Project: Identifying the Optimum Route from a Network for Commuting

Plan 396: Programming Techniques

Student ID: 1815002, 28,29
Level/Term: 3/2
Dept. of Urban and Regional Planning

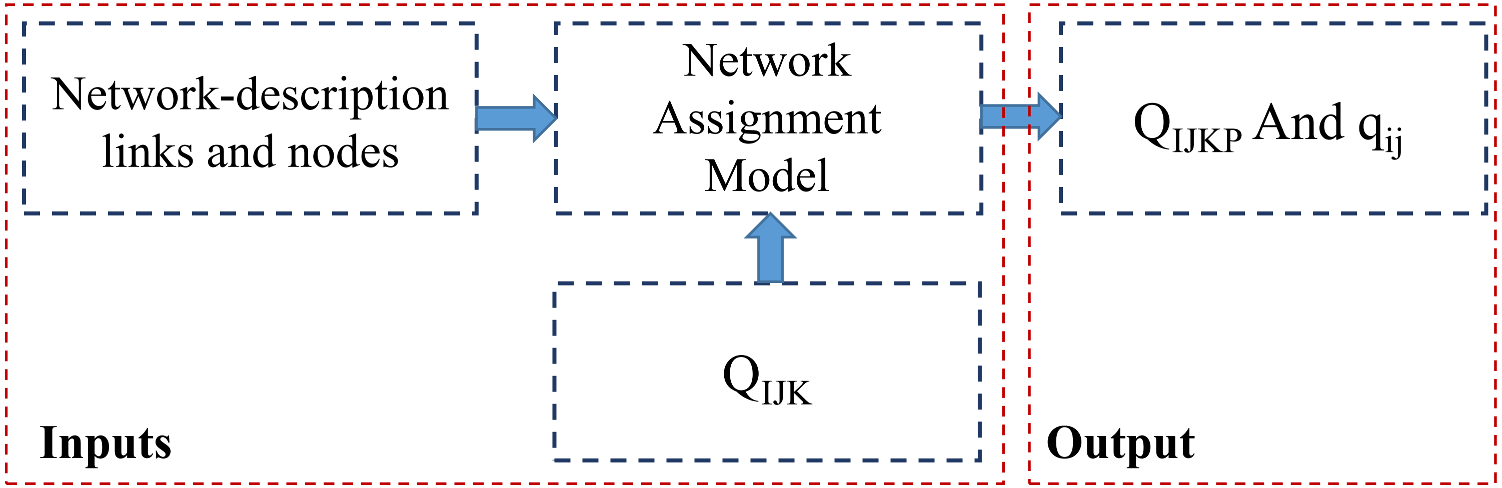
Trip Assignment

- Last phase of the 4 stage model of travel demand
- Concerned with Trip maker's path choice between zones by travel mode and with resulting vehicular flows on the multimodal transportation network
- Equilibrium model between travel demand and supply of transportation in terms of physical facilities.

Problem analysis:

- Determine trip maker's likely choice of paths between all zones I and J along the network of each mode K
- Predict the resulting flows q on individual links that make up the network of that mode

Trip assignment inputs and outputs



Q_{IJK} = Estimate of interzonal demand by mode

Q_{IJKP} = Choice of path between I and J

q_{ij} = flow on individual link

(Papacostas & Prevedouros, 2001)

Trip Assignment on the Principle All-or-Nothing

- All trips between a fixed origin and destination assigned to the links constituting a single shortest connecting path

Assumption

- There are **no congestion effects** and that **all drivers consider the same attributes when choosing their routes**, while perceiving all attributes in the same way and with the same degree of importance.

- Between an origin and a destination point **only a specific route is utilized**, even if other routes have similar travel costs or travel times.

- **Low-density areas** and networks for which there are few alternative routes with large differences (concerning travel times and costs) among them.

