EPITA

Ferdinand Bhavsar - Dorian Poncet

May 31, 2020

(Sparse) Very Large Graphs

Effects of graph reordering on the performance of the Leiden algorithm.

ferdinand.bhavsar@epita.fr
dorian.poncet@epita.fr

Contents

1 Abstract

The present study has investigated the effects of node reordering on the performance of the Leiden community detection algorithm, applied to three real-world very large graphs. The reordered graphs were obtained via a breadth-first traversal, starting from notable nodes. The Leiden algorithm was benchmarked on shuffled and reordered graphs, concluding on a 27% minimal average time reduction with reordered graphs. Moreover, starting the reordering from central nodes results on a 31% minimal average time reduction for the Leiden algorithm, compared to its speed when ran with the shuffled isomorphic graph.

Keywords: graph reordering, community detection, leiden

2 Introduction

For very large real-world graphs, community detection is a useful tool, in domains ranging from social networks to national security. It enables easy and comprehensible visualization of such network, and a study of its internal groups. Such detection can be done using the Leiden algorithm[1]. While an improvement on the Louvain algorithm, it can still be slow on very large graphs, having millions to billions of nodes. In some application, accelerating the detection process can be crucial. Researchers at the LSE showed that graph reordering using a breadth-first search algorithm[2] improved the performance of the Louvain community detection algorithm[3]. Similarly, can the Leiden algorithm be accelerated with a preliminary breadth-first reordering? If so, what are the best root nodes for the breadth-first search? In this paper, we try to answer both questions.

3 Hypothesis

Two graphs which contain the same number of vertices connected in the same way are said to be isomorphic. In theory, isomorphic graphs should behave exactly the same way. However, as shown in previous papers[3][4], reordering a graph using a breadth first search (BFS) algorithm before running an algorithm such as Louvain's community detection algorithm can result in an acceleration of 10% to 50%, depending on the source article, graph and hardware. This is allegedly due to a lower number of cache misses, the graph's adjacent nodes being in average more contiguous in data after the reordering. Also, depending on the root node chosen for the BFS reordering, the average in-memory contiguousness of adjacent graph nodes can differ.

For these reasons:

- We hypothesize that reordering a graph before running the Leiden algorithm can improve performance significantly, with effects similar to those observed with the Louvain algorithm.
- We hypothesize that the choice of root node for the BFS reordering will affect the performance increase.

4 Methodology

4.1 Hardware

Two different computers are used. To match the high RAM needs that come with handling very large graphs, swap memory is allocated on both computers. The first computer worked exclusively on the graph p2p, the second worked exclusively on the inet and ip graphs.

First computer:

- ASUS TUF Gaming A15
- 16GB RAM + 12GB swap file on SSD
- AMD Ryzen 5 3550h with Radeon Vega Mobile Gfx × 8

Second computer:

- LENOVO ThinkPad X230i
- 4GB RAM + 4GB swap file on HDD
- Intel Core i3-3110M

4.2 Software

We use Python 3.7 with the "igraph" [9] and "numpy"[10] libraries to manipulate our graphs. While Python is not the best language for huge sparse graphs, the "igraph" library is coded in C with a Python interface, making it efficient enough. For execution timing, the Python "timeit" module is used. For some statistics, we used "scipy". During benchmarking, the Leiden algorithm is the only user program running on the computer. The Leiden algorithm we use is implemented by the igraph library.[5] We call it with the standard parameters (two iterations), except the objective_function parameter, which is set to "modularity".

```
vertex_clustering = g.community_leiden(
        objective_function="modularity")
```

Function call illustrating our use of igraph's Leiden algorithm on the undirected graph g.

4.3 Graphs

The graphs used were distributed by Clémence Magnien[6] alongside a 2007 publication on graph diameter approximation[7].
Those three graphs are huge sparse graphs, meaning they have more than a million edges and nodes, and |E| = O(|V|), with |E| the number of edges and |V| the number of nodes. [8] They are all coming from real-life massive data sets. [1]

For all experiments, we only study the largest connected component (giant component) of those graphs. The indices of the nodes in this component are shuffled, using the reordering algorithm with a random permutation (see next section), to prevent already partially ordered graphs to bias the experiment. Indeed, graphs produced by crawling graph-like structures (e.g. web pages crawled with a BFS/DFS variant) can have node indices partially ordered. If we don't shuffle the graphs beforehand to have truly non-ordered graphs, the effects of reordering can be potentially lesser for some of the graphs.

After extracting the main connected component, we obtain graphs with the following characteristics:

	inet	ip	p2p
Number of nodes	1694616	2250046	5380491
Number of edges	11094209	19393724	142038351

4.4 Experiments

Goal

We aim at producing relevant statistical results (at confidence level 95% or greater) indicating whether reordering a graph using a breadth-first search prior to running the Leiden community detection algorithm can reduce the time the latter takes to be completed. Similarly, we try to determine whether the choice of root node for the BFS reordering has an impact on that acceleration.

