LAB 1

PROPERTY GRAPHS

Liam James Glennie England
Clara Molins Rosés
16.03.2023

A Modeling, Loading, Evolving

A.1 Modeling

The following image shows the graph we decided to create:

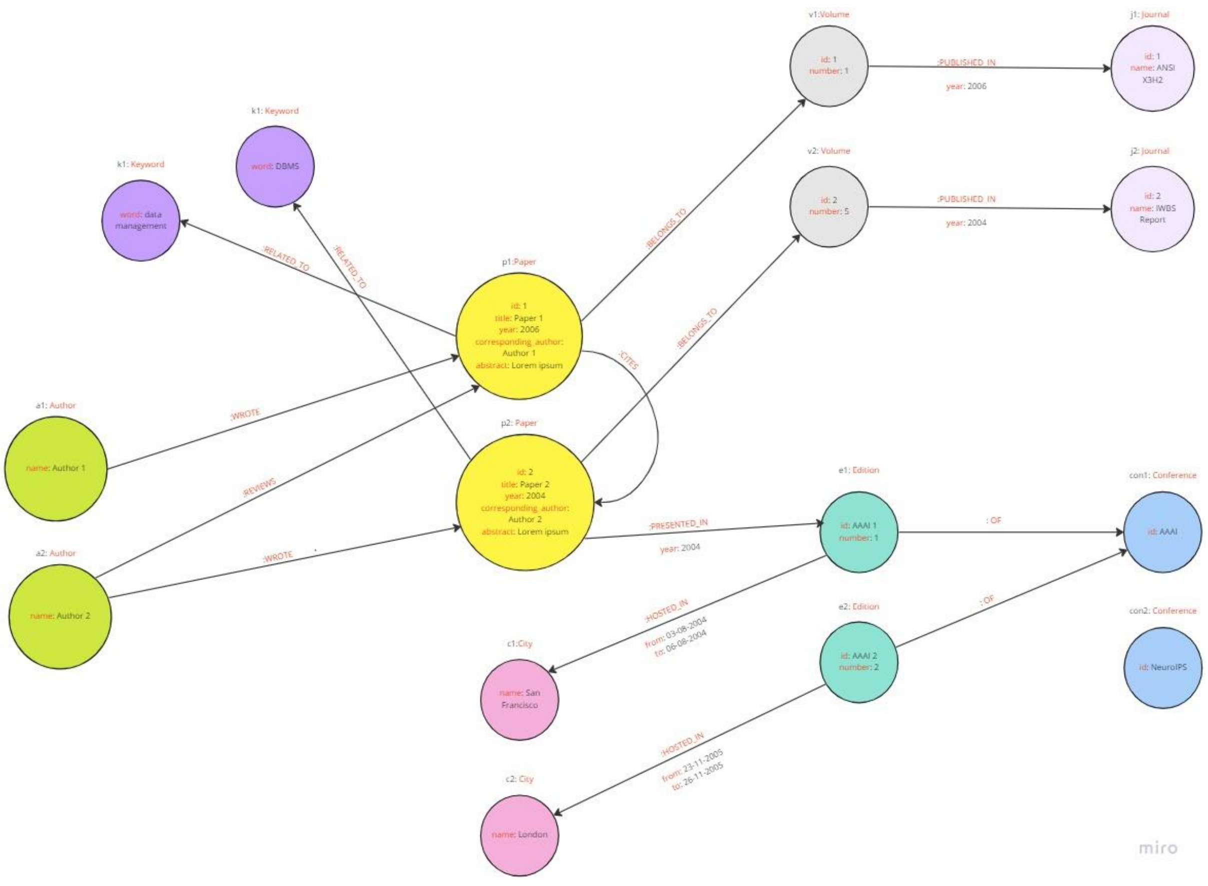


Figure 1: Initial graph.

The words in red color correspond to the schema of the graph, whereas the ones in black correspond to instances. We have decided to represent each node type with a different color. Figure 1 can be observed in better detail in this link.

Regarding the design of the graph, it seemed logical that entities such as **conferences**, **papers** and **journals** should be nodes as they are the main factors of the problem. Authors could have been a property of paper nodes. However, this would have had a detrimental effect on the performance of the queries involving them as to find the papers written by one author, we would have to visit all the papers in the graph. Having authors as nodes also aids maintainability, which will be proven when adding reviewers to the graph.
One of the queries requested that we retrieve authors that have presented papers in at least four different editions. To achieve this efficiently, we have decided to use **edition** as a node. To follow a similar structure, we have also decided to convert **volume** into a node. Additionally, we have thought that the database user may be interested in retrieving the papers from particular editions and volumes. We have also assumed that they may be interested in retrieving the editions of conferences hosted in a particular city. This could be useful in deciding where to host the next conference.