(Profillidis & Botzoris, 2018)

Minimum Path Algorithm – To find minimum (impedance) path between zones

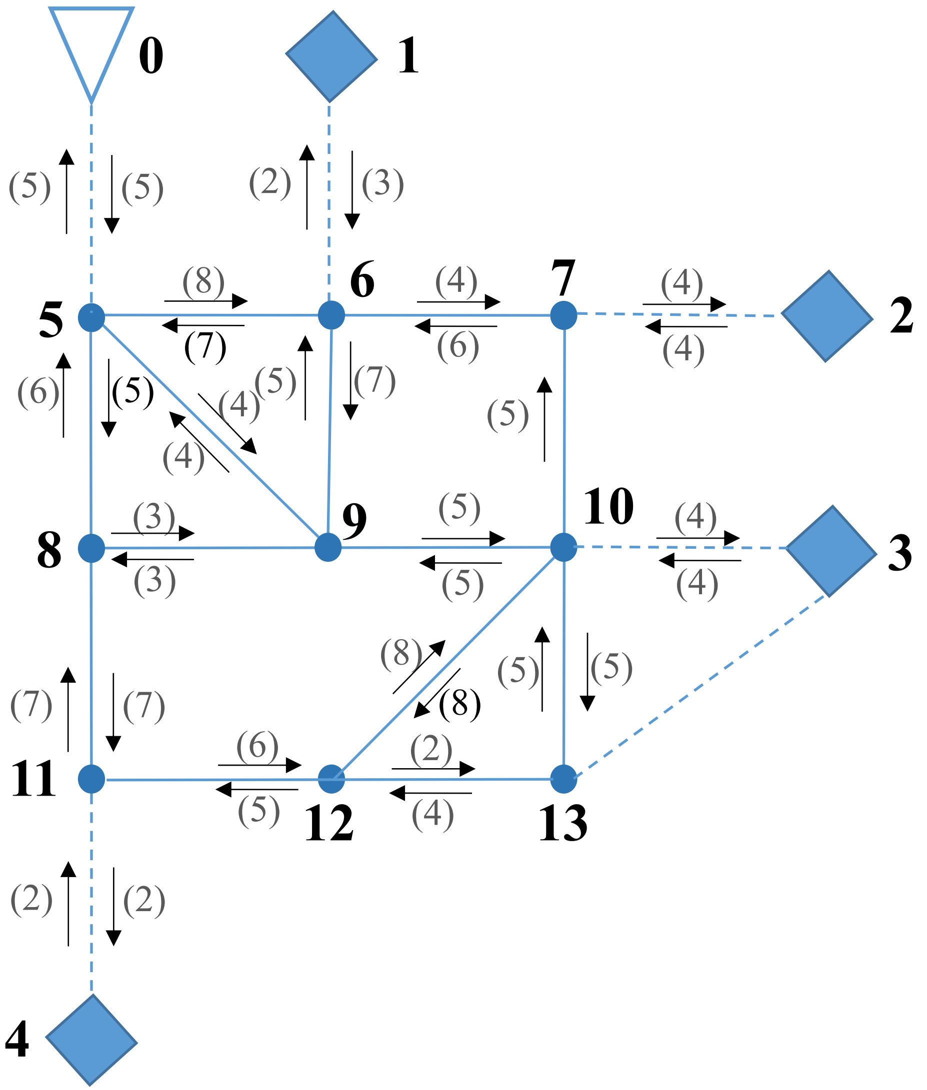
- Identify all possible paths between zones, computing their impedances and choosing the path with lowest impedance

Minimum Tree-Seeking Procedure

- **Step 1:** Initialize the path impedances of the tree table at zero for the node of origin and a very large number for all other nodes. This large number ensures that the first encountered actual path to a node will be chosen
- **Step 2:** Enter into a list the links (i,j) that emanate directly from any node i just added to the tree
- **Step 3:** For each node j included in the list, add the impedance of link (i,j) to the tree table's current total impedance to node i. If the value is smaller than the current tree table entry for node j, replace the current total impedance to j with the new total impedance and enter node I as the node that immediately proceed. Otherwise it proceed to the next link.
- **Step 4:** Return to step 2, unless the list is empty, in which case the tree table contains the solution

