Solutions for HW3

Yunhai Han

January 25, 2020

1 E2.3 The several bridges of a city

1.1 i

From the hint, I know that for this part, I am allowed to connect two nodes with multiple edges. Hence, there are four vertices and seven edges.

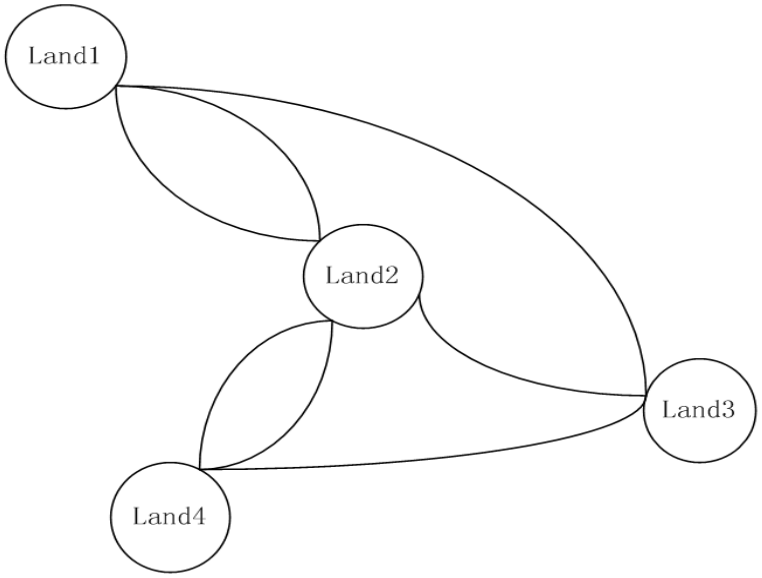

Figure 1: Illustration of lands and bridges

From the above figure, the four different circles represent the four masses of land and the seven edges represent the seven bridges.

1.2 ii

If there exist such a path that walk through the vertex and crosses each edge exactly once, the path could be named **Euler Path**. If we want to examine whether there exist an euler path or not, I need to introduce the concept of degree of vertex. The degree of vertex could be defined as the number of edges which has that vertex as an endpoint.

In this case, we can never find the desired path, the reason is very simple:there are more than two vertices the degrees of which are odd numbers. I could give a quick explanation:

Assume we have more than two such vertices, each time we arrive and leave one of these vertices, two edges emanating from the vertex are used. If the degree of a vertex is an odd number and we want to cross each edge exactly once, the only possible case is when we start from this vertex and end at the other one. Because, if we leave the vertex and never come back(start point), the pair I describe above could be broken and one extra edge is not necessary, which means the degree of the start point could be an odd number. Also, if we end at such a vertex, we will never leave it again and leave one more edge. That is why more then two vertices with degree an odd number could infer that there is no Euler path in the graph.

From the figure **??**, it is obvious that all the four vertices have the odd numbers of degree. Hence, it is impossible to find the desired path.

1.3 iii

If we are allowed to add one more edge into this graph connecting any two points we want, it is possible to find such a path if the single addition edge is located between the Land1 and Land4. Because there are only two vertices which have odd numbers of degree. I draw the modified graph and the Euler path below:

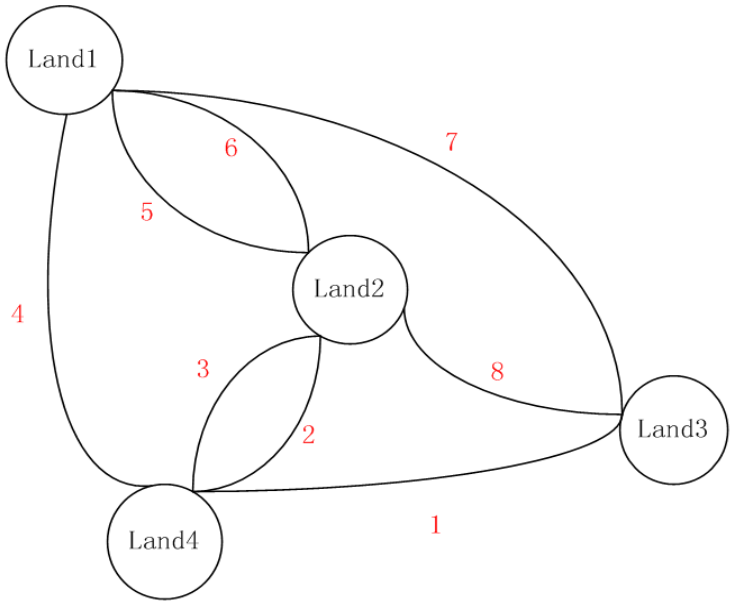

Figure 2: Modified graph and the desired path

I use a sequence of continuous numbers to mark the eight different edges, indicating the order of the desired path. If we follow this order, we could walk through the city and cross each edge exactly once.

2 E2.7 The BFS algorithm for disconnected graphs

From the lecture, the definition of the graph connectivity is: A graph is connected if every two nodes are path-connected, which means we could always find at least one possible path between any two nodes.

2.1 i

f we could find a method to verify that every two nodes are connected, we could easily say that the graph is connected. From the lecture, we could use BFS algorithm to find the shortest path between any two nodes, and this algorithm would return false if there dose't exist such a path. Hence, we could implement this algorithm multiple times to find all the possible test path between any two nodes. Suppose there is m nodes in the graph($m \geq 1$), the stupidest way is to implement it for $\sum_i^m$ times. In this case, we would verity whether the first node in the nodes list connected to any other nodes in the left $m-1$ nodes. Then, we could simply start from the next node the list and verify

the connectivity between this node and any other nodes behind it in the list. Finally, we would implement is for $\sum_i^m$ times and fulfill the goal.

Though it works, it would waste lots of time because most of the verifications are not necessary. So, I could change the strategy and find a much more efficient method. Here, I give a brief description of the new method:

- Create an identity matrix and the size of it is $m * m$.
- We could randomly select a node in the node list, and find all the nodes that connect the selected node. Let say we find k nodes for this time.
- Suppose the index of the original node is i in the node list, and the indexes of all the found nodes are $i^1, i^2...i^k$. We then set these entries in the matrix $(i, i^1)and(i^1, i)$, $(i, i^2)and(i^2, i)...(i, i^k)and(i^k, i)$ to 1.
- for each node in $i^1, i^2...i^k$, we find all the nodes that connect to them respectively. I take the node i^1 as an example. Obviously, its parent node is i and I assume it connect to m nodes including its parent node.
- for each node, I check whether the entry in the Identity matrix at $(i^1, (i^1)^1)and((i^1)^1, i^1)$, $(i^1, (i^1)^2)and((i^1)^1, i^1), ...(i^1, (i^1)^m)and((i^1)^m, i^1)$ is zero or not respectively. If the entry is zero, I would set it as 1 and if it's not, I would unchange it. In this case, there are $2 * (m - 1)$ entries which are zero, so I set them to one. Also, you can see that the entries at $(i^1, i)and(i, i^1)$ has already been set to 1 in the previous procedure, so I don't have to change them. After that, since I know its parent node and its child nodes, I also could set the entries at $(i, (i^1)^1)and((i^1)^1, i)$, $(i, (i^1)^2)and((i^1)^2, i), ...(i, (i^1)^m)and((i^1)^m, i)$ to 1. Of course, if the entries has already been 1, I don't have to change them.
- I would repeat this k times for all the k nodes I found in the previous iteration.
- Obviously, I know that right now I have already found the child nodes of the node $i^1, i^2, ..., i^k$, respectively. So in the next iteration, I would not process these nodes even though some of them would be the child node of the others. It is really useful to help us save lots of time. For the left child nodes, they just appear for the first time, I would find the child nodes of them in the next iteration and do the above procedure again.
- When we should stop doing this is very straightforward:if during an iteration, there is no entry being set to 1, we could stop this.
- We check whether all the entries in the matrix are 1 or not. If they are all 1s, it means any two points are connected, so the graph is connected. Otherwise, the graph is not connected.

