Math 6005

TFL JOURNEY PLANNER

Coursework document

Team member:

Umesh Uddar

Pallavi Ashok

Guangyuan Li

Jiankun Li

1. Introduction

Dijkstra's Algorithm is a famous algorithm used in computer science for finding the shortest path between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956.

The algorithm works by assigning a tentative distance value to every node: set it to zero for our initial node and to infinity for all other nodes. Then it marks the initial node as current and visits all its unvisited neighbors. For each neighbor, it calculates the tentative distance through the current node and compares this to the current assigned value, updating it if the calculated distance is less. This process continues until all nodes have been visited. The algorithm efficiently finds the shortest path between the starting node and all other nodes in the graph.

As a group, a Python program that computes the shortest path around a rail network is going to be written as required by TFL. The following paragraph is about how our group has worked on this task, its summary, and its limitations.

2. Details of group work

1. Member Roles & Problems

Apart from helping each other with coding, the specific roles are:

Umesh	Write **Pure code**
Pallavi Ashok	Write the structure of the **Networkx code** and **Document**
Guangyuan Li	Write **Pandas Code**, write and Edit **Document,** Combine codes
Jiankun Li	Complete **Networkx code**, write and Edit **Document,** Combine codes

Our team encountered challenges in converting subway network data into a usable format. Initial attempts using image recognition proved unsatisfactory. Consequently, the approach was shifted to manual methods, involving two different ways of data input: one considering the connections between each station, and the other focusing on a single station as a reference point. Fortunately, both methods yielded the same results, validating the effectiveness of each approach.

What's more, during the coding process, the team faced various challenges, such as difficulty in untangling nested logic and failures in function writing. However, these issues were successfully resolved with mutual assistance and collaboration. This experience highlights the importance of teamwork and collective problem-solving in technical projects.

2. Meeting Minutes

21/12/2023 14:00 :

All team members participated in the entire meeting online. The team agreed on a specific plan of action, involving collaborative coding and validation tasks, along with clear roles and responsibilities for each member during this meeting. We decided to write three codes using distinct strategies (networkx, pure python, and pandas) for mutually verifying correctness.

3. Algorithm Summary

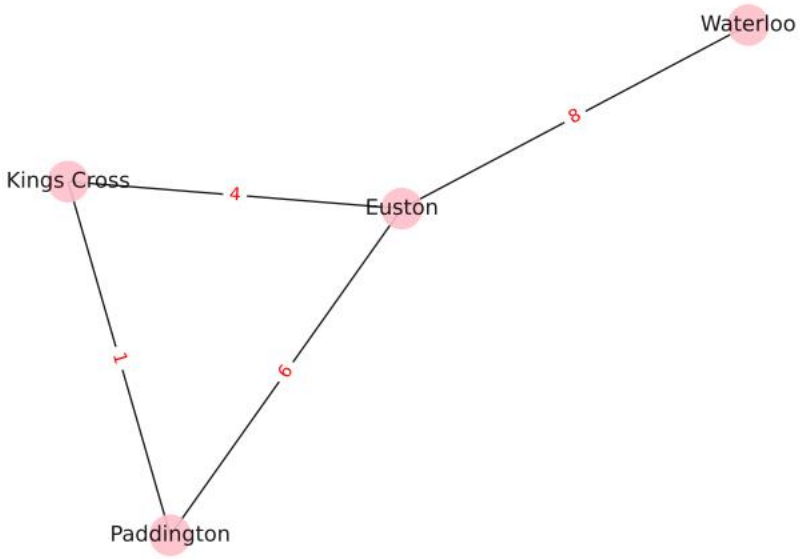
This part is a summary of pandas code.
This algorithm of route finder can find an shortest route and shortest time between any two stations of London metro on the graph. It's made up of a function named Dijkstra_calc and some variables containing the information of the graph. It can also be applied to other similar questions such as finding a shortest distance of two points on a graph.

3.1 Algorithm test

Test the code in a smaller problem at first before applying the algorithm to London metro network graph.

The default parameters of the algorithm are originally set for the given London metro network graph and this smaller test problem is generated from the graph. Therefore, there are only several parameters need to be assigned to apply the algorithm to this smaller problem.

Graph of the smaller problem:



To apply the algorithm to the graph, the graph needs to be converted to variables in the form of a dict and a list or a dataframe at first

When converted to a dict, it is according to every time between stations. The converted dict has two layers with outer keys and inner keys. Select a start station of an edge as an outer key of the dict and destination station as an inner key of this outer key, then the value of this inner key is the time between the two stations. Information of all the routes will be recorded like this. If all the outer keys do not contain names of all the stations, create a list contains all the names of stations. The list is used to be assigned to parameter points.