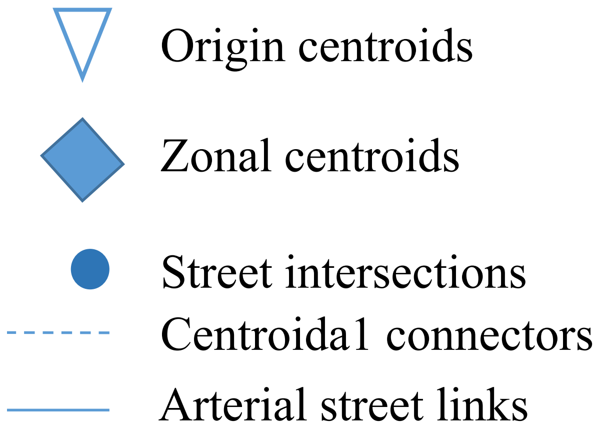
(Papacostas & Prevedouros, 2001)

➢ Problem: Find the optimum routes from a network

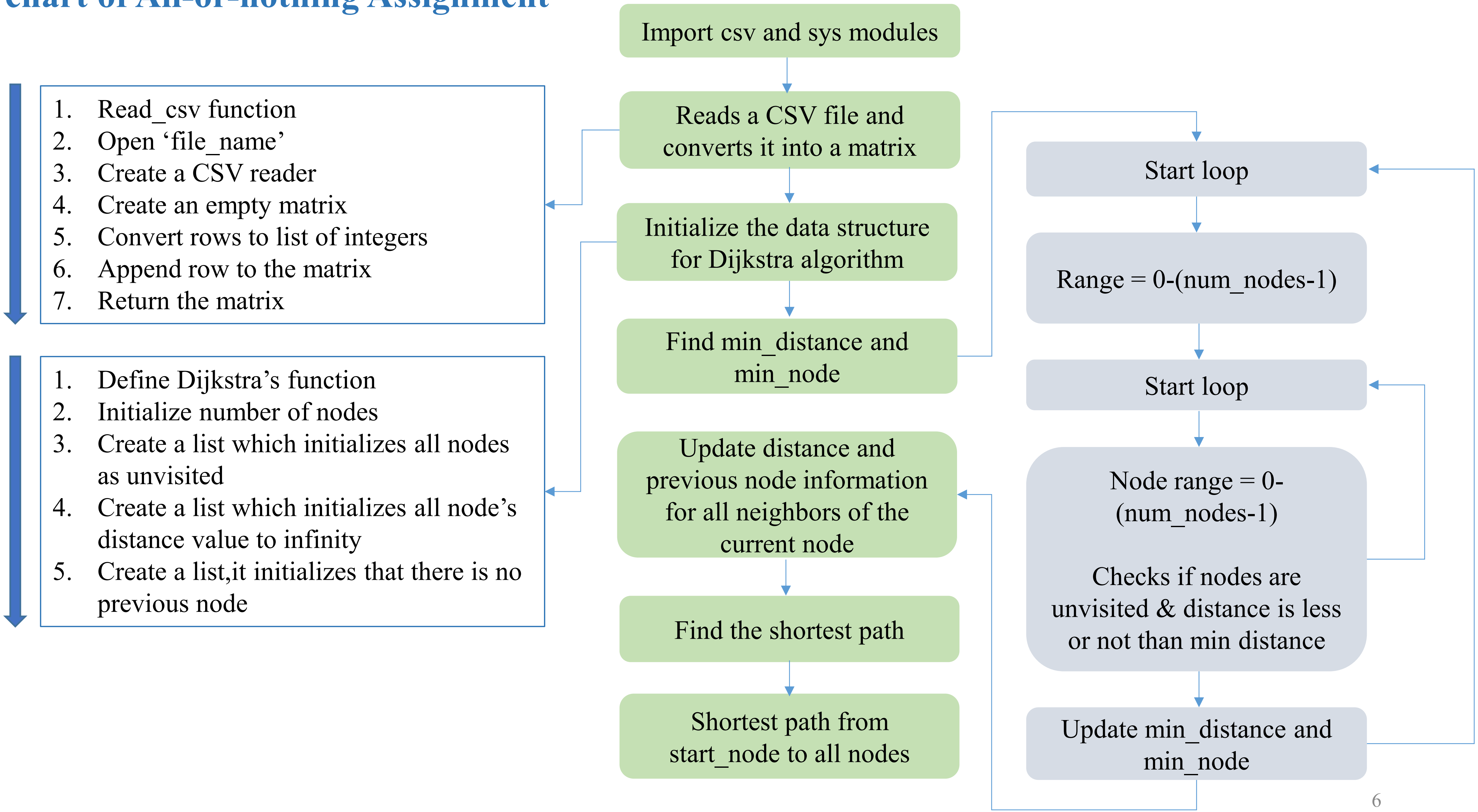


The simple network consists of

- ❑ 5 zonal centroids (nodes 0 to 4)
- ❑ 6 centroida1 connectors
- ❑ 9 street intersections (5 to-13)
- ❑ 13 arterial street links



