08/2024

BIG GRAPH **PROCESSING**

Data Structures - CS163

Task 02.5: Find bus stops' importance

Prepared By
Le Tien Dat

Student ID
23125028

Class
23APCS2

www.facebook.com/ltd.sword
ltdat23@apcs.fitus.edu.vn
University of Science, VNU-HCM

Table of Contents

1. Problem Statement

Given a bus transportation network in HCMC:

- A bus route contains a list of bus stops.
- A bus stop has a list of timestamps.
- A node contains the following attributes: (route_id, var_id, stop_id, time_stamp, node_type, latx, lngy).
- An edge is established between two nodes as follows: NODE_TYPE_DEPART => NODE_TYPE_ARRIVAL, or NODE_TYPE_ARRIVAL => NODE_TYPE_DEPART.
- The weight of each edge is a tuple: (number of transfers, travel time or time difference).
    - number of transfers = 0 when the start node and the end node of an edge are on the same route. This corresponds to type 1 or 2.
    - number of transfers = 1 when the start node and the end node of an edge are on different routes. This corresponds to type 3 or 4.

For each pair of stops, you are asked to find the shortest path with the least weight. And if two paths for the pair (i, j) have the same weight, choose the path with the earliest departure time and, if needed, the earliest arrival time. Use these shortest paths to find the top k most important bus stops.

The provided files are the edges of the graph:

- type12.csv: 500k lines
- type34.csv: 5 million lines

Structure of each line:

```
stop_id1,route_id1,var_id1,timestamp1,stop_id2,route_id2,var_id2,timestamp2,time_dif
f,latx1,lngy1,latx2,lngy2,vehicle_number1,vehicle_number2,node_type1,node_type2,node
_pos1,node_pos2,edge_pos,edge_type
```

Students are asked to complete these tasks, including writing code, documenting the implementation, compiling results, and analyzing both time and space complexity as well as real-world runtime performance.

2. Find the shortest path between all pairs of stops

2.1. Construct the graph

The input file is the list of edges, including that each line is an distinct edge connecting two nodes with the weight is the number of transfer and time difference.
Firstly, we have to read each line of the two files and add it to our graph (we utilize the adjacency list for storing the graph, denotes as `self.graph`)

```
def loadData(self, filename, type):
        # load the data from CSV file
        #cur = times.time()
        file = open(filename, "r")
        for line in file:
            self.count += 1
            data = line.split(",")
            # structure of CSV file:
            # stop_id1,route_id1,var_id1,timestamp1,stop_id2,route_id2,var_id2,timestamp2,
            # time_diff,latx1,lngy1,latx2,lngy2,vehicle_number1,vehicle_number2,
            # node_type1,node_type2,node_pos1,node_pos2,edge_pos,edge_type
            time_diff = float(data[8])
            timestamp1 = int(float(data[3]))
            timestamp2 = int(float(data[7]))
            vertex1 = (data[0], timestamp1)
            vertex2 = (data[4], timestamp2)
            edgetype = data[20]
            # weight of the edge is number of transfer and time difference between two nodes
            # we prioritize the number of transfer over time difference
            if (self.graph[vertex1].get(vertex2) == None): self.graph[vertex1][vertex2] =
(2, 99999999, 'inf')
            if (type == '12'):
                self.graph[vertex1][vertex2] = min(self.graph[vertex1][vertex2], (0,
time_diff, edgetype))
            else:
                self.graph[vertex1][vertex2] = min(self.graph[vertex1][vertex2], (1,
time_diff, edgetype))

            #if (self.count % 50000 == 0): print(f"Processed {self.count} lines in
{times.time() - cur} seconds")
        file.close()
```

Our vertex consists of a `stop_id` and the timestamp of that stop – we actually don't need to add the `route_id` and `var_id` to the vertex (since we have the number of transfer in the weight of the edge). The weight between two nodes is a tuple: `(number of transfer, time difference, edge type)`. Actually, we just need the first 2 attributes, but I need the third one for our optimization in section 2.3.

Since the node contains both the `stop_id` and `timestamp`, the number of vertices is roughly equal to 500000 (see the result in section 4.2). However, through the optimization in section 2.3, this number has decreased dramatically.