Let the dict and the list be named as test_graph and test_stations respectively. Start station is Waterloo and destination station is Paddington.

```
test_graph = {'Kings Cross': {'Euston': 4, 'Paddington': 1},
              'Euston': {'Paddington': 6, 'Waterloo': 8}}

test_stations = ['Kings Cross',
                 'Paddington',
                 'Euston',
                 'Waterloo']
```

Then assign the values above to parameters of function Dijkstra_calc and invoke the function.

```
Dijkstra_calc(test_graph, start: 'Waterloo', end: 'Paddington', points=test_stations)
```

Output:

```
Starting station:Waterloo
Destination station:Paddington
Time: 13minutes
Route: Waterloo->Euston->Kings Cross->Paddington
```

The output is correct.

3.2 Algorithm application for given network graph

Now apply the algorithm to the given London metro network graph.

The first step is to convert the graph as the way on code test. The easiest conversion is a dict and a list. Let the dict and the list be named as network_London_metro and stations respectively. Start stations are Waterloo, Victoria, Holborn and Bank respectively and destination station is Paddington.

```
network_London_metro = {'Paddington':{'Baker Street': 6, 'Notting Hill Gate': 4},
                        'Notting Hill Gate':{'Bond Street': 7, 'South Kensington': 7},
                        'South Kensington':{'Green Park': 7, 'Victoria': 4},
                        'Victoria':{'Green Park': 2, 'Westminster': 4},
                        'Westminster':{'Green Park': 3, 'Embankment': 2, 'Waterloo': 2},
                        'Waterloo':{'Embankment': 2, 'London Bridge': 3, 'Elephant and Castle': 4},
                        'London Bridge':{'Elephant and Castle': 3, 'Bank': 2},
                        'Blackfriars':{'Embankment': 4, 'Bank': 4},
                        'Piccadilly Circus':{'Green Park': 1, 'Oxford Circus': 2, 'Leicester Square': 2, 'Charing Cross': 2},
                        'Bond Street':{'Baker Street': 2, 'Green Park': 2, 'Oxford Circus': 1},
                        'Oxford Circus':{'Green Park': 2, 'Baker Street': 4, 'Warren Street': 2, 'Tottenham Court Road': 2},
                        'Tottenham Court Road':{'Warren Street': 3, 'Holborn': 2},
                        'Leicester Square':{'Tottenham Court Road': 1, 'Holborn': 2, 'Charing Cross': 2},
                        'Charing Cross':{'Embankment': 1},
                        'Kings Cross':{'Baker Street': 7, 'Warren Street': 3, 'Holborn': 4, 'Moorgate': 6, 'Old Street': 6},
                        'Moorgate':{'Old Street': 1, 'Liverpool Street': 2, 'Bank': 3},
                        'Bank':{'Holborn': 5, 'Liverpool Street': 2, 'Tower Hill': 2},
                        'Tower Hill':{'Liverpool Street': 6, 'Aldgate East': 2},
                        'Aldgate East':{'Liverpool Street': 4}}

stations =['Aldgate East',
           'Baker Street',
           'Bank',
           'Blackfriars',
           'Bond Street',
           'Charing Cross',
           'Elephant and Castle',
           'Embankment',
           'Green Park',
           'Holborn',
           'Kings Cross',
           'Leicester Square',
           'Liverpool Street',
           'London Bridge',
           'Moorgate',
           'Notting Hill Gate',
           'Old Street',
           'Oxford Circus',
           'Paddington',
           'Piccadilly Circus',
           'South Kensington',
           'Tottenham Court Road',
           'Tower Hill',
           'Victoria',
           'Warren Street',
           'Waterloo',
           'Westminster']
```

Then assign values, invoke the function and get outputs:

```
Dijkstra_calc(network_London_metro, start: 'Waterloo', end: 'Paddington', stations)

Dijkstra_calc(network_London_metro, start: 'Victoria', end: 'Aldgate East', stations)

Dijkstra_calc(network_London_metro, start: 'Holborn', end: 'Elephant and Castle', stations)

Dijkstra_calc(network_London_metro, start: 'Bank', end: 'Green Park', stations)
```

```
Starting station:Waterloo
Destination station:Paddington
Time: 15minutes
Route: Waterloo->Westminster->Green Park->Bond Street->Baker Street->Paddington


Starting station:Victoria
Destination station:Aldgate East
Time: 15minutes
Route: Victoria->Westminster->Waterloo->London Bridge->Bank->Tower Hill->Aldgate East


Starting station:Holborn
Destination station:Elephant and Castle
Time: 10minutes
Route: Holborn->Bank->London Bridge->Elephant and Castle


Starting station:Bank
Destination station:Green Park
Time: 10minutes
Route: Bank->London Bridge->Waterloo->Westminster->Green Park
```

3.3 Algorithm application for other graphs

3.3.1 Parameter modification

Other parameters are used to allow the algorithm being applied to other questions easily.