Flowchart of All-or-nothing Assignment



Flowchart of All-or-nothing Assignment

1. Set min_node as starting node
2. Mark min_node as visited
3. Check if edge exist between min_node and neighbor node

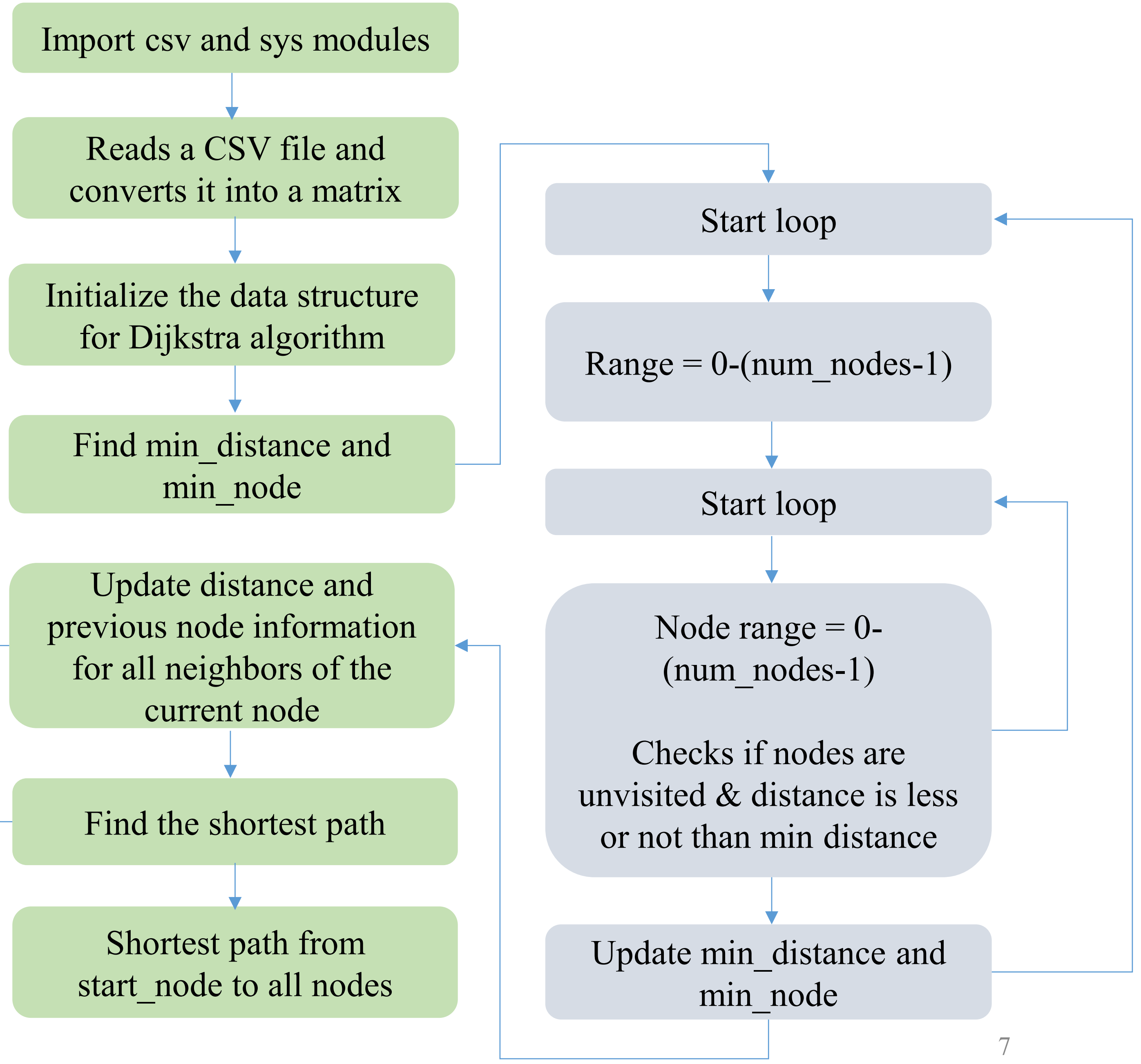
If edge exist

4. Calculate new distance
5. Check if new_dist < dist[neighbor]
6. Update distance and previous node
7. Continue with next neighbor
8. Return final distance and previous node

1. Define shortest path function
2. Create an empty array
3. Initialize node to end

While the node is not none

4. Append node to path
5. Set node=prev[node]
6. Reverse path



Function, loops, statements, keywords, modules, variable

Code: 1st part

Import csv and sys module

```
# Function to read adjacency matrix from CSV file
def read_csv(file_name):
    with open(file_name, 'r') as f:
        reader = csv.reader(f)
        matrix = []
        for row in reader:
            matrix.append(list(map(int, row)))
        return matrix
```

CSV	Provides functionality to work with CSV files
Open()	Used to open files. It takes two arguments: the name of the file to open, and the mode which to open it
as	A keyword to create a variable name which refers to the open file.
reader	Reads CSV data from a file object and returns an iterator object that can be iterated over the rows of the CSV file.
matrix[]	An empty list is assigned to the variable matrix which has no elements
for	Will iterate over each row in the CSV file, one row at a time, assigning the contents of the each row to the variable row in Each iteration.
Matrix append	Appends a new list to the end of the list called matrix.
map	Help to create a new list in a sequence called row, apply int function to each element of the row

```
# Dijkstra's algorithm function

def dijkstra(adj_matrix, start_node):

    num_nodes = len(adj_matrix)

    visited = [False] * num_nodes

    dist = [sys.maxsize] * num_nodes

    prev= [None] * num_nodes

    dist[start_node] = 0
```

a dijkstra = function	Implements Dijkstra’s algorithm for finding the shortest path in a weighted graph