[1] While we ran some tests on random graphs, we personally consider them to be of lesser relevance compared to real-world graphs.

Method

First, we shuffle and save each graph[2]. For each shuffled graphs, we produce and save six reordered versions. [3] To do so, we use the BFS method as described in the LSE "Reordering for fun and profit" article[3], using one of the following node choice as root for the BFS:

- **zero node**: first node of the graph (index=0) ; this is done post-shuffling, so the node is a random one, but it stays the same for each run of the Leiden algorithm on a given graph
- **center node**: one of the center nodes of the graph
- **maximum** (resp. **minimum**) **degree node**: one of the node with the highest (resp. lowest) degree
- **extreme node with double sweep method**: we do a BFS from the node with index 0, then choose the last node visited as the root node for the reordering BFS
- **extreme node with triple sweep method**: we do a BFS from the node with index 0, then choose the last node visited as the root node for a second BFS, then choose the last node visited by this second BFS as the root node for the reordering BFS
- **random node**: a node chosen at random. The graphs reordered with this method are not saved once and reused like the others; this is chosen at random at each run of the Leiden algorithm, since we need a new random node at each run to get baseline statistics for the reordering from a non-significant node.

Once all the new graphs are created using the different method, we run the Leiden algorithm on each of them. We also run the Leiden algorithm on the shuffled graph to have a base value (not reordered). Below, we will refer to these seven graphs produced from the original (one shuffled + six reordered) as the "prepared graphs". For each prepared graph, on each run of the Leiden algorithm, we save the computed modularity, the number of communities, and the time taken by the algorithm to run. Results for a prepared graph are appended in a corresponding results file for later processing.

For a given graph, we do the same number of Leiden runs on each of its corresponding prepared graph. This way, we simplify the statistical analysis of our benchmarking results.

Concerning statistics, we will only be able to conclude on an acceleration of the Leiden algorithm on reordered graphs [4] if we can be sure the mean for the reordered graph time is significantly lower than the mean for the shuffled graph time.

[2] From now on, when refering to a graph, we actually refer to its giant component.

[3] Our BFS reordering being deterministic for a given graph-node couple, we don't have to reorder the graph for each run of the Leiden algorithm. This way, computation time is saved.

[4] At a 95% or more confidence level.

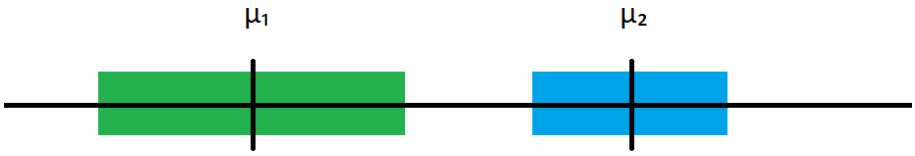

Diagram illustrating our goals regarding confidence intervals. μ_1 and μ_2 are the empirical means of the Leiden execution time respectively for a reordered and a shuffled version of agraph. The colored areas correspond to the confidence intervals at 95% confidence level.

With more test results, we potentially reduce the width of each colored area, corresponding to the error intervals for the means at the 95% confidence level. By running enough tests, we might be able to have no intersection between these colored areas for the shuffled version and a reordered versions of a graph; which corresponds to being sure that the real mean of the Leiden algorithm run time for the reordered version of the graph is lower than the real mean of the Leiden algorithm run time for the shuffled graph. This way, we can confirm the hypothesis that, in our experimental conditions (software and hardware), the reordering of the graphs causes a significant acceleration of the Leiden algorithm.

To this purpose, for each prepared graph, a separate Python script displays the average elapsed time during the execution of the Leiden algorithm. This allows us to monitor the progress of our data collection in real time (number of tests done), as well as basic statistics (average number of communities, average modularity). Once all the data is collected, we can computed advanced statistics with numpy and plot the data with matplotlib.

To reduce computation time, the graphs are reordered only once with each method, and saved in that state. But to get a baseline for reordering statistics from random nodes, we need to choose a random node from which to reorder each time we run the Leiden algorithm.[5]

Whatever the method evaluated (loading already reordered graph for predetermined BFS root nodes or reordering before executing the Leiden algorithm for random BFS root nodes), we try to have a similar memory state when executing the Leiden algorithm, by freeing the unused non-reordered graph copy in memory if present.

[5]The statistics for the Leiden algorithm with a graph reordered from a random node each time (which constitute the baseline for the performance increase when reordering without specifying a node) do not appear in the result section (see section 5). They were benchmarked with the same level of precision (110 iterations on each graph), and were similar in mean to the reordering from zero, with a slightly higher spread.

5 Results

Please note:

1. Some figures are shown twice; the only difference in the second one is the disappearance of the data concerning the shuffled graph. The first figure is used to compare the reordered graphs with the shuffled graph. The second figure is used to compare the reordered graphs with each other.

2. The figure with curves show the observed probability density functions of the studied values. The vertical lines underneath represent the observed values.