2.2. Find the shortest path

In performing the shortest path between all nodes, there are a plethora of algorithms to compute the result for this problem; but I think the most efficient way and convenient way to determine the shortest path between all pairs of nodes is Dijkstra's algorithm.

Firstly, we need to initialize the base things for this algorithm. I introduce the dictionaries called `dist` and `trace` for saving the smallest distance from one vertex to the start stop and the path from this node to one of the start vertex which has the shortest distance. The process of finding the shortest path from all pairs of vertices contains performing N functions (N is the number of vertices). In each function, we need to calculate the shortest path from one stop to others. To do that, we have to get all the vertices with the same `stop_id` and perform the Dijkstra algorithm on these starting points. Since we need to get the minimum distance between the nodes, we can push all of this vertices into a `PriorityQueue`.

This is the preprocessing code:

```python
# perform Dijkstra algorithm to find the shortest path from every start stop to all other stops in all timestamps
        lst = self.stops[startStop]
        record = defaultdict(set)
        mini = {}
        print(f"Processing {startStop} with {len(lst)} vertices")
        dist = defaultdict(lambda: (9999, float(99999999), None))
        trace = defaultdict(lambda: None)
        pq = PriorityQueue()
        for start in lst:
            # dist structure = (number of transfer, time difference, starting point)
            dist[start] = (0, 0, start)
            trace[start] = None
            pq.put((dist[start], start))
```

Here we have `self.stops` is the dictionary that contains the list of nodes corresponding to this `stop_id`. The implementation of this dictionary is straight-forward:

```python
    def loadStopId(self):
        # load the distinct stop id from the graph
        for key in self.graph:
            self.stops[key[0]].append(key) # stop_id, route_id, var_id, timestamp
        print("Number of stops:", len(self.stops))
```

We also put in the starting node for `dist` (`dist[2]`), which makes the tracing phase more efficient.

Now we begin with the processing phase. This one is just similar to the normal Dijkstra algorithm, but to make it faster, I introduce a dictionary called `record`. This thing can identify whether the vertex has been considered or not. If the node has been in `record`, this means that the distance from this node is optimized and we don't need to consider it again. You can see it on the code in the preprocessing phase.

Then, to tracing the path easier, I came up with creating a new attribute called `mini`. This dictionary would *minimize* the distance of the nodes that have the same `stop_id` by choosing the vertex having the shortest distance.

Below is the code for the main algorithm:

```
while (not pq.empty()):
        u = pq.get()[1]
        if (u not in record[u[0]]):
            record[u[0]].add(u)
            if (u[0] not in mini): mini[u[0]] = u
            else:
                if (dist[u] < dist[mini[u[0]]]):
                    mini[u[0]] = u
        else:
            continue
        for v in self.graph[u]:
            edge = self.graph[u][v]
            time_diff = edge[1]
            numoftrans = edge[0]
            if (dist[v] > (dist[u][0] + numoftrans, dist[u][1] + time_diff,dist[u][2])):
                dist[v] = (dist[u][0] + numoftrans, dist[u][1] + time_diff, dist[u][2])
                pq.put((dist[v], v))
                trace[v] = u
```

Combine this one to the preprocessing phase, we have an overall view for this function:

```
def dijkstra1Stop(self, startStop):
        # perform Dijkstra algorithm to find the shortest path from every start stop to all
other stops in all timestamps
        cur = times.time()
        lst = self.stops[startStop]
        record = defaultdict(set)
        mini = {}
        print(f"Processing {startStop} with {len(lst)} vertices")
        dist = defaultdict(lambda: (9999, float(99999999), None))
        defaultTuple = (9999, float(99999999), None)
        trace = defaultdict(lambda: None)
        pq = PriorityQueue()
        for start in lst:
            # dist structure = (number of transfer, time difference, start point)
            #print(f"Processing {start}")
            dist[start] = (0, 0, start)
            trace[start] = None
            pq.put((dist[start], start))
        while (not pq.empty()):
            u = pq.get()[1]
            if (u not in record[u[0]]):
                record[u[0]].add(u)
                if (u[0] not in mini): mini[u[0]] = u
                else:
                    if (dist[u] < dist[mini[u[0]]]):
                        mini[u[0]] = u
            else:
                continue
            for v in self.graph[u]:
                edge = self.graph[u][v]
                time_diff = edge[1]
                numoftrans = edge[0]
                if dist[v] > (dist[u][0] + numoftrans, dist[u][1] + time_diff, dist[u][2]):
                    dist[v] = (dist[u][0] + numoftrans, dist[u][1] + time_diff, dist[u][2])
                    pq.put((dist[v], v))
                    trace[v] = u
```