Finally, keywords could have a property of the paper node, however this would have complicated the creation of communities and consequently finding the top 100 papers within a community.

A.2 Loading

To populate the graph, we downloaded the DBLP data. When converting it from XML to CSV with the command in the code block below, it generated many different CSV files and we decided to only work with 'dblp_article.csv'. This dataset contained most of the data we were interested in considering the model proposed in Section A1 such as paper names, authors, journal names, etc. However, it still missed more or less 50% of the needed data.

```
Unset
python XMLToCSV.py --annotate --neo4j data/dblp.xml data/dblp.dtd dblp.csv
```

Note that we kept a subset of approximately 1000 papers of the original dataset to ensure that we had enough data to answer all the queries whilst not having too much data that would slow down the development of the project.

Although we had some attributes, most of them needed to be synthetically generated. Starting with **citations**, each paper could have up to thirty citations. Papers cannot cite themselves, and neither can two papers cite each other. Let's say *Paper1* cites *Paper2*, *Paper2* cannot cite *Paper1.* Additionally, a paper cannot cite a paper from the future. Next, each paper was assigned a random set of two to six **keywords** of one topic. There were three topics to choose from. For further detail please refer to *dataSampler.ipynb*. We must mention that papers belonging to the first journal of the dataset were forced to only have keywords from the database topic and only be cited by papers of that same journal. This was done to guarantee that queries from future sections always returned a result.
The **corresponding author** for each paper was determined by the first author that appeared in the attribute authors of that same paper. For example, if *Paper1* has as authors *a1*|*a2*|*a3*, the corresponding author would be *a1.* It must be noted that we have assumed that the name of the authors is their identifier as there were no duplicates in the dataset and no true identifier for them.
Each paper is **reviewed** between three and five authors as loosely stipulated in the assignment document. We have guaranteed that an author cannot review their own paper.
We have attempted to recreate the **ISSNs** of journals (id in journal nodes) and their **volume IDs** through random numbers ensuring that all identifiers are unique.
Next, a random number of papers, around 50% of the dataset, were chosen to have been presented at a **conference**. This percentage was chosen to ensure that there would be enough papers presented at conferences for future parts of this project. Each conference has its own **editions**, one for each **year**. Each edition was held in a **city** during a specific **period of the year**. We created a dictionary whose keys were a sequence of years from the year of the first publication in the dataset to 25 years in the future from now. For each year, we have another dictionary whose keys are the five different conference names we chose. The values of this dictionary are the edition, city and time period it was hosted. We created this dictionary, to ensure that two papers presented in the same conference and year, were

presented in the same edition, city and time period. Here is an example of the structure of the dictionary:

```python
conf = {'2012': {'AAAI': {'EditionXX': {'number':XX,,
                                        'city':city,
                                        'start_date':start_date},
                                        'end_date':end_date}}}}
```

A.3 Evolving

We were requested to modify the graph model we had created to store a textual review and decision for each review in our graph. Of course, we did not have this data available to us, thus we created it.

To begin with, we retrieved all the reviewers and the papers they had reviewed from our database. We consider every row from this retrieved data to represent a review. Next, we created another dictionary containing the decision of the review as keys, and as values a list of corresponding textual reviews. Therefore, for each review, we chose a random decision, and depending on this decision, we chose a random corresponding textual review. That is, if the decision chosen was *Accepted*, the textual review would be a positive review of the paper. This dataset of reviews was saved into a .csv for modifying the graph as we will cover shortly. For more detail, please refer to evolving_create.ipynb.
Having created the data, we then had to modify the [:REVIEWS] relationship, as it was already created, to include two new attributes: **decision** and **review**. We retrieved all the [:REVIEWS] relationships from the database and modified them to include the corresponding values for the new attributes from the data previously created. Inserting a new node type for reviews didn't seem logical as it wouldn't have optimised any query requested.

Additionally, we were requested to include affiliations in the model of our graph database. Therefore, following a similar process to before, we created a dataset containing all the authors of our database, the **affiliation** they are associated with, and the **type of affiliation** it is. We chose two types of organizations *University* and *Company*. For each organization we had a list of potential values. Thus, for each author we chose a type of organization they would be affiliated with, and then based on this we randomly chose the name of the said organization from the precomputed list of candidates. We must note that each author can only be affiliated with one organization. We chose to implement affiliation as a node over a property of author because the user could be interested in finding all the authors affiliated with a certain organization. Additionally, if more attributes regarding the affiliation were requested to be added, it would be much easier and efficient to modify the graph if they are attributes.
Finally, to load this into our database, we first created the affiliations with their name and type. Next, we created the relationships between the authors and their corresponding affiliations using the data we had synthetically created.

