Comparison study on different heuristics and measures concerning the Louvain algorithm

June 2022

Kieran van Gaalen (6437397) & Joes de Jonge (6228305)

Abstract

Community detection is an extremely important method of data analysis in graphs. Since there is no objective way to say a community is 'correct', measures are used to get the best communities possible. Which measure to use is therefore extremely important. In this paper we will compare four different measures using the Louvain algorithm and fixed seed LFR generators to get the best measure to use in an algorithm.

1. Introduction

Most of the networks you can observe in the real world have certain communities, specific groups that share a lot of inner edges between the inner nodes. For example in a school there are a lot of friend groups. People inside a friend group have a lot of friends inside the friend group, and fewer friends outside of the friend group. With real world networks and with artificial ones, it is great practise to detect which communities there are. For example, detecting communities on a social media platform may be useful for marketing or recommending new friends [1]. Add to that the value of being able to analyse a dataset of people for example, where you can actually use community detection for more targeted marketing.
The main issue with community detection is the amount of time it takes to calculate the communities. A good community detection algorithm will strike a good balance between delivering results quickly, but also making sure that the results are actually close to an optimal solution.

1.1. Louvain

The Louvain algorithm [2] is a widely applied community detection algorithm. It provides good solutions and runs relatively quickly. The way Louvain works is by setting every node in the graph as its own community, and then moving nodes to neighbouring communities. These moves will then be analysed to see if these would actually improve the community structure or not. This analysis is done by some formula which will give a value indicating how good the communities are. This formula is called a measure, which will always be an approximation, and therefore will not objectively get 'good' communities. Calculating this measure may take a lot of time for a graph, therefore a heuristic may be used, which is approximates the full measure.
Once this measure has been used and the node has or hasn't been moved, the next node is analysed. When every node has been checked, a new total measure will be calculated. If this total measure is a lower increase than some threshold, or no nodes have been moved, the algorithm will move on to the next step, otherwise it will repeat.
The next step is grouping all nodes of a community into a single node and repeating this grouping for all communities. Then the previous steps of putting the nodes into communities and placing nodes into other communities should be repeated with this new network of communities. This should be repeated until there is no improvement anymore in the total measure of the network.

1.2. Measures and heuristics

We were interested in the way measures and heuristics affect the performance of the Louvain algorithm. The heuristic makes an algorithm work faster by making some shortcut solutions in the measure. Normally, you use the Louvain algorithm with a heuristic for the modularity of the graph, where you want to maximize the modularity. However, there are many heuristics with measures one can think of which all have a specific characteristic. To figure out which measure heuristic combination works the best, we implement a few of these into a custom Louvain implementation and test how well each one performed and why.

2. Method

Our plan, to test the different measures and heuristics, was to implement a community detection algorithm in such a way that the measures and heuristics are easily interchangeable. This would give us results in which the only difference is the actual difference in measures and heuristics, and not a specific implementation of the algorithm used. We combined this with fixed seed LFR generators to remove the variance of graphs as well.
We started out by implementing the Louvain algorithm using some different guides found online [3] [4] , which we generalised in such a way that it expects certain functions with certain structures. In these functions we then implemented the specifics of the measure and heuristic we would use. One of the functions is used when you want to get the difference in measure when a node is transferred from one community to another community. In that situation, you only want the community transfer with the highest positive increase. This idea is implemented in the function getDelta. Furthermore, the algorithm also wants to combine nodes from one community to create one big node for the new iteration. For that, we implemented a totalMeasure function. You only want to iterate once more when the totalMeasure is increased compared to the previous step, or from the initial value of minus infinity.

2.1. Measures used

To explain the measures used we will have the following definitions:
C: All communities.
m: The amount of edges in the graph.
n: The amount of nodes in the graph.
n_c: The amount of nodes in community c.
$A_{i,j}$: Whether nodes i and j are neighbours (binary).
$k_{i,in}$: The amount of neighbour nodes of node i that also belong to the community to which node i belongs.
$k_{i,out}$: The amount of neighbour nodes of node i that do not belong to the community to which node i belongs.
k_i: The amount of neighbour nodes of node i.
c_{in}: The total amount of inner edges in community c.
c_{out}: The total amount of outer edges (edges going outside the community) in community c.
c_{tot}: The total amount of edges in community c.
c_{old}: The old community the node was in.
c_{new}: The community the node is being moved to.

As mentioned before, measures have two implementations, a total implementation, which will calculate the entire measure for the graph at once, and a delta implementation which will calculate (or approximate, depending on the complexity of the calculation) the difference in measure when a node is moved. The following measures were used:

Modularity

Total:

$$\frac{1}{2m}\sum_{i,j}\left(A_{i,j}-\frac{k_i k_j}{2m}\right) \tag{1}$$

Delta:

$$\begin{aligned}&\left(\frac{c_{in}+2k_{i,in}}{2m}-\left(\frac{c_{tot}+k_i}{2m}\right)^2\right)\\&-\left(\frac{c_{in}}{2m}-\left(\frac{c_{tot}}{2m}\right)^2-(\frac{k_i}{2m})^2\right)\end{aligned} \tag{2}$$

The first measure we are going to implement, is modularity [5]. This measure uses the assumption that a good community has more inner edges than a random distribution of edges would result in. More precisely, modularity denotes the amount of edges that fall within the current community minus the amount of edges expected to fall within the community if they were distributed randomly. If this value is high, this means that the community is well connected, and therefore would indicate a good community. This measure is also the standard measure used in the Louvain algorithm [2], and therefore we expect to perform quite well.

Edge ratio

Total:

$$\sum_{c\in C}\frac{c_{in}}{n_c} \tag{3}$$

Delta:

$$\begin{aligned}&\left(\frac{c_{old,in}-k_{in,c_{old}}}{n_{c_{old}}-1}+\frac{c_{new,in}-k_{in,c_{new}}}{n_{c_{old}}+1}\right)\\&-\left(\frac{c_{old,in}}{n_{c_{old}}}+\frac{c_{new,in}}{n_{c_{new}}}\right)\end{aligned} \tag{4}$$

The edge ratio once again relies on the fact that a good community has a lot of inner connections. This measure doesn't really care if the amount of connections to the

outside of the community is high or low, as long as the amount of inner connections is as high as possible. It does this by simply looking at all the nodes in a community and all the inner edges in the community and divides all of the edges by all of the nodes. This is the average degree of a community, only taking into account the inner edges. This is an extremely simple measure, and therefore we expect it to be quite rough, but quick.

Minimum Inner Degree Fraction

Total:

$$\sum_{c \in C} \min_{i \in c} \frac{k_{i,in}}{k_i} \tag{5}$$

Delta:

$$\begin{aligned} &\left(\min_{i \in c_{new} \cup \{u\}} \frac{k_{i,in}}{k_i} + \min_{i \in c_{old} \setminus u} \frac{k_{i,in}}{k_i} \right) \\ &- \left(\min_{i \in c_{new}} \frac{k_{i,in}}{k_i} + \min_{i \in c_{old}} \frac{k_{i,in}}{k_i} \right) \end{aligned} \tag{6}$$

Where u is the node that is considered to be moved from c_{old} to c_{new}.

The minimum inner degree fraction takes the node that belongs the least in each community, and checks how little it belongs. It does this by checking the total degree of the node and comparing it to the inner degree. The lower the inner degree is compared to the total degree, the worse it is. The worst of these gets taken and added to a sum value. This sum value tells us how bad the worst node of every community is. The better our bad nodes are, the higher this value, thus the better our communities. Because this measure also takes into account the amount of outer edges of a node, compared to edge ratio which doesn't, we expect that this measure will perform better than the edge ratio. But still worse than the modularity.