These parameters all have their default values. If all these values are default, the graph will be regarded as undirected, output will not include total table, name of points on the graph is station, unit is minutes, name of edges on graph is Time, the shortest value is rounded to integer and the function will print a summary of information about the shortest value and route.

To apply the algorithm to other graphs, these parameters can be modified according to the graphs.

Parameter end: specify the destination point. Its default value is a null string and the algorithm will only calculate a total table in this case.

For parameter undirected and points:

If the graph is undirected and converted to a dict, the dict can be created according to every edge between points like the operation on algorithm test.

If the graph is directed and converted to a dict, what to do is like the operations above, but outer keys must be start stations and inner keys must be destination stations. The parameter undirected should be set as False.

If the graph is converted to a dataframe, the names of columns are start points and indexes are destination points, each entry is the value of corresponding edge. Parameter undirected doesn't take effect in this case. The algorithm will start calculation according to the origin data in parameter graph. Therefore, a dataframe is required to record all the values of edges according to directions.

Parameter pointname: specify the name of points on the graph such as station and site

Parameter edgename: specify the name of edges on the graph such as time and distance

Parameter unit: specify the unit name of values of edges such as minutes and meters

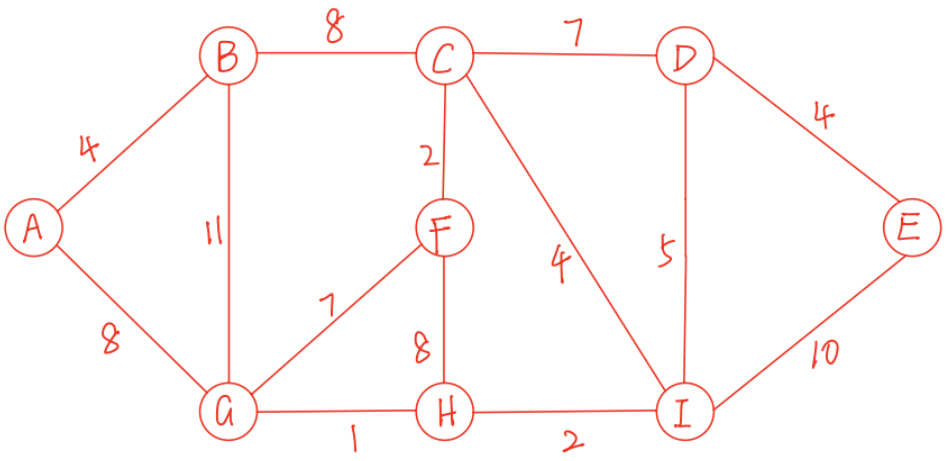
Parameter place: specify the demical places of the shortest values in summary.

For parameter totalprint whose default value is False, if it's True, the output will print a total table that contains all the shortest routes and time.

For parameter summaryprint whose default value is True, if it's True, the output will print a summary of the shortest value and

3.3.2 Apply the algorithm to other graphs:

Graph1:



Convert graph1 to a dataframe:

```
graph1 = pd.DataFrame( data: [[0,4,None,None,None,None,8,None,None],
                              [4,0,8,None,None,None,11,None,None],
                              [None,8,0,7,None,2,None,None,4],
                              [None,None,7,0,4,None,None,None,5],
                              [None,None,None,4,0,None,None,None,10],
                              [None,None,2,None,None,0,7,8,None],
                              [8,11,None,None,None,7,0,1,None],
                              [None,None,None,None,None,8,1,0,2],
                              [None,None,4,5,10,None,None,2,0]],
                              columns=['A','B','C','D','E','F','G','H','I'],
                              index=['A','B','C','D','E','F','G','H','I'])
```