Here is the pseudocode:

```
Input: a graph G(m nodes) and a start point k(randomly selected)
Output: true->the graph is connected; false->the graph is
    disconnected
1.Create a parent node list P and insert k into this list
2.Create a parent node list P_store and insert k into this list
3.Create a child nodes list C
4.Create a Identity matrix(m*m) M
5.find all the nodes connected to k, and put them into a list, then
    insert the list into C
6.create a integer T, set it as 1
7.while(T!=0):
     T=0
     for each i in P:
        for each node j in C(i):
          if(j is not in P_store):
          if(M(i,j)!=0)
            set M(i,j) and M(j,i) to 1
            find all the nodes connected to j and put them into a
               list, then insert the list into C
    insert j into P_store
            insert j into P
            T+=1
        remove C(i) in C
        remove i in P
8.check whether the sum of all the entries in the matrix is equal to
    m*m or not:
9:yes->return true, no->return false
```

Also, there is another method for this problem as described on the page 44 in the lecture. We could check the output value of the parent node list to see whether the graph is connected or not. If the value of any elements in the list are NONE, which is the initialization value, it verifies that the graph is connected. The main principle behind the algorithm I define myself above is the same as the one described in the lecture but is expressed in different ways(I use matrix but the algorithm in the book uses list).

2.2 ii

If the graph is connected, I think it only has one connected components which is itself. If it is disconnected, it may have more than one connected components.

First, we would have to check whether the graph is connected or not. we could use the method described above. In order to find the connected components, I could the matrix I defined before because it could provide us with enough information to figure out the number connected components. When the graph is disconnected, the sum of all

the entries in the matrix is not equal to m^2. So, there would exist some entries in each row which are zero. I draw a figure to illustrate it:

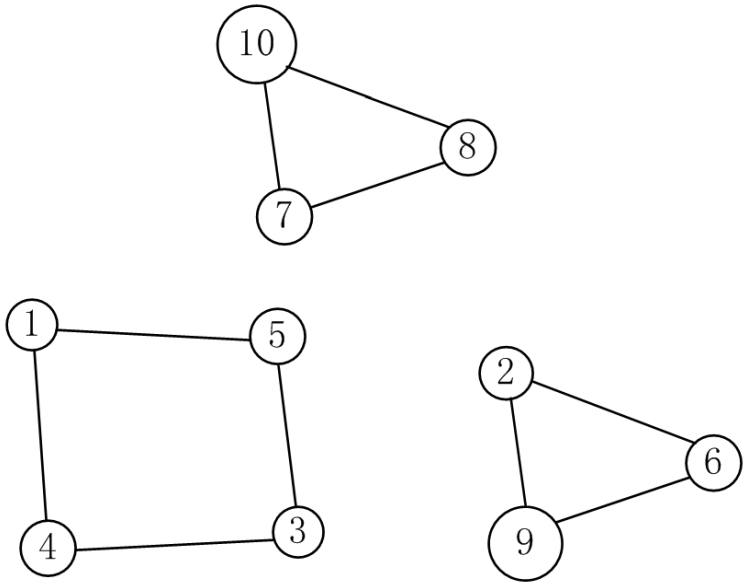

Figure 3: An example

The matrix representing such a graph should be like this(assume the start point is node 1):

M

10x10 double

	1	2	3	4	5	6	7	8	9	10
1	1	0	1	1	1	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0
3	0	0	1	1	1	0	0	0	0	0
4	0	0	1	1	1	0	0	0	0	0
5	0	0	1	1	1	0	0	0	0	0
6	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	1	0	0
9	0	0	0	0	0	0	0	0	1	0
10	0	0	0	0	0	0	0	0	0	1

