Similarity Modeling on Citation Network

Cs 424 Term Project
Group 9

Selçuk Gülcan
Halil İbrahim Özercan
Pelin Deniz
Eren Ekinci

21.12.2015

1 Introduction

The problem we tried to solve is to find scientific articles with most similar citations. Articles with most similar citations may sound ambiguous because articles can be similar in a way that they cited similar articles or they are cited by similar articles. These definitions are called bibliographic coupling and co-citation. “Papers are bibliographically coupled when different authors cite one or more papers in common. On the other hand, co-citation analysis is based primarily on identifying pairs of highly-cited papers.“[1] In other words, bibliographically coupled documents have reference to common papers but co-cited documents are referenced by common papers. Figure - 1 shows this difference on a simple citation graph.

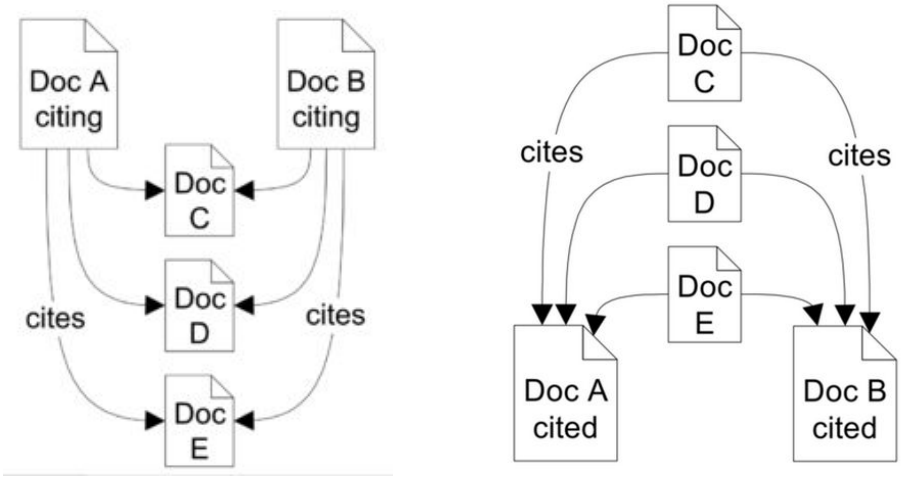

Figure 1 - Bibliographically coupled documents vs co-cited documents respectively

We decided to do bibliographically coupling analysis (left graph on figure 1) on citation network so our goal is to find article pairs whose references are similar.

1.1 Dataset Explanation

We chose our dataset as Arxix HEP-TH Dataset (High Energy Physics Theory)[2] from Stanford Network Analysis Project. This dataset offers a citation graph which is from the e-print arXiv. I contains all the citations within a dataset of 27.770 papers and 352.807 edges. In the dataset, citations from outside papers or citations to outside papers removed so dataset contains information about paper with given topic (High energy physics theory). If a paper A cites paper B, the network contains a directed edge from A to B. Edges are given as node pairs in each line.

[1] http://garfield.library.upenn.edu/papers/drexelbelvergriffith92001.pdf
[2] https://snap.stanford.edu/data/cit-HepTh.html

Other than citation network, this dataset includes abstract information of articles and information about paper submission time to Arxix. However, we did not use this part of dataset since our aim to find similarity between articles by their citation graph.

First 10 lines of dataset is given below (Figure - 2) to show format of data. First 4 lines of data file consists of some comment about data. Other lines contain 2 numbers, these 2 numbers are id numbers of articles representing a citation. For example in the first line, 1001 and 9304045 numbers are located, It means article with id of 1001 cites article with id of 9304045. File contains 352.807 such lines to show edges. vertices are not shown explicitly but they can be retrieved from edges. Unique id numbers in edges may represent vertices of graph.

```
# Directed graph (each unordered pair of nodes is saved once):
Cit-HepTh.txt
# Paper citation network of Arxiv High Energy Physics Theory
category
# Nodes: 27770 Edges: 352807
# FromNodeId    ToNodeId
1001  9304045
1001  9308122
1001  9309097
1001  9311042
1001  9401139
1001  9404151
```

Figure 2 - First 10 lines of dataset

One thing to be noted about these id numbers is that id numbers are not consecutive. For example, id 1001 exists in the file but id 1002 may not appear.

2 Approach

We can consider vertices as sets. Each vertex represent a set of edges. Similarity of articles (vertices) is simply similarity between these sets. This approach leads us to the same solution. Since we want to compute similarity between sets, jaccard similarity can be used to determine similarity as a numerical value.

Jaccard similarity of two sets is computed by dividing size of intersection set by size of union set.

$$\mathrm{J}(A,B) = \frac{\left|A \cap B\right|}{\left|A \cap B\right|}$$

Figure 3 - Jaccard Similarity

First of all, brute force method is considered as a method but clearly, it is an ineffective approach. In brute force method, jaccard similarity of all vertex pairs should be computed. There are 27770 vertices so 385,572,565 set comparison should be done. It basically gives us $O(n^2 m)$ time complexity which is inefficient.

In similarity modeling part of our textbook, content based similarity is discussed. In textbook, 3 steps while finding content based similarity are explained. In first step, shinglings are created from documents that are considered as set of words. After that big sets are converted to small signatures in minhashing step. At last step, locality sensitive hashing is applied to generate set of candidate pairs. By doing so the number of set comparison is reduced to the number of candidate pairs.

We would like to follow very similar approach because finding content based similarity and our problem has many common points. Both problems consist of sets and the goal is to find set pairs with higher jaccard similarity. Both problems can utilize locality sensitive hashing. The only difference is shingling step. Content based similarity modeling hash words into shinglings to generate set from character sequence. In our problem however, sets are given already so there is no need to perform shingling step.

Our approach can be divided into these parts:

1. Preprocess raw data to get edge sets.
2. Perform minhashing to generate signatures from sets.
3. Do locality sensitive hashing (LSH) to find candidate pairs.
4. Compare these candidate pairs to find their similarity.
5. Postprocess final data.

The following section explains these parts in detail.

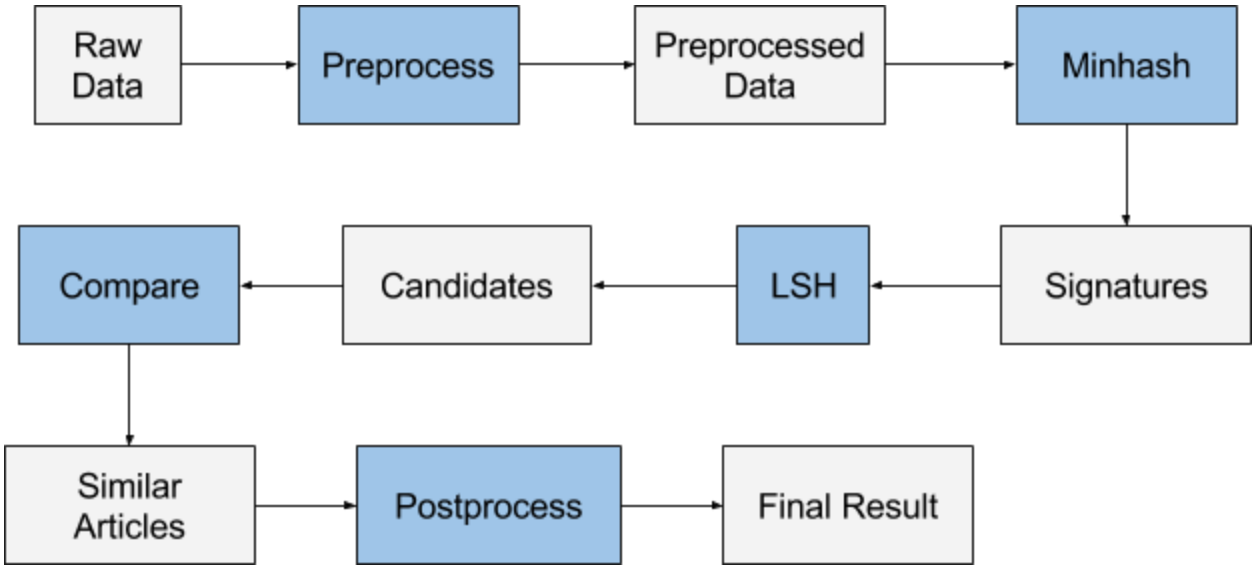

Figure 4 - Flowchart of our approach; blue boxes represent processes, white boxes represent data files

3 Implementation

This section covers implementations steps mentioned in previous section. The project is implemented in C language on linux platform. No additional library is used. Project code can be found on Github.

3.1 Preprocessing

As mentioned in dataset explanation part, id numbers in the data file are not consecutive. This situation may create problem while processing data because vertex access operation is highly used. With raw data, It takes O(n) (or O(logn) if we sort data) time to access a node. Searching the whole node array each time we try to access a node is not very efficient. Therefore we map these irregular id numbers to consecutive numbers from 0 to 27769 so that access operation takes O(1) time. We first sorted the nodes in ascending order and then map them into consecutive numbers. After that we performed all other operations to these abstract id numbers. At the end of execution, we can remap these numbers to their original ids (We need to store map table to revert id numbers).

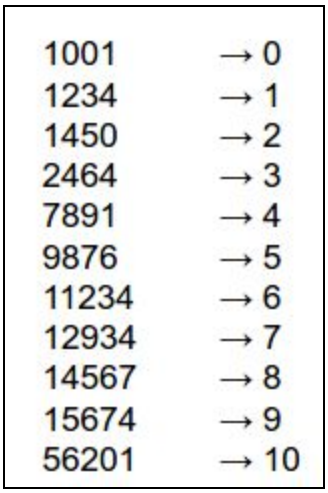

Figure 5 - Mapping example

After mapping id numbers, we convert list of id pairs to list of vectors. Before and after preprocess data slices are given below.

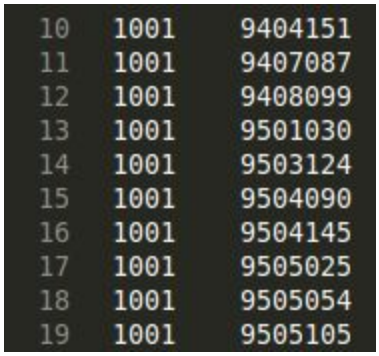

Figure 6 - Raw data (Data before this step)

Line numbers in figure 7 represent node ids. First integer in line shows how many edges this node has. Other numbers in the line represent node ids that this node has an edge to. For example Line 6 in figure 7 says that Node 6 has only 1 outgoing edge and this edge points node 27220. In a way, we generated sets whose similarity should be computed.

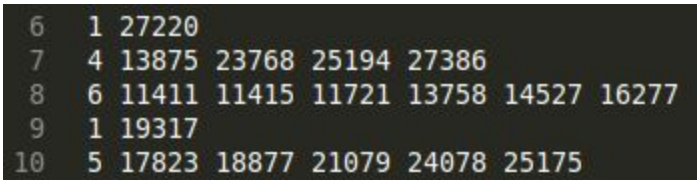

Figure 7 - Data after preprocessing (Data after this step)

3.2 Minhashing

Minhashing is a technique to convert a set into signature sequence. It is very useful technique to find jaccard similarity of two sets because jaccard similarity of two set is equal to jaccard similarity of two signature vectors of infinite length. Of course infinitely long signatures are not practical to implement but even with smaller signature vectors (such as K=100) similarity of signature vectors gives us pretty good estimation. Another reason we perform minhashing i s to generate fixed length sets for locality sensitive hashing. Without minhashing band sizes and the number of bands are not equally distributed among documents in lsh step, this leads to inconsistency in result because set size of nodes affects result. In textbook minhash method is described as following : To minhash a set represented by a column of the characteristic matrix, pick a permutation of the rows. The minhash value of any column is the number of the first row, in the permuted order, in which the column has a 1. It is easy to visualize this method in adjacency matrix of graph. Therefore, in previous versions of our project, we converted network into adjacency matrix and then performed minhashing. However, converting data to adjacency matrix is costly and this matrix is taking a lot of space because of the sparseness of adjacency matrix. In addition to this, we do not need to compute adjacency matrix to generate signatures. For these reasons, we gave up computing adjacency matrix and we generated signatures directly from edge sets (preprocess.txt). To do so, we needed to permute edges K (the number

of signatures generated) times and find the the number of the first row which column has value of 1. Instead of this costly method we can generate K hash functions, then hash edges with these functions and pick lowest hash value.

First we needed to generate K = 100 random hash function. For this purpose, we used hash function of $h(x) = ((a * x + b) \ mod \ P) \ mod \ N$. a and b numbers are random integers. N is the number of vertices in graph. P is smallest prime number bigger than N. In our implementation N is 27770 and P is 27773. After generating hash functions, we hash values and pick lowest one. Pseudo code of this step is demonstrated in figure 8.

```
for( i = 0 to VERTEXSIZE)

    for( j = 0 to |Node[i]|)

        for each hash_function h(x) = a*x + b

            hash_value = ((a * Node[i][j] + b) % P) % VERTEXSIZE

            if( signature_matrix[i][k] > hash_value)
                signature_matrix[i][k]=hash_value;
```

Figure 8 - Pseudo code of signature generation step

The result of this process is 27770 signature vectors of size 100 (K). Result is stored in signature.txt, A slice of this file is given in Figure 9.

```
1   43 269 39 48 195 536 244 109 85 110 2
    714 10 166 296 89 74 557 238 95 365 3
    73 176 0 70 65 158 324 131 27 230 123
    336 86 196 25 118 155 93 25 51 104 18
2   112 310 155 27 65 336 298 944 62 324
    351 34 28 192 1012 155 62 108 163 31
    481 50 419 55 160 382 883 149 264 193
    802 774 150 300 31 560 303 130 384 55
3   3275 24353 15520 14625 6796 6442 2687
    4401 20258 8215 1578 2020 23912 5896
    17299 10213 13091 7483 2301 9508 2011
    20749 20618 21837 22151 9667 1578 738
```

Figure 9 - Data after minhashing. signature.txt (Each line consists of set of signatures)

3.3 Locality Sensitive Hashing

Locality sensitive hashing is key part of our model. It helps us to minimize the number of candidate pairs which should be compared. This is the step which reduce 385,572,565 set

comparison (brute force approach) to less than 10.000 set comparison. In this step, we divide signatures into bands (a part of signature vector), we hash these band and place them according to hash value.

This part is the key of the project and key part of this step is to find a good hash function. We want to hash bands so that same bands are hashed to the same value and different bands are hashed to the different value. We want to minimize clusters. We tried many hash functions to see effect of choosing hash function. First we used associate hash functions like hash_value = signature_1 XOR signature_2 XOR signature_3 … However, these hash functions did not give good distribution as we expected. This is because we want to hash a vector not a set but associative hash function does not respect order, they consider band as set so they are problematic. To overcome this issue, we used the following hash function:

$$h(x) = p^0 * sign[0] + p^1 * sign[1] + p^2 * sign[2] + ... + p^r * sign[r]$$

In function declaration, p is a big prime number. sign is signature vector and r is band size. This function respects for order of signatures so it gives better distribution. After hashing each band with this function, nodes are put into bucket according to their bands' hash value.

This approach may produce false negative candidates which are not similar but indicated as candidate by lsh process. However, we compare these candidates to see if they are actually similar and if not, we can remove from similar pairs list. For our case, it is better to have more false negatives than false positives. This is because, having more false negatives does not decrease performance of our project and more false negatives mean less false positive so it gives us better accuracy. Band size and the number of bands are tuned to balance these values and we decided that band size is 20 and the number of bands is 5.

```
 2  2 15919 17749
 3  3 16795 18660 23742
 4  2 2159 5492
 5  2 22912 25417
 6  2 8745 22926
 7  2 3880 11590
 8  2 8586 20416
 9  2 10076 21950
10  2 5957 25977
11  2 23141 23792
12  2 14015 15657
```

Figure 10 - Candidate pairs (candidate.txt)

The output of this process is candidate.txt. This file contains all candidate pairs computed by this step. Each line represents a bucket. First integer in each line shows how many items the bucket has. Other integers shows ids of nodes in the buckets. For example, in

line 3 of figure 10, first number is 3 so there are 3 nodes in this bucket. These nodes are 16795, 18660, 23742.

3.4 Comparing Candidate Pairs

After lsh, we have candidate pairs, now we have to look these pairs to check if they are actually similar. We determined similarity threshold as 0.8. This step is straightforward, the program computes jaccard similarity of each candidate pairs in candidate.txt, if similarity is above threshold, the pair is printed in file named similar.txt with its similarity value.

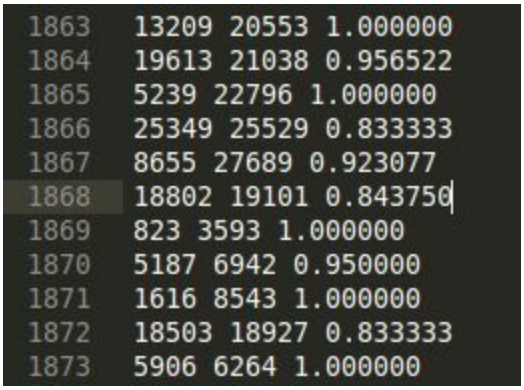

Figure 11 - Similar pairs (similar.txt)

Figure 11 shows a slice of file generated after executing this step. Each line corresponds a similar pair. First two integers shows ids of similar pair and last integer represents jaccard similarity of the pair.

3.5 Postprocessing

Although all work seems to be finished, there is one thing left to do before solving the problem. At the beginning of implementation we changed original vertex ids. Now we have to change ids to their original ids. We kept track how we map ids in map.txt file. By using that file, we generated our final data file and the problem is solved. For convenience, we bundled executables of these steps into a shell script. With the two lines below, the project can be run:

```
./compile.sh
./run.sh
```

It produces all intermediate and final data files in /data folder.

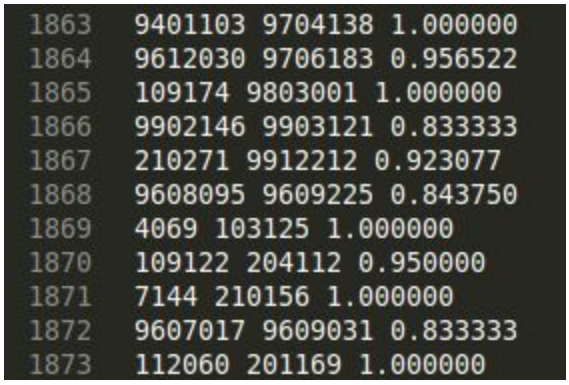

Figure 12 - Final result (final.txt)

4 Analysis & Conclusion

At the end, we accomplished our aim, we found articles with similar citation. Although the code is written for one particular dataset; with a little tweak, the project can be used to find node similarity of any graph. With this project, we found a chance to implement what we learnt in class and learnt lsh application in detail.

There are some observations we made during developing the project:

- Hashing is a powerful tool of computer science. In almost every step of project, we used hashing.
- The power of hashing comes from its hash function. If we can find a good hash function, we can overcome many performance issues like we did in the project by using lsh.
- The number of signatures, band size, the number of bands and the number of buckets are the parameters affecting false negative and false positive rates. We can tune these parameters to find suitable false negative and false positive rates.
- Increasing false negative rate make us compute more candidates and increasing false positive rate decreases our accuracy.
- Minhashing is great to generate small signatures if we have lots of edges but even if we have few edges, lsh can still utilize minhashing.

In terms of teamwork, we can say that work is divided equally. Although each team member has worked in different areas of project, everybody knew what other members were doing. Our general work division was as follows Pelin worked on project report, Selçuk worked on implementation part, Halil was responsible for presentation and Eren did researches and was responsible for team organisation. However, each member helped whenever another member had difficulty on his/her part.