3. The other figures show the observed means (fully-opaque vertical line) of the studied values, with a confidence interval specified under each concerned graph (semi-transparent area); in these figures, the vertical axis has no meaning, and is there only for readability.

4. "Running time" is in seconds, "Modularity" is in percent, "Number of cluster" has no unit.

5. The "conclusive", "partially conclusive" and "inconclusive" figure labels are only meant to facilitate a fast reading.

5.1 inet

On the graph "inet", the Leiden algorithm was run 110 times for each prepared version of the graph. The computer used was the LENOVO ThinkPad X230i (second computer, see 4.1). On inet, the reordering takes an average of 7.5 seconds.

Means for graph inet	Modularity	Number of clusters	Running time
No reordering	0.85	304.86	48.19
Center node	0.85	305.36	32.82
Node zero	0.85	301.32	33.00
Max degree node	0.85	295.49	33.36
Min degree node	0.85	304.15	33.14
Double Sweep	0.85	310.67	32.85
Triple Sweep	0.85	306.35	33.03

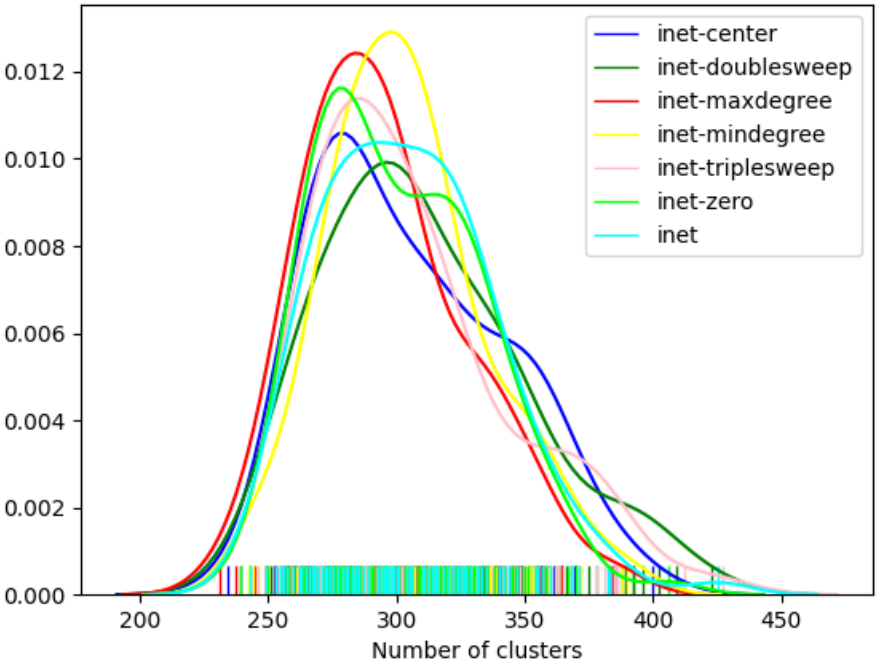

Figure 1: inconclusive

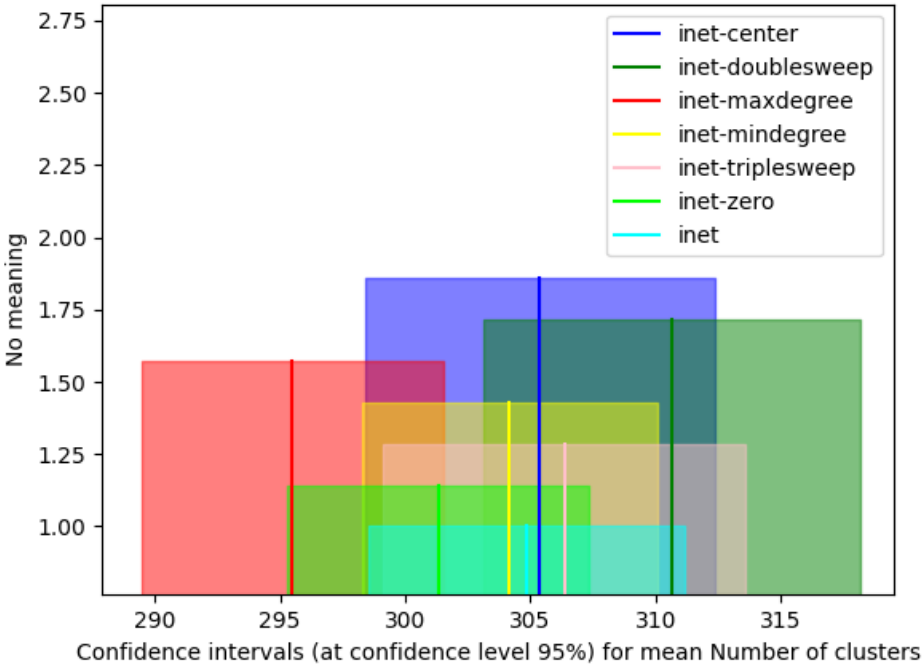

Figure 2: partially conclusive

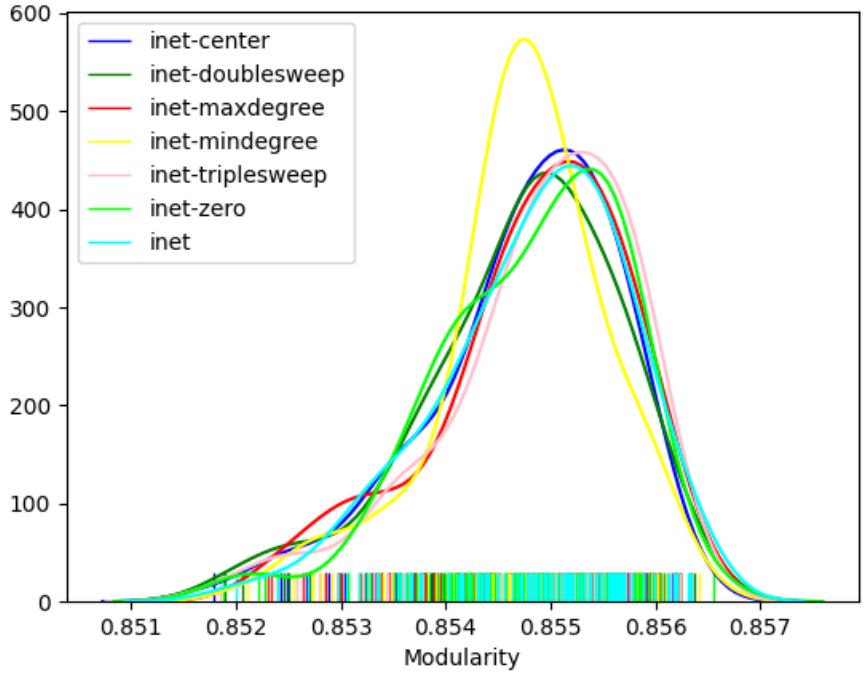

Figure 3: inconclusive

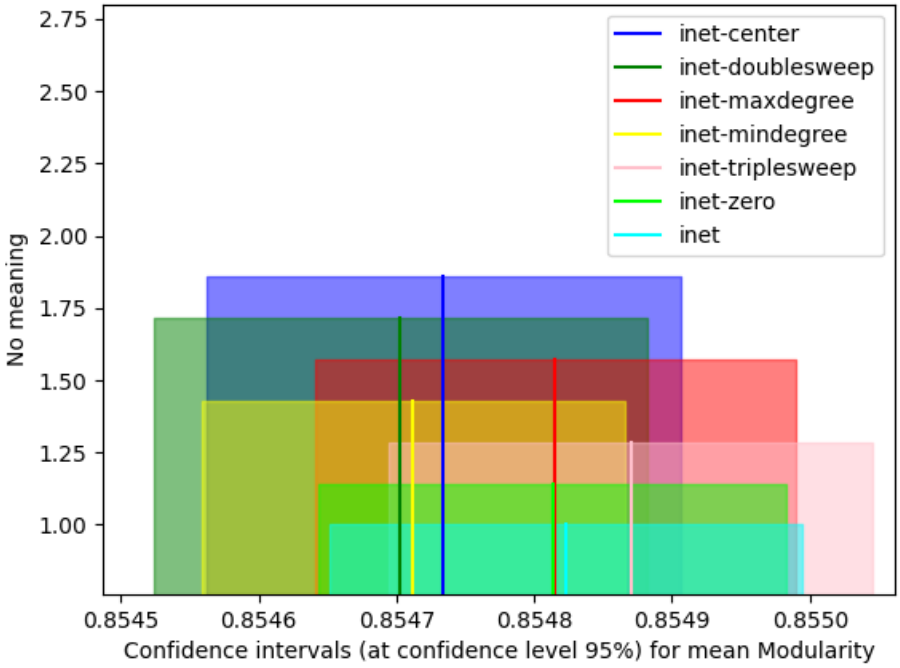

Figure 4: inconclusive

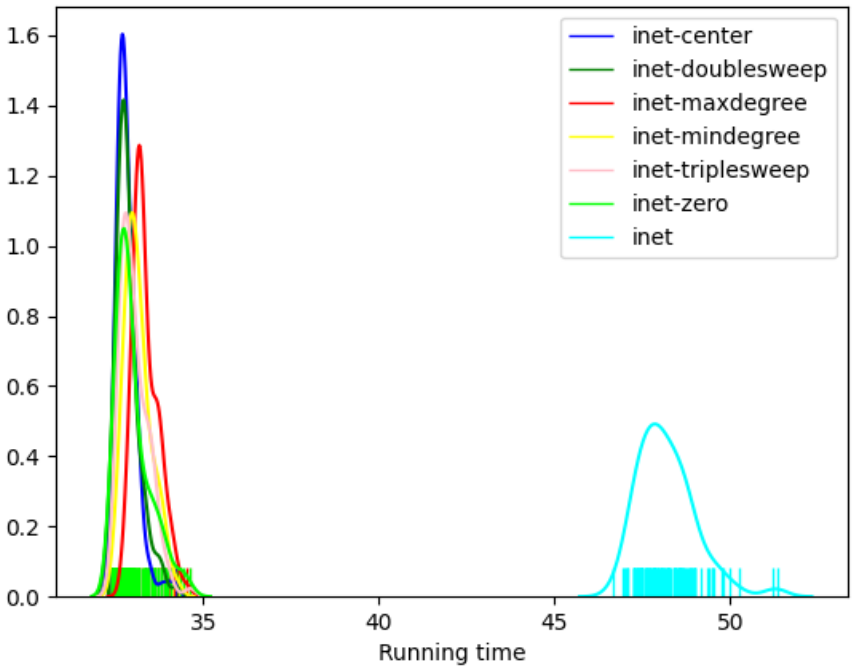

Figure 5: conclusive

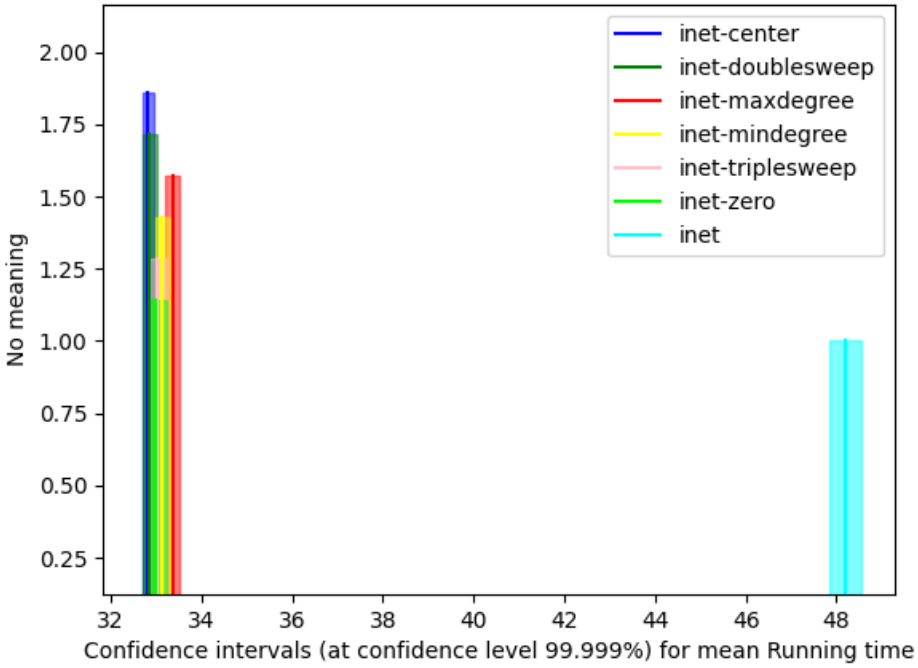

Figure 6: conclusive

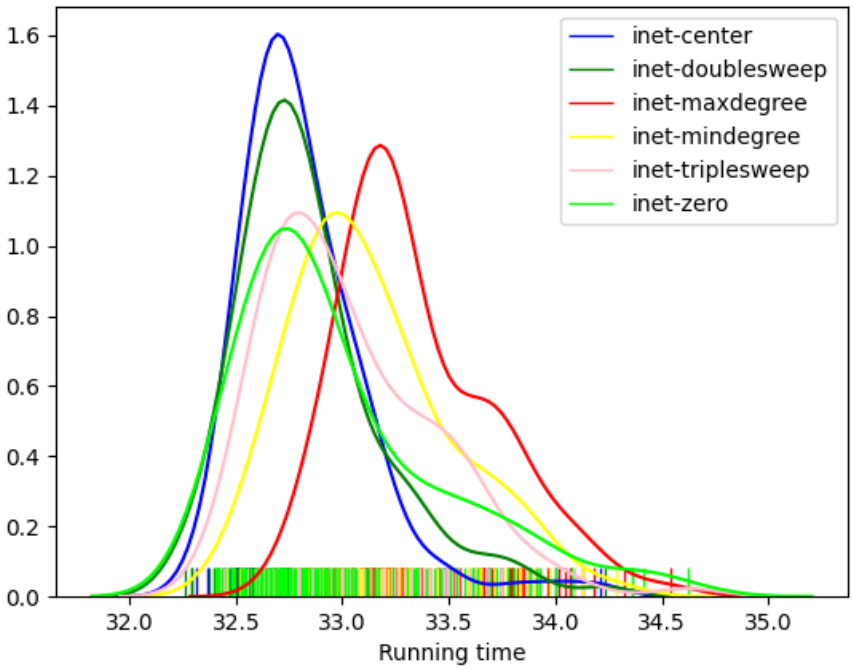

Figure 7: partially conclusive

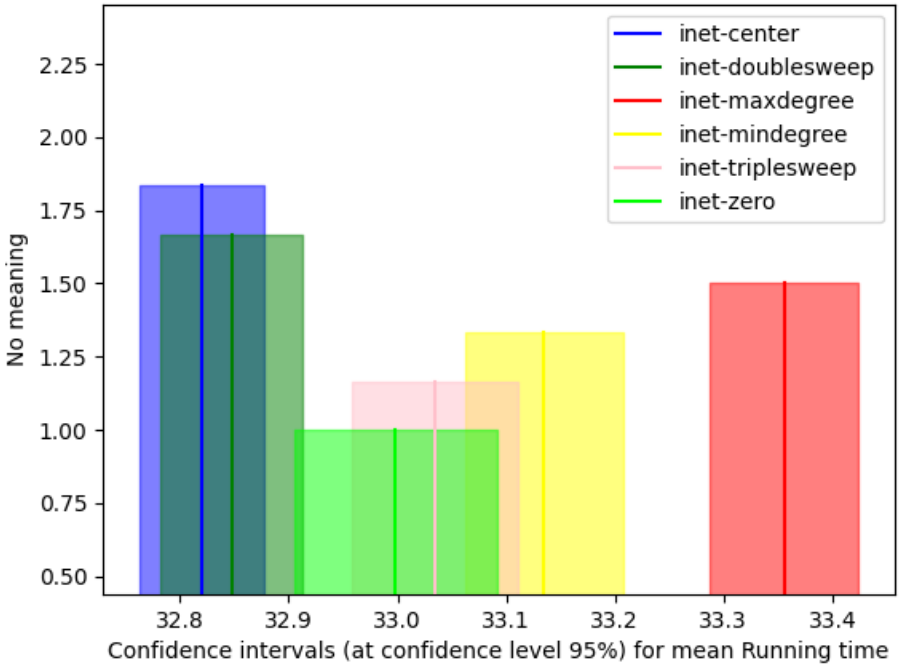

Figure 8: conclusive

5.2 ip

On the graph "ip", the Leiden algorithm was run 110 times for each prepared version of the graph. The computer used was the LENOVO ThinkPad X230i (second computer, see 4.1). On ip, the reordering takes an average of 9.5 seconds. See the next pages for the corresponding figures.

Means for graph ip	Modularity	Number of clusters	Running time
No reordering	0.27	132.31	141.15
Center node	0.27	134.52	89.28
Node zero	0.27	136.34	91.41
Max degree node	0.27	136.35	100.30
Min degree node	0.27	136.28	91.47
Double Sweep	0.27	136.43	95.20
Triple Sweep	0.27	135.88	91.10

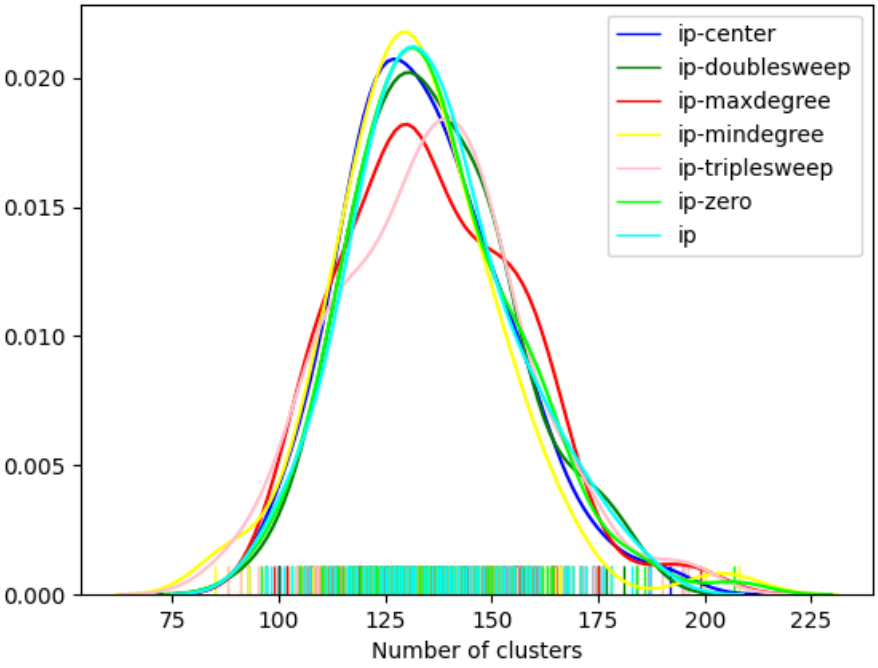

Figure 9: inconclusive

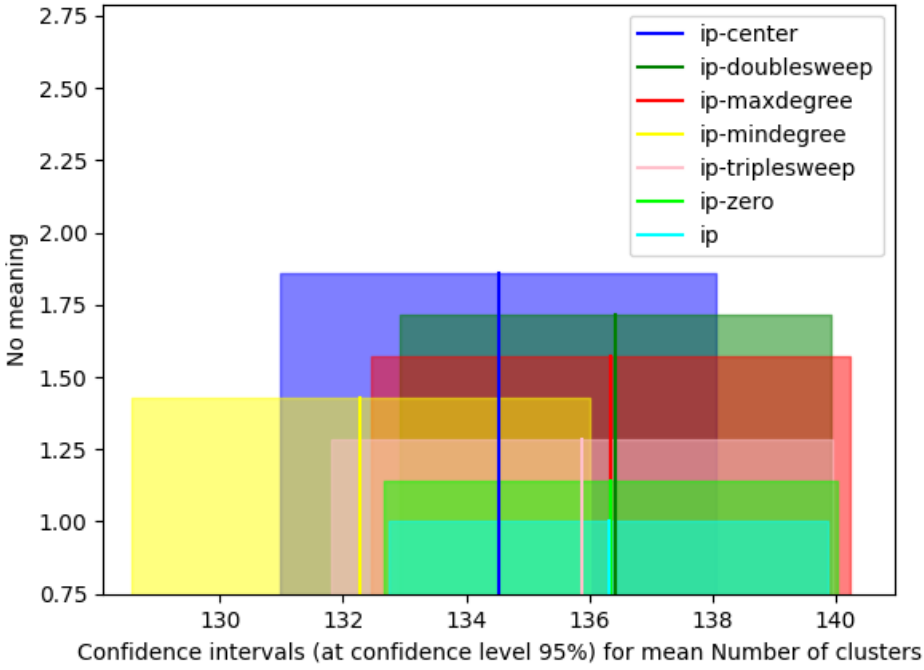

Figure 10: inconclusive

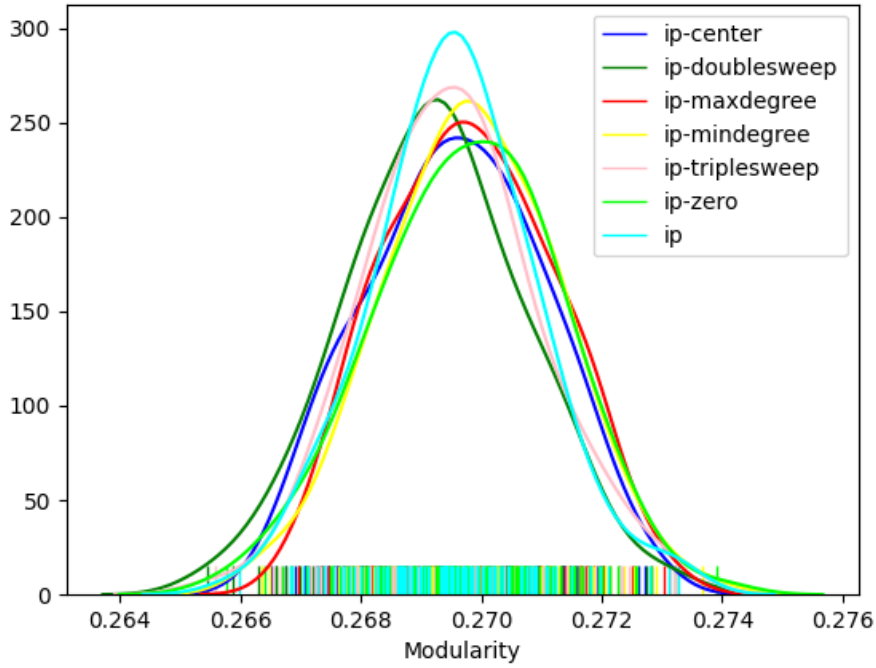

Figure 11: inconclusive