Figure 4: The corresponding matrix

If we divide the matrix into two small matrix:one is the 4*4 matrix which contains all the 1s in the original large matrix, the other one is the 6*6 identity matrix. For this 4*4 matrix, it corresponds to the connected subgraph in the original matrix and the indexes are 1,3,4,5. Then I change the start point,for example, 2, and implement this algorithm again in the 6*6 matrix. For the real implementation, we have to pay a little attention to the index conversion(the indexes 2,6,7,8,9,10 should be converted into new set of indexes like 1,2,3,4,5,6). Here is the another matrix:

M

6x6 double

	1	2	3	4	5	6
1	1	1	0	0	1	0
2	0	1	0	0	1	0
3	0	0	1	0	0	0
4	0	0	0	1	0	0
5	0	1	0	0	1	0
6	0	0	0	0	0	1

Figure 5: The corresponding matrix

This time, we could extract the columns 1,2,5 and divide the 6*6 matrix into two 3*3 matrices:one is the identity matrix and the other one contains all the 1s. For the same reason, the 3*3 submatrix correspond to the connected subgraph and the indexes are 1,2,5. In the figure 4, these indexes represent the node 2,6,9.

We could change the start point again and this time, there is only one 3*3 matrix left with all the 1s and it represent the subgraph with node 7,8,10.

Hence, for this graph, there are totally three subgraphs, which means the number of connected components is three.

Here is the pseudocode:

```
Input:the matrix M obtained from the part i.
output:the number of connected components
1.Set k equal to 0.
2.while(M is not an identity matrix):
3. extract the columns and rows which contain 1s off the diagonal
4. divide M into two smaller matrix.
5. k+=1
6. set the smaller identity matrix as M
7  if(the size of M is 1*1):
      break;
8. change the start point and implement the algorithm described in
   part i again
```

2.3 iii

For this part, what I need to do it to just add one simple function into the algorithm in the part iii: each time I extract a smaller ones matrix, we count its size and this is the number of nodes in each connected component.

The pseudocode is nearly the same as the previous one:

```
Input:the matrix M obtained from the part i.
output:the number of connected components and the number of nodes in
    each connected component
0.Create an empty list L
1.Set k equal to 0.
2.while(M is not an identity matrix):
3. extract the columns and rows which contain 1s off the diagonal
4. divide M into two smaller matrix.
5. k+=1
6. insert the size of the ones matrix into L
7. set the smaller identity matrix as M
8  if(the size of M is 1*1):
      break;
9. change the start point and implement the algorithm described in
    part i again
```

3 E2.8 Programming:BFS algorithm

3.1 computeBFStree

The problem requires us to represent the graph by its adjacency table **AdjTable**. There are two different cases:

- The graph is connected
- The graph is disconnected

The method to distinguish such two cases are described above. Also, as described on the page 44 in the lecture, we could check whether all the elements in the parent node list are another nodes in the graph. Because, all the elements are initialized as NONE, and if the node has been found as the child node of some other nodes, the corresponding element in the parent node list would be set to the index of the parent node. Hence, if we check the output of the algorithm, we could decide whether or not the graph is connected. Besides, each element in the list represents the indexes of the parent nodes, which are the solutions for the problem. Here is the procedure:

1. create an empty queue Q and insert(Q, v_{start})
2. **for** each node v in G:
3. parent(v) = NONE
4. parent(v_{satrt}) = SELF
5. **while** Q is not empty:
6. v = retrieve(Q)
7. **for** each node u connected to v by an edge:
8. **if** parent(v) == NONE:
9. set paren(u) = v and insert(Q, u)
10. **return** parent list

