University of Studies of Florence

Department of Engineering Information

Pistoia, July 15, 2020

Advanced Algorithm and Graph Mining Exam

An implementation of Floyd-Warshal Algorithm and the Clustering coefficent on Italian Provinces Graph

In [3]:
```
__AUTHOR__ = {'lp': ("Lorenzo Pisaneschi",
                     "lorenzo.pisaneschi1@stud.unifi.it",
                      "https://github.com/pisalore/AAGM_exam")}

__TOPICS__ = ['Graphs algorithms', 'Data Analysis']


__KEYWORDS__ = ['Python', 'Jupyter', 'Graphs', 'NetworkX', 'pandas',]
```

Introduction

Big Data are everywhere. For this reason we must know them and know how to read and manage them in order to obtain useful information for our community. This concept is stronger during such an event as a pandemic as we are living just now with the COVID-19 caused by the new SARS-CoV-2.

Main topics

In this work I present two main topic:

- Graphs generation and graphs' algorithms application with NetworkX
- Data analysis with Pandas

Both of them are based on official Italian CIvil Protection COVID-19 data.

In the first part, Italian provinces graphs and some random dimension graphs are build. They will be used to run two algorithms, the Floyd-Warshall algorithm and one which computes graphs' clustering coefficient. In the second part, Italian COVID-19 pandemic evolution is investigated thank to Pandas, Matplotlib and Geopandas, which are very powerful Python's modules for Data Analysis.

Data format

Data of interest are the ones related to Italian Regions and Provinces. Both are available in JSON format and one completes the other: information about Provinces make Region Data analysis more exhaustive.

Algorithms and performance analysis have been ran on **Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz** using Ubuntu 19.10.

Example: Region JSON object


```
{
        "data": "2020-06-25T17:00:00",
        "stato": "ITA",
        "codice_regione": 9,
        "denominazione_regione": "Toscana",
        "lat": 43.76923077,
        "long": 11.25588885,
        "ricoverati_con_sintomi": 24,
        "terapia_intensiva": 6,
        "totale_ospedalizzati": 30,
        "isolamento_domiciliare": 294,
        "totale_positivi": 324,
        "variazione_totale_positivi": -6,
        "nuovi_positivi": 2,
        "dimessi_guariti": 8799,
        "deceduti": 1101,
        "casi_da_sospetto_diagnostico": 10107,
        "casi_da_screening": 117,
        "totale_casi": 10224,
        "tamponi": 323864,
        "casi_testati": 222294
}
```

Example: Province JSON object


```
{
        "data": "2020-06-25T17:00:00",
        "stato": "ITA",
        "codice_regione": 9,
        "denominazione_regione": "Toscana",
        "codice_provincia": 47,
        "denominazione_provincia": "Pistoia",
        "sigla_provincia": "PT",
        "lat": 43.933465,
        "long": 10.91734146,
        "totale_casi": 743
    }
```

Build the Provinces graph and random graphs

In the first part, data of interest are only the provincial ones, in particular the **longitude and latitude**. This is because the goal is to build an **undirected weighted Italian Provinces graph** starting from these two geographic coordinates. Moreover, two cities are connected if, given x (longitude) and y (latitude) for city "a", then city "b" is in position z (longitude), w (latitude) with z in [x-d,x+d] and w in [y-d, y+d], where d=0.8.

The graph is weighted: an edge E which connects the nodes (cities) "a" and "b" has a weight which is the euclidean distance between the two cities.

Then, in order to perform some computation analysis on Floyd-Warshall algorithm and to compute clustering coefficients, ten random graphs and one 2000 nodes sparser graph are build.

Build weighted undirected graphs: first strategy

The first graph build-up strategy is the expensive one. Its implementation is very easy: iterate over all the nodes computing all possible combinations and calculating all distances (weights) If the distance between two cities is less than 0.8, they are connected. Its complexity is $O(n^2)$

```
def set_provinces_edges_expensive(graph, thrs):
    for a in graph.nodes(data=True):
        for b in graph.nodes(data=True):
            if (a[1]['long'] - thrs < b[1]['long'] < a[1]['long'] + thrs) \
                and (a[1]['lat'] - thrs < b[1]['lat'] < a[1]['lat'] + thrs):
                graph.add_edge(a[0], b[0], a=a[1]['city'], b=b[1]['city'],
                weight=d(a[1]['long'], a[1]['lat'], b[1]['long'],b[1]['lat']))
```

Build weighted undirected graphs: second more efficient strategy. Idea.

It is possible to do better than $O(n^2)$. It is mandatory to do better, moreover if we have to manage larger and larger graphs, as in this case. So, **binary search** could help us, as long as we work with **sorted lists**. In this case, we can store sort cities by longitude in a data structure; in another data structure, we can memorize sort cities by latitude. Then, for each city we store those that are close, by longitude and by latitude, in two separated lists, using binary search. Finally, for each city we compute the intersection between its above mentioned special lists. This intersection will output the near cities for each one.

Build weighted undirected graphs: second more efficient strategy. Implementation and analysis.

The complexity of this procedure is $O(nlogn)$, where the sort procedure dominates, but it is a good trade-off, since our graph node lists can be very large. So, here all the steps followed in order to generate efficiently a graph.

Given a graph $G = (V, E)$,

1. $sorted_x = TimSort(V, long)$
2. $sorted_y = TimSort(V, lat)$
3. $for\ each\ city\ compute\ nearX\ and\ nearY$
4. $for\ each\ city\ compute\ A = nearX \cap nearY$

Note that "*TimSort* is an hybrid stable sorting algorithm, derived from merge sort and insertion sort, used in Python. The algorithm finds subsequences of the data that are already ordered (runs) and uses them to sort the remainder more efficiently" from [Timsort Wikipedia (https://en.wikipedia.org/wiki/Timsort)](https://en.wikipedia.org/wiki/Timsort)

Intersection is computed efficiently using Python **Sets**:
$O(min(len(list1), len(list2)))$

Random graphs

Now we have correctly build a weighted graph containing all (107) italian provinces, where weights are distances between connected provinces. Obviously, other random graphs are build up with the strategy shown above.

```
import networkx as nx
import random
def construct_random_graph(nodes_num, x_inf, x_sup, y_inf, y_sup):
    graph = nx.Graph()
    for node_id in range(nodes_num):
        graph.add_node(node_id, city=str(node_id), long=random.uniform(x_inf,
x_sup), lat=random.uniform(y_inf, y_sup))
    return graph
```

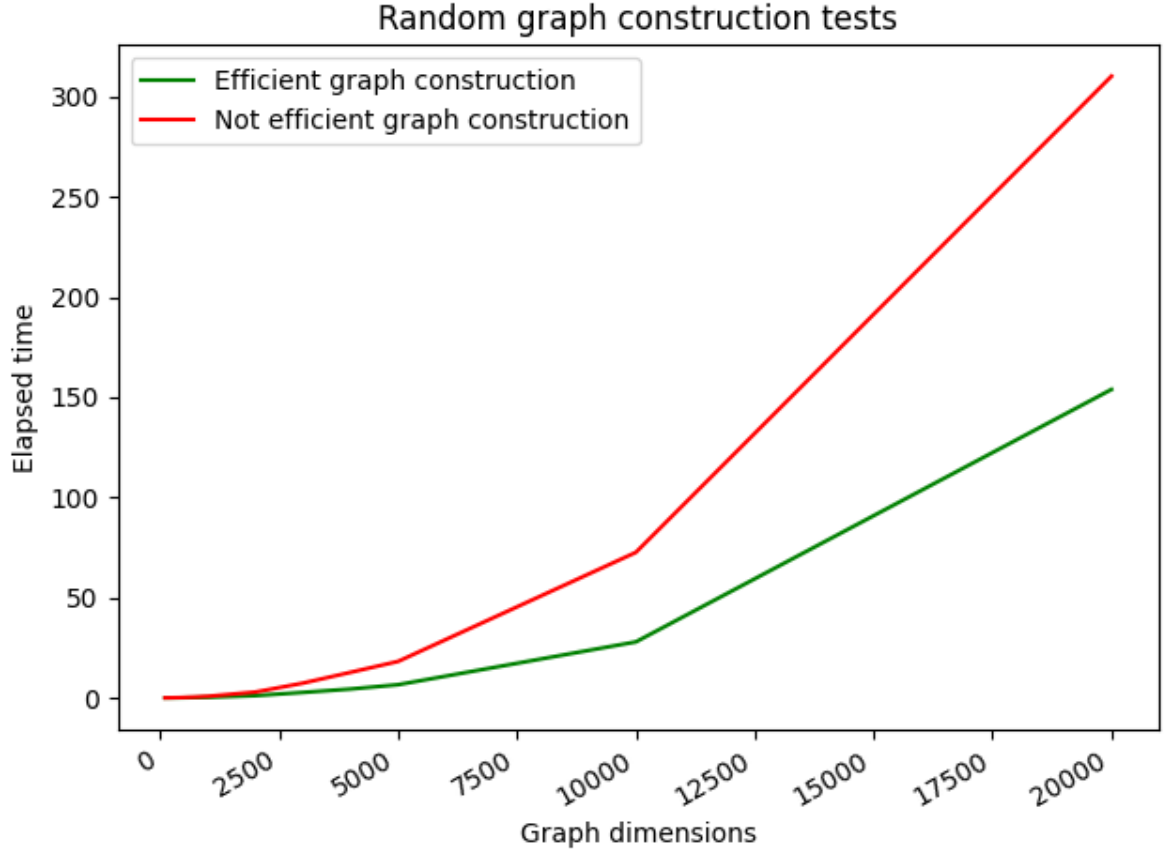
Where:

1. **nodes_num** is n
2. **(x_inf, x_sup)** is the range for random longitude
3. **(y_inf, y_sup)** is the range for random latitude

A random graph is treated as the "special" provinces graph in the edge set up procedure. In this procedure, a "threshold" parameter is indicated: it is used in defining the "closeness" criteria between nodes/cities.

Experimental graph construction test

This graph shows why the "binary search strategy" is really better then the "expensive" one.



Floyd-Warshall algorithm

Now it is possible to play with some algorithm, using the provinces and random graphs. The first algorithm here presented is the **Floyd-Warshall algorithm**. Floyd-Warshall algorithm is used for finding the shortest path between all the pairs of vertices in a weighted graph, with positive or negative edge weights (but without negative cycles).

The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with $O(|V|^3)$, even if there may be up to $\Omega(|V|^2)$ edges in the graph. Every combination is tested, incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

Floyd-Warshall algorithm. Idea. I

The algorithm basis is the **dynamic programming**. It refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. In the Floyd-Warshall algorithm is possible to use this technique, since sub-problem can be nested in recursively inside larger problems.

Floyd-Warshall algorithm. Idea. II

Consider a graph $G = (V, E)$, and a function $shortestPath(i, j, k)$ that returns the shortest possible path from from i to j using vertices only from the set $\{1, 2, .., k\}$ as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each from each i to j using any vertex in the set $\{1, 2, .., N\}$. The $shortestPath(i, j, k)$ could be:

1. a path that does not go through $\{1, 2, .., k\}$, (uses only vertices in $\{1, 2, .., k-1\}$)
2. a path that does go through $\{1, 2, .., k\}$ (from i to k and then from from k to j, both only using intermediate vertices in $\{1, 2, .., k-1\}$)

Floyd-Warshall algorithm. Idea. III

If $w(i, j)$ is the weight of the edge between vertices k and j, we can define $shortestPath(i, j, k)$ in terms of the following recursive formula: the base case is

- $shortestPath(i, j, 0) = w(i, j)$

and the recursive case is

- $shortestPath(i, j, k) = min(shortestPath(i, j, k - 1),$
  $shortestPath(i, k, k - 1) + shortestPath(k, j, k - 1))$

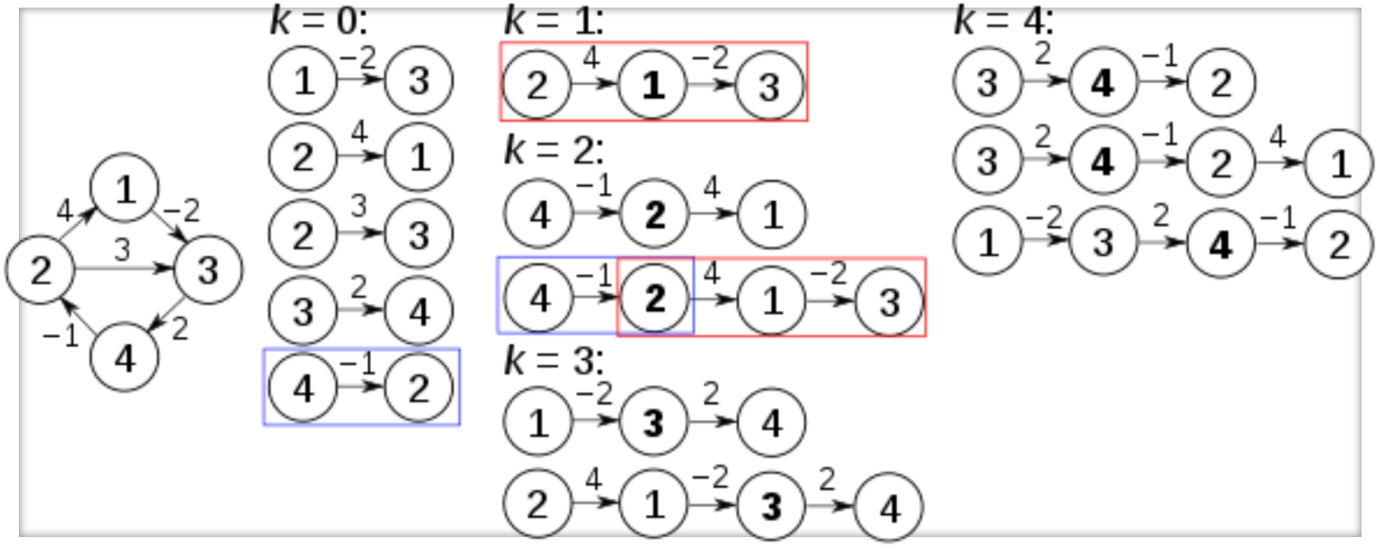
This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing $shortestPath(i, j, k)$ for all (i, j), pairs for $k = 1, 2, \ldots, N$ finding all the shortest paths.

Floyd-Warshall algorithm. Pseudo-code

```
def floyd_warshall(graph):
    n = graph.number_of_nodes()
    sparse_adj = nx.adjacency_matrix(graph, nodelist=None, weight='weight')
    shortest_paths = sparse_adj.toarray()
    for i in range(n):
        for j in range(n):
            if i != j and shortest_paths[i][j] == 0:
                shortest_paths[i][j] = INFINITY
    # Core algorithm
    for k in range(n):
        for i in range(n):
            for j in range(n):
                shortest_paths[i][j] = min(shortest_paths[i][j],
                shortest_paths[i][k] + shortest_paths[k][j])
    print(shortest_paths, '\n')
    return shortest_paths
```

Floyd-Warshall algorithm. Toy example

This example is taken from: [Wikipedia Floyd-Warshall Algorithm (https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm)](https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm)



$k=0$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	3	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

$k=1$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	**2**	∞
	3	∞	∞	0	2
	4	∞	−1	∞	0

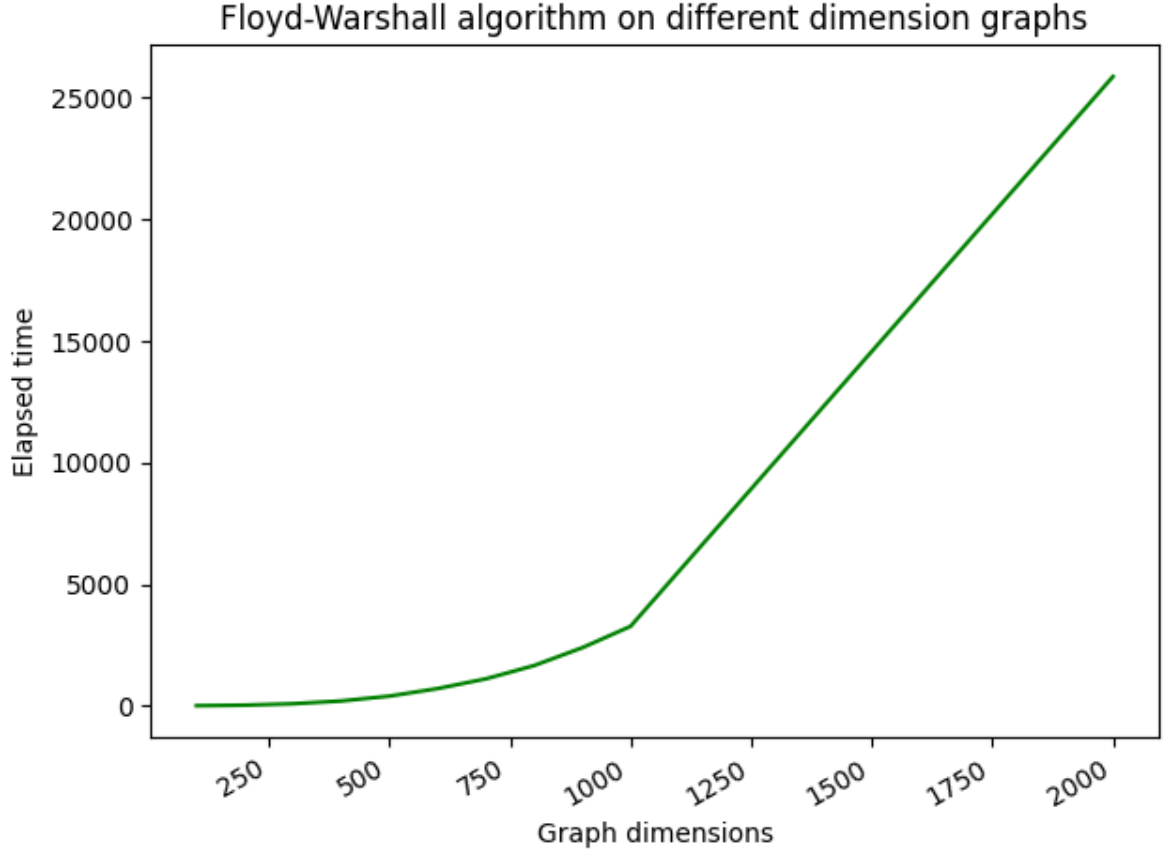
$k=2$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	∞
	2	4	0	2	∞
	3	∞	∞	0	2
	4	**3**	−1	**1**	0

$k=3$		j			
		1	**2**	**3**	**4**
i	**1**	0	∞	−2	**0**
	2	4	0	2	**4**
	3	∞	∞	0	2
	4	3	−1	1	0

$k=4$		j			
		1	**2**	**3**	**4**
i	**1**	0	**−1**	−2	0
	2	4	0	2	4
	3	**5**	**1**	0	2
	4	3	−1	1	0

Floyd-Warshall algorithm. Execution times.

Graph dimensions	100	200	300	400	500	600	700	800	900	1000	2000
Elapsed times	3.0552s	25.067s	83.724s	196.82s	396.74s	707.23s	1107.1s	1653.5s	2388.5s	3267.2s	25878s



Clustering coefficient

In graph theory, the **clustering coefficient** measures how much nodes tends to be connected between them. In real world graph, it is evident that nodes tend to form highly connected clusters, with an high number of links.

There are two different clustering coefficients:

1. **Local clustering coefficient**
2. **Global clustering coefficient**

Local clustering coefficient

The local clustering coefficient of a node is a metric which measures how much node's neighbours tend to form a clique. Given a graph $G = (V, E)$

- e_{ij} is a connection between nodes $i, j \in V$
- $N_i = \left\{ v_j : \langle e_{ij}, e_{ji} \rangle \in E^2 \right\}$
- $k_i = |N_i|$

Then, if the graph is oriented, the clustering coefficient if

- $C_i = \frac{\{|e_{jk} : v_j, v_k \in N_i e_{jk} \in E|\}}{k_i(k_i - 1)}$

Else, if G is not oriented

- $C_i = 2 * \frac{\{|e_{jk} : v_j, v_k \in N_i e_{jk} \in E|\}}{k_i(k_i - 1)}$

Global clustering coefficient

The global clustering coefficient concept is based on triples. A **triple** consists in three nodes connected with two (open triple) or three (closed triple) links. A triple is related to a node. A **triangle** consists in three closed triples, one for each triangle's node.

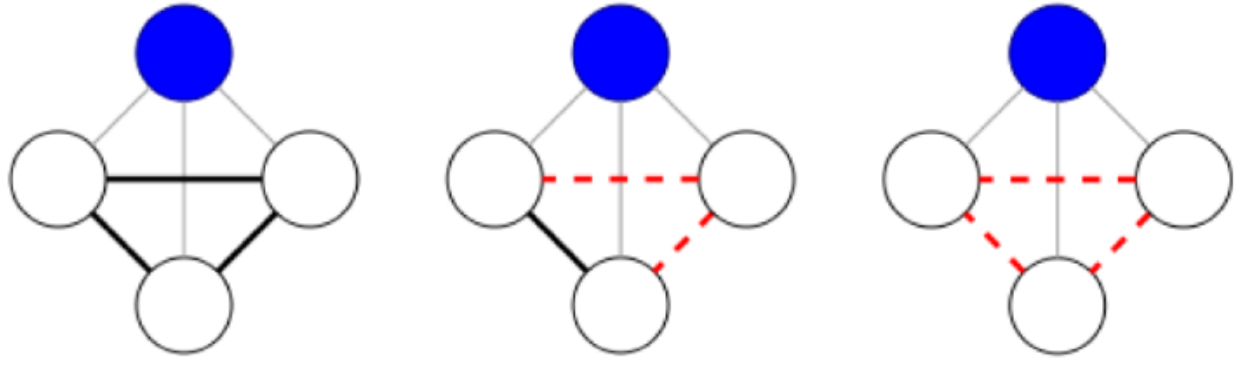
Global clustering coefficient is the closed triples to total triples (closed and open) ratio. It can be viewed also as the weighted average of all the local clustering coefficients for a network.

$$C_i = \frac{3*n_\Delta(G)}{n_\wedge(G)} = \frac{\sum_{i=1}^{n}(C_i*w_i)}{\sum_{i=1}^{n} w_i}$$

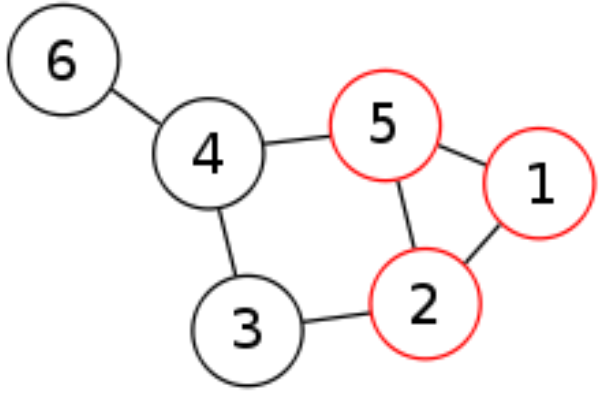
Where:

- $n_\Delta(G)$ are the triangles in G
- $n_\wedge(G)$ are total triples in G
- w_i is the weight for node v_i (number of triples where v_i is *central*)

Toy example (Local clustering coefficient).



Toy example (Global clustering coefficient).



Node	Neighbours links	Weight	Triples where node is central
1	1	1	⟨2,1,5⟩
2	1/3	3	⟨1,2,3⟩, ⟨1,2,5⟩, ⟨3,2,5⟩
3	0	1	⟨2,3,4⟩
4	0	3	⟨3,4,5⟩, ⟨3,4,6⟩, ⟨5,4,6⟩
5	1/3	3	⟨1,5,2⟩, ⟨1,5,4⟩, ⟨2,5,4⟩
6	0	0	-

$C_i = \frac{3}{11}$

Source: [Wikipedia Clustering Coefficient for Graphs (https://it.wikipedia.org/wiki/Coefficiente_di_clustering)]

Thanks for your attention!