
UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Covid-19 Analysis

ADVANCED ALGORITHMS AND GRAPH MINING

*Professors:*
Marino Andrea
Nocentini Massimo

*Students:*
Chaudhry Abdullah
Raffaelli Claudia

AY 2019 - 2020

# Contents


1 Abstract 1

2 Introduction 2

3 Adding edges 3

4 Shortest paths 3

5 Bellman-Ford algorithm 4
- 5.1 First intuition . . . 4
- 5.2 Toy Example . . . 5
- 5.3 Our First Implementation . . . 5
  - 5.3.1 Computation time . . . 6
- 5.4 New strategy . . . 6
- 5.5 Our Second Implementation . . . 6
  - 5.5.1 Computation time . . . 7
- 5.6 Putting all together . . . 8

6 Centrality 9

7 Betweenness Centrality 9
- 7.1 First intuition . . . 9
- 7.2 Toy example . . . 10
- 7.3 Betweenness centrality in large graphs . . . 10
- 7.4 A handy benefit to betwenness centrality . . . 10
- 7.5 Some common algorithms . . . 11
- 7.6 Our implementation . . . 11

8 Implementation decisions making 12

Bibliography 12


# 1 Abstract


These slides describe a technique to calculate the Betweenness Centrality of a graph's nodes using Bellman-Ford's algorithm


[1]:
```
__AUTHORS__ = {'ac': ("Abdullah Chaudhry",
                      "abdullah.chaudhry@stud.unifi.it",
                      "https://github.com/chabdullah"),
               'cr': ("Claudia Raffaelli",
                      "claudia.raffaelli@stud.unifi.it",
                      "https://github.com/ClaudiaRaffaelli",)}

__KEYWORDS__ = ['Python', 'Jupyter', 'notebooks', 'Bellman-Ford',␣
 ↪'Betweenness-Centrality', 'graphs']
```

## 2 Introduction

During the next slides we will see how to:

- Handle large graphs
- Implement two variations of the Bellman-Ford algorithm
- Implement the Betweenness Centrality algorithm

We start by loading the project:

[2]:
```
from graph_manager import GraphManager
import networkx as nx
import json
import math
import random
import time
from collections import deque
import matplotlib as plt
```

[3]:
```
with open("./dati-json/dpc-covid19-ita-province.json") as f:
    parsed_file = json.load(f)

# Building the graph of provinces
provinces_already_annotated = []
P = GraphManager()
for province_data in parsed_file:
    # extracting information from the JSON
    if province_data['sigla_provincia'] != '' and province_data[
        'sigla_provincia'] not in provinces_already_annotated:
        provinces_already_annotated.append(province_data['sigla_provincia'])
        province = province_data['denominazione_provincia']
        position_x = province_data['long']
        position_y = province_data['lat']
        # adding each province to the graph
        P.add_node_to_graph(province, position_x, position_y)

# Building the graph of doubles
R = GraphManager()
# Generate 2000 pairs of double (x,y)
for i in range(2000):
    x = round(random.uniform(30, 50), 1)
    y = round(random.uniform(10, 20), 1)
    R.add_node_to_graph(str(i), x, y)
```

## 3 Adding edges

While handling large graphs, heavy operations like adding edges to the graph itself can be challenging. Here, we see our implementatin of this task.

We keep in mind that two nodes a and b are connected by an edge if the following holds:

- if x,y is the position of a, then b is in position z,w with z in [x-d,x+d] and w in [y-d, y+d], with d=0.8.

Here we see the main steps:

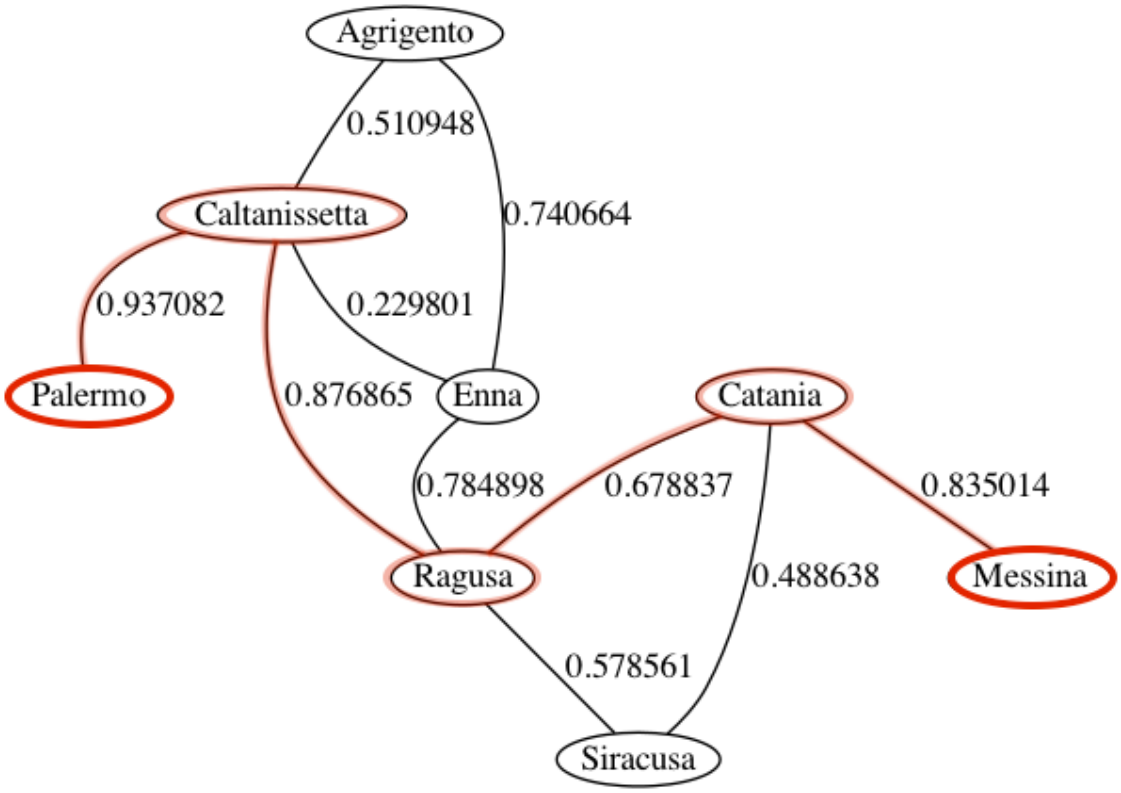
- Is created a list of pairs with this structure: (node_name, (x, y)),
- This list is sorted by the x value,
- We iterate over the sorted list with an index i,
- We also keep an index j that can help us compare the node i to this other node,
- If the nodes indexed by i and j are close enough relatively to x and y, then we add an edge, with weight the Eucledian distance between the two,
- If the node i and j are close only with respect to x, we must check the node i with j+1, since the list is only ordered wrt x, and not y,
- If the node indexed by i and the node indexed by j are not close enough relatively to x, of course i and j+1 will also not be close wrt x because of the sorting. In this case we increment i and set j=i+1.

[4]:
```
# Inserting the edges according to the distance between each node
%timeit P.add_edges()
%timeit R.add_edges()
```

```
2.26 ms ± 21.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
327 ms ± 4.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

## 4 Shortest paths

In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.




Applications:

- Shortest path algorithms are applied to automatically find directions between physical locations, such as driving directions on web mapping websites like Google Maps
- Considering a nondeterministic abstract machine as a graph where vertices describe states and edges describe possible transitions, shortest path algorithms can be used to find an optimal sequence of choices to reach a certain goal state

## 5 Bellman-Ford algorithm

The Bellman–Ford algorithm [2] is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a **weighted digraph**. It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.

Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm. If a graph contains a **"negative cycle"** (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no cheapest path: any path that has a point on the negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect and report the negative cycle.

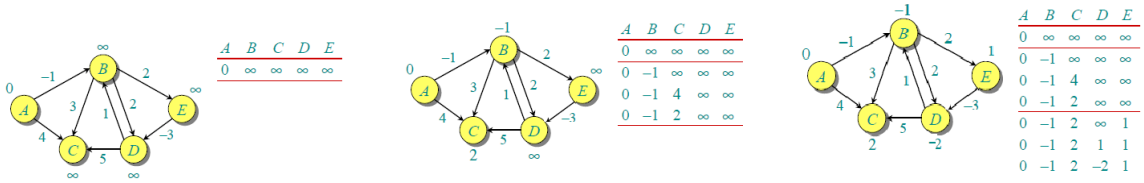
### 5.1 First intuition

Like Dijkstra's algorithm, Bellman–Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution. In both al-

gorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value and the length of a newly found path.

Bellman–Ford algorithm simply relaxes all the edges, $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances.

## 5.2 Toy Example




- **Iteration 0:** all distances are initialized from source vertex A,
- **Iteration 1:** all edges are processed in the order (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). Found all shortest paths which are at most 1 edge long,
- **Iteration 2:** processing again all edges. All shortest paths which are at most 2 edges long are found,
- **Iteration 3 and 4:** useless.

## 5.3 Our First Implementation