3.1.1 Verification

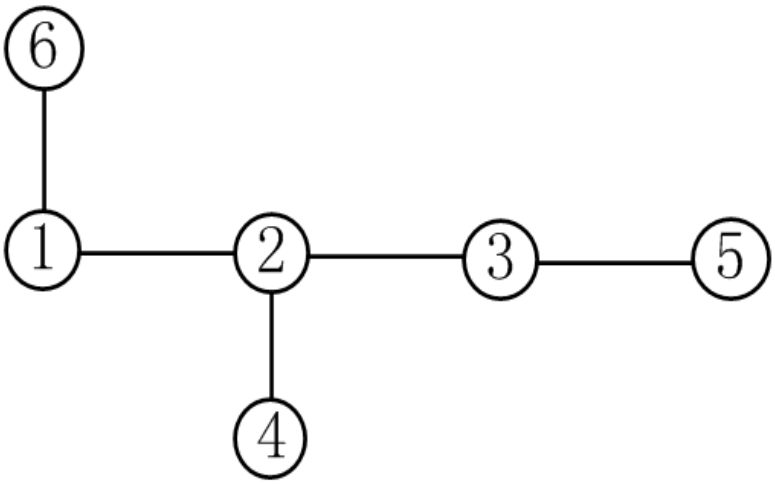

Figure 6: A graph

```
if __name__=='__main__':
    number_nodes=6
    Adjtable=[]
    for i in range(0,number_nodes):
        Adjtable.append(0)
    Adjtable[0]=[2,6]
    Adjtable[1]=[1,3,4]
    Adjtable[2]=[2,5]
    Adjtable[3]=[2]
    Adjtable[4]=[3]
    Adjtable[5]=[1]
    computeBFStree(Adjtable,2)
```

Figure 7: Description of the graph by its adjacency table

```
'''
 RESTART: C:/Users/Administrator/Desktop/winter quarter/MAE145/Homework/Homework
3/program_homework3.py
The graph is connected!
Here is the vector of pointers describing the BFS tree rooted as start:
[2, 2, 2, 2, 3, 1]
'''
```

Figure 8: Result

You can see from the above three figures that the function could give the right solution. Hence, I remove the edge between node 1 and node 6.

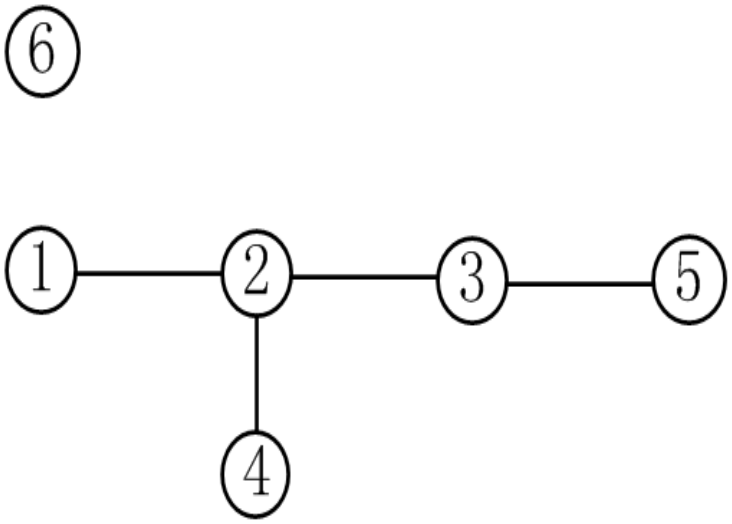

Figure 9: A graph

```
if __name__=='__main__':
    number_nodes=6
    Adjtable=[]
    for i in range(0,number_nodes):
        Adjtable.append(0)
    Adjtable[0]=[2]
    Adjtable[1]=[1,3,4]
    Adjtable[2]=[2,5]
    Adjtable[3]=[2]
    Adjtable[4]=[3]
    Adjtable[5]=[]
    computeBFStree(Adjtable,2)
```