2.3. Optimization

Recall for the details of the input files, we can see that there are 4 types of edges:

- Type 1: The edge indicates that the it takes `time_diff` to go from one bus stop to the next stop in the same route.
- Type 2: This edge refers to the *waiting time* of the same bus at a bus stop.
- Type 3: This edge means that it takes `time_diff` to change the route in a bus stop (we have to *wait* for another route to come at the bus stop).
- Type 4: This edge takes us to the another stop with another route.

From the `edge_type` explanation, we can combine the edges in type 2 and 3 (both refer to the "*waiting*" time) to the next edges and contract the destination nodes of this kind of edge to reduce the numbers of vertices and edges we have to check when performing the Dijkstra algorithm.

The steps we conduct the optimization will be:

- Iterate for the nodes that need to contract.
- Add the new edges connecting the previous nodes to the nodes that links with the contracting nodes (that's why I call it "combine").
- Delete the contracting nodes.

This is the implementation for this function. You can go to my GitHub repository for more information about the code in [5].

```
def contract(self):
        cur = times.time()
        # contract the edge having type = 2 or 3
        count = 0
        delete = defaultdict(set)
        lst = set()
        addi = defaultdict(dict)
        for key in self.graph:
            for key2 in self.graph[key]:
                if (self.graph[key][key2][2] == '2' or self.graph[key][key2][2] == '3'):
                    if (key2 not in self.graph): continue
                    for key3 in self.graph[key2]:
                        if (key3 == key): continue
                        if (key3 not in self.graph[key2]): continue
                        addi[key][key3] = (self.graph[key][key2][0] +
self.graph[key2][key3][0], self.graph[key][key2][1] + self.graph[key2][key3][1],
self.graph[key2][key3][2])
                        count += 1
                        delete[key2].add(key3)
                    delete[key].add(key2)
                    lst.add(key2)
                    count += 1
        print(f"Number of edges to be contracted: {count}")
        count = 0
        # add the new edge
        for key in addi:
            count += len(addi[key])
            for key2 in addi[key]:
                self.graph[key][key2] = addi[key][key2]
        print(f"Number of edges added: {count}")
        print("Number of vertices:", len(self.graph))
        print("Number of edges:", self.getLenEdges())
        # delete the edge having type = 2 or 3
        count = 0
        for key in delete:
            for key2 in delete[key]:
                count += 1
                del self.graph[key][key2]
        print(f"Number of edges deleted: {count}")
        print(f"Number of nodes deleted: {len(lst)}")
        print("Number of edges:", self.getLenEdges())
        for key in lst:
            self.graph.pop(key)
        print("Number of vertices:", len(self.graph))
        print("Number of edges:", self.getLenEdges())
        print(f"Done contracting in {times.time() - cur} seconds")
```

The result of this contracting function is in section 4.2, and it really can decrease the number of nodes to 250000, which is a significant figure.

3. Find the top k most important bus stop

From the above Dijkstra algorithm, we can easily find the path from the starting stop to each stop by using the dictionary `mini`. And on this path, we will increase the stops that we pass through by 1. This will result to the importance of all stops after we perform the Dijkstra algorithm for all stops:

```
def dijkstraAllStop(self):
        self.loadStopId()
        cur = times.time()
        count = 0
        for stop in self.stops:
            self.dijkstra1Stop(stop)
            count += 1
            print(f"Processed {stop}, ({count}/4148) in {times.time() - cur} seconds")
```

After sorting the key, we will get the list of the most important stops to the least important stops. Moreover, we can combine the tracing process into the function `dijkstra1Stop()` to make it easier.
Below is the code for trace back:

```
# (in function self.dijkstra1Stop())…
for stop in mini:
            if (stop == startStop): continue
            minDist = dist[mini[stop]]
            minStop = mini[stop]
            # the minimum distance = minDist
            # the stop that has the minimum distance = minStop
            # trace back to increase the importance of the stop
            # increse by 1 in self.vertices[stop]
            minStart = minDist[2]
            while (minStop != minStart and minStop != None):
                self.vertices[minStop[0]] += 1
                minStop = trace[minStop]
        print(f"Processed {startStop}, trace back in {times.time() - cur} seconds")
```

The `self.vertices` is the place where I store the *importance* (in other words, frequency) of the stops. Each time the path goes through a certain stop, it will augment the value of this dictionary to 1.