```
def bellman_ford(self, source_vertex):
    vertices = list(self.graph.nodes())

    distances = dict.fromkeys(self.graph.nodes(), math.inf)
    predecessors = dict.fromkeys(self.graph.nodes(), None)
    distances[source_vertex] = 0
    # relax edges
    count = len(vertices) - 1
    while count > 0:
        something_has_changed = False
        for (u, v) in self.graph.edges():
            # considering both the symmetric edges in the form (u, v) and (v, u)
            if distances[u] + float(self.graph[u][v]['label']) < distances[v]:
                distances[v] = distances[u] + float(self.graph[u][v]['label'])
                predecessors[v] = u
                something_has_changed = True
            if distances[v] + float(self.graph[v][u]['label']) < distances[u]:
                distances[u] = distances[v] + float(self.graph[v][u]['label'])
                predecessors[u] = v
                something_has_changed = True
        if something_has_changed is False:
            break
        count -= 1
    return distances, predecessors
```

### 5.3.1 Computation time

Bellman–Ford runs in $O(|V| \cdot |E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges respectively at its worst case.

The Bellman–Ford algorithm may be improved in practice (although not in the worst case), as we did, by the observation that, if an iteration of the main loop of the algorithm terminates without making any changes, the algorithm can be immediately terminated, as subsequent iterations will not make any more changes. With this early termination condition, the main loop may in some cases use many fewer than $|V| - 1$ iterations, even though the worst case of the algorithm remains unchanged.

## 5.4 New strategy

The **Shortest Path Faster Algorithm (SPFA)**[6] is an improvement of the Bellman–Ford algorithm which computes single-source shortest paths in a weighted directed graph. The algorithm is believed to work well on random sparse graphs and is particularly suitable for graphs that contain negative-weight edges. However, the worst-case complexity of SPFA is the same as that of Bellman–Ford, so for graphs with nonnegative edge weights Dijkstra's algorithm is still preferred.

The basic idea of SPFA is the same as Bellman–Ford algorithm in that each vertex is used as a candidate to relax its adjacent vertices. The improvement over the latter is that instead of trying all vertices blindly, SPFA maintains a **queue** of candidate vertices and adds a vertex to the queue only if that vertex is relaxed. This process repeats until no more vertex can be relaxed.

## 5.5 Our Second Implementation

```
def bellman_ford_SPFA(self, source_vertex):
    distances = dict.fromkeys(self.graph.nodes(), math.inf)
    already_in_queue = dict.fromkeys(self.graph.nodes(), False)
    predecessors = dict.fromkeys(self.graph.nodes(), math.inf)
    distances[source_vertex] = 0

    q = deque()
    q.append(source_vertex)
    already_in_queue[source_vertex] = True
    while len(q) > 0:
        u = q.popleft()
        for (u, v) in self.graph.edges(u):
            if distances[u] + float(self.graph[u][v]['label']) < distances[v]:
                distances[v] = distances[u] + float(self.graph[u][v]['label'])
                if not already_in_queue[v]:
                    q.append(v)
                    already_in_queue[v] = True
                predecessors[v] = u
            if distances[v] + float(self.graph[v][u]['label']) < distances[u]:
                distances[u] = distances[v] + float(self.graph[v][u]['label'])
                if not already_in_queue[u]:
                    q.append(u)
```

```
                    already_in_queue[u] = True
                predecessors[u] = v
    return distances, predecessors
```

### 5.5.1 Computation time

The worst-case running time of the algorithm is $O(|V| \cdot |E|)$ , just like the standard Bellman-Ford algorithm. However experiments suggest that the average running time is $O(|E|)$.

We can now compare the two strategies.

[5]:
```
%timeit P.bellman_ford("Firenze")
%timeit P.bellman_ford_SPFA("Firenze")
```