All the changes we applied to the graph can be found in Figure 2 in the Annex marked in red. For a better view of the image, check this link.

B Querying

The queries we created for each petition are the following:

Petition 1: Find the top 3 most cited papers of each conference.

```
Unset
MATCH (p1:Paper)-[:CITES]->(p2:Paper)-[:PRESENTED_IN]->(:Edition)--(c:Conference)
WITH c, p2, count(*) as cites
ORDER BY c.name, cites DESC
WITH c, collect({title: p2.title, citations: cites}) as papers
UNWIND papers[0..3] as paper
RETURN  c.id  AS  conference_id,  paper.title  AS  paper_title,  paper.citations  AS
num_citations
```

The first step in this query is a pattern matching to find all the papers in each conference that are cited at least once by another paper. Then in the 'WITH' it counts the number of citations of each of these papers and orders them in descending order in the 'ORDER BY'. Afterwards, it collects the top 4 cited papers for each conference and returns the conference ID, paper title, and number of citations for each of these papers.

Petition 2: For each conference find its community: i.e., those authors that have published papers on that conference in, at least, 4 different editions.

```
Unset
MATCH (aut:Author)-[:WROTE]->(p:Paper)-[:PRESENTED_IN]->(e:Edition)--(c:Conference)
       WITH aut, c, count(e) as num_editions
       ORDER BY num_editions DESC
       WHERE num_editions >= 4
       RETURN c.id, aut.name AS author_name,  num_editions
```

The first thing this Cypher query does is find all authors that have written at least one paper that has been published in an edition of a conference. Then, it groups the authors and conferences by the number of editions the author has presented in, sorts the results in descending order, and filters for authors who have presented in at least 4 editions of a conference. Finally, as wanted, it returns each conference id and those authors that have published papers in at least four of its editions.

Petition 3: Find the impact factors of the journals in your graph

```
Unset
MATCH (j:Journal)
       UNWIND [1990, 1999, 2000, 2001, 2002, 2003, 2004, 2018] as year
       WITH j, year
       MATCH
(p1:Paper)-[cit:CITES]->(p2:Paper)-[:BELONGS_TO]->(v:Volume)-[pub:PUBLISHED_IN]-(j)
       WHERE pub.year = year
       WITH j, year, count(cit) AS num_citations
       ORDER BY j, num_citations
       WITH j, year, num_citations
       MATCH (p3:Paper)-[:BELONGS_TO]->(v:Volume)-[pub1:PUBLISHED_IN]-(j)
       WHERE pub1.year > year - 3 AND pub1.year < year
       WITH j, year, num_citations, count(p3) AS num_publications
       RETURN     j.name,     j.id,     year,     num_citations,     num_publications,
toFloat(num_citations)/num_publications
```

This query matches journals and then unwinds a list of years which can be modified to the user's needs. Then, for each journal and year, it matches papers that are cited by other papers that belong to a volume published in that same journal during the corresponding year.
After that, it groups the results by journal and year, counts the number of citations for each paper as num_citations, and sorts the results by journal name and number of citations in ascending (default) order.
Next, it matches papers that belong to a volume published in the same journal within the 2 years before the current year in the list, counts the number of publications for each journal, and returns the name and ID of the journal, year, number of citations, number of publications, and the two-year impact factor of the journal.

Petition 4: Find the h-indexes of the authors in your graph

```
Unset
MATCH (a:Author)-[:WROTE]->(p1:Paper)<-[:CITES]-(p2:Paper)
WITH a, p1, count(*) as num_citations
ORDER BY num_citations DESC
WITH a, collect(num_citations) as list_num_citations
WITH  a,  list_num_citations,  [x  in  range(0,  size(list_num_citations)-1)  WHERE  x  <
list_num_citations[x] | x+1] AS res
RETURN a.name, res[-1]
```

This query matches authors who wrote papers that are cited by other papers. Then, it groups the results by author and paper, counts the number of citations for each paper as num_citations, and sorts the results num_citations in descending order.
Afterwards, it collects the number of citations for each paper into a list and calculates the number of times the value in each position is less than the position itself. This gives the h-index of the author, which is the largest number h such that the author has published h papers with at least h citations each. Finally, it returns the name of the author and their h-index.