dijkstra 2 parameters: 1. adj matrix 2. start node	1.a two-dimensional matrix that represents the adjacency matrix of the graph. The matrix contains the weights of the edges between the nodes of the graph. If there is no edge between two nodes, the corresponding value in the matrix is infinity. 2. an integer that represents the starting node for the algorithm.

num_nodes = len(adj_matrix)	Get the number of nodes in the graph by getting the length of the adjacency matrix

visited = [False] * num_nodes:	Create a boolean list visited of length num nodes to keep track of which nodes have been visited during the algorithm.

dist = [sys.maxsize] * num_nodes:	Create a list dist of length num_nodes to store the shortest distance from the start node to each node in the graph. Initialize all values in the list to the maximum possible value to start

dist[start_node] = 0:	Initialize the distance from the start node to itself as 0, since the distance from a node to itself is always 0

```
for _ in range(num_nodes):
    min_dist = sys.maxsize
    min_node = None
    for node in range(num_nodes):
        if not visited[node] and dist[node]
< min_dist:
            min_dist = dist[node]
            min_node = node
```

for _ in range(num_nodes):	This loop runs for num_nodes iterations. Since Dijkstra's algorithm visits every node in the graph, this loop ensures that the algorithm visits every node
min_dist = sys.maxsize:	Initialize min_dist to the maximum possible value to start with.
min_node = None:	Initialize min_node to None. If the loop below doesn't find a minimum distance node that hasn't been visited yet, min_node will remain None.
for node in range(num_nodes):	Loop through every node in the graph.
if not visited[node] and dist[node]	< min_dist: Check if the current node has not been visited yet and if its distance from the start node is less than the current minimum distance found so far.
min_dist = dist[node]:	If the current node's distance is less than the current minimum distance found so far, update min_dist.
min_node = node:	Set min_node to the current node with the smallest distance found so far.

```
visited[min_node] = True

for neighbor in range(num_nodes):

    if

adj_matrix[min_node][neighbor] > 0:

        new_dist = dist[min_node] +

adj_matrix[min_node][neighbor]

        if new_dist < dist[neighbor]:

            dist[neighbor] = new_dist

        prev[neighbor] = min_node
```