Figure 10: Description of the graph by its adjacency table

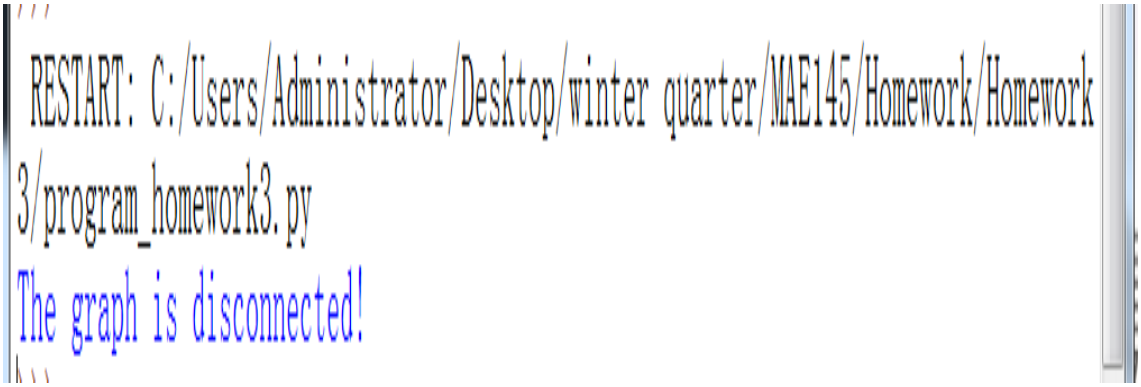

Figure 11: Result

The function could also give the right solution.

3.2 computeBFSpath

For this problem, I only have to add a few codes into the previous function to find the path. These codes could help us save some time. If we find the goal node, we could return the path, instead of scanning the whole graph. Then, I need to extract the path from the goal node and the present nodes list. The algorithm here for the extraction is really simple:during each iteration, find the parent node of the input node and then set the parent node as the input node of the next iteration. Repeat this until the parent node is the start node.

Input: a goal node v_{goal}, and the parent list

Output:a path from v_{start} to v_{goal}

1. create an array P = [v_{goal}]
2. set u = v_{goal}
3. **while** $parent(u)$ is not SELF:
4. u = $parent(u)$
5. insert u at the beginning of P
6. **return** P

3.2.1 Verification

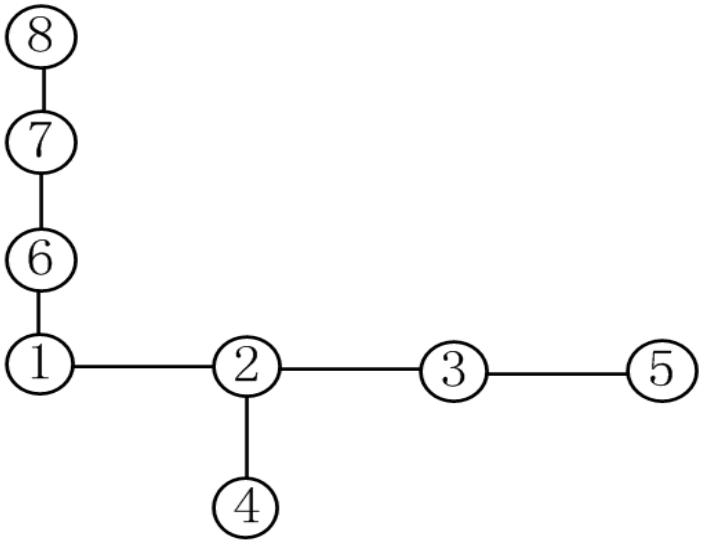

Figure 12: A graph

```
if __name__=='__main__':
    number_nodes=8
    Adjtable=[]
    for i in range(0,number_nodes):
        Adjtable.append(0)
    Adjtable[0]=[2,6]
    Adjtable[1]=[1,3,4]
    Adjtable[2]=[2,5]
    Adjtable[3]=[2]
    Adjtable[4]=[3]
    Adjtable[5]=[1,7]
    Adjtable[6]=[6,8]
    Adjtable[7]=[7]
    #computeBFStree(Adjtable,2)
    path=computeBFSpath(Adjtable,2,6)
    print(path)
    P=extractPath(path,6,2)
    print(P)
```

Figure 13: Description of the graph by its adjacency table

```
'''
RESTART: C:/Users/Administrator/Desktop/winter quarter/MAE145/Homework/Hom
3/program_homework3.py
[2, 2, 2, 2, 3, 1, -1, -1]
[2, 1, 6]
\\\
```

Figure 14: Result

In the figure 14, there are some elements -1, which means since we know the goal node, we could stop scanning the whole graph once we find the goal node.

