                 problem += (x[i][j] + x[j][i]) <= 1
227
228     # Constraint 4: No Shuttle Run
229     for j in range(1, self.concated_num_of_sites):
230         problem += pulp.lpSum(x[i][j] if i != j else 0
231                                for i in range(self.concated_num_of_sites)) \
232         - pulp.lpSum(x[j][i] if i != j else 0
233                     for i in range(self.concated_num_of_sites)) \
234         == 0
235
236     # Constraint 5: time constraint
237     problem += pulp.lpSum(self.cost_matrix[i][j]
238                            * x[i][j] if i != j else 0
239                            for i in range(self.concated_num_of_sites)
240                            for j in range(self.concated_num_of_sites)) <= self.time_limit
241
242
243     # Constraint 6: pass through edges that exist
244     for i in range(self.concated_num_of_sites):

```

```

245         for j in range(self.concated_num_of_sites):
246             if i != j:
247                 problem += (x[i][j] <= int(self.adj_matrix[i][j]))
248
249
250     # Constraint 7: remove subtours
251     for i in range(1, self.concated_num_of_sites):
252         for j in range(1, self.concated_num_of_sites):
253             if i != j and (i != 0 and j != 0):
254                 problem += t[j] >= t[i]
255                 + self.cost_matrix[i][j] * x[i][j]
256                 - self.time_limit * (1 - x[i][j])
257
258
259     if problem.solve() == 1:
260         print('Maximum_weights: ', pulp.value(problem.objective))
261
262     varValues = np.zeros((self.concated_num_of_sites,
263 self.concated_num_of_sites))
264     for i in range(len(x)):
265         for j in range(len(x[i])):
266             if x[i][j] != None:
267                 varValues[i][j] = x[i][j].varValue
268
269     self.varValues = varValues
270     return varValues
271
272 T = 1000
273 solverObject = MultiFloorSolver([G1,G3], T)
274 varValues = solverObject.floor_routing_solver()
275 solverObject.visualize_routing_results(varValues)

```

References

- [BG14] Tolga Bektaş and Luis Gouveia. Requiem for the miller–tucker–zemlin subtour elimination constraints? *European Journal of Operational Research*, 236(3):820–832, 2014. Vehicle

- [BNK09] Thomas Becker, Claus Nagel, and Thomas H. Kolbe. A Multilayered Space-Event Model for Navigation in Indoor Spaces. In Jiyeong Lee and Sisi Zlatanova, editors, *3D Geo-Information Sciences*, pages 61–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISSN: 1863-2246, 1863-2351 Series Title: Lecture Notes in Geoinformation and Cartography.
- [CE14] G.C. Calafiore and L. El Ghaoui. *Optimization Models*. Control systems and optimization series. Cambridge University Press, October 2014.
- [CLY22] R Cheng, X Lu, and X Yu. A Mathematical Model for the Routing Optimization Problem with Time Window. *Journal of Physics: Conference Series*, 2219(1):012038, April 2022.
- [LZ11] Liu Liu and Sisi Zlatanova. A "door-to-door" path-finding approach for indoor navigation. 05 2011.
- [MDS17] Pedro Munari, Twan Dollevoet, and Remy Spliet. A generalized formulation for vehicle routing problems, 2017.
- [YW15] Liping Yang and Michael Worboys. Generation of navigation graphs for indoor space. *International Journal of Geographical Information Science*, 29(10):1737–1756, 2015.