visited[min_node] = True:	Mark the minimum distance node as visited
for neighbor in range(num_nodes):	Loop through every neighbor of the minimum distance node.
if adj_matrix[min_node][neighbor] > 0:	Check if there is an edge between the minimum distance node and the neighbor. If there is no edge, then there is no need to update the neighbor's distance.
new_dist = dist[min_node] + adj_matrix[min_node][neighbor]:	Calculate the new distance from the start node to the neighbor through the minimum distance node. This is done by adding the weight of the edge between the minimum distance node and the neighbor to the distance from the start node to the minimum distance node
if new_dist < dist[neighbor]:	Check if the new distance is smaller than the current distance from the start node to the neighbor. If it is, update the neighbor's distance to the new distanc
dist[neighbor] = new_dist:	Update the neighbor's distance to the new distance.
prev[neighbor] = min_node:	Keep track of the previous node in the shortest path from the start node to the neighbor. This is used to reconstruct the shortest path later on.

```
def shortest_path(prev, start_node, end_node):
    path = []
    node = end_node
    while node is not None:
        path.append(node)
        node = prev[node]
    path.reverse()
    return path
```

path = []:	Create an empty list to store the nodes in the shortest path
node = end_node:	Start at the end node
while node is not None:	While there is a previous node in the shortest path to the current node
path.append(node):	path.append(node): Add the current node to the path
node = prev[node]:	Move to the previous node in the shortest path
path.reverse():	Since we started at the end node and worked backwards to the start node, the path is currently in reverse order. Reverse the order of the path to get the correct order
return path:	Return the shortest path from the start node to the end node as a list of nodes.