We can also save the importance of all stops to a JSON file for easier to query. To get the top k most important bus stop, we need to sort the dictionary from most to least importance:

```
def outputImportance(self, filename):
        # output the importance of each stop to a file
        # the vertices should be sorted by the largest importance to the smallest importance
        vertices = sorted(self.vertices.items(), key=lambda x: x[1], reverse=True)
        with open(filename, 'w', encoding='utf8') as file:
            for vertex in vertices:
                json.dump(vertex, file, ensure_ascii=False)
                file.write("\n")
```

Get the top k important bus stops via the file:

```
    def loadImportance(self, filename):
        with open(filename, 'r', encoding='utf8') as file:
            for line in file:
                data = json.loads(line)
                self.vertices[data[0]] = data[1]

    def findTopK(self, k):
        self.loadImportance("importance.json")
        vertices = sorted(self.vertices.items(), key=lambda x: x[1], reverse=True)
        file = open(f"top{k}.txt", "w")
        for i in range(k):
            file.write(str(vertices[i]) + "\n")
        return vertices[:k]
```

4. Complexity, Result and Testing

4.1. Complexity

- For loading the data:

- Time complexity: $O(L.C)$, where L is the number of lines and C is number of components we have split out.

- Space complexity: $O(E)$, where E is the total edges of the graph.

- Running the Dijkstra algorithms:

- Time complexity: $O(V.ElogV)$, where V is the number of vertices and E is the number of edges after optimizing. In real-life testing, the running time can be much smaller than the expecting value since the graph is NOT connected - not all the vertices can be accessed by one node.

Space complexity: $O(V)$, where V is the number of vertices (the space complexity comes from `dist, trace` and other list and dictionary)

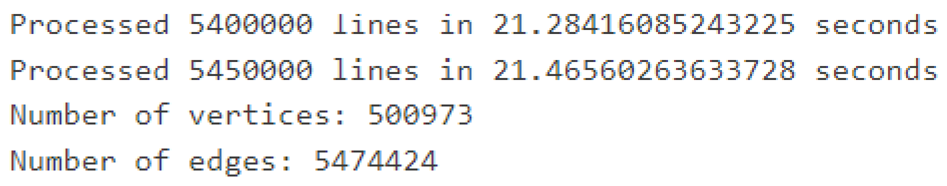
- Query of getting the top k most important stops:

- Time complexity: $O(N)$, where N is the number of stops.

Space complexity: $O(N)$, where N is the number of stops.

4.2. Result and Testing

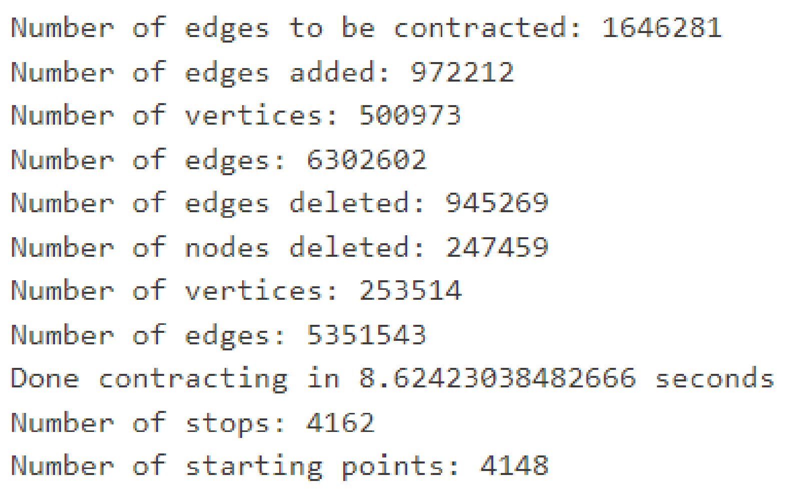
When loading the graph, we have this output:



```
Processed 5400000 lines in 21.28416085243225 seconds
Processed 5450000 lines in 21.46560263633728 seconds
Number of vertices: 500973
Number of edges: 5474424
```

So, we have $V = 500973$ and $E = 5474424$.

