ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

Dataset report

School Donation Dataset

DATA & AI - Graphs and mining

BEN SOUNA Nacima

BENJIRA Wissal

HAOUAS Lies

FADILI Yanis

Professors

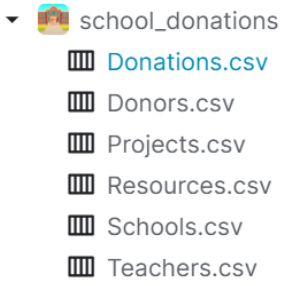
TRAVERS Nicolas

OULED DLALA Imen

A5 - DIA2

1. Introduction

The 4GB dataset contains 6 csv files :



First of all, we started by creating the whole relational graph below in order to have a general idea of the relationships between each csv file..

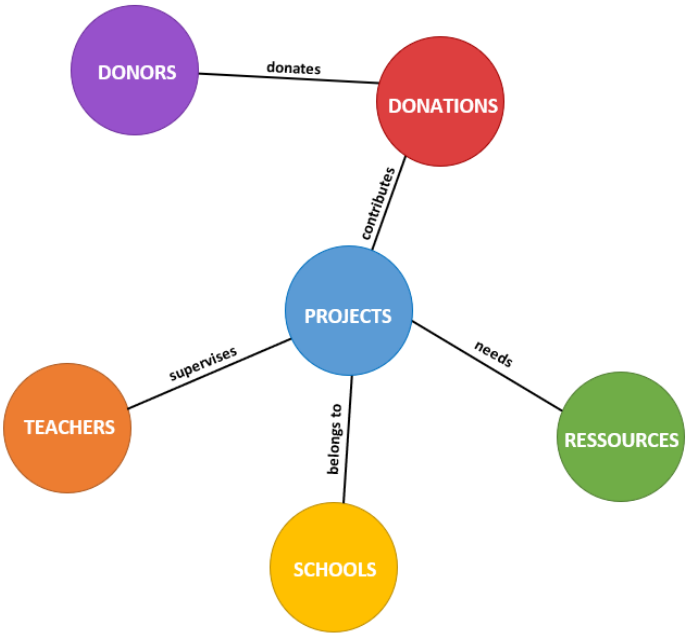


Figure 1:

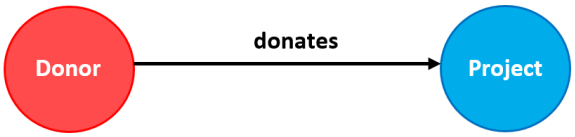
After downloading the files and trying to import them, we figured out that they were too voluminous to be used on a student laptop. Therefore, we decided to do some data pre processing through the following python code.

2. Bi-partite graph

2.1. Model creation

2.1.1. Description

First of all, in order to find similar nodes in the network and help determine the closeness of a pair of nodes , we have to use a bipartite graph. So, we extracted from the graph (figure 1) the bipartite graph.



We chose to use the donor nodes and the project nodes because they are linked by the donates relationship. To explain in detail our choice, we saw in the donations file that we have the Project ID and the Donor ID for each donation. Thus, we can create a relationship between Project node and Donor node, and this is our first bi-partite graph.

Donations.csv

Project ID	Donation ID	Donor ID
000009891526c0ade7180f8423792063	68872912085866622120 8529ee3fc18e	1f4b5b6e68445c6c4a05 09b3aca93f38

2.1.2. Data pre-processing

Firstly, we need 3 csv files to obtain, this graph:

- *donors.csv*
- *donations.csv*
- *projects.csv*

Secondly, before doing anything, we need to do some data processing, like data cleaning. We found that some columns of files contain some null values.

```
Entrée [12]: df_project.isnull().sum()
```

```
Out[12]: Project ID                              0
         School ID                               0
         Teacher ID                              0
         Teacher Project Posted Sequence         0
         Project Type                            0
         Project Title                           6
         Project Essay                           1
         Project Short Description               3
         Project Need Statement                  3
         Project Subject Category Tree          29
         Project Subject Subcategory Tree       29
         Project Grade Level Category            0
         Project Resource Category              36
         Project Cost                            0
         Project Posted Date                     0
         Project Expiration Date                14
         Project Current Status                  0
         Project Fully Funded Date          283253
```

We created a function *cleanDf() that takes the data frame that needs to be cleaned*, the function deletes rows which contain null values, so we will not have a problem when we import the data on Neo4j of null values. This function also renamed the columns of each file by replacing the space by an underscore to not have a formatting problem.

```
def cleanDf(df):
    df.columns = [c.replace(" ","_") for c in df.columns]
    for column in df.columns:
        if df[column].isnull().sum() > 0:
            df.drop(df[df[column].isnull()].index, inplace=True)
```

After cleaning, the donations, donors, projects dataframes, then we sampled the data by taking only 1 000 000 rows of every file because we are limited by the computing power of our computers. In addition to that, we noticed that some donors id from some donations are not in the donors dataframe. In order to remove the donor id from the donation dataframe that are not in the donor dataframe and vice versa, we created a function deleteNotIn, that takes in argument 2 dataframes and the column name to check.

```
def deleteNotIn(df1,df2,columnid):
    test = set(df1[columnid]) & set(df2[columnid])
    indexNames1 = df1[df1[columnid].apply(lambda x: False if x in test else True)].index
    indexNames2 = df2[df2[columnid].apply(lambda x: False if x in test else True)].index
    df1.drop(indexNames1 , inplace=True)
    df2.drop(indexNames2 , inplace=True)
```

```
deleteNotIn(df_donor,df_donation,"Donor_ID")
```

To illustrate, the problem that we have, imagine the blue circle, is a set that contains all the different donor id from the donor dataframe, the orange circle, is the same but of the donation dataframe. What we need, is the intersection of the two sets, to not have a donor id that doesn't exist in the other set.

Thirdly, we sampled the donations dataframe because it contained 3 millions rows and that was too heavy for our computers. When we imported it on Neo4j, we only took the first 1 million rows, to reduce the number of rows.

```
df_donation_sample_copy = df_donation[:1000000]
```

Finally, we converted the 3 dataframes to csv file, and then to import them on Neo4J. We renamed the new files with the letter "A" at the end to better recognize them.

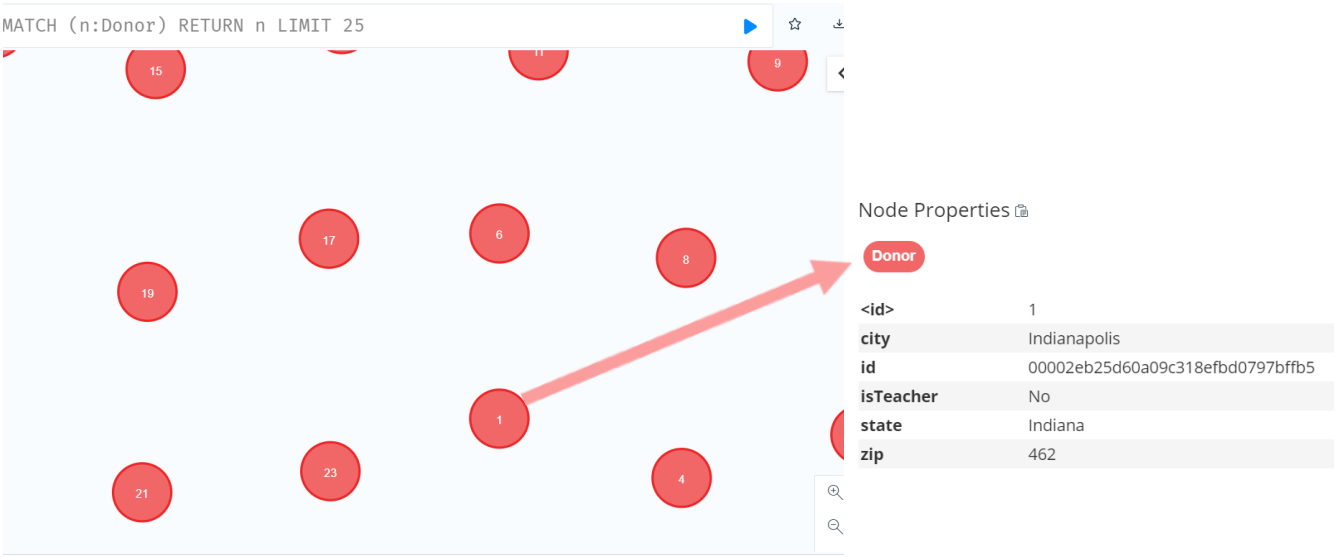
```
df_donor_sampled.to_csv(pathDBMS+fileName_donor+"A.csv",index=False)
```

2.1.3. Nodes

Then, we loaded the files corresponding to the nodes in Neo4j and we included their respective properties.

```
//1) LOAD Donor
:auto LOAD CSV WITH HEADERS FROM "file:/DonorsA.csv" as
l
        MERGE (donor:Donor{
          id:l.Donor_ID,
          city:l.Donor_City,
          state: l.Donor_State,
          isTeacher: l.Donor_Is_Teacher,
          zip: l.Donor_Zip
                    })
        RETURN l limit 5;
```

This is the donor grah we obtain after executing the load query above :

