

# BFS, Dijkstra's Shortest Path Algorithm & Betweenness Centrality Implementations in Application of OpenFlights Dataset

Cheng Ji, Jing Zhan, Jinzhe Cheng

Fall 2020, CS 225

## Contents

<b>1</b>	<b>Overview</b>	<b>1</b>
1.1	How we establish a graph	1
1.2	Algorithms implemented	2
<b>2</b>	<b>Traversal: BFS</b>	<b>2</b>
2.1	Usage	2
2.2	Application	2
<b>3</b>	<b>Dijkstra's Shortest Path Algorithm</b>	<b>2</b>
3.1	Usage	2
3.2	Application on openflights	3
<b>4</b>	<b>Betweenness Centrality</b>	<b>3</b>
4.1	Usage	3
4.2	Application on Openflights	3
<b>5</b>	<b>Analysis on the algorithms</b>	<b>4</b>

## 1 Overview

### 1.1 How we establish a graph

By implementing a directed graph template, we import data from the Open Flights. A graph using adjacency lists was implemented. Each vertex in the graph represents an airport, and the edge between two nodes represents the routes between two airports with weight representing distance. Algorithms could be applied to this graph to give further useful information.

## 1.2 Algorithms implemented

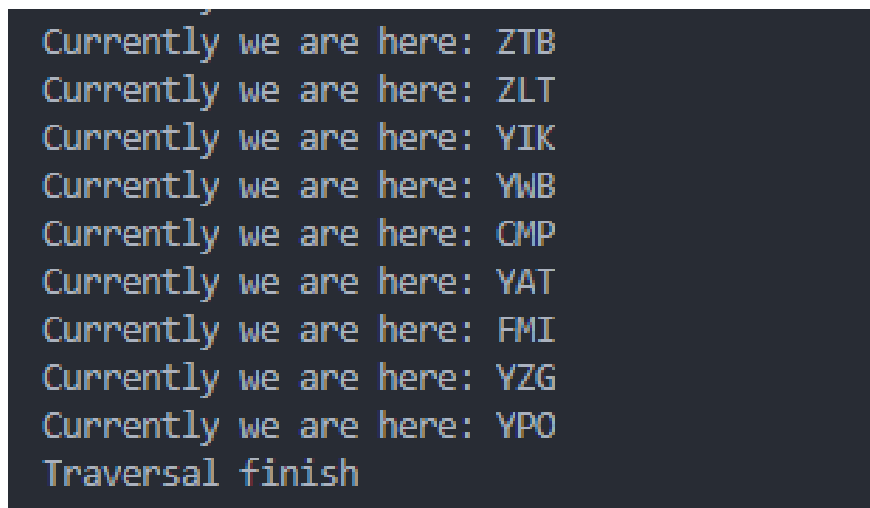
BFS is the approach to traverse, then we use Dijkstra's Algorithm to compute the shortest path and shortest path length between two airports, and we use betweenness centrality to compute the importance of an airport. Dynamic programming strategy has been applied to improve efficiency of finding shortest path from one airport to all other connected airports, which helps to improve the efficiency of betweenness centrality algorithm.

## 2 Traversal: BFS

### 2.1 Usage

Given start node, We could use BFS to traverse.

### 2.2 Application



```
Currently we are here: ZTB
Currently we are here: ZLT
Currently we are here: YIK
Currently we are here: YWB
Currently we are here: CMP
Currently we are here: YAT
Currently we are here: FMI
Currently we are here: YZG
Currently we are here: YPO
Traversal finish
```

Figure 1: Screenshots of part of results on traversal starting from LAE

## 3 Dijkstra's Shortest Path Algorithm

### 3.1 Usage

Given two vertices in the graph, we may first check whether these two are connected first to guarantee there is a path between them. Then we may perform Dijkstra's Algorithm to find the shortest path between these two. Using this algorithms, given two airports, we may find the shortest path between them, and thus give useful suggestions on how to make flight plans.

## 3.2 Application on openflights

For example, given airports LAE and YPO, we can compute the shortest path and shortest path's relative length on these two airports. Figure(Figure 1) is shown as below:

```
The shortest distance between LAE and YPO: 192.312
The shortest path accordingly:
LAE
POM
SYD
LAX
LAS
YWG
YQT
YTS
YMO
YFA
ZKE
YAT
YPO
```

Figure 2: Screenshots of results on Shortest path between LAE and YPO

## 4 Betweenness Centrality

### 4.1 Usage

Given a vertex in the graph, we may find its centrality index in the graph. This centrality number may indicate the importance of each airport. The higher the centrality is, the more importance the airport can be.

### 4.2 Application on Openflights

For example, given two airports, LAE(Nadzab Airport) and POM(Jacksons International Airport), we may find the centrality of each airports. By comparing the centrality of these two airports, it is indicated that Jacksons International Airport is more important than Nadzab Airport. Figures are shown as below:

```
The Centrality of POM: 185355
```

Figure 3: Screenshots of results on betweenness centrality of POM

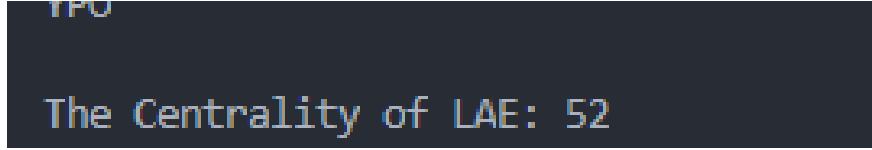


Figure 4: Screenshots of results on betweenness centrality of LAE

## 5 Analysis on the algorithms

- Dijkstra's Shortest Path:  $O(E + V \log V)$

```

1  template<typename K, typename V>
2  vector<K> Graph<K, V>::shortestPath(K v1, K v2) const {
3      if (!ifConnected(v1, v2))
4          return vector<K> {};
5
6      unordered_map<K, K> prev;
7      unordered_map<K, V> dist;
8      std::priority_queue<pair<K, V>, vector<pair<K, V>>,
9                          std::greater<pair<K, V>>> Q;
10
11     for (auto it = vertices_.begin(); it != vertices_.end(); ++it)
12         dist[it->first] = std::numeric_limits<V>::max();
13
14     dist[v1] = 0;
15     Q.push(std::make_pair(v1, 0));
16
17     while (!Q.empty()) {
18         K curr = Q.top().first;
19         Q.pop();
20
21         for (auto adj: vertices_[curr]) {
22             auto neighbor = adj.first;
23             auto edge = adj.second;
24             if (dist[curr] + edge.weight_ < dist[neighbor]) {
25                 dist[neighbor] = dist[curr] + edge.weight_;
26                 prev[neighbor] = curr;
27                 Q.push(std::make_pair(neighbor, dist[neighbor]));
28             }
29         }
30     }
31
32     vector<K> path;
33     K curr = v2;
34     while (curr != v1) {
35         path.push_back(curr);
36         curr = prev[curr];
37     }
38     path.push_back(v1);
39     std::reverse(path.begin(), path.end());
40     return path;
41 }

```

- Betweenness Centrality:  $O(VE + V^2 \log V)$

```

1 template<typename K, typename V>
2 unsigned Graph<K, V>::BetweennessCentrality(const K& v) const {
3     unsigned res = 0;
4
5     for (auto src = vertices_.begin(); src != vertices_.end(); ++src)
6     {
7         K current = src->first;
8         if (current != v) {
9             if (!ifConnected(current, v)) {
10                 continue;
11             }
12             auto currentPaths = shortestPath(current);
13             for (auto& des : currentPaths) {
14                 if (des.first != v) {
15                     std::unordered_set<K> path(des.second.begin(),
16 des.second.end());
17                     if (path.find(v) != path.end()) {
18                         ++res;
19                     }
20                 }
21             }
22         }
23     }
24     return res;
25 }

```