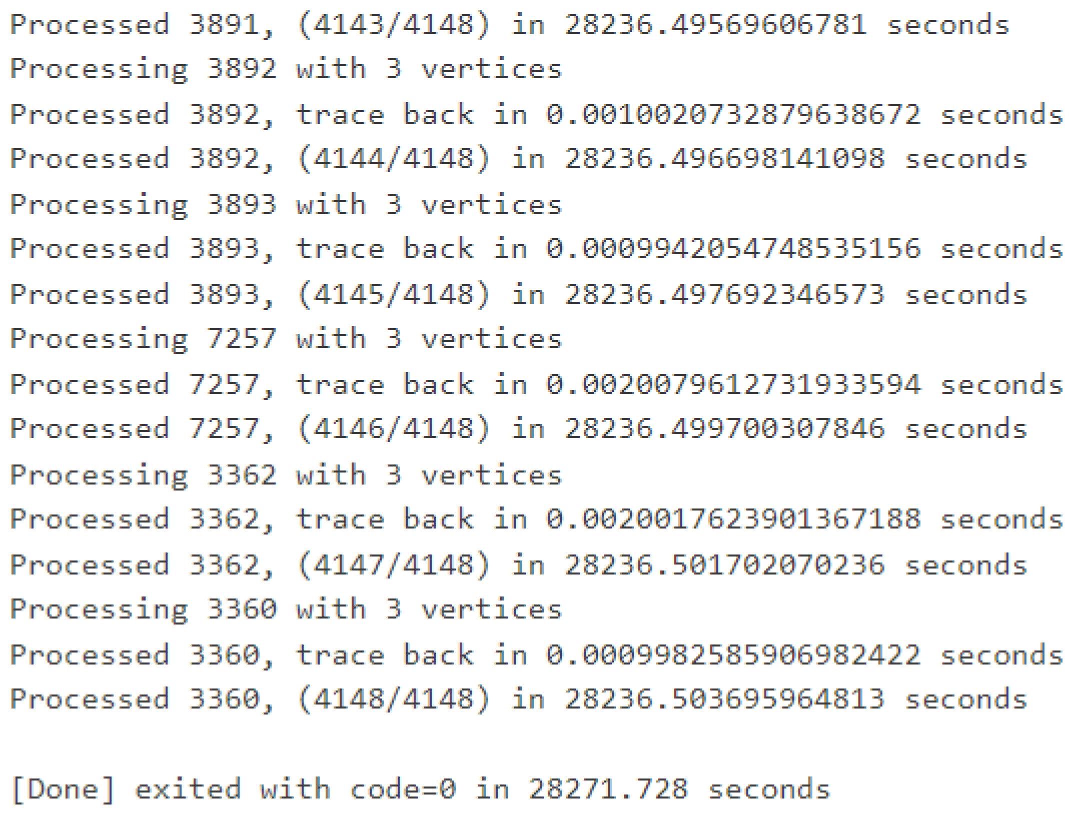
After conducting the optimization, the number of the vertices and edges have decreased, in details:



```
Number of edges to be contracted: 1646281
Number of edges added: 972212
Number of vertices: 500973
Number of edges: 6302602
Number of edges deleted: 945269
Number of nodes deleted: 247459
Number of vertices: 253514
Number of edges: 5351543
Done contracting in 8.62423038482666 seconds
Number of stops: 4162
Number of starting points: 4148
```

Since V and E is significantly large (after contraction, $V = 253514$ and $E = 5351543$), the running time for the Dijkstra algorithm is very huge and this took me approximately 8 hours of running this program to find the top k most important bus stop.

This is the real-life running time of the Dijkstra algorithm. You can see the full output at my result file (`output.txt`) in my GitHub link at [5] or in the collection of result files in [6].



```
Processed 3891, (4143/4148) in 28236.49569606781 seconds
Processing 3892 with 3 vertices
Processed 3892, trace back in 0.0010020732879638672 seconds
Processed 3892, (4144/4148) in 28236.496698141098 seconds
Processing 3893 with 3 vertices
Processed 3893, trace back in 0.0009942054748535156 seconds
Processed 3893, (4145/4148) in 28236.497692346573 seconds
Processing 7257 with 3 vertices
Processed 7257, trace back in 0.0020079612731933594 seconds
Processed 7257, (4146/4148) in 28236.499700307846 seconds
Processing 3362 with 3 vertices
Processed 3362, trace back in 0.0020017623901367188 seconds
Processed 3362, (4147/4148) in 28236.501702070236 seconds
Processing 3360 with 3 vertices
Processed 3360, trace back in 0.0009982585906982422 seconds
Processed 3360, (4148/4148) in 28236.503695964813 seconds

[Done] exited with code=0 in 28271.728 seconds
```

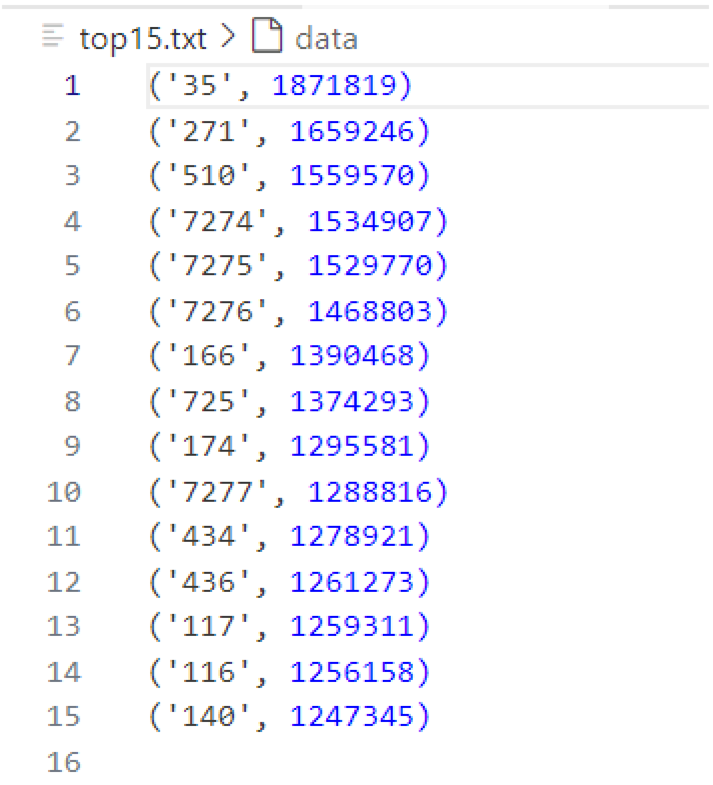
Testing for top k ($k = 15$) most important bus stops:

```
# driver code
```

```
graph = Graph("graph.json", "", mode="load")
print("Number of vertices:", graph.getLenVertices())
print("Number of edges:", graph.getLenEdges())
# query phase
print(graph.findTopK(15))


sys.exit(0)
```

And this is the result:



top15.txt > data

```
('35', 1871819)
('271', 1659246)
('510', 1559570)
('7274', 1534907)
('7275', 1529770)
('7276', 1468803)
('166', 1390468)
('725', 1374293)
('174', 1295581)
('7277', 1288816)
('434', 1278921)
('436', 1261273)
('117', 1259311)
('116', 1256158)
('140', 1247345)
```

The first column is the `stop_id`, and the second column is the importance (or the frequency of the corresponding `stop_id`). You can visit my GitHub link at [5] or the Result Files collections at [6] for more information.

5. Reference

These are the website that I have looked for information about JSON files and algorithms, including:

[1]. Shortest Path Algorithm Tutorial with Problems - GeeksforGeeks

[2]. Find Shortest Paths from Source to all Vertices using Dijkstra’s Algorithm (geeksforgeeks.org)

[3]. On The Optimization of Dijkstra’s Algorithm - Seifedine Kadry, Ayman Abdallah, Chibli Joumaa

[4]. Loading And Parsing JSON In Python - ExpertBeacon

This is my GitHub repository link:

[5]. GitHub Source Code for CS163 Solo Project (github.com)

And this is my Google drive link for the result files:

[6]. CS163-23APCS2-Solo-23125028-ResultFiles