```
6.51 ms ± 61.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
2.51 ms ± 19.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[6]:
```
distances, predecessors = P.bellman_ford_SPFA("Firenze")
```

[7]:
```
distances
```

[7]:
```
{'Chieti': 3.470136,
 "L'Aquila": 2.701029,
 'Pescara': 3.59269,
 'Teramo': 3.134951,
 'Matera': 7.103830000000001,
 'Potenza': 7.232769000000001,
 'Bolzano': 2.907397,
 'Catanzaro': inf,
 'Cosenza': inf,
 'Crotone': inf,
...}
```

[8]:
```
predecessors
```

[8]:
```
{'Chieti': "L'Aquila",
 "L'Aquila": 'Terni',
 'Pescara': 'Chieti',
 'Teramo': "L'Aquila",
 'Matera': 'Barletta-Andria-Trani',
 'Potenza': 'Barletta-Andria-Trani',
 'Bolzano': 'Trento',
 'Catanzaro': inf,
 'Cosenza': inf,
 'Crotone': inf,
 'Reggio di Calabria': inf,
 ...}
```

### 5.6 Putting all together

```
def bellman_ford_shortest_path(self, source_vertex, SPFA=True):
    if SPFA:
        distances, predecessors = self.bellman_ford_SPFA(source_vertex)
    else:
        distances, predecessors = self.bellman_ford(source_vertex)
    all__shortest_paths = []
    nodes = list(self.graph.nodes(data=True))
    for target_vertex in nodes:
        target_vertex = target_vertex[0]
        if predecessors[target_vertex] == math.inf:
            continue
        # using the predecessors of each node to build the shortest path
        shortest_path = []
        current_node = target_vertex
        shortest_path.append(target_vertex)
        while current_node != source_vertex:
            current_node = predecessors[current_node]
            # no path between the two nodes: exiting from the loop
            if current_node is None:
                break
            shortest_path.append(current_node)
        if len(shortest_path) != 1:
            all__shortest_paths.append(shortest_path)
    return all__shortest_paths
```

The output obtained by this computation:

[9]:
```
P.bellman_ford_shortest_path("Firenze", SPFA=True)[:4]
```

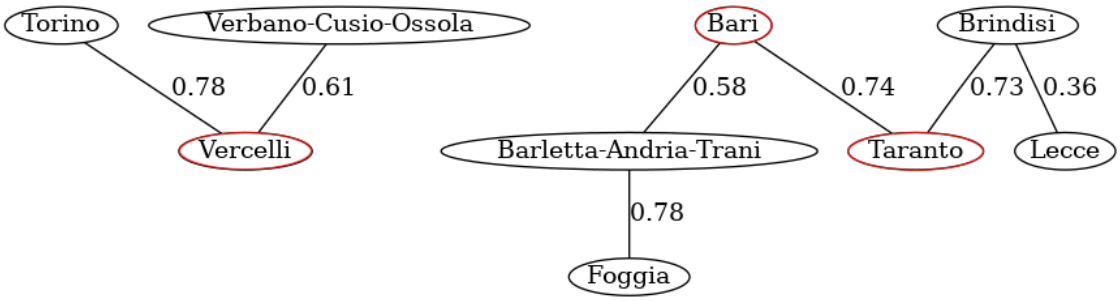
[9]:
```
[['Chieti', "L'Aquila", 'Terni', 'Perugia', 'Arezzo', 'Firenze'],
 ["L'Aquila", 'Terni', 'Perugia', 'Arezzo', 'Firenze'],
 ['Pescara', 'Chieti', "L'Aquila", 'Terni', 'Perugia', 'Arezzo', 'Firenze'],
 ['Teramo', "L'Aquila", 'Terni', 'Perugia', 'Arezzo', 'Firenze'],
 ['Matera',
  'Barletta-Andria-Trani',
  'Foggia',
  'Benevento',
  'Campobasso',
  'Chieti',
  "L'Aquila",
  'Terni',
  'Perugia',
  'Arezzo']]
```