From the above three figures, you can see that the result is right.

The problem also requires us to verify our algorithms given a graph, which takes me a while to represent it.

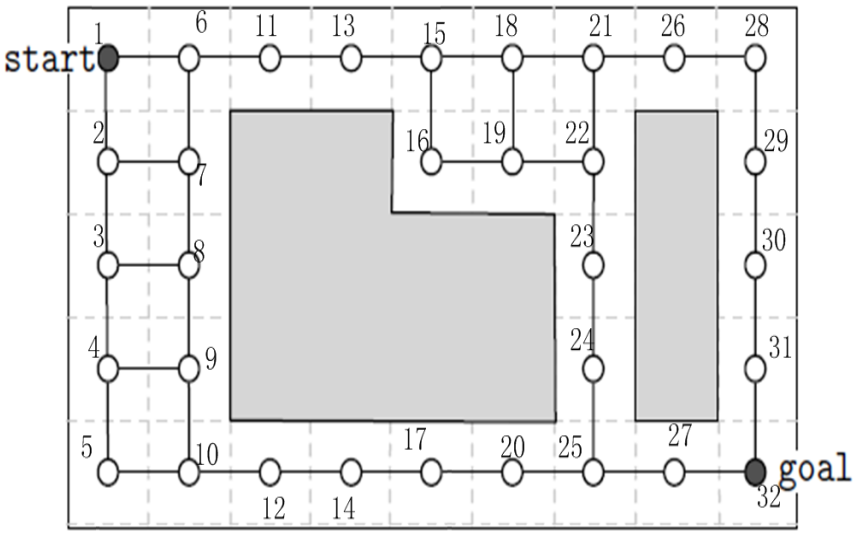

Figure 15: The given graph and its nodes

In the above figure, I mark all the nodes with different numbers.

```
Adjtable[0]=[2, 6]
Adjtable[1]=[1, 3, 7]
Adjtable[2]=[2, 4, 8]
Adjtable[3]=[3, 5, 9]
Adjtable[4]=[4, 10]
Adjtable[5]=[1, 7, 11]
Adjtable[6]=[2, 6, 8]
Adjtable[7]=[3, 7, 9]
Adjtable[8]=[4, 8, 10]
Adjtable[9]=[5, 9, 12]
Adjtable[10]=[6, 13]
Adjtable[11]=[10, 14]
Adjtable[12]=[11, 15]
Adjtable[13]=[12, 17]
Adjtable[14]=[13, 16, 18]
Adjtable[15]=[15, 19]
Adjtable[16]=[14, 20]
Adjtable[17]=[15, 19, 21]
Adjtable[18]=[16, 18, 22]
Adjtable[19]=[17, 25]
Adjtable[20]=[18, 22, 26]
Adjtable[21]=[19, 21, 23]
Adjtable[22]=[22, 24]
Adjtable[23]=[23, 25]
Adjtable[24]=[20, 24, 27]
Adjtable[25]=[21, 28]
Adjtable[26]=[25, 32]
Adjtable[27]=[26, 29]
Adjtable[28]=[28, 30]
Adjtable[29]=[29, 31]
Adjtable[30]=[30, 32]
Adjtable[31]=[27, 31]
```

Figure 16: Codes

```
'''
 RESTART: C:/Users/Administrator/Desktop/winter quarter/MAE145/Homework/Homework
3/program_homework3.py
[1, 1, -1, -1, -1, 1, 6, -1, -1, -1, 6, -1, 11, -1, 13, 15, -1, 15, 18, -1, 18,
21, -1, -1, -1, 21, -1, 26, 28, 29, 30, 31]
[1, 6, 11, 13, 15, 18, 21, 26, 28, 29, 30, 31, 32]
'''
```

Figure 17: Result

You can see that the result is right(it is the shortest path between the start node and the goal node for on the breadth-first search algorithm).