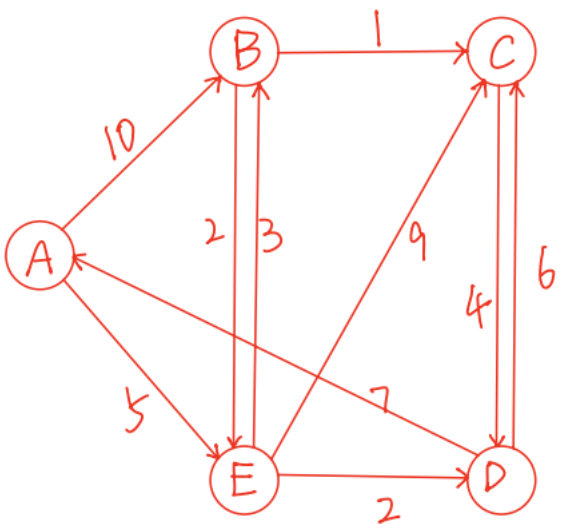
Invoke function:

```
Dijkstra_calc(graph1, start: 'A', end: 'E', pointname='point', edgename='Distance', unit='')
```

Output:

```
Starting point:A
Destination point:E
Distance: 20
Route: A->G->H->I->D->E
```

Graph2:



Convert graph2 to a dataframe:

```
graph2 = pd.DataFrame( data: [[0,None,None,7,None],
                        [10,0,None,None,3],
                        [None,1,0,6,9],
                        [None,None,4,0,2],
                        [5,2,None,None,0],],
                        columns = ['A','B','C','D','E'],
                        index = ['A','B','C','D','E'])
```

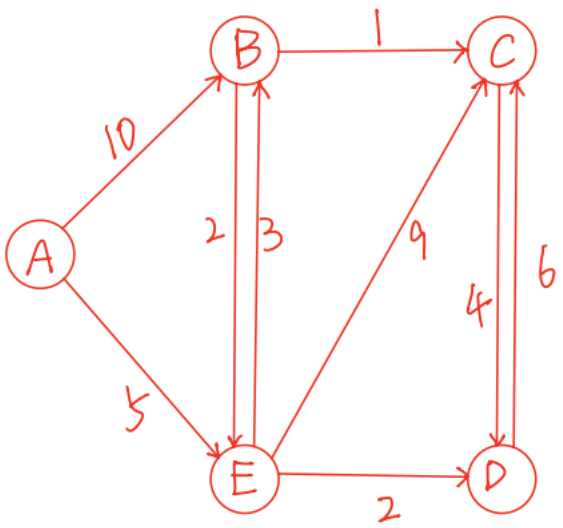
Invoke function:

```
Dijkstra_calc(graph2,  start: 'D',  end: 'B', undirected = False,
              pointname='point', edgename='Distance', unit='')
```

Output:

```
Starting point:D
Destination point:B
Distance: 15
Route: D->A->E->B
```

Graph3:
Delete the connection between A and D and convert it to dataframe graph3



For this graph, if parameters is input like them on graph2, there will be an error since there is no route from D to B
We can set start point as D and output a total table:

```
Dijkstra_calc(graph3, start: 'D', undirected = False,
              pointname='point', edgename='Distance', unit='', printtotal=True)
```

```
The total optimal distance and route:
   Distance from D Route from D
A              NaN
B              NaN
C              6.0         D->C
D              0.0            D
E              NaN
```

4. Limitations of algorithms

4.1 Limitations of Pandas Code

Limitation1:
The algorithm calculates all the shortest routes and values like distances and times at first and then select route and values according to input start point and end point. The benefit is that we can calculate and get all the minimum time and route from a single table. However, for the situation where only the minimum time and route between start station and end station need to be calculated, there are some redundant computations. To optimize computation, the algorithm can be modified to end calculation when the end station is in the series of shortest time.

Limitation2:
Another limitation is that the algorithm can only find one route from start station to destination station while there may exist other routes with same shortest time.
To output more routes, some new series could be created and when there is a route with a same shortest time appears, record it in these series and carry out calculation on these series after the prior calculation is finished.

4.2 Limitations of Networx Code

The efficiency of performance on large-scale graphs might be suboptimal. This limitation is primarily due to the system being restricted to handling unidirectional edges and specific incidents.

5. Real world problem consideration

For the real world problem, if only the time consumption between each station is considered, these algorithms can calculate the shortest time and route accurately. However, it can be noticed that all routes between any two stations are not all in a same metro line and different colored lines on the graph represent different metro lines. A passenger may have to transfer to another metro line at a station to get to destination station. Obviously, there is a time consumption when a passenger has a metro transfer. This time consumption is not considered in these algorithms.