Average Inner Degree Fraction

Total:

$$\sum_{c \in C} \frac{1}{n_c} \sum_{u \in c} \frac{k_{i,in}}{k_i} \tag{7}$$

Delta:

$$\begin{aligned} &\left(\frac{1}{n_{c_{new} \cup \{u\}}} \sum_{u \in c_{new} \cup \{u\}} \frac{k_{i,in}}{k_i} + \frac{1}{n_{c_{old} \setminus u}} \sum_{u \in c_{old} \setminus u} \frac{k_{i,in}}{k_i} \right) \\ &- \left(\frac{1}{n_{c_{new}}} \sum_{u \in c_{new}} \frac{k_{i,in}}{k_i} + \frac{1}{n_{c_{old}}} \sum_{u \in c_{old}} \frac{k_{i,in}}{k_i} \right) \end{aligned} \tag{8}$$

Where u is the node that is considered to be moved from c_{old} to c_{new}.

The average inner degree fraction uses, just like the minimum inner degree fraction, the fraction between the amount of inner edges of a node inside a community and the total amount of edges of that node. But instead of getting the minimum value of that community it will get the average of that community. In general you would like to have a high average inner degree fraction. It would mean that there a lot of nodes inside the community that have a high amount of inner edges. Because this measure looks a lot like the minimum inner degree fraction, we do not expect it will perform that differently to the minimum inner degree fraction.

The minimum inner degree fraction and average inner degree fraction follow both from Leskovec, Lang and Mahoney [6]. They mention a maximum outer degree fraction and an average outer degree fraction. We changed that into minimum inner degree fraction and average inner degree fraction, because in the paper the authors optimise the algorithm by minimising the measures. However, in the Louvain algorithm the measure is maximised instead. Changing to inner degree instead of outer degree means that anything 'bad' in the original measures, such as a lot of connections to nodes outside the community, will be changed to a lot of connections to nodes inside the community, which is 'good'. We used these two measures from their paper, because as they discussed they are the ones that have the most interesting results. For example, the minimum inner degree fraction prefers small communities.

These measures will then get run through the Louvain algorithm we implemented.

2.2. Graph generation

This more generalised version of the Louvain algorithm is used 20 times with randomly generated graphs using the LFR benchmark function of NetworkX. An LFR benchmark is a great way to produce

graphs with a strong ground truth when it comes to communities. Because it explicitly tells you what the communities are. It allows for a lot of control over the different variables. To make sure we can indeed compare these graphs and their outputs from the generalised Louvain algorithm, we also have to make sure these 20 different graphs were created in a similar way. Therefore, we use a fixed seed. This seed is used in the LFR benchmark function to randomize the creation of nodes and edges. A stochastic graph generator isn't very useful is this case since such graphs do not automatically generate lists of correct communities. Which would make it quite difficult to come to a good conclusion when the ground truth is becoming less strong.
The exact settings we used for LFR took a lot of tweaking and searching around. Eventually we found a paper [7] using the following variables, which worked very well.

Variable	Value
Node Count (n)	250-2000
τ 1	3
τ 2	1.5
μ	0.03-0.75
average degree	20
max communities	0.1n
max degree	0.1n

Table 1. LFR variables

2.3. Results gathering

To compare these communities, we needed to determine how good their partition is on the whole network. We used the end total measure, the run time, the amount of edges, the amount of nodes and the Jaccard coefficient. This last measure tells us how many pairs of nodes are correctly put together in communities according to the community labeling that results from the LFR benchmark function. The higher this result, the better our partition is. The Jaccard coefficient was calculated as follows:

$$J(X,Y) = \frac{a_{11}}{a_{11} + a_{01} + a_{10}} \tag{9}$$

Where a_{11} is the number of pairs of nodes which are in the same community in both partition of X and Y. a_{10} is the number of pairs of elements which are in the same community in Y, but in different communities in X, and vice versa for a_{01}. And then X shall be the partition that is the result from running our algorithm and Y is the partition following from the LFR benchmark. [8]
The experiments were run on a computer using an AMD Ryzen 9 3900X 12-Core Processor @ 3.80 GHz and 32GB CL16 3600Mhz RAM.

3. Results

	Average	SD	CV
Modularity	0.1296	0.1126	0.8688
Edge ratio	0.0484	0.0079	0.1631
Min IDF	0.0095	0.0082	0.8611
AVG IDF	0.0133	0.0132	0.9956

Table 2. Jaccard coefficients of measures

	Average	SD	CV
Modularity	62.9	57.8	0.9184
Edge Ratio	55.2	29	0.5246
Min IDF	590.4	280.7	0.4754
Avg IDF	529.5	261.8	0.4754
LFR	32.8	11	0.3355

Table 3. Amount of communities in measures and correct amount of communities

	Average	SD	CV
Modularity	217.17	218.69	1.007
Edge Ratio	18.03	29	0.7709
Min IDF	3.58	1.77	0.4944
Avg IDF	4.32	2.44	0.5648

Table 4. Runtimes of the measures (s)

Table 2 shows the average Jaccard coefficient and the standard deviation and coefficient of variation for the Jaccard coefficient. This tells us the correctness of the communities generated by the measure. Table 3 shows the amount of communities generated per measure and the average amount of communities as well as once more showing the standard deviation and coefficient of variation. These can be compared to the community count set by the LFR generator. These may not be as accurate as the Jaccard coefficients in denoting correctness, but they do give an interesting insight into the structure of generated communities.
Finally table 4 shows the average runtime as well as the standard deviation and coefficient of variation once again.

4. Discussion

To properly evaluate our measures we need to look at both runtime and correctness.

4.1. Runtime

Runtime is a very important aspect of every algorithm. To get a result you don't want to wait too long. One of the biggest factors for this is the time complexity. This means the increase in time given a certain increase in input. In our case, that would be an increase in graph size. If a graph becomes twice as large, what happens to the runtime.
In table 4, we can see that modularity has by far the worst runtime, with the average run taking about 3.5 minutes. This is a lot slower than some other implementations of Louvain with modularity, which does tell us that our Louvain method is relatively slow. However, we can still compare the results, since they all use the same Louvain algorithm. Both IDF measures are extremely quick, being done within five seconds on average in both cases. Finally, the edge ratio falls somewhere in the middle, not as quick as the IDF measures, but still pretty fast.
To measure the time complexity we can use the coefficient of variation as a rough indicator. Since the time increases when input increases, any variation we can see in time will be due to input. Since every measure had the same input, any large variations in runtime will indicate that the time complexity of said measure is worse. As we can see, the variation for modularity is the highest, with edge ratio coming up behind. We can also look at the curve when these runtimes are plotted to see the time complexity. These graphs are found in the appendix in figures 1 through 4. In these figures the same can be seen as with the coefficient of variation, modularity has the largest curve, and the IDF measures the flattest. However, from both the coefficient of variation and the graphs we can see that the time complexities aren't that different overall.

4.2. Correctness

Evaluating runtime is very valuable of course, but an algorithm which runs quickly is of no use if the results for that algorithm may as well be useless. Therefore we will now evaluate the correctness of our results.
To evaluate the correctness of a solution we have to compare it to some ground truth. The LFR generator we used generates a graph with every node having an attribute which contains a list of the nodes in its community. We can compare this community structure to the community structure generated by our measures to find out the correctness of these results. To grade the correctness we used the Jaccard coefficient. This will give us a result between zero and one, zero meaning absolutely nothing in common and one meaning the communities are identical. In Table 2 we can see that the average Jaccard coefficient for modularity is way higher than for the rest of the measures used. This would indicate that the communities generated using the modularity measure are, on average, closer to the ground truth than the communities generated by other measures. Overall the Jaccard coefficient is quite low for all measures, however in comparison modularity is miles ahead.
Apart from getting good results on average, getting consistent results is also important. The best way to do this is by using the coefficient of variation. This value is the standard deviation divided by the average. The lower the value, the closer values in the dataset are relative to one another. Due to big differences in ranges of communities, the standard deviation alone is not a good way to compare these. Looking at the coefficient of variation, we can see that the edge ratio gets very consistent results, whereas the other measures are a lot less consistent.
Another thing to look at for the correctness is the amount of communities generated by each algorithm. This can tell us a lot about why the Jaccard coefficients for measures are the way they are. This can be found in table 3. All community counts are higher on average than the correct community counts generated by the LFR generator, however there is a clear divide in these differences. The average community counts for both IDF measures are extremely high, in most cases about half of the node counts found in table 7 in the appendix. This makes sense, since the result Jaccard coefficients are so low. This would also explain the runtimes of these measures, since a high amount of communities means not many combining steps have been taken until the measure didn't increase anymore, resulting in a quick stop to the algorithm, and therefore a low runtime.
Modularity actually has a higher difference between the community count generated and the correct community count than edge ratio does, even though the Jaccard coefficients for modularity tell us the communities are of higher quality. This would make it seem that the communities generated by modularity are either very close to the correct communities, or otherwise fragmented into multiple smaller communities.

5. Conclusion

From the results the measure with the best runtime is clearly either of the IDF measures, however runtime

doesn't mean much when the results are of a low quality, which is the case for these measures, as well as for the fast edge ratio. The correctness results of these are significantly worse than those for modularity, and therefore not really that useful. Modularity however does have a good result, however the runtime is not ideal. In general however, modularity does appear to be the best option, since overall, fast runtime isn't really worth it if the results are not up to scratch. This is probably also the reason modularity is the standard for the Louvain algorithm.

5.1. Reflection

What we can see from our runtime results, is that the total runtime for any measure was pretty slow. Even though this isn't desired when actually using the measures, this shouldn't affect our conclusions, since any comparisons made were relative comparisons. However, ideally this algorithm was more optimised and the results could be more true to life, and not only used as a relative comparison but maybe also a way to analyse a singular measure.

5.2. Future research

In the future, more different measures could be added to compare one to another. We tried our best to implement the structure of measures in the codebase in such a way that this is relatively easy to do. Furthermore, different or more optimised algorithms could be implemented, since a different algorithm may actually result in a different measure coming out on top. For example, we initially also implemented a Flake-IDF measure, which counts the amount of nodes for which more neighbours exist in the current community than outside of the current community. This measure would increase the more nodes belong in their current communities. The issue with this measure however was that the way Louvain starts with communities with only one node, the total measure would be zero. However, after attempting to move, the result would still always be zero, since the total amount of neighbours a node had would almost always be larger or equal to two. If this was the case, adding another neighbour to its community would not actually cause it to finally be a 'belonging' node, and the value would still be zero. In a different algorithm with a different start position however, this measure could have been usable, and may have had good results.

6. Appendix

6.1. Tables

Graph	Modularity	Edge Ratio	Min IDF	Avg IDF
1	0.1146	0.0554	0.0210	0.0269
2	0.0818	0.0396	0.0039	0.0049
3	0.0322	0.0417	0.0019	0.0016
4	0.0386	0.0388	0.0023	0.0025
5	0.286	0.0506	0.0107	0.0166
6	0.0468	0.0477	0.0107	0.0162
7	0.0966	0.0615	0.012	0.0129
8	0.0416	0.0471	0.0033	0.0043
9	0.0294	0.0498	0.0031	0.0037
10	0.3448	0.0493	0.0252	0.0421
11	0.051	0.0509	0.014	0.013
12	0.0452	0.0589	0.0072	0.0094
13	0.2994	0.0561	0.0057	0.0085
14	0.0862	0.0454	0.0056	0.0069
15	0.1785	0.0486	0.0111	0.0146
16	0.3003	0.0434	0.0098	0.013
17	0.0303	0.0419	0.0011	0.001
18	0.0363	0.0545	0.0012	0.002
19	0.1516	0.0292	0.0095	0.0138
20	0.3008	0.0581	0.031	0.0511

Table 5. Jaccard coefficients of measures per individual graph

Graph	Modularity	Edge Ratio	Min IDF	Avg IDF	LFR
1	26	36	214	179	19
2	30	84	820	771	46
3	159	51	535	485	34
4	63	100	1008	951	50
5	37	75	576	507	33
6	1	93	957	810	43
7	31	26	312	275	21
8	42	37	498	447	30
9	50	8	335	303	25
10	83	24	347	280	25
11	32	25	209	182	21
12	22	22	205	171	18
13	43	64	834	747	29
14	79	68	676	623	35
15	29	51	511	466	29
16	261	98	960	860	47
17	46	66	873	811	44
18	91	59	797	733	33
19	52	92	847	754	53
20	81	25	294	234	21

Table 6. Amount of communities in measures and correct amount of communities per individual graph

Graph	Node Counts	Edge Counts
1	421	5551
2	1650	19696
3	1073	13466
4	1979	24377
5	1151	14320
6	1943	22669
7	634	7967
8	1008	12457
9	695	8863
10	693	8409
11	435	5886
12	429	5726
13	1647	21026
14	1363	17495
15	1040	12912
16	1937	22999
17	1751	23364
18	1587	20318
19	1665	21292
20	605	7287
Average	1185.3	14804

Table 7. Node and edge counts per graph

Graph	Modularity	Edge Ratio	Min IDF	Avg IDF
1	10.9	1.9	2	1.2
2	421.2	25.6	4.3	5.6
3	138.6	12.9	3	3.1
4	674.3	40.9	4.7	5.7
5	155.7	13.1	2.8	4.9
6	627.2	30	6.5	9.9
7	25.9	4.6	1.8	1.9
8	107	11.3	2.9	3.4
9	43.3	8.4	2.1	2.4
10	16.8	5.1	2.1	4.6
11	11.1	2.3	0.9	1.2
12	15.6	2.2	1.6	1.2
13	346.6	29.2	5.3	5.9
14	151.7	19.3	3.6	4.9
15	168	10.4	2.6	3.7
16	153.1	35.8	6	8.6
17	581.7	43.4	7.2	4.7
18	309.2	31.7	4.1	4.8
19	373.2	28.8	5.5	7.2
20	12.2	3.7	2.6	1.6

Table 8. Runtimes for measures per graph

6.2. Runtime graphs

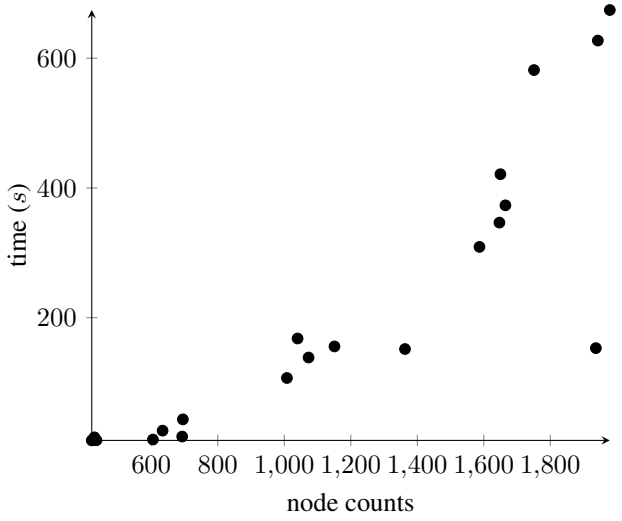

Figure 1. Runtimes for modularity

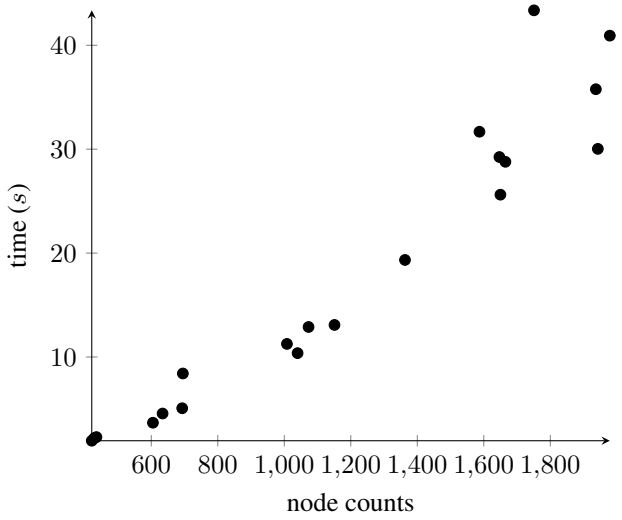

Figure 2. Runtimes for edge ratio

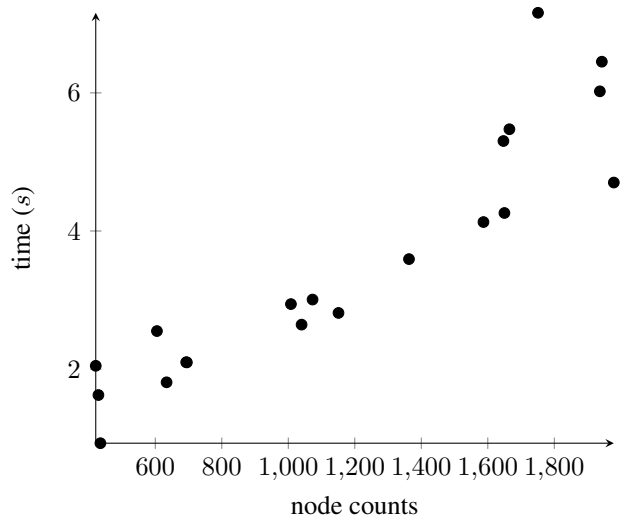

Figure 3. Runtimes for minimum IDF

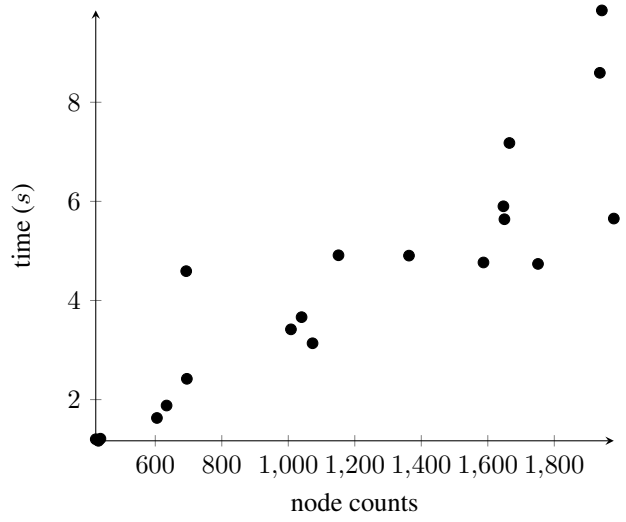

Figure 4. Runtimes for average IDF

References

[1] P. Bedi and C. Sharma, "Community detection in social networks," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 6, no. 3, pp. 115–135, Feb. 2016. DOI: 10.1002/widm.1178. [Online]. Available: https://doi.org/10.1002/widm.1178.

[2] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," 2008. DOI: 10.48550/ARXIV.0803.0476. [Online]. Available: https://arxiv.org/abs/0803.0476.

[3] L. Rita, *Louvain algorithm*, Apr. 2020. [Online]. Available: https://towardsdatascience.com/louvain-algorithm-93fde589f58c.

[4] N. Ozaki, H. Tezuka, and M. Inaba, "A simple acceleration method for the louvain algorithm," *International Journal of Computer and Electrical Engineering*, vol. 8, no. 3, pp. 207–218, 2016. DOI: 10.17706/ijcee.2016.8.3.207-218. [Online]. Available: https://doi.org/10.17706/ijcee.2016.8.3.207-218.

[5] M. E. J. Newman, "Fast algorithm for detecting community structure in networks," *Phys. Rev. E*, vol. 69, p. 066 133, 6 Jun. 2004. DOI: 10.1103/PhysRevE.69.066133. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.69.066133.

[6] J. Leskovec, K. J. Lang, and M. W. Mahoney, "Empirical comparison of algorithms for network community detection," *CoRR*, vol. abs/1004.3539, 2010. arXiv: 1004.3539. [Online]. Available: http://arxiv.org/abs/1004.3539.

[7] Z. Yang, R. Algesheimer, and C. J. Tessone, "A comparative analysis of community detection algorithms on artificial networks," *Scientific Reports*, vol. 6, no. 1, Aug. 2016. DOI: 10.1038/srep30750. [Online]. Available: https://doi.org/10.1038/srep30750.

[8] S. Fortunato and D. Hric, "Community detection in networks: A user guide," *Physics Reports*, vol. 659, pp. 1–44, Nov. 2016. DOI: 10.1016/j.physrep.2016.09.002. [Online]. Available: https://doi.org/10.1016%5C%2Fj.physrep.2016.09.002.