C Recommender

In this part of the project, we aim to create a reviewer recommender for editors and chairs. In particular, the focus is put on identifying potential reviewers for the database community.

First of all, we defined three different research communities and their corresponding associated keywords. In the code below you can see their names and the keywords that define them.

```
Unset
research_communities = [['database', ['data management', 'indexing', 'data modeling',
'big    data',    'data    processing',    'data    storage',    'data    querying']],
['artificial_intelligence', ['machine learning', 'neural networks', 'deep learning',
'natural language processing', 'computer vision', 'reinforcement learning', 'expert
systems', 'knowledge representation', 'genetic algorithms', 'bayesian networks']],
['cybersecurity',  ['cybersecurity',  'network  security',  'information  security',
'vulnerability',  'penetration  testing',  'threat  detection',  'malware  analysis',
'security policies', 'risk management', 'cybercrime']]
]
```

Once these communities were defined, we created a node for each of them and an edge between each of the keywords and its corresponding community called [:DEFINES], as keywords define communities. The following query shows how we added this data to the graph. Note that keywords were incorporated during loading in Section A.2.

```
Unset
UNWIND [['database', ['data management', 'indexing', 'data modeling', 'big data',
'data processing', 'data storage', 'data querying']],['artificial_intelligence',
['machine learning', 'neural networks', 'deep learning', 'natural language
processing', 'computer vision', 'reinforcement learning', 'expert systems', 'knowledge
representation', 'genetic algorithms', 'bayesian networks']], ['cybersecurity',
['cybersecurity', 'network security', 'information security', 'vulnerability',
'penetration testing', 'threat detection', 'malware analysis', 'security policies',
'risk management', 'cybercrime']]] AS rsrch_com
    MERGE (c:Community {name: rsrch_com[0]})
    WITH c, rsrch_com
    UNWIND rsrch_com[1] as keyword
    MERGE (key:Keyword {word: keyword})
    MERGE (key)-[:DEFINES]->(c)
```

First, it accesses each research community inside the list we previously showed. Then it creates the three community nodes, and then given a community and its list of keywords it creates a [:DEFINES] edge between the community and each of the keywords. Note that the 'UNWIND' statement is used twice because the data fed to the query is a list of lists.

The following step was to find the conferences and journals related to the database community, in other words, write a query to extract journals and conferences where at least 90% of the papers contain one or more keywords related to the database community. In addition, as the found data must be added to the graph, we added a new type of edge to the graph called [:LINKED_TO] relating the extracted conferences and journals to the database community. The query we wrote is the following:

```
Unset
MATCH
(n:Conference|Journal)-[:OF|PUBLISHED_IN]-(:Edition|Volume)-[:PRESENTED_IN|BELONGS_TO]
-(p:Paper)-[:RELATED_TO]-(:Keyword)-[:DEFINES]-(c:Community)
WHERE c.name = "database"
WITH n, c, count(distinct p) as num_papers_com

MATCH (n)-[:OF|PUBLISHED_IN]-(:Edition|Volume)-[:PRESENTED_IN|BELONGS_TO]-(p1:Paper)
WITH n, c, num_papers_com, count(p1) AS total_num_papers
WHERE 100*toFloat(num_papers_com)/total_num_papers >= 90
WITH n, c, num_papers_com, total_num_papers

MERGE (n)-[:LINKED_TO]->(c)

RETURN n.id, n.name, num_papers_com, total_num_papers,
100*num_papers_com/total_num_papers
```

This query first finds through pattern matching all papers of each conference that have at least a keyword from the database community. Afterwards, again through pattern matching, it finds the total number of papers that belong to each conference/journal. Finally, it calculates the percentage of papers from each conference/journal that belong to the database community and keeps only those that have a percentage equal to or higher than 90%.
Lastly, the results are added to the graph. In particular, the [:LINKED_TO] edges as we mentioned in the previous paragraph.

The next step consisted in identifying the top 100 papers of the conferences/journals belonging to the database community. In other words, we want to find the 100 papers with the highest page rank provided the number of citations from the papers of the same community. To find them we wrote the following query. Note that the final part of the query adds the results to the graph through edges between the top 100 (or fewer) papers and the database community and it also prints the results of the query so that we can see exactly what it returns. In the corresponding python script, it can be seen that we transform the format of the returned data into a dataframe so it is easier to interpret.

```
Unset
MATCH
(c:Community)<-[r:LINKED_TO]-(:Conference|Journal)<-[:OF|PUBLISHED_IN]-(:Edition|Volum
e)<-[:PRESENTED_IN|BELONGS_TO]-(p1:Paper)<-[:CITES]-(p2:Paper)-[:PRESENTED_IN|BELONGS_
TO]->(:Edition|Volume)-[:OF|PUBLISHED_IN]->(n:Conference|Journal)
WHERE c.name="database"
WITH c, n, p1
MATCH (c)--(n)
WITH c, n, p1, count(*) AS cites
```

```
ORDER BY cites DESC LIMIT 100
MERGE (p1)-[:TOP_100]->(c)
RETURN c.name, p1.id, p1.title, cites
```