[10]:
```
%timeit P.bellman_ford_shortest_path("Firenze", SPFA=True)
```

```
2.58 ms ± 30.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

# 6 Centrality

In network analysis, indicators of **centrality** identify the most important vertices within a graph.




Applications:

- Identifying the most influential person(s) in a social network
- Key infrastructure nodes in the Internet
- Super-spreaders of disease.

# 7 Betweenness Centrality

## 7.1 First intuition

It was introduced as a measure for quantifying the control of a human on the communication between other humans in a social network by *Linton Freeman*. In his conception, vertices that have a high probability to occur on a randomly chosen **shortest path** between two randomly chosen vertices have a high betweenness.

Betweenness centrality [3], in Graph Theory, is a measure of centrality in a graph based on shortest paths. Vertices with high betweenness may have considerable influence within a network by virtue of their control over information passing between others. They are also the ones whose removal from the network will most disrupt communications between other vertices because they lie on the largest number of paths taken by messages.

More compactly the betweenness can be represented as:

$$g(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

where $\sigma_{st}$ is total number of shortest paths from node $s$ to node $t$ and $\sigma_{st}(v)$ is the number of those paths that pass through $v$.

### 7.2 Toy example

Betweenness Centrality for A = ?

BC = 0/1 = 0

BD = 0/1 = 0

**BE = 1/1 = 1**

BF = 0/1 = 0

CD = 0/1 = 0

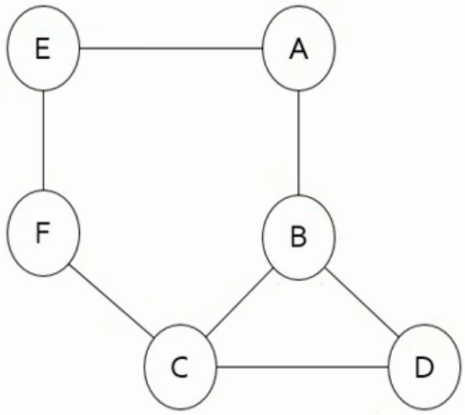
CE = 0/1 = 0

CF = 0/1 = 0

**DE = 1/2 = 0.5**

DF = 0/1 = 0

EF = 0/1 = 0



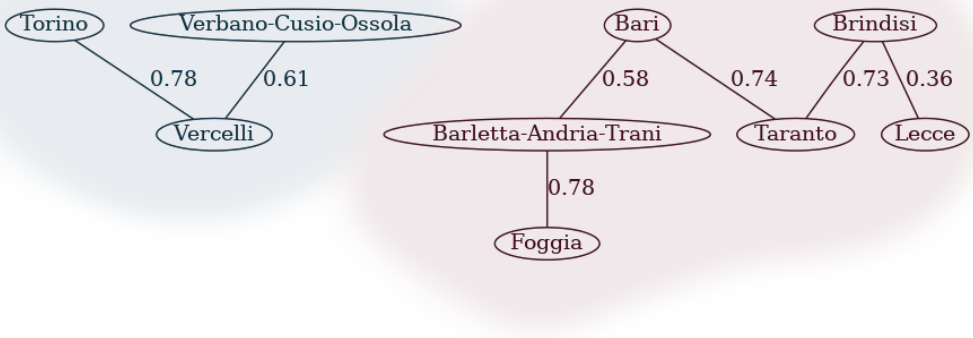

Betweenness Centrality for A = **1.5**

### 7.3 Betweenness centrality in large graphs

The betweenness centrality of a node scales with the number of pairs of nodes. Therefore, the calculation may be rescaled by dividing through by the number of pairs of nodes not including $v$, so that $g(v) \in [0,1]$. The division is done by $(N1)(N2)$ for directed graphs and $\frac{(N1)(N2)}{2}$ for undirected graphs, where $N$ is the number of nodes.

### 7.4 A handy benefit to betwenness centrality

We don't need a (fully) connected graph to calculate it



[12]:
```
print(P_toy.betweenness_centrality())
```