Solution of the selected problem

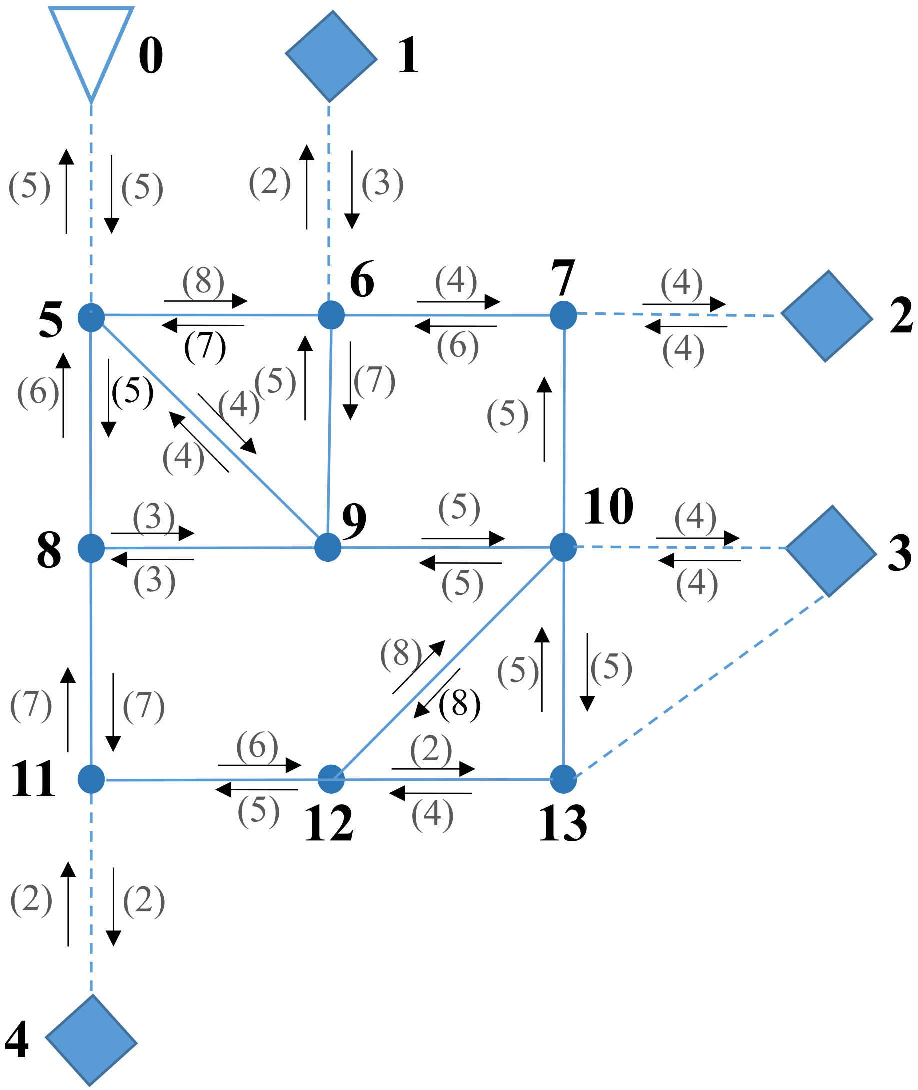
```
>>>
========== RESTART: C:\Users\rafiu\OneDrive\Documents\3-2\python\f.py ==========
Distances from node 0 : [0, 15, 21, 18, 19, 5, 13, 17, 10, 9, 14, 17, 22, 19]
Path from  0  to  0  :  [0]
Path from  0  to  1  :  [0, 5, 6, 1]
Path from  0  to  2  :  [0, 5, 6, 7, 2]
Path from  0  to  3  :  [0, 5, 9, 10, 3]
Path from  0  to  4  :  [0, 5, 8, 11, 4]
Path from  0  to  5  :  [0, 5]
Path from  0  to  6  :  [0, 5, 6]
Path from  0  to  7  :  [0, 5, 6, 7]
Path from  0  to  8  :  [0, 5, 8]
Path from  0  to  9  :  [0, 5, 9]
Path from  0  to  10  :  [0, 5, 9, 10]
Path from  0  to  11  :  [0, 5, 8, 11]
Path from  0  to  12  :  [0, 5, 9, 10, 12]
Path from  0  to  13  :  [0, 5, 9, 10, 13]
>>>
                                                              Ln: 37  Col: 0
```

Node	Total impedance to node j	Node preceding j
0	0	-
1	15	6
2	21	7
3	18	10
4	19	11
5	5	0
6	13	5
7	17	6
8	10	5
9	9	5
10	14	9
11	17	8
12	22	10
13	19	10