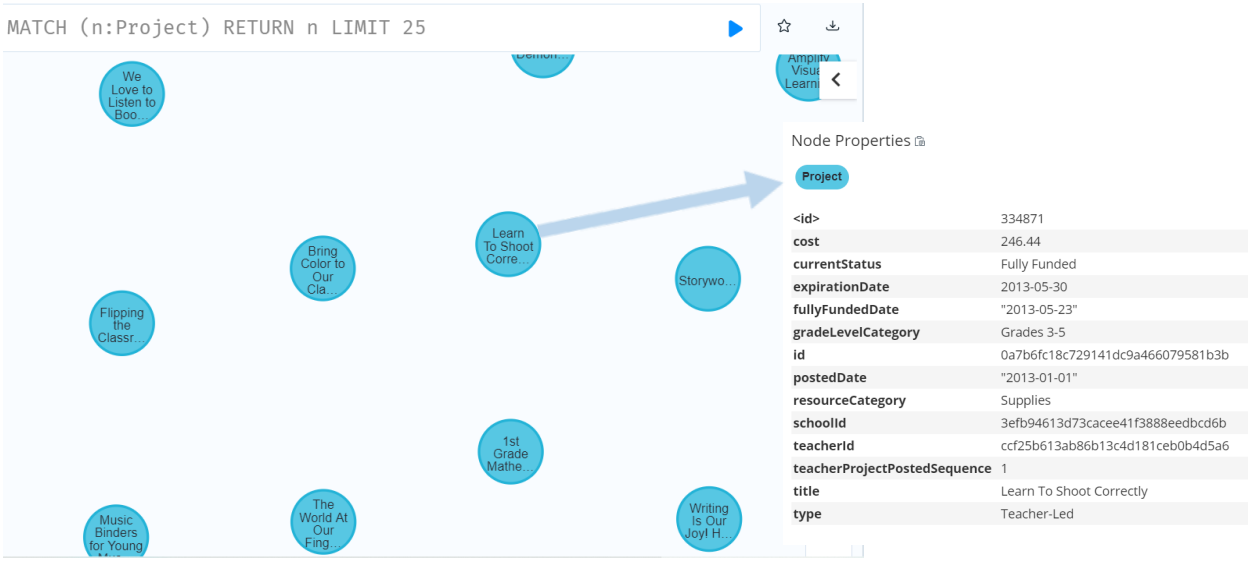


As for Donor nodes, we did the same for Projects

```
//2) LOAD Project
:auto LOAD CSV WITH HEADERS FROM "file:/ProjectsA.csv" as l
        MERGE (project:Project{
          id:l.Project_ID,
          schoolId:l.School_ID,
          teacherId: l.Teacher_ID,
          teacherProjectPostedSequence: l.Teacher_Project_Posted_Sequence,
          type: l.Project_Type,
          title: l.Project_Title,
          gradeLevelCategory: l.Project_Grade_Level_Category,
          resourceCategory: l.Project_Resource_Category,
          cost : toFloat(l.Project_Cost),
          postedDate: date(l.Project_Posted_Date),
          expirationDate: l.Project_Expiration_Date,
          currentStatus:  l.Project_Current_Status,
          fullyFunded date(l.Project_Fully_Funded_Date)

        RETURN l limit 5;
```

This is the Project grah we obtain after executing the load query above :

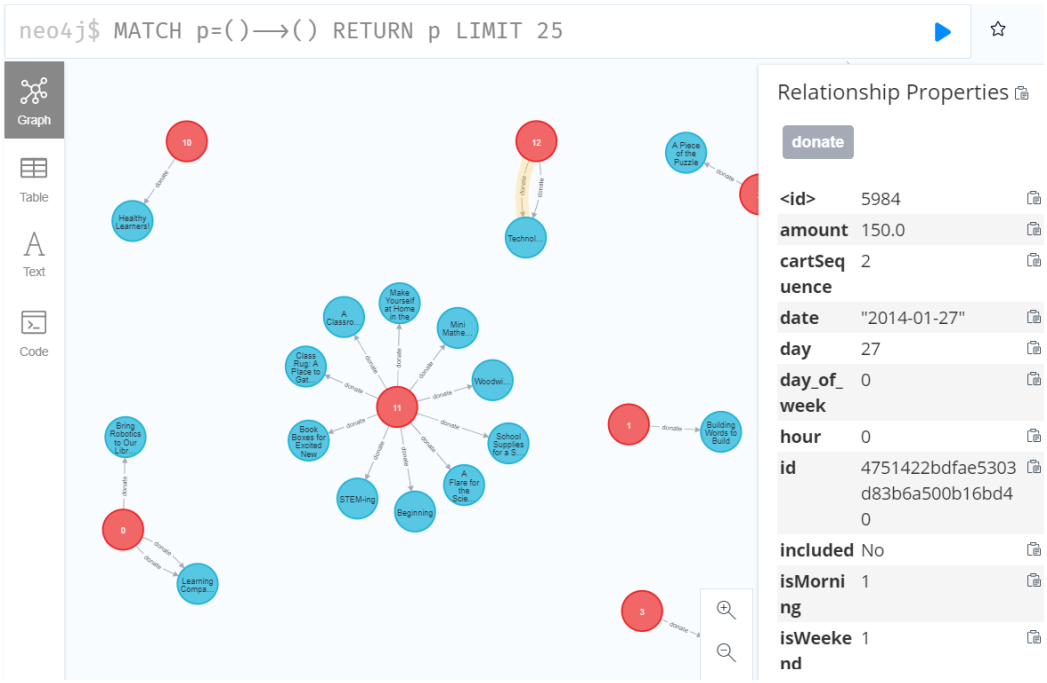


2.1.4. Relationship

Finally, we loaded the relation between the nodes Project and Donor. This relationship is given by the donation file that contains all the properties of the donation between a donor and the project that he finances.

```
//3) RELATION Donor → Project New
:auto LOAD CSV WITH HEADERS FROM "file:/donationsA.csv" as l
MERGE (donor:Donor{id:l.Donor_ID} )
MERGE (project:Project{id:l.Project_ID} )
MERGE (donor) -[:donate{
            id:l.Donation_ID,
            included: l.Donation_Included_Optional_Donation,
            amount: toFloat(l.Donation_Amount),
            cartSequence: toInteger(l.Donor_Cart_Sequence),
            date: date(l.Donation_Received_Date),
            year: toInteger(l.year),
            month: toInteger(l.month),
            day: toInteger(l.day),
            hour : toInteger(l.hour),
            min: toInteger(l.min),
            day_of_week: l.day_of_week,
            isWeekend: l.isWeekend,
            isMorning: l.isMorning
                    }]→ (project);
```

As a final result, we obtain the following graph that represents all the relations between all donors and projects.



2.1.5. Indexes

After doing some queries, we noticed that we have some node properties that often come up. Hence, for query performance purposes and with the aim of making searches more efficient, we decided to add an index on some nodes and relationship properties.

INDEX		
Donor	**donate**	**Project**
id	id	id
state	amount	year
city		

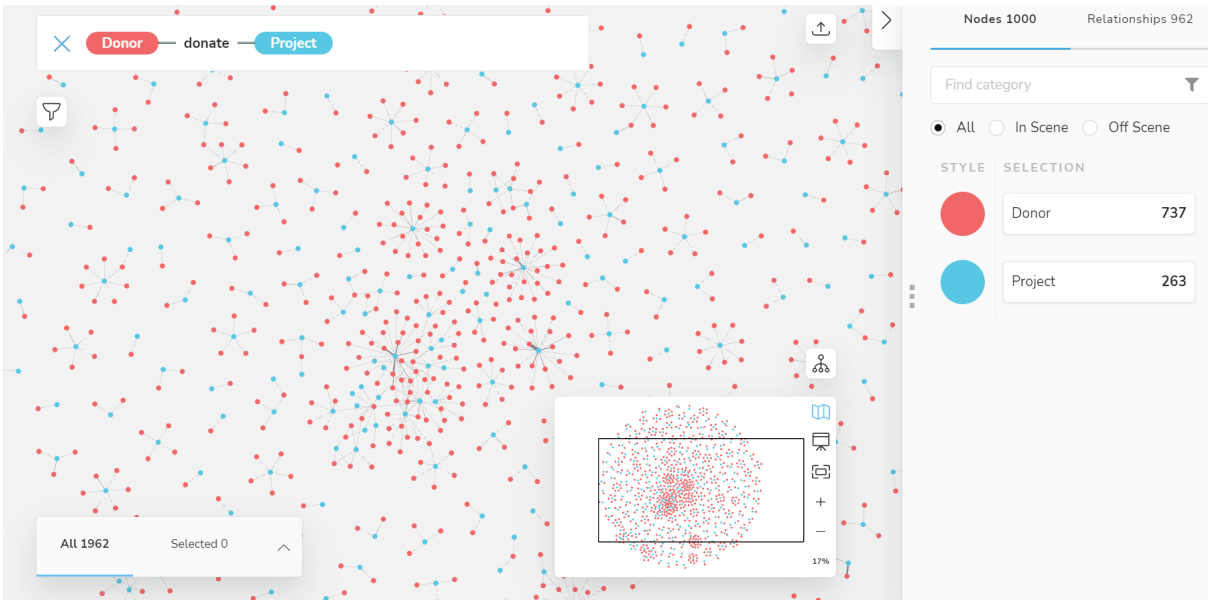
To create these indexes, we used the following queries :

```
neo4j$ CREATE INDEX ON:Donor(id)
```

```
neo4j$ CREATE INDEX donate_index FOR ()-[d:donate]→() ON (d.year)
```

2.2. Cypher projections

Before making the different cypher projection, we use BLOOM, to make some visualization on the data, to make the best cypher projection and to see if we can make some assumptions and to verify them.



We created the general cypher projection of the bi-partite graph by executing the following cypher creation query and we called it “Bi-partite” :

```
CALL gds.graph.create.cypher(
                "Bi-partite",
                "MATCH (donor:Donor)-[d:donate]→(p:Project)
                WITH donor
                RETURN DISTINCT id(donor) as id",
                "MATCH (source:Donor)-[d:donate]→(p:Project)
                WITH source
                MATCH (target:Donor)-[t]→(p:Project)
                WHERE target.id = source.id
                RETURN id(source) AS source, id(target) as target
                "
            ) YIELD graphName AS graph,
```

	graph	nodes	rels
1	"Bi-partite"	334865	59275978

Then, we thought that it would be interesting to create a subgraph in order to understand various behavior from the donor in California in 2014.

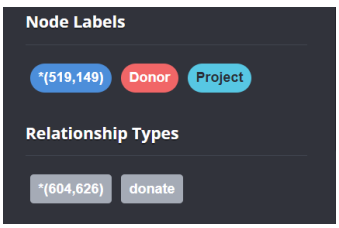
```
CALL gds.graph.create.cypher(
                "California2017",
                "MATCH (donor:Donor{state:'California'})-[d:donate{year:2017}]→()
                 RETURN DISTINCT id(donor) as id",
                "MATCH (source:Donor{state:'California'})-[d:donate{year:2017}]→()
                WITH source
                MATCH (target:Donor)-[t]→()
                WHERE target.id = source.id
                RETURN id(source) AS source, id(target) as target"
            ) YIELD graphName AS graph,
                    nodeCount AS nodes,
                    relationshipCount AS rels
```

	graph	nodes	rels
1	"California2017"	12710	12864248

Finally, we thought that it would be interesting to create a cypher projection of the donor who donated to the same project.

```
1  //TOP 50 donors
2  CALL gds.graph.create.cypher(
3                  "TOP 50 donors",
4                  "MATCH (donor:Donor)-[d:donate{year:2017}]→(p:Project)
5                  WITH donor, COUNT(DISTINCT d) as NB
6                  RETURN DISTINCT id(donor) as id, NB
7                  ORDER BY NB DESC LIMIT 50",
8                  "MATCH (source:Donor)-[d:donate{year:2017}]→(p:Project)
9                  WITH source, COUNT(DISTINCT d) AS nb
.0                 ORDER BY nb DESC LIMIT 50
.1                 MATCH (target:Donor)-[t]→(p:Project)
.2                 WHERE target.id = source.id
```

	graph	nodes	rels
1	"TOP 50 donors"	50	29182



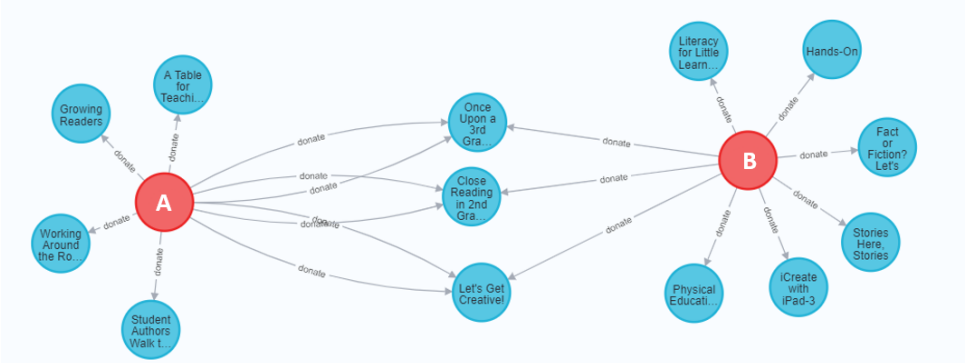
After obtaining the bi-partite graph, we have in total more than 500k Nodes and more than 600k relations between them all. We notice that there are more relationships than nodes. This is explained by the fact that a donor can donate to different projects, and can donate multiple times to the same project.

2.3. Node Similarity

The Node Similarity algorithm compares a set of nodes based on the nodes they are connected to. Two nodes are considered similar if they share many of the same neighbors. The input of this algorithm is usually a bipartite graph containing two disjoint node sets. Then, the algorithm compares all nodes from the first node set based on their relationships to nodes in the second set.

Therefore, in order to get node similarity, we have to choose 2 sets of nodes to compare. For that we did the query below that returns the 5 states that donated the most (in terms of donation amount) and their number of total donations.

To evaluate node similarity, we will use jaccard and overlap similarities. But before that, we will give an example of how these 2 similarity scores are calculated. The set of nodes that we will use for the example is the one below.



Jaccard Similarity :

This is the jaccard similarity formula : $J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$

1°) We calculate the value of the different terms of the equation separately :

$$|A| = 7$$
$$|B| = 9$$
$$|A \cap B| = 3$$

2°) We can replace the terms by their values on the formula :

$$J(A,B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} = \frac{3}{(7+9)-3} = \frac{3}{13} = 0.23$$

Overlap Similarity :

This is the overlap similarity formula : $O(A,B) = \frac{|A \cap B|}{\min(|A|, |B|)}$

1°) We calculate the value of differents terms of the equation separately :

$$|A| = 7$$
$$|B| = 9$$
$$|A \cap B| = 3$$
$$min(|A|, |B|) = 7$$

2°) We can replace the terms by their values on the formula :

$$O(A, B) = \frac{|A \cap B|}{min(|A|,|B|)} = \frac{3}{7} = \frac{3}{7} = 0.42$$

In our dataset, we will give jaccard and overlap similarities between donors who did the most donations (number) and donors who donated the most (amount). First, we will get the 5 states with the highest number of donations. Then we will focus on the first state and a specific year to find the top 2 donors and finally compute their similarity scores.

The 5 states who made the most donations (number of donations) :

```
1 //1) top states donation & amount
2 MATCH (donor:Donor)-[d:donate]→ ()
3 WITH count(d) as NB,donor, sum(d.amount) as Total_Amount
4 RETURN DISTINCT donor.state, sum(NB) as Total_Donation,
  sum(Total_Amount) as Total_Amount
5 ORDER BY Total_Donation DESC limit 5
```

Table | Text | Code

"donor.state"	"Total_Donation"	"Total_Amount"
"California"	96883	6546035.460000006
"New York"	47102	3429222.110000008
"Texas"	36501	2464152.209999998
"Illinois"	32549	2236279.7600000035
"Florida"	30126	1841008.380000004

This result stresses that California is the state that presents the highest number of donations. Thus, we will focus on this state especially for the city of Los Angeles in 2014 for the rest of the study. In particular, we will evaluate node similarity between the 2 donors who made the most donations (number of donations) and the 2 donors who donate the most (sum of donation amount) to highlight a possible pattern between them. Therefore, we first made queries to get their id, and then we computed the jaccard and overlap similarities between both of them.

The 2 donors from Los Angeles who made the most donations in 2014 (number of donations) :

```
1  //2) Ranked Donor by total donation in California
2  MATCH (donor:Donor{state: "California",city:"Los Angeles"})
   -[d:donate{year:2014}]→ ()
3  RETURN donor, count(d) as Total_Donation
4  ORDER BY Total_Donation DESC limit 2
```

Graph | Table | Text | Code

"donor"	"Total_Donation"
{"zip":"900","isTeacher":"No","state":"California","id":"0e345dcdef0d2a36c9bd17bf1ac3e10a","city":"Los Angeles"}	277
{"zip":"900","isTeacher":"Yes","state":"California","id":"4a38ea1371c10c8b5fb8658ea42a8f1f","city":"Los Angeles"}	118

We get the id of these 2 donors and reinject this value in the next query that computes their jaccard and overlap similarity.

The Overlap and Jaccard similarity of the different year of donations between the 2 donors we found above :

```
1  //3)Jacar & Overlap California total donations 2014
2  MATCH (donor1:Donor{id: "0e345dcdef0d2a36c9bd17bf1ac3e10a"})-[d1:donate{year:2014}]→(p1:Project)
3  WITH donor1, collect(distinct id(p1)) as donor1Project
4
5
6  MATCH (donor2:Donor{id: "4a38ea1371c10c8b5fb8658ea42a8f1f"})-[d2:donate{year: 2014}]→(p2:Project)
7  WITH donor1,donor1Project, donor2,collect(distinct id(p2)) as donor2Project
8
9  RETURN donor1.id as donor1, donor2.id as donor2,
10    gds.alpha.similarity.overlap(donor1Project, donor2Project) AS Overlap,
11    gds.alpha.similarity.jaccard(donor1Project, donor2Project) AS Jaccard
```

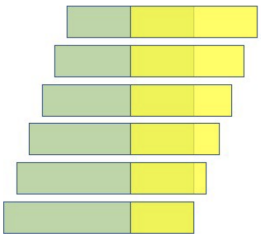
Table | Text

"donor1"	"donor2"	"Overlap"	"Jaccard"
"0e345dcdef0d2a36c9bd17bf1ac3e10a"	"4a38ea1371c10c8b5fb8658ea42a8f1f"	0.05434782608695652	0.015527950310559006

It appears that we have a very low similarity value for the project id for both donors. It means that they donate to different projects and, depending on the similarity technique, they have approximately between 1% and 5% donations given to the same project. We can explain that out of 100 donations for the donor1, we only gave 1-5 five times to the same project as the donor2. In conclusion, even if both did the most donations (number of donation) in the city of Los Angeles in 2014, that doesn't mean they donated to the same project, the probability is from 1% to 5%,we can say this hypothesis is wrong.

We noticed that the Jaccard algorithm doesn't have a sensibility on the size of the sets, because if one set is smaller than the other, the similarity will be not as accurate as the overlap algorithm.

To demonstrate that we have two sets A,B, that have 50 similar elements. We will increase the size of A, and decrease the size of B but there are still 50 similar elements between A and B. By changing the size of the sets, we will compare the different values of jaccard and overlap.



A	B	Jaccard	Overlap
100	100	0.333	0.500
110	90	0.333	0.556
120	80	0.333	0.625
130	70	0.333	0.714
140	60	0.333	0.833
150	50	0.333	1.000

We see that the jaccard similarity doesn't change while the size of the set is different. It's still 0.333 whereas the overlap value is changing and is more accurate than the jaccard, because in the overlap formula, it takes the minimum size between the both sets. In conclusion, if we have two sets that haven't the same size, it's better to use overlap.

The 2 donors from Los Angeles donate the most (sum of donation amount) in 2014 :

```
1 //4) Ranked Donor by total donation amount  in California
2 MATCH (donor:Donor{state: "California",city:"Los Angeles"})
  -[d:donate{year:2014}]→ ()
3 RETURN donor, sum(d.amount) as Total_Amount
4 ORDER BY Total_Amount DESC limit 2
```

Graph Table Text

"donor"	"Total_Amount"
{"zip":"900","isTeacher":"No","state":"California","id":"64b026d79da9063bae49f74f50856aec","city":"Los Angeles"}	5814.579999999999
{"zip":"900","isTeacher":"Yes","state":"California","id":"1f5fc18d62377e3a99e712b46484b66e","city":"Los Angeles"}	4750.0

We get the id of these 2 donors and reinject this value in the next query that computes their jaccard and overlap similarity

The Overlap and Jaccard similarity of the different year of donations between the 2 donors we found above :

```
1 //5) Jacar & Overlap California total amount donations 2014
2 MATCH (donor1:Donor{id:
  "64b026d79da9063bae49f74f50856aec"})-[d1:donate]→
  (p1:Project)
3 WITH donor1, collect(d1.year) as donor1Years
4
5
6 MATCH (donor2:Donor{id:
  "1f5fc18d62377e3a99e712b46484b66e"})-[d2:donate]→
  (p2:Project)
7 WITH donor1, donor1Years, donor2, collect(d2.year) as
  donor2Years
```

Table Text

"user1"	"user2"	"Overlap"	"Jaccard"
"64b026d79da9063bae49f74f50856aec"	"1f5fc18d62377e3a99e712b46484b66e"	0.16666666666666666	0.20833333333333334

To summarize, we made a comparison between the similarity of the 2 donors who did the most number of donation and the 2 donors who gave the biggest amount.

	Jaccard	Overlap
Top 2 donors (number of donations)	0.0543	0.0155
Top 2 donors (total amount)	0.1667	0.2083

For both of them, the similarity is higher for the 2 donors who gave the biggest amount. It can be explained by a greater intersection for sum of amounts. Moreover, to explain the difference between jaccard and overlap scores for both of them, it can be noted that the jaccard similarity is a very powerful analysis technique, but it has a major drawback when the two sets being compared have different sizes. And that is the case here. This is the reason why jaccard score is not enough because it has no sensitivity to the sizes of the sets.

Euclidean similarity:

We wanted to calculate the Euclidean similarity, for the the both donors who donates the most in number of donation but the query gave this result :

```
MATCH (donor1:Donor{id: "0e345dcdef0d2a36c9bd17bf1ac3e10a"})-[d1:donate]→(p1:Project)
WITH donor1, collect(id(p1) ) as donor1Project


MATCH (donor2:Donor{id: "4a38ea1371c10c8b5fb8658ea42a8f1f"})-[d2:donate]→(p2:Project)
WITH donor1, donor1Project, donor2, collect(id(p2)) as donor2Project

RETURN donor1.id as user1, donor2.id as user2,
    gds.alpha.similarity.euclidean(donor1Project, donor2Project) AS Euclidean
```

ERROR **Neo.ClientError.Procedure.ProcedureCallFailed**

```
Failed to invoke function `gds.alpha.similarity.euclidean`: Caused by: java.lang.RuntimeException: Vectors must be non-empty and of the same size
```

The euclidean algorithm need to have the same size whereas the the first donor "0e345dcdef0d2a36c9bd17bf1ac3e10a" have donated 277 times, the other one did 118 times, the euclidean algorithm will not work. We tried to use distinct on the collect of the project id, maybe they donated to the same number of different projects but same error.

```
MATCH (donor1:Donor{id: "0e345dcdef0d2a36c9bd17bf1ac3e10a"})-[d1:donate]→(p1:Project)
WITH donor1, collect(distinct id(p1) ) as donor1Project


MATCH (donor2:Donor{id: "4a38ea1371c10c8b5fb8658ea42a8f1f"})-[d2:donate]→(p2:Project)
WITH donor1, donor1Project, donor2, collect(distinct id(p2)) as donor2Project

RETURN donor1.id as user1, donor2.id as user2,
    gds.alpha.similarity.euclidean(donor1Project, donor2Project) AS Euclidean
```

ERROR **Neo.ClientError.Procedure.ProcedureCallFailed**

```
Failed to invoke function `gds.alpha.similarity.euclidean`: Caused by: java.lang.RuntimeException: Vectors must be non-empty and of the same size
```

We decided to use the different attribute of the relationship "donate" to calculate the euclidean similarity, like the year of the donation, the amount, but always the same error "*Vectors must have the same size*".

Then, we tried on the donors who contributed the most (sum of donation amount) in 2014. At the beginning, we had the issues with the size of the vector, then we tried to calculate the similarity on the year of the donation of both donors.

```
MATCH (donor1:Donor{id: "64b026d79da9063bae49f74f50856aec"})-[d1:donate]→(p1:Project)
MATCH (donor2:Donor{id:"1f5fc18d62377e3a99e712b46484b66e"})-[d2:donate]→(p2:Project)

RETURN donor1.id AS donor1, donor2.id AS donor2,
gds.alpha.similarity.euclidean(collect(d1.year),collect(d2.year)) AS euclidean
```

	user1	user2	Euclidean
1	"64b026d79da9063bae49f74f50856aec"	"1f5fc18d62377e3a99e712b46484b66e"	1.0

We got 1 in result, that literally means they made donations in exactly the same years.

```
MATCH (donor:Donor{state: "California",city:"Los Angeles"}) -[d:donate{year:2014}]-
> (p:Project)
WITH sum(d.amount) as Total_Amount,donor,collect(distinct d.year) as Year_Donation
RETURN donor,Total_Amount,Year_Donation
ORDER BY Total_Amount DESC limit 2
```

Graph Table Text

"donor"	"Total_Amount"	"Year_Donation"
{"zip":"900","state":"California","id":"64b026d79da9063bae49f74f50856aec","isTeacher":"No","city":"Los Angeles"}	5814.579999999999	[2014]
{"zip":"900","state":"California","id":"1f5fc18d62377e3a99e712b46484b66e","isTeacher":"Yes","city":"Los Angeles"}	4750.0	[2014]

We noticed, they made all their donations in 2014. So, without surprise their euclidean similarity will be one.

Little reminder: the euclidean similarity is between 0 and 1, the more the value is closer to one the more 2 nodes are similar and the closer to 0 the more the nodes are far away in distance and not quite the same.

2.4. Link prediction

Link prediction algorithms help determine the closeness of a pair of nodes using the topology of the graph. The computed scores can then be used to predict new relationships between them. There are various algorithms for calculating link prediction :

- Adamic Adar : a measure used to compute the closeness of nodes based on their shared neighbors. It is using the following formula

  $$A(x,y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{\log |N(u)|}$$ where N(u) is the set of nodes adjacent to u.

  A value of 0 indicates that two nodes are not close, while higher values indicate nodes are closer.

- Common neighbors : common nodes of a set of 2 nodes
  It is computed using the following formula:

  $$CN(x,y) = |N(x) \cap N(y)|$$ where N(x) is the set of nodes adjacent to node x, and N(y) is the set of nodes adjacent to node y.

  A value of 0 indicates that two nodes are not close, while higher values indicate nodes are closer.

- Total neighbors: based on the idea that the more connected a node is, the more likely it is to receive new links.It is computed using the following formula:

  $$TN(x,y) = |N(x) \cup N(y)|$$ where `N(x)` is the set of nodes adjacent to `x`, and `N(y)` is the set of nodes adjacent to `y`.

- Preferential Attachment : means that the more connected a node is, the more likely it is to receive new links. It is how much they have the capacity to spread information togetherIt is computed using the following formula:

  $$PA(x,y) = |N(x)| * |N(y)|$$ where N(u) is the set of nodes adjacent to u.

A value of 0 indicates that two nodes are not close, while higher values indicate that nodes are closer.

- Resource Allocation : How much the common neighbors are “famous” . It is computed using the following formula:

$$RA(x,y) = \sum_{u \in N(x) \cap N(y)} \frac{1}{|N(u)|}$$ where N(u) is the set of nodes adjacent to u.

We are going to give the link prediction based on the different algorithm we presented above, on the 2 donors who donated the most (sum of donation amount) in 2014, and who made the most donations in 2014 (number of donations):

```
MATCH (d1:Donor)-[r1:donate]→(p1:Project) WITH d1
WHERE d1.id in
["0e345dcdef0d2a36c9bd17bf1ac3e10a","4a38ea1371c10c8b5fb8658ea42a8f1f","64b026d79da9063bae49f74f50856aec","1f5fc18d62377e3a99
e712b46484b66e"]

MATCH (d2:Donor)-[r2:donate{year:2014}]→(p2:Project)
WHERE d2.id in
["0e345dcdef0d2a36c9bd17bf1ac3e10a","4a38ea1371c10c8b5fb8658ea42a8f1f","64b026d79da9063bae49f74f50856aec","1f5fc18d62377e3a99
e712b46484b66e"] and d1.id < d2.id
WITH d1, d2
RETURN d1.id as u1, d2.id as u2,
gds.alpha.linkprediction.adamicAdar(d1, d2, {relationshipQuery:"donate"}) AS aa,
gds.alpha.linkprediction.commonNeighbors(d1, d2, {relationshipQuery:"donate"}) AS cn,
gds.alpha.linkprediction.totalNeighbors(d1, d2, {relationshipQuery:"donate"}) AS tn,
gds.alpha.linkprediction.preferentialAttachment(d1, d2, {relationshipQuery:"donate"}) AS pa,
gds.alpha.linkprediction.resourceAllocation(d1, d2, {relationshipQuery:"donate"}) AS ra
ORDER BY aa DESC
```

	u1.id	u2.id	aa	cn	tn	pa	ra
1	"0e345dcdef0d2a36c9bd17bf1ac3e10a"	"4a38ea1371c10c8b5fb8658ea42a8f1f"	14.597062385949426	40.0	1010.0	299808.0	2.870299719305666
2	"0e345dcdef0d2a36c9bd17bf1ac3e10a"	"64b026d79da9063bae49f74f50856aec"	0.0	0.0	855.0	6246.0	0.0
3	"0e345dcdef0d2a36c9bd17bf1ac3e10a"	"1f5fc18d62377e3a99e712b46484b66e"	0.0	0.0	856.0	23943.0	0.0
4	"4a38ea1371c10c8b5fb8658ea42a8f1f"	"64b026d79da9063bae49f74f50856aec"	0.0	0.0	207.0	1728.0	0.0
5	"1f5fc18d62377e3a99e712b46484b66e"	"4a38ea1371c10c8b5fb8658ea42a8f1f"	0.0	0.0	208.0	6624.0	0.0
6	"1f5fc18d62377e3a99e712b46484b66e"	"64b026d79da9063bae49f74f50856aec"	0.0	0.0	13.0	138.0	0.0

We can observe that except the first line, all the rows have 0 common neighbors, this why the AcademicAdar and the Resource Allocation are equal to 0. The first line is the pair of nodes who made the most donation, this is the only row that have common neighbors. We can conclude that 2 donors who made more donations even if it's a little bit of money, have more capacity to spread the information (Preferential Attachment) than 2 donors who gave more money but made less donations. According to this fact, most of the time, they will be connected to common projects that are more popular than donors who gave more money.

We can see that the 2 donors who donated the most together have an AcademicAdar value 4 times greater than the other pairs, which means the common projects they have donated, have received a lot of donations. Moreover, the most important is the projects they have donated are very popular, this explains why they have the best Preferential Attachment. In addition to that, they have the most common neighbors, 4 times more than others, this explains why the AcademicAdar is very high compared to the other pairs of nodes.

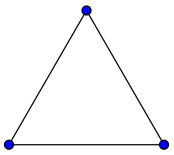
```
MATCH (u1:Donor{state: "California",city:"Los Angeles"})-[r1:donate{year:2014}]→(a1:Project)
WITH u1, count(r1) as nb_donations
ORDER BY nb_donations DESC LIMIT 5

MATCH (u2:Donor{state: "California",city:"Los Angeles"})-[r2:donate{year:2014}]→(a2:Project)
WHERE u1.id < u2.id
WITH u1, u2, count(r2) as nb_donations
ORDER BY nb_donations DESC LIMIT 5
RETURN u1.id, u2.id,
gds.alpha.linkprediction.adamicAdar(u1, u2, {relationshipQuery:"donate"}) AS aa,
gds.alpha.linkprediction.commonNeighbors(u1, u2, {relationshipQuery:"donate"}) AS cn,
gds.alpha.linkprediction.totalNeighbors(u1, u2, {relationshipQuery:"donate"}) AS tn,
gds.alpha.linkprediction.preferentialAttachment(u1, u2, {relationshipQuery:"donate"}) AS pa,
gds.alpha.linkprediction.resourceAllocation(u1, u2, {relationshipQuery:"donate"}) AS ra
ORDER BY aa DESC LIMIT 5
```

	u1.id	u2.id	nb_Donation1+nb_Donation2	aa	cn	tn	pa	ra
1	"0e345dcdef0d2a36c9bd17bf1ac3e10a"	"4a38ea1371c10c8b5fb8658ea42a8f1f"	395	14.597062385949426	40.0	1010.0	299808.0	2.870299719305666
2	"4a38ea1371c10c8b5fb8658ea42a8f1f"	"50fcefb289b3eca9d5f00802edb06eab"	168	3.04680643203368	9.0	256.0	23616.0	0.4925982270302757
3	"0e345dcdef0d2a36c9bd17bf1ac3e10a"	"50fcefb289b3eca9d5f00802edb06eab"	327	1.5512658409487454	5.0	908.0	85362.0	0.24007388580691932
4	"43bc5c6892c61fef11719982c90d8a98"	"4a38ea1371c10c8b5fb8658ea42a8f1f"	161	1.0551557211398157	2.0	252.0	28512.0	0.3
5	"43bc5c6892c61fef11719982c90d8a98"	"50fcefb289b3eca9d5f00802edb06eab"	93	0.0	0.0	117.0	8118.0	0.0

2.5. Community detection

We are going to calculate the number of triangles in our graph. A triangle exists in a graph when 3 nodes are connected all connected together.



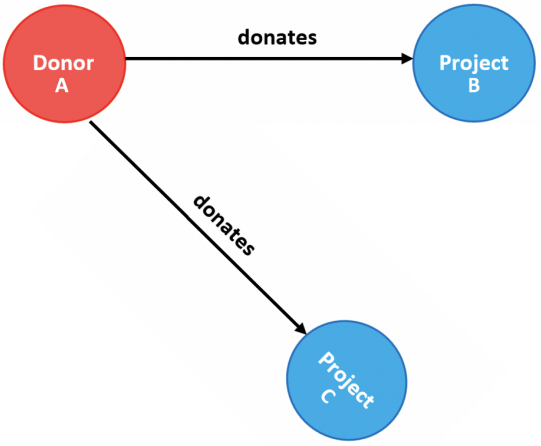
The number of triangle in the cypher projection “Bi-partite”:

```
CALL gds.triangleCount.stream("Bi-partite")
YIELD nodeId, triangleCount
RETURN gds.util.asNode(nodeId).id, triangleCount ORDER BY triangleCount DESC
```

	gds.util.asNode(nodeId).id	triangleCount
1	"00002d44003ed46b066607c5455a999a"	0
2	"00002eb25d60a09c318efbd0797bffb5"	0
3	"00005f52c98eeaf92b2414a352b023a4"	0
4	"00008eec5aab2228652e22457881f2d0"	0
5	"0000a2175753bc165e53c408589a3bd6"	0
6	"0000b282b7070b4dd31c05210f82453b"	0

In our cypher projection “Bi-partite”, we have 0 triangle, this completely normal because to have a triangle we need to have a donor A who donates to a project B and project C and the project B and C have to be linked but in our graph we doesn't have this kind of relationship.Furthermore, Bipartite graphs are 2-colorable, but a triangle requires three colors.

Example of our Bi-partite graph:



2.6. Pathfinding

Path finding algorithms find the shortest path between two or more nodes or evaluate the availability and quality of paths. This is the graph that we created to use the Dijkstra algorithm.

```
CALL gds.alpha.allShortestPaths.stream(
        {
                nodeQuery:"MATCH (donor:Donor)-[d:donate{year:2017}]→(p:Project)
                WITH donor, COUNT(DISTINCT d) as NB
                RETURN DISTINCT id(donor) as id, NB
                ORDER BY NB DESC LIMIT 50",
                relationshipQuery:"MATCH (source:Donor)-[d:donate{year:2017}]→(p:Project)
                WITH source, COUNT(DISTINCT d) AS nb
                ORDER BY nb DESC LIMIT 50
                MATCH (target:Donor)-[t]→(p:Project)
                WHERE target.id = source.id
                RETURN id(source) AS source, id(target) as target, sum(t.amount) as NB",
```

	sourceNodeId	targetNodeId	distance
1	1168852	1168852	0.0
2	1123037	1123037	0.0
3	1072358	1072358	0.0
4	1274057	1274057	0.0
5	1118729	1118729	0.0
6	1099027	1099027	0.0

2.7. Centrality

We are going to use a Page Rank algorithm to measure the average, min and max score for corresponding cities in the state of California.

The Page rank score in city in the state of California an on the cypher projection “California2018”:

```
CALL gds.pageRank.stream("California2018")
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).city AS city,score as score, AVG(score), MIN(score),
MAX(score)
ORDER BY score DESC, city ASC
```

	city	score	AVG(score)	MIN(score)	MAX(score)
1	"Lake Elsinore"	0.9612404689163623	0.9612404689163623	0.9612404689163623	0.9612404689163623
2	"Sylmar"	0.9612404689163329	0.9612404689163329	0.9612404689163329	0.9612404689163329
3	"Los Angeles"	0.9612404689159593	0.9612404689159593	0.9612404689159593	0.9612404689159593
4	"Bakersfield"	0.961240468915659	0.961240468915659	0.961240468915659	0.961240468915659
5	"La Crescenta"	0.9612404689156154	0.9612404689156154	0.9612404689156154	0.9612404689156154
6	"Altadena"	0.9612404689155767	0.9612404689155767	0.9612404689155767	0.9612404689155767
7					

As we can see, all the cities have the same page rank score because it has incoming links from the same donors.

3. Mono-partite graph

3.1. Model creation

3.1.1. Description

This dataset is a transformation of the main graph (figure 1) presented previously. We created an undirected and unweighted mono-partite graph that shows the relation between two teachers. A teacher is linked with another one if they know each other. The relationship "knows" means that these teachers supervised projects belonging to the same school. Therefore two teachers know each other if they are colleagues i.e. working in the same school.

3.1.2. Data pre-processing

First of all, we need 3 csv files to obtain, this graph:
"*teachers.csv*","*project.csv*"," *schools.csv*"

Secondly, before doing anything, we need to do some data processing, like data cleaning. We found that some columns of files contain some null values.

```
Entrée [12]: df_project.isnull().sum()

Out[12]: Project ID                            0
         School ID                             0
         Teacher ID                            0
         Teacher Project Posted Sequence       0
         Project Type                          0
         Project Title                         6
         Project Essay                         1
         Project Short Description             3
         Project Need Statement                3
         Project Subject Category Tree        29
         Project Subject Subcategory Tree     29
         Project Grade Level Category          0
         Project Resource Category            36
         Project Cost                          0
         Project Posted Date                   0
         Project Expiration Date              14
         Project Current Status                0
         Project Fully Funded Date        283253
```

We use the same function *cleanDf that we used before for the bi-partite graph, that takes the data frame that needs to be cleaned*, the function deletes rows which contain null values, so we will not have a problem when we import the data on Neo4j of null values. This function also renamed the columns of each file by replacing the space by an underscore to not have a formatting problem.

In addition to that, we noticed the same problems as in bi-partite graph that some teachers ids from some project are not in the teacher dataframe. In order to remove the teachers ids from the teacher dataframe that are not in the project dataframe and vice versa, we used the function deleteNotIn, that takes in argument 2 dataframes and the column name to check.

```
def deleteNotIn(df1,df2,columnid):
    test = set(df1[columnid]) & set(df2[columnid])
    indexNames1 = df1[df1[columnid].apply(lambda x: False if x in test else True)].index
    indexNames2 = df2[df2[columnid].apply(lambda x: False if x in test else True)].index
    df1.drop(indexNames1 , inplace=True)
    df2.drop(indexNames2 , inplace=True)
```

```
deleteNotIn(df_teacher,df_project,"Teacher_ID")
```

To illustrate, the problem that we have, imagine the blue circle, is a set that contains all the different teachers ids from the teacher dataframe, the orange circle, is the same but of the project dataframe. What we need, is the intersection of the two sets, to not have a teacher id that doesn't exist in the other set.

Thirdly, we grouped the project dataframe by *school id* and *the year of the project posted date*. In order to create all the pairs of teachers for each school and year, we used the function createColumns that takes a row of the grouped by dataframe and creates all the possible pairs and adds for each element the year and the school id .

```
def createColumns(row,teacher1List,teacher2List,schoolList,yearList):
    arrTeacherId = row["Teacher_ID"]
    schoolId = row["School_ID"]
    teacherIds= list(set(arrTeacherId))
    for pair in itertools.combinations(teacherIds,2):
            teacher1List.append(pair[0])
            teacher2List.append(pair[1])
            yearList.append(str(set(row['year'])))
            schoolList.append(str(set(schoolId)))
```

```
teacher1List = list()
teacher2List = list()
schoolList = list()
yearList= list()
groupBY.apply(lambda x: createColumns(x,teacher1List,teacher2List,schoolList,yearList))
```

Fourthly, we created a new dataframe with the array that we retrieved with the createColumns function.

```
df = pd.DataFrame(columns=[])
df["Teacher1"] = teacher1List
df["Teacher2"] = teacher2List
df["School_ID"] = schoolList
df["Year"] = yearList
```

Fifthly, we merged this new data frame with the school dataframe, to have more information about where each teacher met, like the city, the state, the school name.

```
result = pd.merge(df, df_school, on=["School_ID"])
```

Finally, we converted the result dataframes to a csv file. We renamed the new file “knows” because we will use it to create different relationships between teachers. We converted the teacher dataframe too to csv, we will use it to import on Neo4J to create all the nodes *Teacher*.

```
result.to_csv("knows.csv",index=False)
```

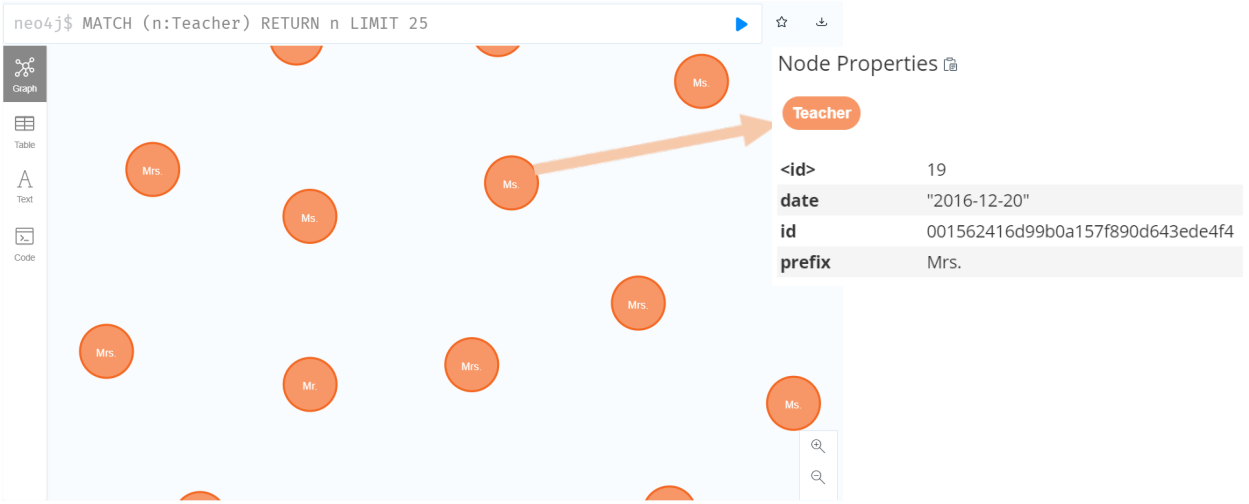
```
df_teacher.to_csv(pathDBMS+fileName_teacher+"A.csv",index=False)
```

3.1.3. Nodes

Then, we loaded the files corresponding to the nodes in Neo4j and we included their respective properties.

```
1 //LOAD Teacher
2 :auto LOAD CSV WITH HEADERS FROM "file:/TeacherA.csv" as l
3       MERGE (teacher:Teacher{
4           id:l.Teacher_ID,
5           prefix:l.Teacher_Prefix,
6           date:date( l.Teacher_First_Project_Posted_Date)
7       }) RETURN l limit 5;
```

This is the teacher grah we obtain after executing the load query above :



3.1.4. Relationship

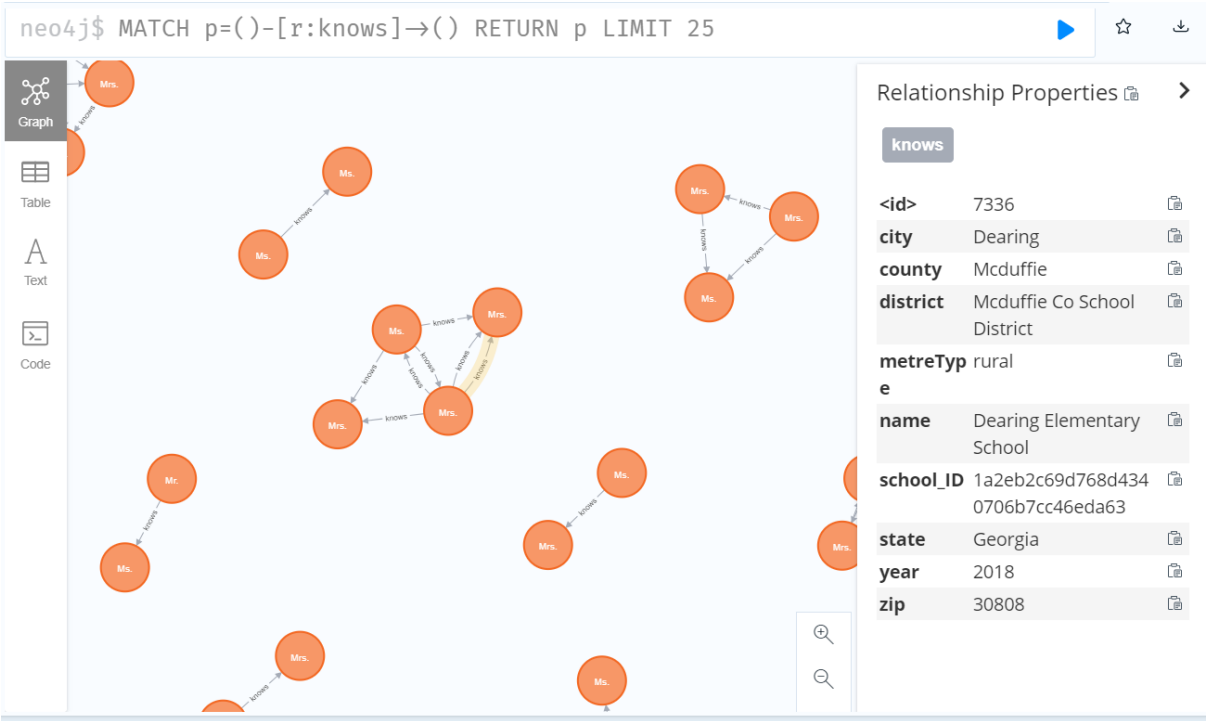
Next, we loaded the relation between the teacher's nodes. This relationship displays if a teacher knows the other one. To create the relationship, we had to take a closer look at the csv files. The file that made the link between a teacher and the school where he is working, is project.csv. Indeed, this dataset is composed by various properties and especially a Teacher ID and a School ID; Thus, after some python processing to create the relationship from the dataset, we obtained a new csv file containing all the information of the common school where the teachers met.

```
1  //RELATION Teacher → Teacher
2  :auto LOAD CSV WITH HEADERS FROM "file:/knowsA.csv" as l
3  MERGE (teacher1:Teacher{id:l.Teacher1} )
4  MERGE (teacher2:Teacher{id:l.Teacher2} )
5  MERGE (teacher1) -[:knows{
6      school_ID:l.School_ID,
7      year:toInteger(l.Year),
8      name:l.School_Name,
9      metreType:l.School_Metro_Type,
10     state:l.School_State,
11     zip:l.School_Zip,
12     city:l.School_City,
```

Table

Added 40414 labels, created 40414 nodes, set 679126 properties, created 70968 relationships, completed after 12535 ms.

As a final result, we obtain the following graph that represents all the relations between all donors and projects.



3.2. Cypher projections

We created the general cypher projection of the mono-partite graph by executing the following cypher creation query and we called it “Mono-partite” :

```
CALL gds.graph.create.cypher(
    "Mono-partite",
    "MATCH (t:Teacher) RETURN DISTINCT id(t) AS id",
    "MATCH (t1:Teacher) -[k:knows]→ (t2:Teacher)
    RETURN id(t1) as source, id(t2) as target")
    YIELD
    graphName AS graph,
    nodeCount AS nodes,
    relationshipCount AS rels
```

	graph	nodes	rels
1	"Mono-partite"	69579	430818

Then, we thought that it would be interesting to treat the question of gender or prefix for each teacher. Therefore, we created another cypher projection between the teachers who know each other and have the same prefix. We observed half less number of relations compared to the mono-partite graph.

```
//Cypher projection : Mono-prefix
CALL gds.graph.create.cypher(
    "mono-prefix",
    "MATCH (t:Teacher) RETURN DISTINCT id(t) AS id",
    "MATCH (t1:Teacher) -[k:knows]→ (t2:Teacher)
    WHERE t1.prefix = t2.prefix
    RETURN id(t1) as source, id(t2) as target")
    YIELD
    graphName AS graph,
    nodeCount AS nodes,
    relationshipCount AS rels
```

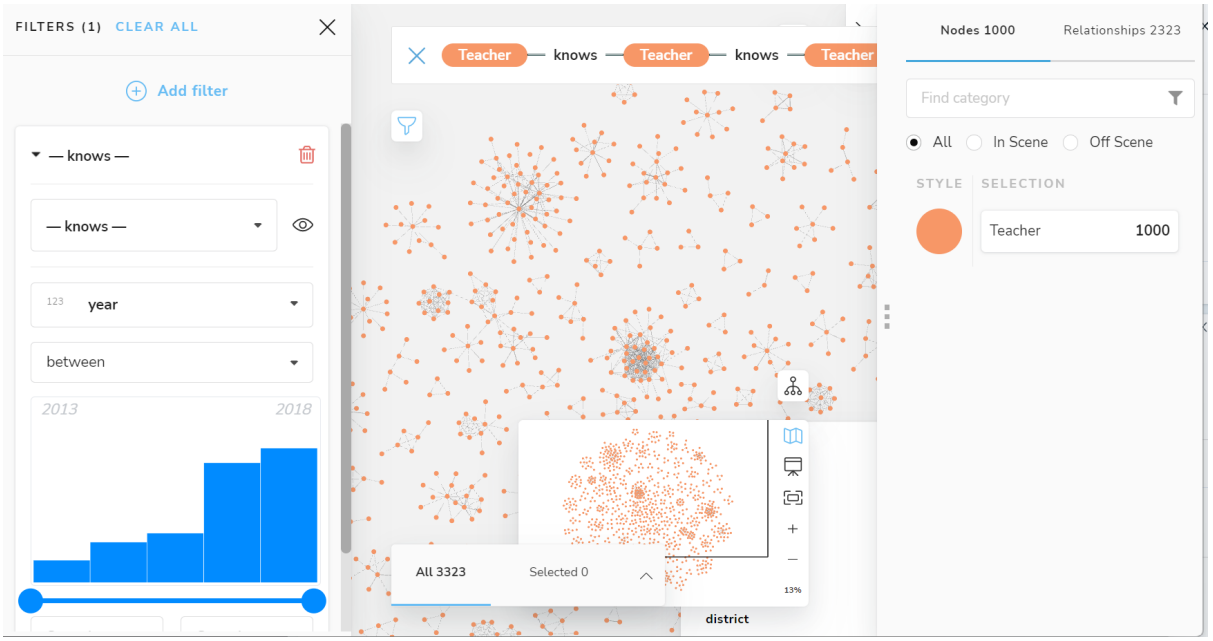
	graph	nodes	rels
1	"mono-prefix"	69579	199286

Finally, we thought that it would be interesting to create a cypher projection of the teachers who met in a specific year and here we chose 2018.

```
CALL gds.graph.create.cypher(
    "Teacher2018",
    "MATCH (t1:Teacher) RETURN DISTINCT id(t1) AS id",
    "MATCH (n:Teacher) -[k:knows{year:2018}]→ (m)
    RETURN id(n) as source, id(m) as target"
    )
    YIELD
    graphName AS graph,
    nodeCount AS nodes,
    relationshipCount AS rels
```

	graph	nodes	rels
1	"Teacher2018"	109993	8204

To visualize this projection, we used BLOOM visualization and got the following image. We were able to add filters and adapt the queries.



3.3. Node similarity

3.4.

We will study node similarity for teachers coming from rural zones. We calculated node similarity between the 2 most popular teachers with the following ids :

- id: 13bbaa62d7d310a898f6e8194a0f3efd
- id: 67b638392192e41e1b530ff6e641930a
-

```
1 MATCH (t1:Teacher)-[k:knows]→ (t2:Teacher)
2 WHERE k.metreType = "rural"
3 RETURN t1, COUNT(*) as nb_teachers
4 ORDER BY nb_teachers DESC LIMIT 2
5
```

Graph
Table
Text

"t1"	"nb_teachers"
{"date":"2015-08-13","id":"13bbaa62d7d310a898f6e8194a0f3efd","prefix":"Mrs."}	41
{"date":"2015-06-17","id":"67b638392192e41e1b530ff6e641930a","prefix":"Mrs."}	40

This query calculates jaccard and overlap similarity for the 2 selected teachers.

```
MATCH (t1:Teacher{id:"13bbaa62d7d310a898f6e8194a0f3efd"})-[]→
(t:Teacher)
WITH t1, collect(id(t)) AS liste_t1
MATCH (t2:Teacher{id:"67b638392192e41e1b530ff6e641930a"})-[]→
(t:Teacher)
WITH t1, liste_t1, t2, collect(id(t)) AS liste_t2
RETURN t1.id AS t1, t2.id AS t2,
gds.alpha.similarity.jaccard(liste_t1, liste_t2) AS jaccard,
gds.alpha.similarity.overlap(liste_t1, liste_t2) AS overlap
```

	t1	t2	jaccard	overlap
1	"13bbaa62d7d310a898f6e8194a0f3efd"	"67b638392192e41e1b530ff6e641930a"	0.975609756097561	1.0

We note that the similarity between these 2 teachers is really close to 1 meaning that they have a strong correlation and present approximately comparable conduct. That also means that these rural teachers have nearly the same relatives. In order to interpret an overlap score of 1, we can say that the teachers that one of them knows are included in the set of teachers that the second one knows.

3.5. Link prediction

In order to, we computed some link prediction algorithms scores that can then be used to predict new relationships between them.

```
MATCH (t1:Teacher{id:"13bbaa62d7d310a898f6e8194a0f3efd"})-[]→(t:Teacher)
WITH t1, collect(id(t)) AS liste_t1
MATCH (t2:Teacher{id:"67b638392192e41e1b530ff6e641930a"})-[]→(t:Teacher)
WITH t1, liste_t1, t2, collect(id(t)) AS liste_t2
RETURN
gds.alpha.linkprediction.adamicAdar(t1, t2) AS adamicAdar,
gds.alpha.linkprediction.commonNeighbors(t1, t2) AS commonNeighbors,
gds.alpha.linkprediction.totalNeighbors(t1, t2) AS totalNeighbors,
gds.alpha.linkprediction.preferentialAttachment(t1, t2) AS preferentialAttachment,
gds.alpha.linkprediction.resourceAllocation(t1, t2) AS resourceAllocation
```

	adamicAdar	commonNeighbors	totalNeighbors	preferentialAttachment	resourceAllocation
1	10.750443312909718	40.0	42.0	1681.0	0.9690766550522648

We obtain 40 common neighbors between these 2 teachers who have in total 42 neighbors. That means that they have approximately all 95% neighbors in common which is a lot. For the preferential attachment, we have 1681 corresponding to their capacity to spread information together. Then for the resourceAllocation we have nearly 97%. It's a high score that is normal because they have almost all their acquaintances in common, so almost all these neighbors are "famous". Finally, the adamic adar of 10,75 computes the same as previously but because it is using logs, it increases the effect of very connected common neighbors.

3.6. Community detection

Community detection algorithms are used to evaluate how groups of nodes are clustered or partitioned, as well as their tendency to strengthen or break apart. In this part, we will calculate triangle count, local clustering coefficient, label propagation and louvain on the Teacher2018 projection.

```
CALL gds.triangleCount.stream("Teacher2018")
YIELD nodeId, triangleCount
RETURN gds.util.asNode(nodeId).prefix, SUM(triangleCount) as
triangleCount
ORDER BY triangleCount DESC
```

Table | Text | Code

	gds.util.asNode(nodeId).prefix	triangleCount
1	"Mrs."	346
2	"Ms."	186
3	"Mr."	103
4	"Teacher"	17
5	"Dr."	0
6	"Mx."	0

Here we can observe that women represent the biggest number of triangles. and we note that the nodes are not stable and it is not balanced and fairly distributed.

Then, for the local Clustering Coefficient on the same cypher projection as before. We can observe that it is diverging for the 3 first ones with an infinity value. That's explained by the fact that the graph that we are working on is heavily connected. One node has a lot of neighbor nodes in common with the others. And that's keeping with our previous result on common neighbors. Resultantly, we can say that the teachers know each other very well and especially for the 3 main prefixes that are represented here.

```
CALL gds.localClusteringCoefficient.stream("Teacher2018")
YIELD nodeId, localClusteringCoefficient
RETURN gds.util.asNode(nodeId).prefix,
SUM(localClusteringCoefficient) as localClusteringCoefficient
ORDER BY localClusteringCoefficient DESC
```

Table | Text | Code

	gds.util.asNode(nodeId).prefix	localClusteringCoefficient
1	"Mrs."	Infinity
2	"Ms."	Infinity
3	"Mr."	Infinity
4	*null*	Infinity
	"Teacher"	5.166666666666667

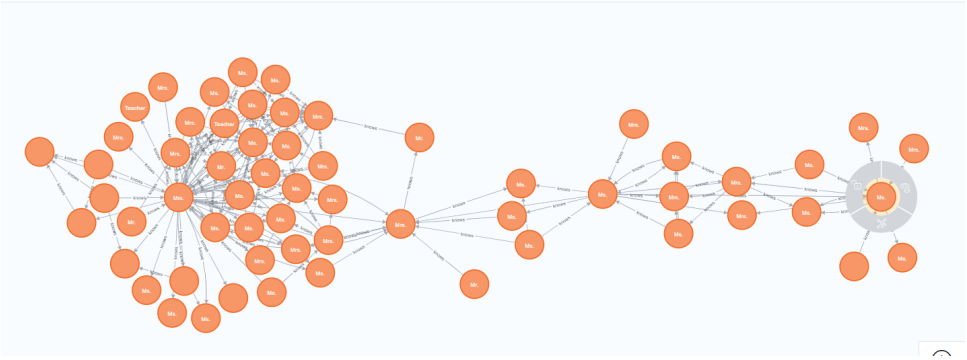
For label propagation, we obtained a large community with 34423 teachers followed by 26199 other women. In total, it's the largest community of this projection in 2018 because these women represent more than 60k members (the following community is 10 times smaller).

```
CALL gds.labelPropagation.stream("Teacher2018")
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).prefix, collect(distinct
communityId) as communityId,count(communityId)
ORDER BY communityId
```

	gds.util.asNode(nodeId).prefix	communityId	count(communityId)
1	"Mrs."	[0, 1, 2, 3, 4, 7, 9, 12, 13, 15, 16, 18, 19, 20, 21, 24, 25, 18526, 30, 31, 32, 33, 34, 36	34423
2	"Ms."	[5, 6, 10, 11, 14, 17, 22, 18609, 28, 29, 37, 40, 43, 45, 46, 47, 50, 52, 54, 57, 59, 60,	26199
3	"Mr."	[8, 23, 35, 50500, 74, 82, 85, 98, 110, 120, 176, 180, 189, 194, 197, 202, 207, 213, 2	7815
4	"Teacher"	[51, 76, 127, 178, 220, 344, 356, 439, 497, 536, 539, 555, 892, 976, 1034, 1073, 112	1117
5	"Dr."	[4122, 4285, 9508, 12183, 12246, 19257, 23322, 23618, 33038, 37482, 39203, 4533	19
6	"Mx."	[18720, 39001, 57158, 58893, 65718, 67553]	6

3.7. Pathfinding

Path finding algorithms find the shortest path between two or more nodes or evaluate the availability and quality of paths. This is the graph that we created to use the Dijkstra algorithm.



```
CALL gds.graph.create(
    'Mono-partite2',
    'Teacher',
    'knows',
    {
        relationshipProperties: 'cost'
    }
)
```

nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	createMillis
{ "Teacher": { "properties": { }, "label": "Teacher" } }	{ "knows": { "orientation": "NATURAL", "aggregation": "DEFAULT", "type": "knows", "properties": { "cost": { "property": "cost", "aggregation": "DEFAULT", "defaultValue": null } }	"Mono-partite2"	109993	67291	608

Dijkstra algorithm:

```
MATCH
(source:Teacher{id:"88d99ccd29c97bf2e91d244f745652c5"}),
(target:Teacher{id:"adcfbc342457bd9017666bdb55382c48"})
CALL gds.shortestPath.dijkstra.stream(
    'Mono-partite2', {
        sourceNode: source,
        targetNode: target,
        relationshipWeightProperty: 'cost'
    })
YIELD sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
    gds.util.asNode(sourceNode).id AS sourceNodeName,
```

"sourceNodeName"	"targetNodeName"	"totalCost"	"nodeNames"	"costs"	"path"
"88d99ccd29c97bf2e91d244f745652c5"	"adcfbc342457bd90176 66bdb55382c48"	7.0	["88d99ccd29c97bf2e91d244f745652c5","60c2df8cd5425e7e0115351 5f14330e4","04f68ffd8bfb690f84dba9793dbe d5f1","cbbe6d4fb0c14330d815c91b7bf85f2f ,"e72310cfbb6e940616 04aa28a6266510","84d716ec3f04e38734339d9 b92207e6e","898bc1cbc9f4cb5b16b774aa7feb	[0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0]	[{"date":"2013-12-09","id":"88d99ccd29c97bf2e91d244f745652c5","prefix":"Ms."},{"date":"2010-12-03","id":"60c2df8cd5425e7e01153515f14330e4","prefix":"Mrs."},{"date":"2013-12-17","id":"04f68ffd8bfb690f84dba9793dbed5f1","prefix":"Ms."},{"date"

This is the shortest path given by the Dijkstra algorithm to go

- from the node id “88d99ccd29c97bf2e91d244f745652c5”
- to the node id “adcfbc342457bd9017666bdb55382c48”

We can see in the column path in the result, the path with the different nodes we need to visit to reach the target node with a total cost of 7.

All Shortest paths Algorithm:

```
CALL gds.alpha.allShortestPaths.stream(
        {
                nodeQuery:"MATCH (t1:Teacher)-[r:knows{state:'California',year:2017}]-()  RETURN DISTINCT id(t1) AS id",
                relationshipQuery:"MATCH (t:Teacher)-[r:knows{state:'California',      year:2017}]-(m)
                RETURN id(t) as source, id(m) as target, r.cost as cost",
                relationshipWeightProperty: "cost"
        })
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
WHERE gds.util.isFinite(distance) = true
```

	source	target	distance
1	"0d1e4f18a7e4d90a806239c0b9c23d92"	"33f5783210d5929bcae3b1c14a0b1275"	3.0
2	"0d1e4f18a7e4d90a806239c0b9c23d92"	"9d62b0fd0b8500bbd6206441aa043ce7"	3.0
3	"0d1e4f18a7e4d90a806239c0b9c23d92"	"9d871f2ae5acf24d04c00484c8ecef90"	3.0
4	"0d1e4f18a7e4d90a806239c0b9c23d92"	"aa4198820e134ce01739eb11b6820e7f"	3.0
5	"0d1e4f18a7e4d90a806239c0b9c23d92"	"deb7efe125a9203799a3585a949d2764"	3.0
6	"0e9423eeb6ee7a7d1ffbbd100fb2fa5f"	"33f5783210d5929bcae3b1c14a0b1275"	3.0

We can see the shortest distance that separates a source node from a target node.

3.8. Centrality

Centrality algorithms are used to determine the importance of distinct nodes in a network. The Degree Centrality algorithm can be used to find popular nodes within a graph. It measures the number of incoming and/or outgoing relationships from a node, depending on the orientation of a relationship projection.

We note for this mono-partite graph the 5 most popular teachers and their scores, therefore we can consider them as famous teachers.

```
CALL gds.degree.stream('Mono-partite2')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).id AS id,score
ORDER BY score DESC LIMIT 5
```

	id	score
1	"d15b52f5901293d5b2925fa687122851"	36.0
2	"da1ca9a90a8b46442e7cd7fa93f71f5c"	32.0
3	"78caa48d3a83228858efc95453941c9f"	30.0
4	"2218dda84eee109cf2f16cc415d10e9f"	29.0
5	"898bc1cbc9f4cb5b16b774aa7feb4838"	29.0

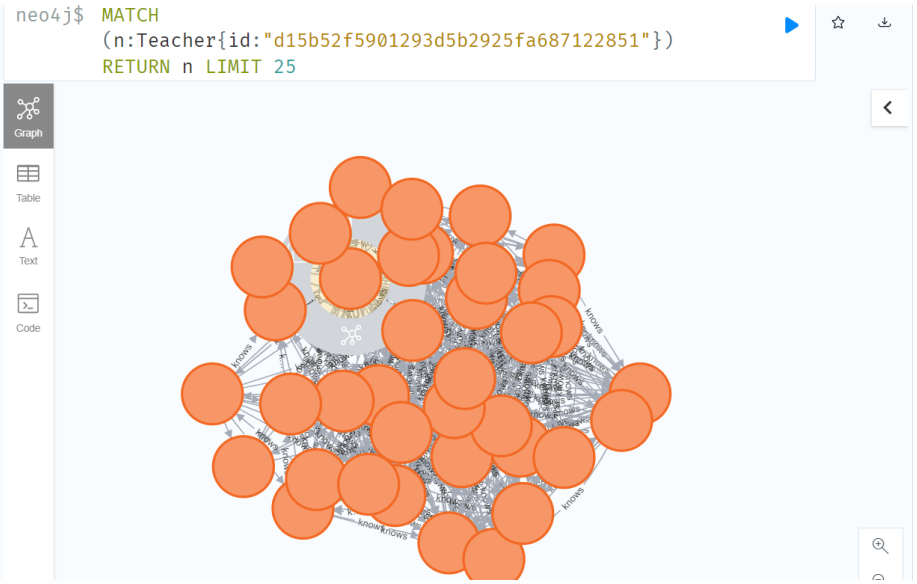
After presenting the direct measure of influence often called Degree Centrality. We will then focus on PageRank, on the other hand, that shows how to look at the influence of nodes in consideration of their neighbors, connections, their importance and the influence each one of those neighbors has, which makes it more iterative.

PageRank is computed by either iteratively distributing one node's rank over its neighbors or by randomly traversing the graph and counting the frequency of hitting each node during these walks. The algorithm counts the number and quality of links to a page, which determines an estimation of how important the page is. higher volume of links from other influential pages. We used the PageRank algorithm that measures the importance of each node within the graph, based on the number of incoming relationships and the importance of the corresponding source nodes.

```
CALL gds.pageRank.stream('Mono-partite2')
YIELD nodeId, score
RETURN DISTINCT gds.util.asNode(nodeId).id AS id, score
ORDER BY score DESC
```

	id	score
1	"3089d7b1d405b69a3dc9bde3c5c6da7f"	8.099573535481722
2	"2e5c5f5af9bcc565e1cf18ba657f5a80"	6.516807535425761
3	"9513b21d5cc6007d87b603ddcd608a60"	6.105798059688296
4	"9351fe89157685802bcc1c524299c204"	5.169532589307293
5	"ac348e52f81eec502987fc0cabcd5da9"	4.8254522338190995

As we can see, the id “d15b52f5901293d5b2925fa687122851” has the highest PageRank because it has incoming links from a lot of teachers. Also, it’s not only the number of incoming links that is important, but also the importance of the teachers behind those relations. Indeed, when we tried to visualize the node, it was very hard due to the huge number of neighbors as we see here :



Conclusion

At the end of this project, we reached the final goal of using different algorithms on graphs from a dataset. The different algorithms that had to be used have all been successfully executed. So, we can only be satisfied that we met the expectations of the project and have completed it.

We were happy to experiment, for the first time, a graph and mining project in which we were able to get a brief overview of a professional project in the field of data science. Moreover, the realization of this project has been more than useful to us. It allowed us to learn real techniques for the future and brought us a concrete learning experience.

At first, it allowed us to develop, to better understand and to put into practice the knowledge that we acquired during the beginning of our course that we have acquired during this beginning of the year. Furthermore, this project allowed us to expand our capacities to manage our time, to collaborate and to work in groups in order to adopt the best working methods.

However, being still beginners, many difficulties were encountered and many points can still be improved. This made us realize that there is still a lot to learn and that we need to keep working on it. The first difficulties we encountered were at the beginning of the project. We felt a bit lost since we didn't really know where to start. Thanks to the teachers and the courses, we were fastly able to start our project on a good basis.

Lastly, we also had difficulties in understanding the algorithms. Finally, the major difficulty was during the interpretation of the last algorithm, we could not find real interpretations to this problem and that precisely waits (/remains) to be improved. Concerning the coding of this algorithm, we must have had a lack of analysis. Another point to improve concerns the interpretations that we consider sometimes too vague and sometimes too descriptive but we imagine that it is a problem that is solved with time and experience.