```
{'Torino': 0.0, 'Verbano-Cusio-Ossola': 0.0, 'Vercelli': 0.03571428571428571,
'Bari': 0.21428571428571427, 'Barletta-Andria-Trani': 0.14285714285714285,
'Brindisi': 0.14285714285714285, 'Foggia': 0.0, 'Lecce': 0.0, 'Taranto':
0.21428571428571427}
```

### 7.5 Some common algorithms

Calculating the betweenness centrality of all the vertices in a graph involves calculating the shortest paths between all pairs of vertices on a graph, which takes:

- $\theta(|V|^3)$ for weighted graphs *(Floyd–Warshall algorithm [4])*.
  $O(|V|^2 log|V| + |V||E|)$ on sparse graphs *(Johnson's algorithm [5] or Brandes' algorithm [1])*
- $O(|V||E|)$ for unweighted graphs *(Brandes' algorithm)*.

A single execution of the algorithms will find the shortest paths between all pairs of vertices. In the last case (Brandes' algorithm) it also calculate the betweenness value for each vertex.

### 7.6 Our implementation

Here is a snippet of our implementation which calculates the betwenness value for each node in $O(|V|^3|E|)$. We used *Bellman-Ford's alghoritm* to find the shortest paths from a source node $i$ to $v$, with $v \in V$

```
def betweenness_centrality(self, SPFA=True):
    nodes = list(self.graph.nodes(data=True))
    N = len(nodes)
    BC = {nodes[i][0]: 0 for i in range(N)}
    for i in range(N):
        paths_lists = self.bellman_ford_shortest_path(nodes[i][0], SPFA=SPFA)
        for path in paths_lists:
            for node in path[1:-1]:
                BC[node] += 1
    # Normalize
    for i in BC:
        BC[i] /= (N - 1) * (N - 2)
    return BC
```

[13]:
```
%timeit P.betweenness_centrality()
%timeit nx.betweenness_centrality(P.graph, weight='label', normalized=True)
%timeit R.betweenness_centrality()
%timeit nx.betweenness_centrality(R.graph, weight='label', normalized=True)
```

```
241 ms ± 3.18 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
67.4 ms ± 257 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
4min 49s ± 5.22 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
1min 9s ± 615 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

## 8 Implementation decisions making

- In our implementation we used the Bellman-Ford algorithm (which is not good for this kind of problem) to calculate the shortest path between two nodes. This forced us to calculate the shortes path in both directions, therefore we needed to divide by 2 the result of the betweenness of the considered node. We have calculated all the shortest paths in the graph in $O(|V|^2|E|)$ while Johnson's algorithm takes $O(|V|^2 log|V| + |V||E|)$
- Due to the great precision of the distances between the various nodes (graph **P** and **R**) and for the definition of the Bellman-Ford's algorithm itself, we thought it was appropriate to simplify the algorithm assuming that there could be at most one shortest path between each pair of nodes.

[14]:
```
print("Our Implementation: ",list(P.betweenness_centrality().items())[0:3])
print("\n\nNetworkx method: ",list(nx.betweenness_centrality(P.graph,␣
 ↪weight='label').items())[0:3])
```

```
Our Implementation:  [('Chieti', 0.19748427672955976), ("L'Aquila",
0.06037735849056604), ('Pescara', 0.005750224618149146)]


Networkx method:  [('Chieti', 0.19748427672955976), ("L'Aquila",
0.06037735849056604), ('Pescara', 0.005750224618149146)]
```

- Due to the large number of nodes we normalized the value of the betweenness so that $g(v) \in [0,1]$. The division is done by $\frac{(N1)(N2)}{2}$ cause we are working with an undirected graph.
- Based on the considerations made, the final formula for calculating the betweenness centrality of a node $v$ is
  $g(v) = \frac{\sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}}{2} \frac{2}{(N-1)(N-2)} = \frac{\sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}}{(N-1)(N-2)}$
  where $\sigma_{st}(v), \sigma_{st} \in \{0,1\}$.

## Bibliography


[1] Ulrik Brandes. "A Faster Algorithm for Betweenness Centrality". In: *Journal of Mathematical Sociology*. 25.2 (2001), pp. 163–177.

[2] Wikipedia. *Bellman-Ford algorithm*. URL: https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm.

[3] Wikipedia. *Betweenness centrality*. URL: https://en.wikipedia.org/wiki/Betweenness_centrality.

[4] Wikipedia. *Floyd–Warshall algorithm*. URL: https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm.

[5] Wikipedia. *Johnson's algorithm*. URL: https://en.wikipedia.org/wiki/Johnson%27s_algorithm.

[6] Wikipedia. *Shortest Path Faster Algorithm*. URL: https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm.