CPA project

Edwin Ansari
Maria Sidko

1 Introduction

Real world implementation of the mathematical notion of graph theory has countless applications in modern computer science, which have become crucial in our daily life, from efficient handling of data to pertinent responses of search engines. Throughout this project, we have attempted to understand, implement and optimize some of those unavoidable algorithms, following the guidelines provided by our professor for the subject. In order to reach that goal, we have decided to implement our code using the Java language, given its reasonable calculation efficiency, in addition to its object oriented paradigm and provided APIs, reaching an optimal middle ground between speed and coding comfort.

2 TME 1 - Handling a large graph

Exercise 1 - Preparation The provided files are comprised of the edges of the graph, so we can easily count the edges by looping through it. The nodes on the other hand should only be counted once, so we used an HashSet for that purpose. The class `Counter` implements this process. Following are the results of the counting on the examples:

Amazon benchmark :	number of vertices	334863
	number of edges	925872
Live journal benchmark:	number of vertices	3997962
	number of edges	34681189
Orkut benchmark:	number of vertices	3072441
	number of edges	117185083

N. B. The last provided benchmark, Friendster, which has 65608366 Nodes and 1806067135 Edges according to the website it was taken from. That gives us $|E| \approx 10^9$ and $|G| \approx 10^9 + 10^7$ Just reading this file and storing it in a simple list structure would take $10^9 \times sizeof(int) = 10^9 \times 4Bytes \approx 4GB$ of RAM and then the whole program that counts it would take on top of it around 5GB of RAM (that is nearly impossible with our basic computers having 8GB of RAM in total). The same reasoning could be applied in terms of time complexity. We were therefore unfortunately unable to use that benchmark throughout the project.

Exercise 2 - Three graph data structures Given that the graphs in the provided files are essentially already stored as a list of edges, the parsing, done by the `GraphLoader Class`, creates automatically a list of edges representation (`GraphList Class`), from which are derived the two other representations, `GraphArray Class` and `GraphMatrix Class`. The number of vertices of the two latter representations is `MAX NODE ID + 1`. In an attempt to improve speed, we decided to replace java Scanner with a buffered `FastReader.java`. Our results were the following:

Amazon :	List of edges List of edges using FastReader Adjacency array Adjacency matrix	1934ms 939ms 89ms OutOfMemoryError
Live-journal :	List of edges List of edges (using FastReader) Adjacency array List of edges (allocating 4500MB of RAM) Adjacency array (allocating 4500MB of RAM)	91363ms 31400ms OutOfMemoryErrorr 32031ms 17712ms
Orkut :	List of edges (using FastReader) Adjacency array List of edges (allocating 5800MB of RAM) Adjacency array (allocating 5800MB of RAM)	OutOfMemoryError OutOfMemoryErrorr 441201ms OutOfMemoryErrors

Space Complexity:
List of Edges : $O(|E|)$
Adjacency Array : $O(|V| + |E|)$
Adjacency Matrix : $O(|V|^2)$

We have the same time complexity for structures reading.

It is noticeable right away that, while the list of edges representation is scaled up reasonably easily, and so is the adjacency array representation, the matrix representation quickly gets out of range, and despite additional memory allocation, reaches the hardware limits of our machines.

Exercise 3 - BFS Diameter results using `GraphDiameter` class:

Amazon	Lower-Bound Lower-Bound time Upper-Bound Upper-Bound time (after Lower Bound executed)	47 2330ms 57 296ms
Live-Journal	Lower-Bound Lower-Bound time Upper-Bound Upper-Bound time (after Lower Bound executed)	21 74739ms 24 10685ms

Exercise 4 - Triangles The intuitive way to calculate the triangles would be for each edge of the graph to compute the intersection of each two nodes. However, due to efficiency issues, some adjustments can be made to that algorithm. In order to avoid registering the same triangle multiple times, as well as to optimize processing time, we can visit the vertices in a non increasing degree order, and only considering neighbor vertices of lower degree. This is implemented by the `TriangleFinder` class

amazon	triangles counting time	667129 2840ms

3 TME2

Exercise 1 - PageRank (Directed Graph) Implementing the Pagerank algorithm in the `PageRank` class we could achieve accurate results after about 5 or 6 iterations, using the Wikipedia database. Here are the top 5 listed pages:

```
max 1 : United States
max 2 : United Kingdom
max 3 : 2007
max 4 : Germany
max 5 : 2006
```

and the 5 bottom:

```
min 1 : Schismogenesis
min 2 : Thomas W. Thompson
min 3 : Thomas W. Thompson
min 4 : List of examples in general topology
min 5 : James Dempsey
```

Exercise 2 - Correlations In order to plot we used linear scales.

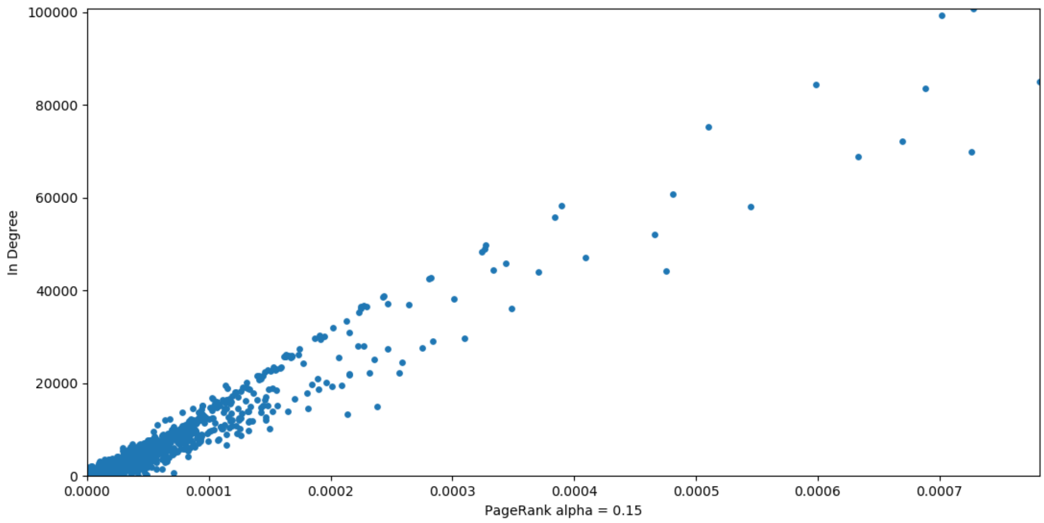

Figure 1: Scatter plot using the Wkikpedia dataset

Intuitively, the more references(In Degree) a node has, the higher its rank is. We can notice that on the scatter plot, as it is somehow linearly correlated to the Pagerank results, which means that it is linked to the ranking, but not identical, given that other factors enter in the Pagerank calculation.

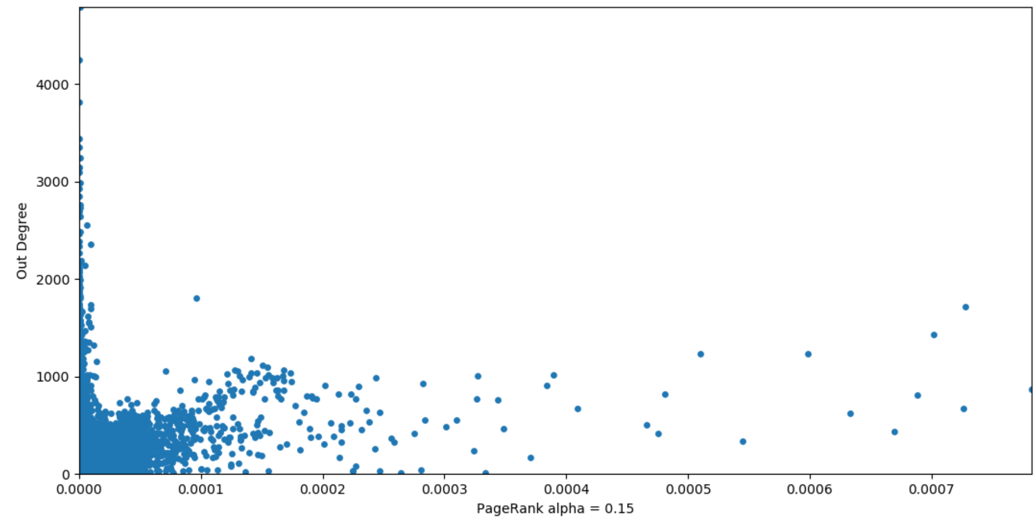

Figure 2: Scatter plot using the Wkikpedia dataset

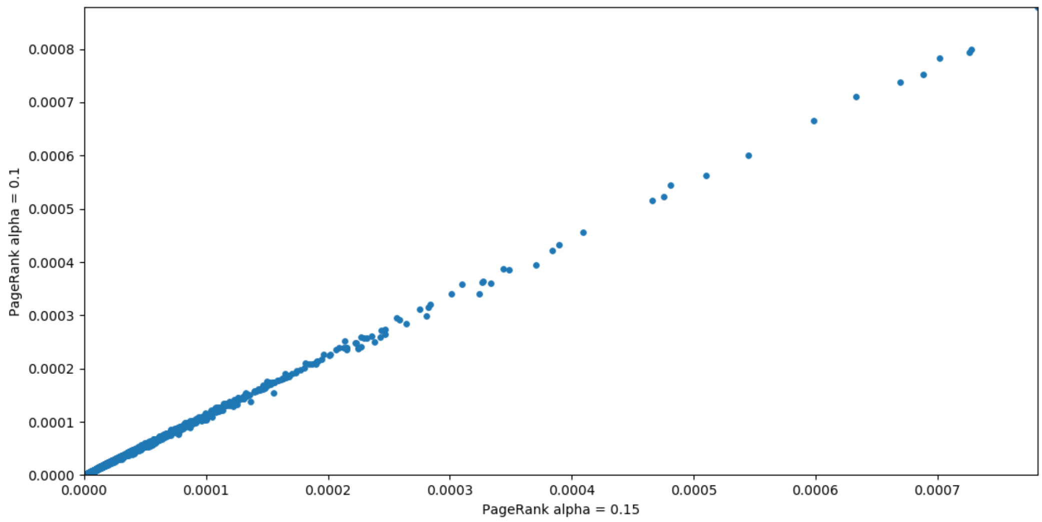

Figure 3: Scatter plot using the Wkikpedia dataset

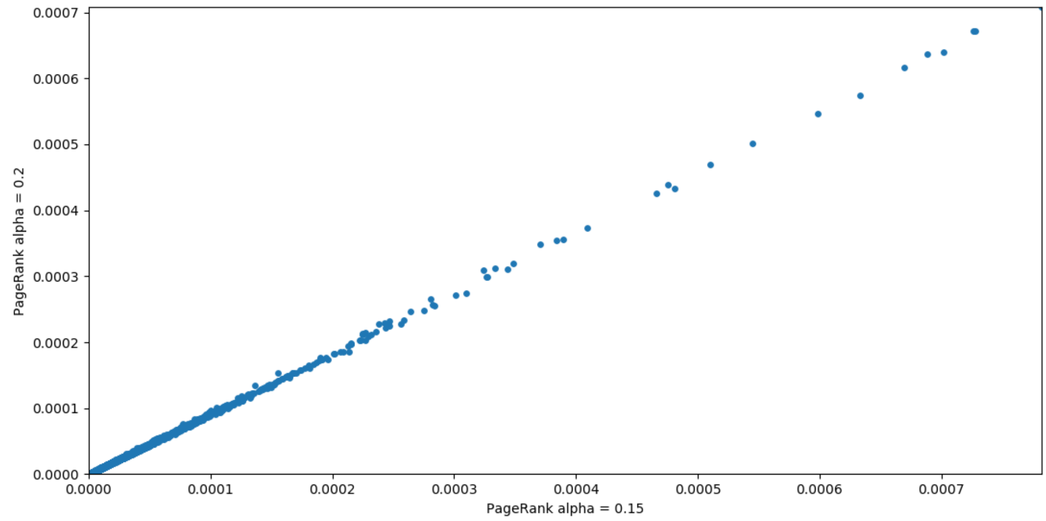

Figure 4: Scatter plot using the Wkikpedia dataset

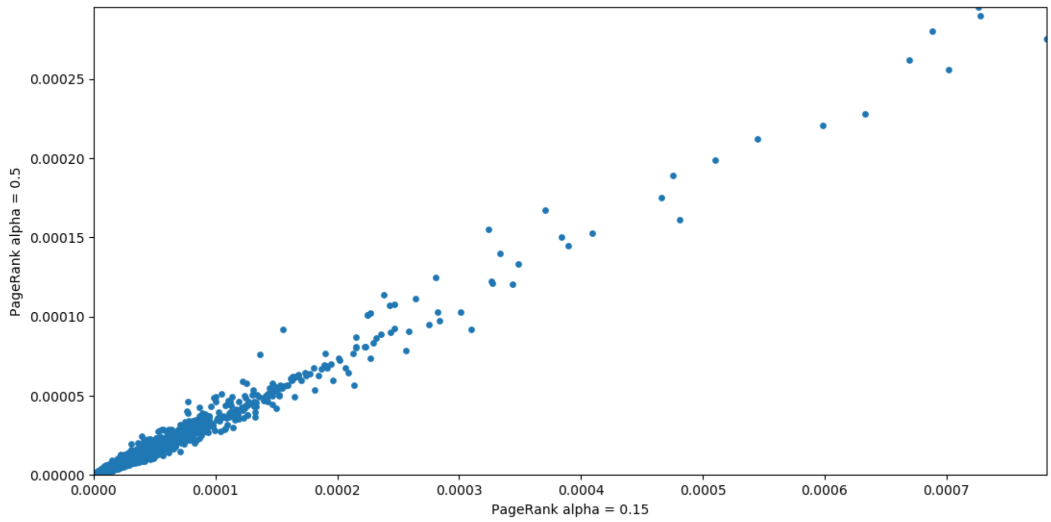

Figure 5: Scatter plot using the Wkikpedia dataset

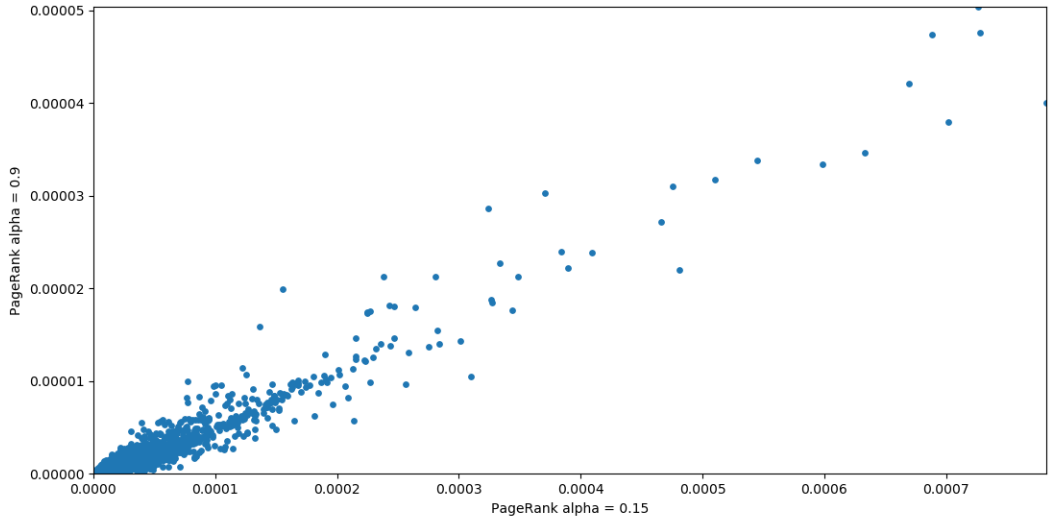

Figure 6: Scatter plot using the Wkikpedia dataset

We can notice that the α value of the PageRank algorithm has an influence on its results, the closer they are the more linear the plot is, meaning that the PageRank results of the two calls are similar. When the values become more distant, the rankings change, leaving the scatter plot more sparse.

Exercise 3 - Personalized Page Rank

Magnus Carlsen: To find very close results to node(Magnus Carlsen) we first found its Page Id : 442682 and then run the *rootedPageRank* using an initial P_0 vector such that:
$P_0(\texttt{magnus\_id}) = 1$
$P_0(i) = 0\ \forall i \neq \texttt{magnus\_id}$

By computing personnalized PageRank using *Magnus Carlsen* as the root node, we obtained the following top and bottom pages:

```
max 1 : Magnus Carlsen
max 2 : Chess
max 3 : Grandmaster (chess)
max 4 : Federation Internationale des echecs
max 5 : Elo rating system

min 1 : Transmission and infection of H5N1
min 2 : Ital
min 3 : (Sittin' on) the Dock of the Bay
min 4 : Ministry of New and Renewable Energy (India)
min 5 : Superclub
```

Chess Boxing:
In order to have a relevant restart vector using a set of subjects (Chess and Boxing in this case) we offer two methods.

1. In this first method we simply set the restarting vector P_0 such that:
   $P_0(\texttt{id}) = \frac{1}{N}$ if $id \in S$
   $P_0(\texttt{id}) = 0\ \forall id \notin \texttt{S}$
   Where S is the set of pages that are linked to any of the related categories and N is their quantity.

   The results using this method are :

```
max 1 : Chess
max 2 : Grandmaster (chess)
max 3 : Federation Internationale des echecs
max 4 : Chess Olympiad
max 5 : United States
max 6 : Boxing
max 7 : International Master
max 8 : Elo rating system
max 9 : World Chess Championship
max 10 : Chess opening

min 1 : Villeray-Saint-Michel-Parc-Extension
min 2 : Jessica Cutler
min 3 : Electrical connection
min 4 : Bass player
min 5 : Caribou-Targhee National Forest
min 6 : Above the Rim
min 7 : Charlesbourg, Quebec
min 8 : Gaius Calpurnius Piso
min 9 : Lisa Opie
min 10 : Persistence
```

2. We fetch, using all the provided files, the number of categories related to the requested subjects each page is linked to, alongside with searching whether the actual names of the pages contain those subjects. We use this information to weight the restarting vector P_0 such that: $P_0(i) = \frac{r+10}{t}$ if $ct = 0$
   $P_0(i) = \frac{r+ct*k}{t}$ if $ct > 0$
   $P_0(i) = 0 \; \forall i \notin \mathtt{S}$

   Where r is the number of appearances of the related subjects in the page categories, ct is number of subjects that are a sub-string of the page name, k is a factor that can be customized in order to reach convergence (we chose 20 in our tests). and $t = \sum_{i=0}^{n} P_0(i)$

   Using this second method we get the following top and bottom results:

```
max 1 : Chess
max 2 : Grandmaster (chess)
max 3 : Boxing
max 4 : United States
max 5 : Federation Internationale des     checs
max 6 : Chess variant
max 7 : Chess Olympiad
max 8 : Chess problem
max 9 : Chess boxing
max 10 : World Chess Championship

min 1 : Pioneer High School
min 2 : Umea IK
min 3 : Guadalupe River (California)
min 4 : Brian Ashton (rugby player)
min 5 : Grammy Award for Best Latin Rock/Alternative Album
min 6 : Brampton Thunder
min 7 : Pump Up the Volume (song)
min 8 : Sigurd Evensmo
min 9 : Ethel Snowden
min 10 : St Edward's School (Oxford)
```

4 TME 3

Exercise 1 - Simple Benchmark Following are some of the generated random graphs :

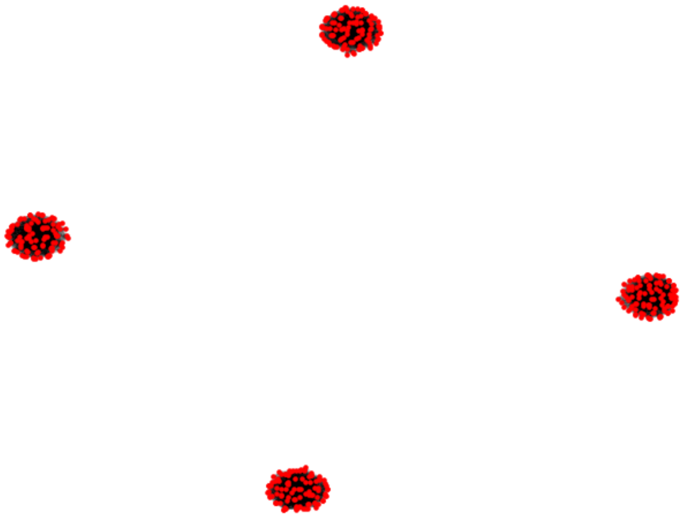

Figure 7: Random generated graph with p = 1.0 and q = 0.0

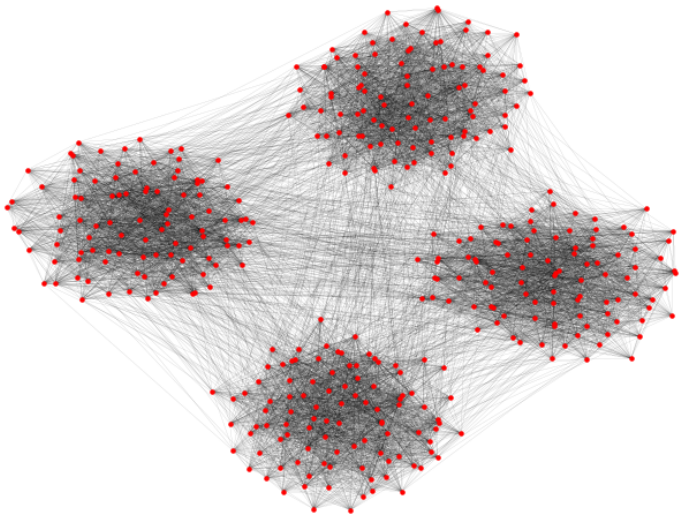

Figure 8: Random generated graph with p = 0.4 and q = 0.02

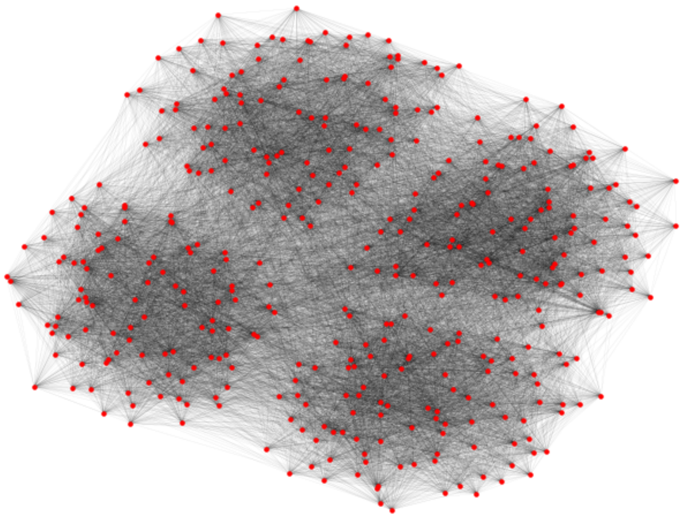

Figure 9: Random generated graph with p = 0.6 and q = 0.1

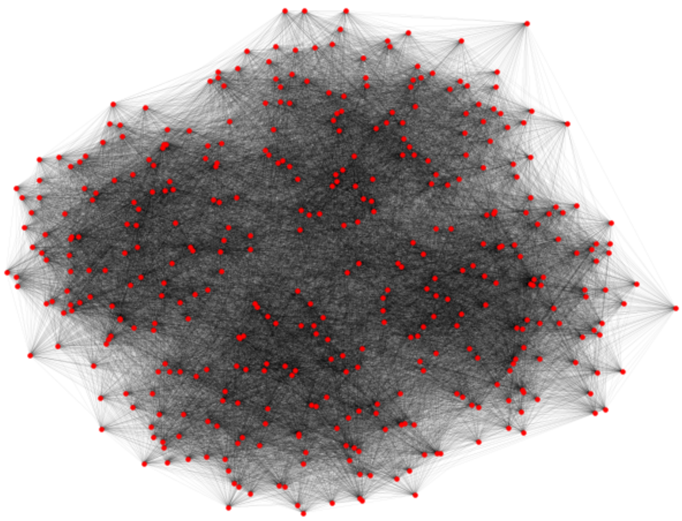

Figure 10: Random generated graph with p = 0.7 and q = 0.2

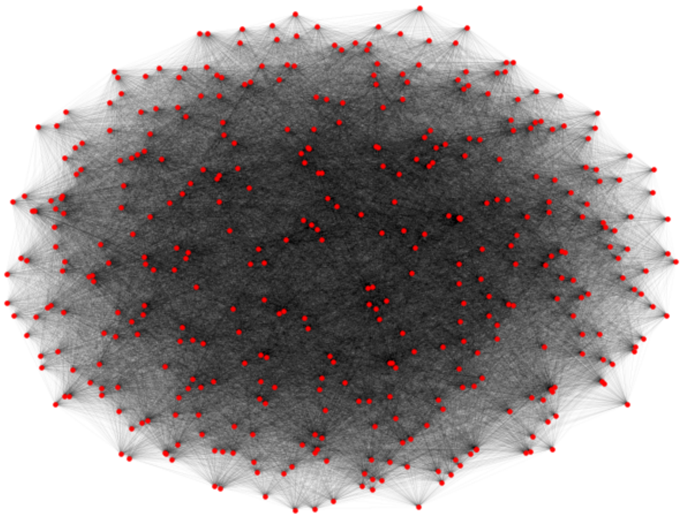

Figure 11: Random generated graph with p = 0.5 and q = 0.5

As we can see on the graphical rendition of the graphs, increasing $\frac{p}{q}$ ratio leads to more noticeable separation in communities, while decreasing it creates a more homogeneous graph with only one big community.

Exercise 2 - Label propagation We have implemented the Label Propagation label in the `LabelPropagator` class. Following are the results of the call of this algorithm of some random generated graphs.

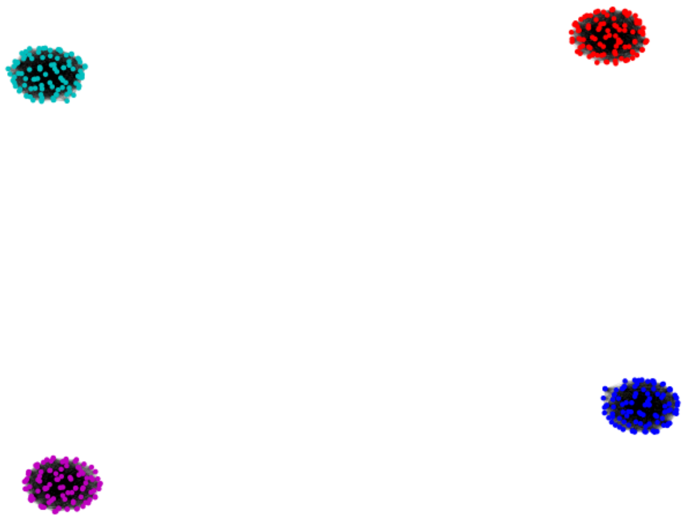

Figure 12: Results of the label propagation algorithm on a random generated graph with p = 1.0 and q = 0.0

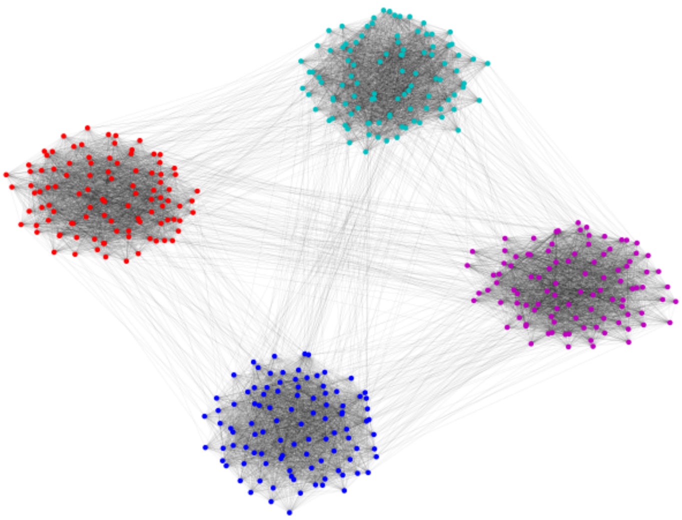

Figure 13: Results of the label propagation algorithm on a random generated graph with p = 0.6 and q = 0.01

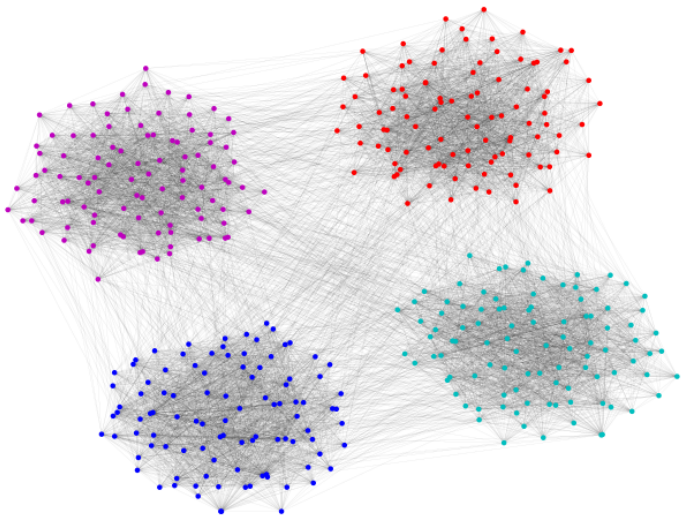

Figure 14: Results of the label propagation algorithm on a random generated graph with p = 0.4 and q = 0.02

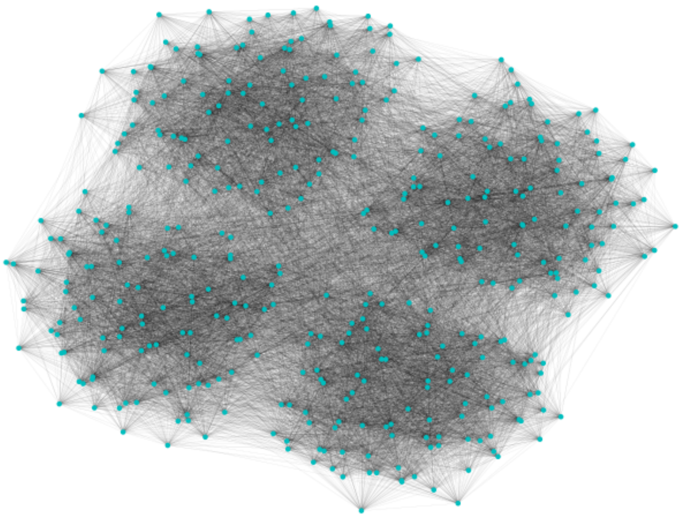

Figure 15: Results of the label propagation algorithm on a random generated graph with p = 0.6 and q = 0.1

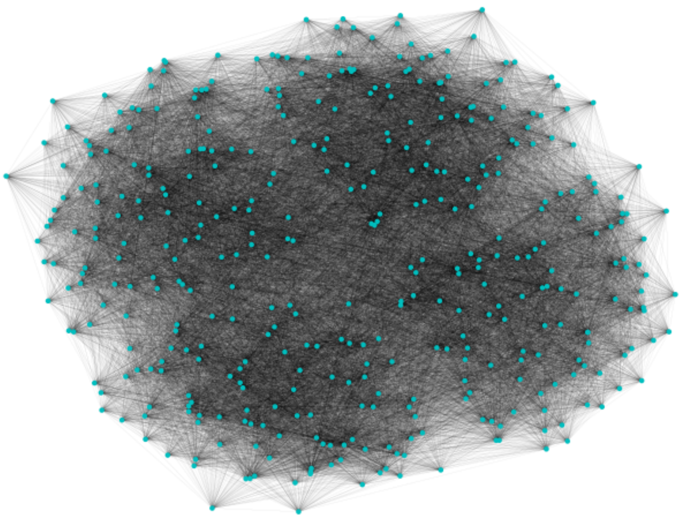

Figure 16: Results of the label propagation algorithm on a random generated graph with p = 0.7 and q = 0.2

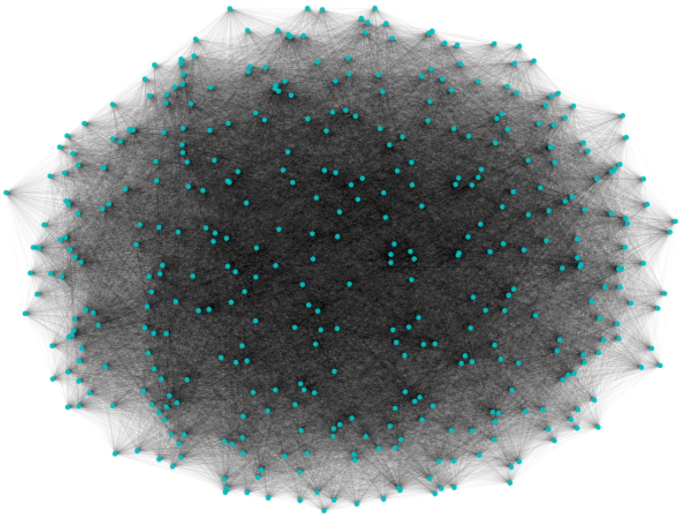

Figure 17: Results of the label propagation algorithm on a random generated graph with p = 0.5 and q = 0.5

While the algorithm is efficient at recognizing communities with a $\frac{p}{q}$ ratio higher than 20, when it decreases it has more trouble and only recognizes one unique community comprising the whole graph.

Exercise 3 - New algorithm For this matter, after searching for different existing algorithms, and in order to experiment on a different approach from label propagation and Louvain algorithms, we decided to turn to divisive approaches, and, after looking more in depth we settled on the Girvman-Newman algorithm. This algorithm computes the communities by successively removing edges from the graph by a certain measure, and then computing the connected components of the graph, which are the communities. The measure, called betweenness is defined as the extent to which that edge lies along shortest paths between all pairs of nodes.The edge with the highest betweenness will be removed at each iteration. We experimented on this algorithm using the `Jung` java API, in the `Clustering` class.
Analysis: The time complexity of the EdgeBetweennessClusterer is: $O(kmn)$ where k is the number of edges to remove, m is the total number of edges, and n is the total number of vertices. For very sparse graphs the running time is closer to $O(kn^2)$ and for graphs with strong community structure, the complexity is even lower. Girvman-Newman algorithm

Exercise 4 Experimental results from different benchmarks including LFT confirm us the following time complexity comparison:
Execution times
LPA < Louvain << Girvnam-Newman
$O(m) < O(n \cdot log_2 n) << O(kmn)$

Algorithm	Number of nodes	Execution time (ms)
Label propagation	300	101.0
	400	77.0
	600	29.0
Louvain	300	150.3
	400	246.3
	600	635.9
Girvnam-Newman	300	286310.0
	400	376473.0
	600	729681.0

The Louvain algorithm Label propagation algorithm are much more efficient and faster than Girvman-Newman; However some of these algorithm are randomised and depend on the structure of the graph so it is possible in some special cases to obtain better results using Girvman-Newman algorithm.

Algorithm	p/q ratio	Accuracy (abs(expected clusters - found clusters)
Label Propagation	1.0	3
	6.0	2
	20.0	0
Louvain	1.0	3
	6.0	1
	20.0	0
Girvman-Newman	1.0	3
	6.0	3
	20.0	0

5 TME4

Exercise 1 — k-core decomposition We implemented an O(m) Algorithm for Cores Decomposition of Networks (1) where m is the number of edges in the undirected graph. To optimize the complexity of the classic solution, the latter algorithm uses bin-sort (Bucket sort) to sort and reorder the set of vertices V in increasing order of their degrees. Here are the results obtained by applying this algorithm to some benchmarks.

Benchmark	nb of nodes	nb of edges	core value	avg degree density	edge density	size of densest core ordering
Amazon	334863	925872	6	3.375695	0.014188	497
Live Journal	3997962	34681189	360	17.183631	0.000009	377
Orkut	3072441	117185083	253			

Execution times of core-value decomposition: amazon : 396ms, live-journal : 8017ms, orkut : 23730ms

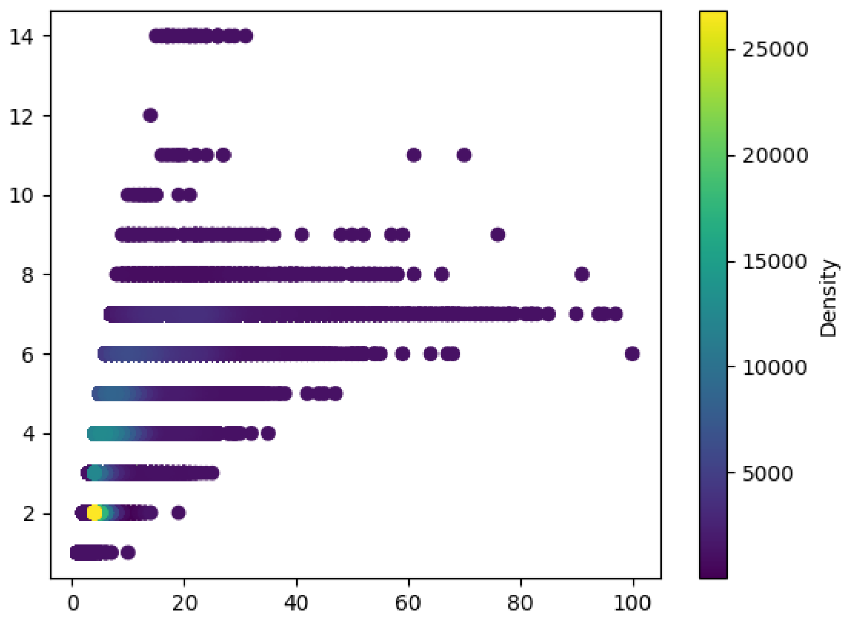

Figure 18: Scatter density plot representing coreness on y-axis and degree of nodes on x-axis

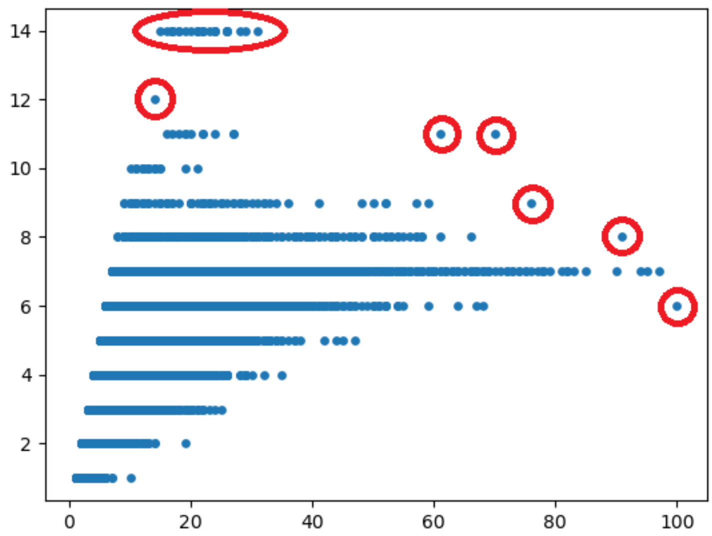

Figure 19: Highlighted anomalies on scatter density plot representing coreness on y-axis and degree of nodes on x-axis

```
We found 32 anomalous authors listed here:

#DEG #Coreness #ID #Author
91 8 2987 Derek Abbott
76 9 11913 Carole Goble
70 11 19991 Marc Pollefeys
100 6 25501 Ali Khademhosseini
61 11 46967 Roland Siegwart
14 12 123684 hoon (donghoon) choi
_ 14 55807 Sa–kwang Song
_ 14 55809 Sung–Pil Choi
_ 14 55813 Chang–Hoo Jeong
_ 14 55815 Yun–soo Choi
_ 14 55816 Hong–Woo Chun
_ 14 71188 Jinhyung Kim
_ 14 123670 Hanmin Jung
_ 14 123671 Do–Heon Jeong
_ 14 123672 Myunggwon Hwang
_ 14 123673 Won–Kyung Sung
_ 14 123675 Hwamook Yoon
_ 14 123676 Minho Lee
_ 14 123677 Won–Goo Lee
_ 14 123678 Jung Ho Um
_ 14 123680 Dongmin Seo
_ 14 123681 Mi–Nyeong Hwang
_ 14 123682 Sung J. Jung
_ 14 123688 Minhee Cho
_ 14 123689 Sungho Shin
_ 14 123694 Seungwoo Lee
_ 14 123695 Heekwan Koo
_ 14 123696 Jinhee Lee
_ 14 123697 Taehong Kim
_ 14 192697 Mikyoung Lee
_ 14 192698 Ha–neul Yeom
_ 14 192699 Seungkyun Hong
_ 14 192700 Yun–ji Jang
```

References

[1] Batagelj, V., and M. Zaversnik. "An O(m) Algorithm for Cores Decomposition of Networks." ArXiv:Cs/0310049, Oct. 2003. arXiv.org, http://arxiv.org/abs/cs/0310049.