This query uses pattern matching to find all papers that belong to a conference/journal linked to the database community such that are cited by at least one paper belonging to the same community. This pattern had to be achieved by performing two different matches because Neo4j does not allow the exact same relationship to appear twice in one matching. Afterwards, it makes a count to know the number of citations (in database community papers) of each paper, orders them in descending order and keeps the first 100 using the ORDER BY command. Lastly, as previously stated, it adds the [:TOP_100] edges between these papers and the database community.
The RETURN in the end, just as before, is so that we could easily check the results.

The final step was to write a query to detect 'database' gurus: very reputed authors that have at least two papers among the top 100 found with the previous query. Once we knew who they were, we linked them with the database community through an edge called [:GURU_OF]. The corresponding query is the following:

```
Unset
MATCH (c:Community)-[r:TOP_100]-(p:Paper)<-[:WROTE]-(a:Author)
           WHERE c.name = "database"
           WITH a, c, count(r) as numPapers
           WHERE numPapers > 1
           MERGE (a)-[:GURU_OF]->(c)
           RETURN a.name, numPapers
```

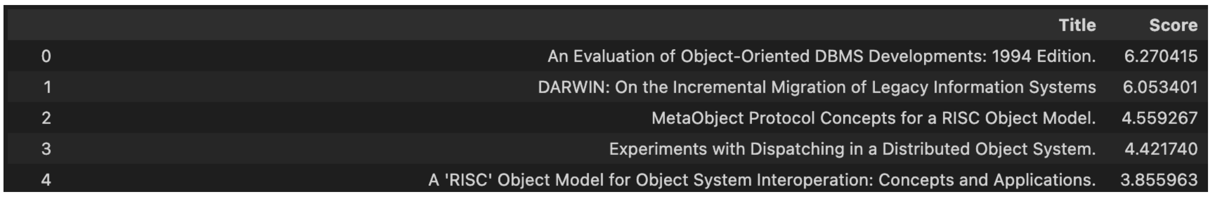
The query starts by finding all authors we have written papers belonging to the top 100 of the database community through a 'MATCH'. Then, in the 'WITH', we count the number of papers each author has in this top 100 and name it numPapers. In addition, we keep the community node to later use it in the 'MERGE'. Finally, the 'WHERE' clause right after keeps only those authors that have more than one paper in the top 100.
The 'MERGE' clause adds the results to the graph by means of edges. To finalize, as before, we added a return so that we could see the results of the query.

D Graph algorithms

Graph algorithms can be used to exploit the structure of a graph and extract useful information from it, such as the importance of a node, calculating the similarity between two nodes, detecting community within the graph… In our solution, we have opted for using graph algorithms to discover the importance of a paper based on the papers that cite and the similarity between the two authors.

PageRank

PageRank is an algorithm developed by Google which has been used to determine which websites are more important than others. In our case, we have applied it to discover the papers that are most important in our graph. The importance of a paper is measured by the number of papers that cite it and their importance. Here are the top 5 papers in our graph. Note that the results may change if the data is regenerated again.



	Title	Score
0	An Evaluation of Object-Oriented DBMS Developments: 1994 Edition.	6.270415
1	DARWIN: On the Incremental Migration of Legacy Information Systems	6.053401
2	MetaObject Protocol Concepts for a RISC Object Model.	4.559267
3	Experiments with Dispatching in a Distributed Object System.	4.421740
4	A 'RISC' Object Model for Object System Interoperation: Concepts and Applications.	3.855963

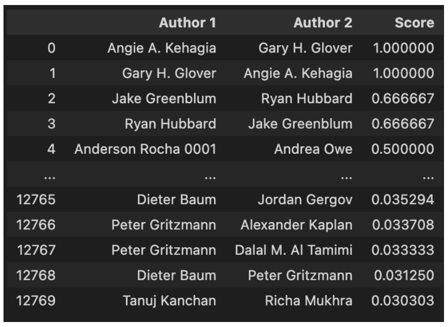
Table 1: PageRank algorithm results.

The most important paper according to the PageRank algorithm is "An Evaluation of Object-Oriented DBMS Developments: 1994 Edition.". Figure 3, which can be found in the Annex, shows that this paper, marked by the red rectangle, is not the one with the most incoming CITED relationships, nonetheless, it is still the most important paper due to it having important papers cite it, like the second most important paper shown in Table 1.
We should mention that in an attempt to avoid the algorithm entering a never-ending loop, we have set the parameters of maxIterations and dampingFactor to 20 and 0.85, respectively.

Finally, Neo4j includes an adapted version of the PageRank algorithm called ArticleRank. PageRank works on the basis that low-degree nodes have a higher influence than high-degree ones, meaning that a paper cited by one other paper could potentially have more influence than a paper that is cited by many papers. ArticleRank lowers the influence of low-degree nodes by lowering the scores being sent to their neighbours in each iteration. We did not use this algorithm as it was not specified in the assignment, although it may be interesting to study further the differences between PageRank and ArticleRank, and compare the results they yield.

Node Similarity

The node similarity algorithm computes the similarity between two nodes based on the neighbouring nodes they share in common. Therefore, two nodes are considered to be similar if they share many neighbours. As part of our solution, we have opted to use this algorithm to calculate the similarity between two authors based on the relationships [:WROTE], [:REVIEWED], :[GURU_OF] and [:AFFILIATED_TO]. Essentially, we have used all the relationships that an author can have in our graph to calculate similarity.



	Author 1	Author 2	Score
0	Angie A. Kehagia	Gary H. Glover	1.000000
1	Gary H. Glover	Angie A. Kehagia	1.000000
2	Jake Greenblum	Ryan Hubbard	0.666667
3	Ryan Hubbard	Jake Greenblum	0.666667
4	Anderson Rocha 0001	Andrea Owe	0.500000
...	...	...	...
12765	Dieter Baum	Jordan Gergov	0.035294
12766	Peter Gritzmann	Alexander Kaplan	0.033708
12767	Peter Gritzmann	Dalal M. Al Tamimi	0.033333
12768	Dieter Baum	Peter Gritzmann	0.031250
12769	Tanuj Kanchan	Richa Mukhra	0.030303

Table 2: Node similarity algorithm results.

Table 2 showcases pairs of authors and the similarity score between them. The pair of authors with the maximum similarity, both wrote the same paper and are affiliated with the same organisation as seen in Figure 4 found in the Annex. On the other side of the spectrum, we find that Tanuj Kanchan and Richa Mukhra have the lowest similarity score of all authors that have at least one neighbour in common. Figure 5 illustrates the relationship between these two authors. As we can see, they only have one similarity: they wrote a paper together. Additionally, Tanuj Kanchan has many more neighbouring nodes than Richa Mukhra, reducing the similarity score between them.

ANNEX

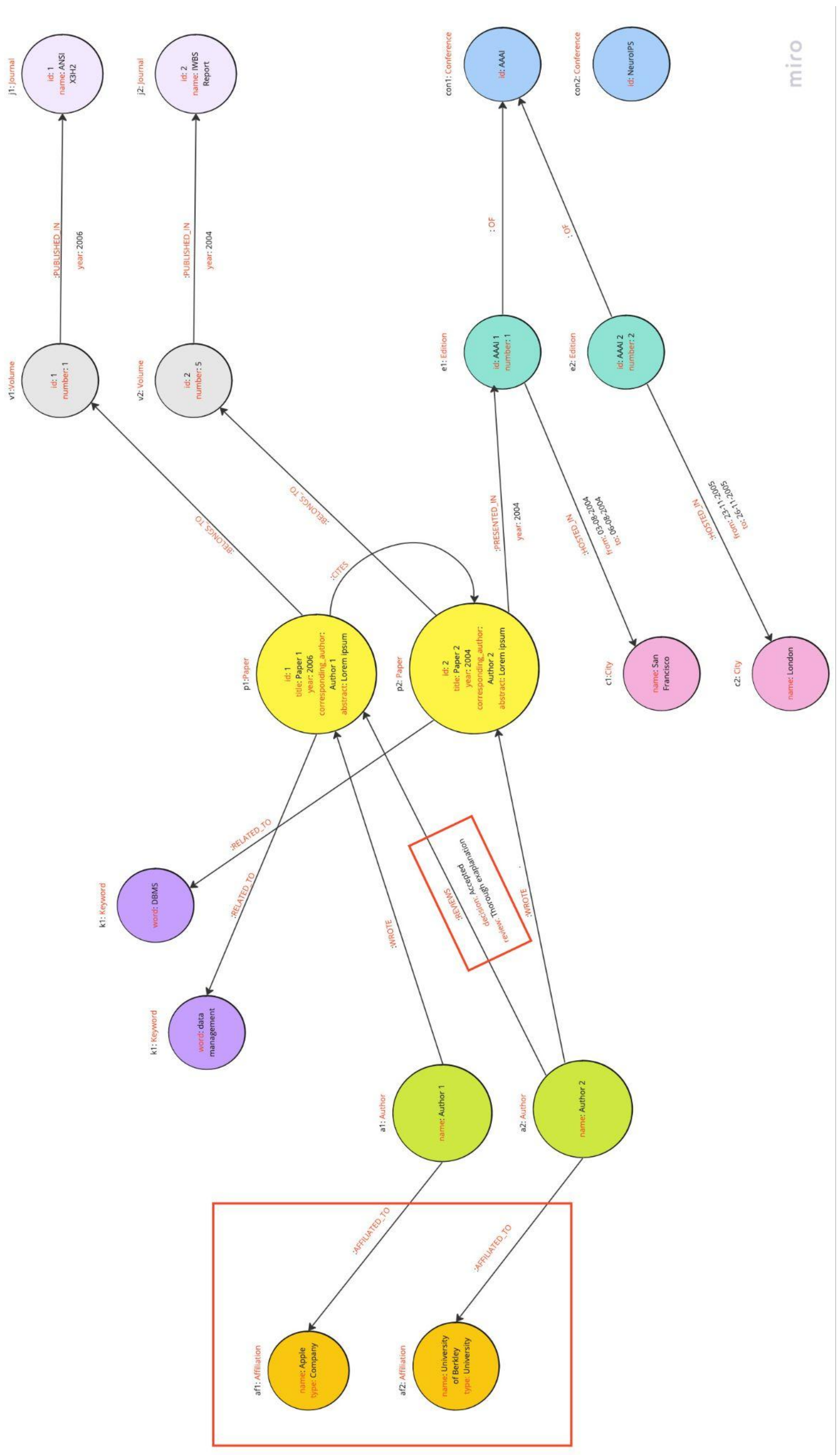


Figure 2: Evolved Graph.

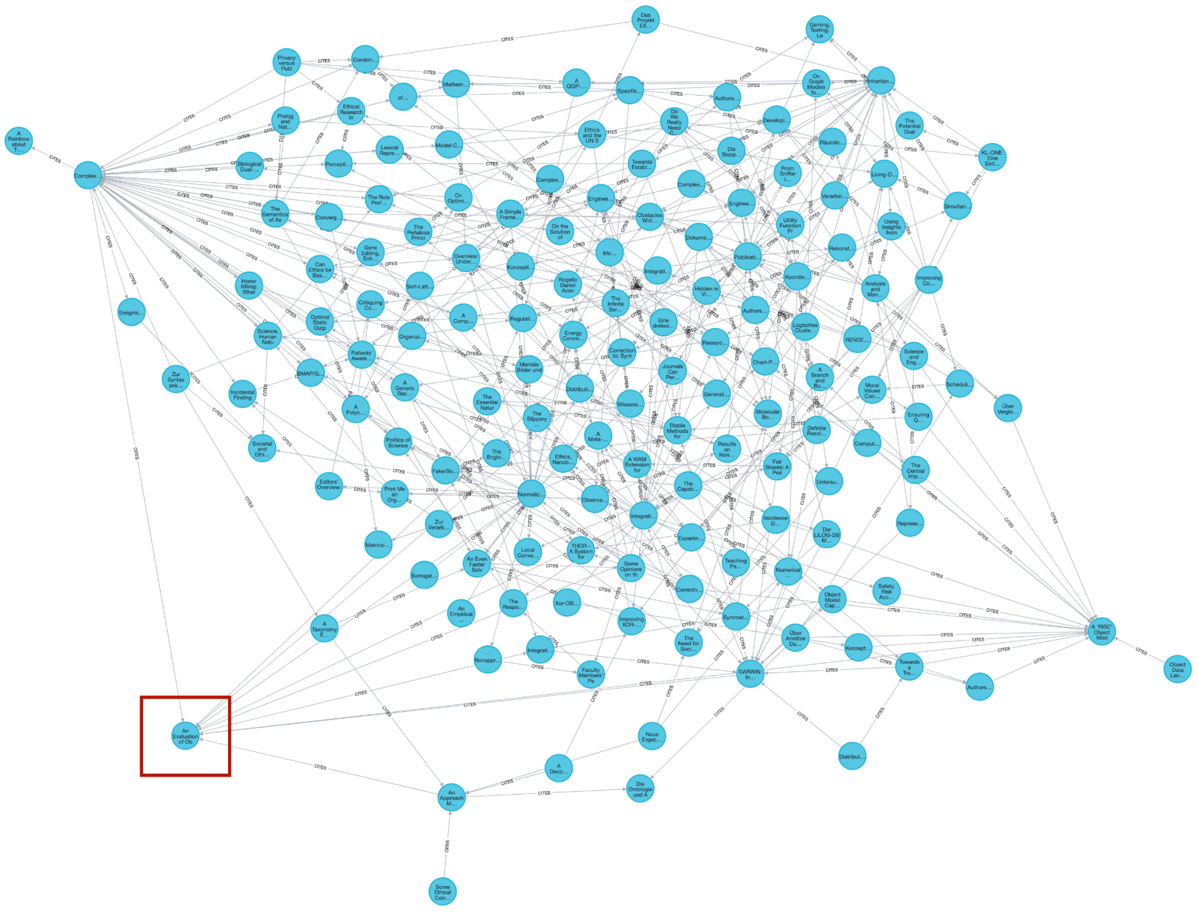


Figure 3: Page Rank results shown in a graph.

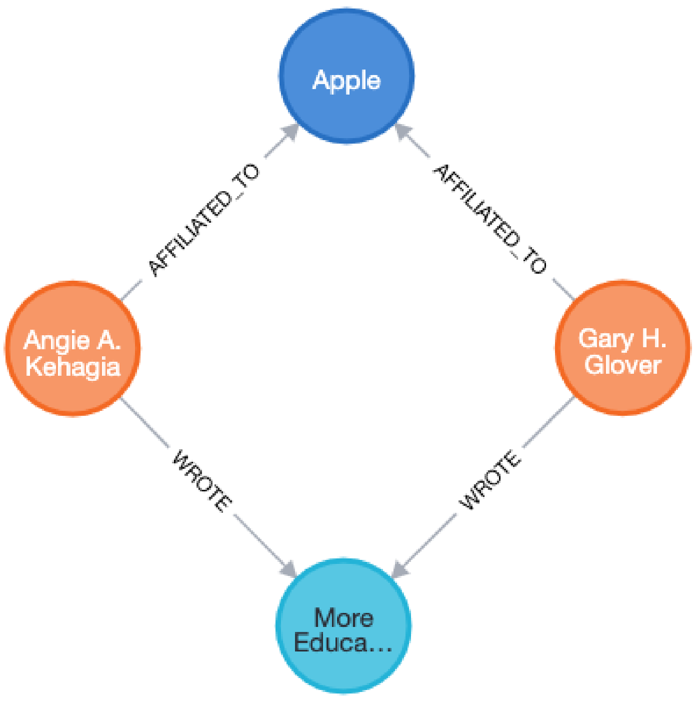


Figure 4: Authors with maximum node similarity score.

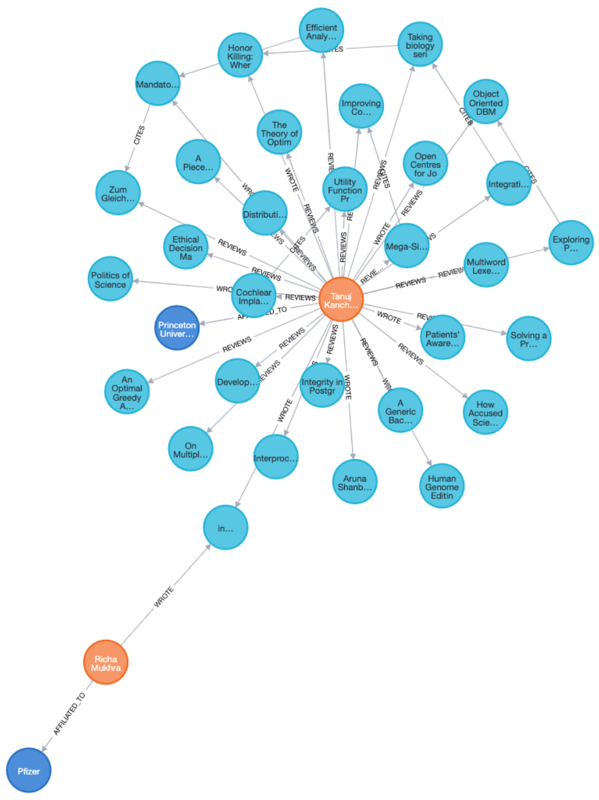


Figure 5: Authors with minimum node similarity score.