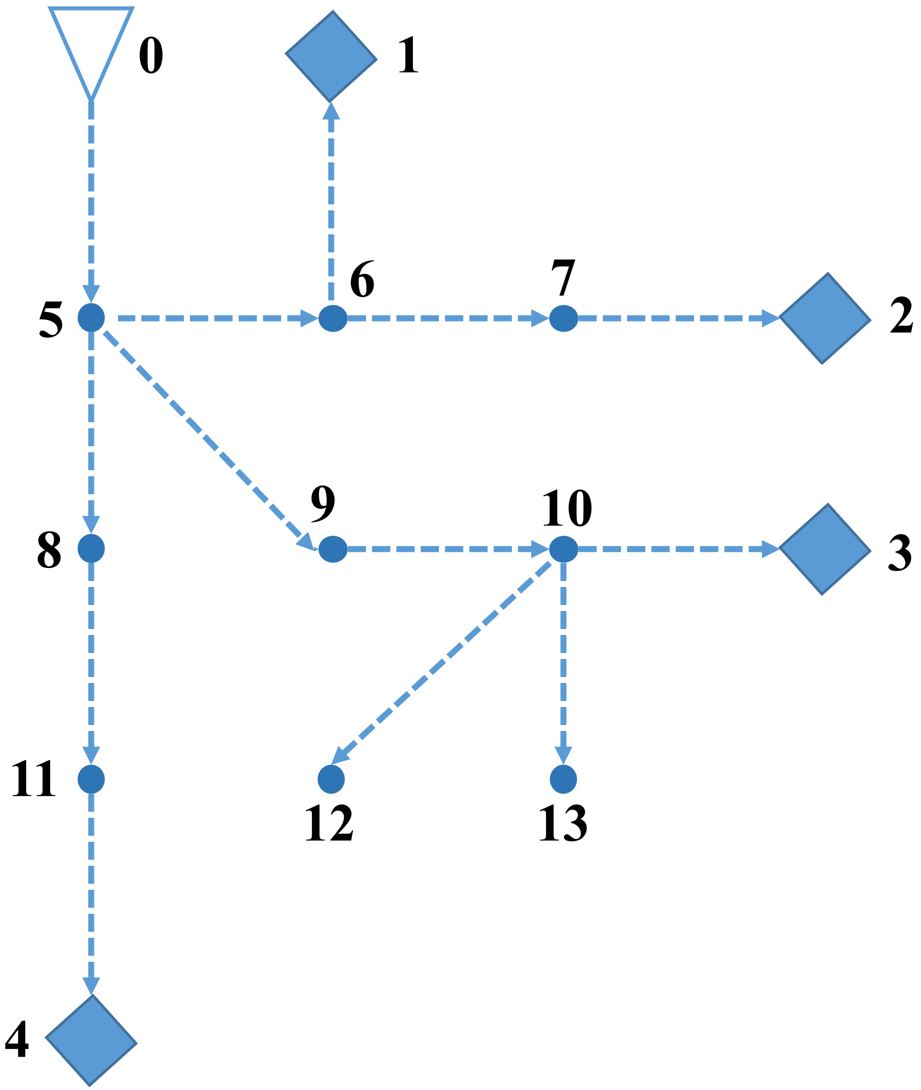
File Edit Format Run Options Window Help

```
    visited = [False] * num_nodes
    dist = [sys.maxsize] * num_nodes
    prev = [None] * num_nodes
    dist[start_node] = 0

    for _ in range(num_nodes):
        min_dist = sys.maxsize
        min_node = None
        for node in range(num_nodes):
            if not visited[node] and dist[node] < min_dist:
                min_dist = dist[node]
                min_node = node

        visited[min_node] = True
        for neighbor in range(num_nodes):
            if adj_matrix[min_node][neighbor] > 0:
                new_dist = dist[min_node] + adj_matrix[min_node][neighbor]
                if new_dist < dist[neighbor]:
                    dist[neighbor] = new_dist
                    prev[neighbor] = min_node

    return dist, prev

def shortest_path(prev, start_node, end_node):
    path = []
    node = end_node
    while node is not None:
        path.append(node)
        node = prev[node]
    path.reverse()
    return path


# Test the function with an example adjacency matrix
adj_matrix = read_csv('C:/Users/rafiu/OneDrive/Documents/3-2/python/Book4.csv')
start_node = 0
distances, prev = dijkstra(adj_matrix, start_node)
print('Distances from node', start_node, ':', distances)


for i in range(0,14):
    path = shortest_path(prev, 0, i)
    print('Path from ',0, ' to ', i, ' : ' ,path )
```

- Problem



- Solution: Shortest network assignment



Shortcoming of All-or-Nothing Assignment

1. **Instability:** A **trivial change in a network's link** times can **cause gross changes in the forecasted link** volumes. In some cases a link can unrealistically shift from being the most heavily used in the system to having too few trips to justify its construction.

2. **Failure to reflect actual behavior:** It **contradicts actual trip behavior**, due to effects of trip volumes on travel time and the **trip maker's non-deterministic choice function**

3. **Inaccuracy:** Total vehicle hours are biased because **all trips are assumed to use the shortest path**. Such an assumption minimizes total travel time. Since this measure is often used by the planner as a macro-evaluator, it means he will always **overestimate the value of his design**.

(Dial, 1971)

References:

1. Profillidis, V. A., & Botzoris, G. N. (2018). Modeling of transport demand: Analyzing, calculating, and forecasting transport demand.
2. Papacostas, C. S., & Prevedouros, P. D. (2001). Transportation engineering and planning.
3. Dial, R. B. (1971). A probabilistic multipath traffic assignment model which obviates path enumeration. *Transportation Research*, *5*(2), 83–111.