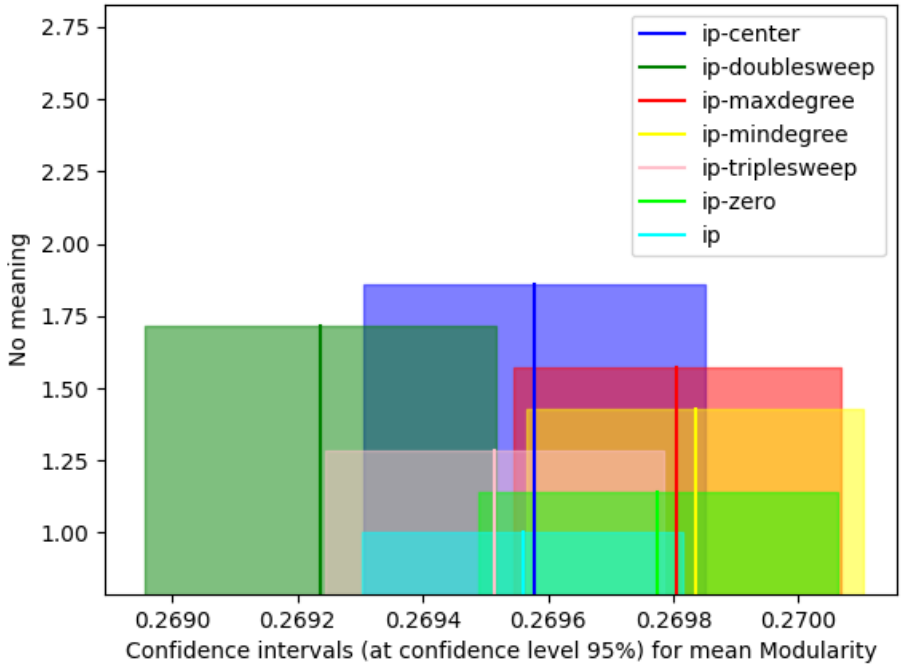

Figure 12: inconclusive

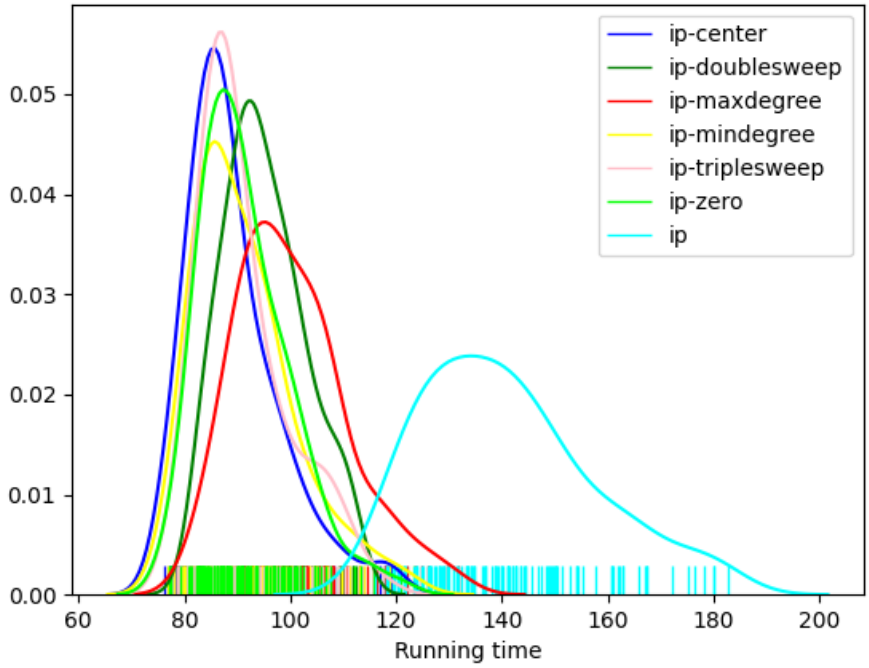

Figure 13: conclusive

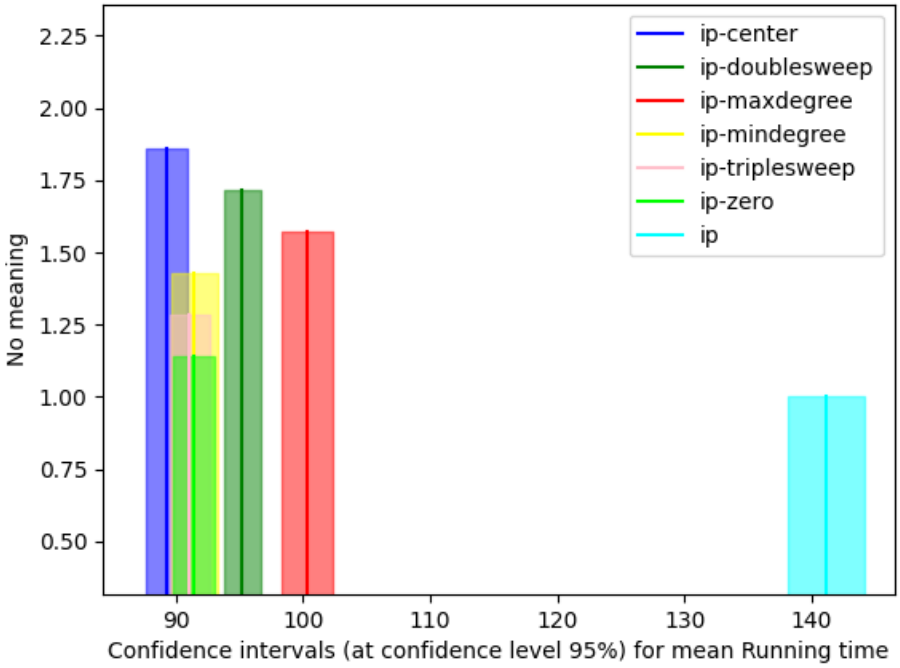

Figure 14: conclusive

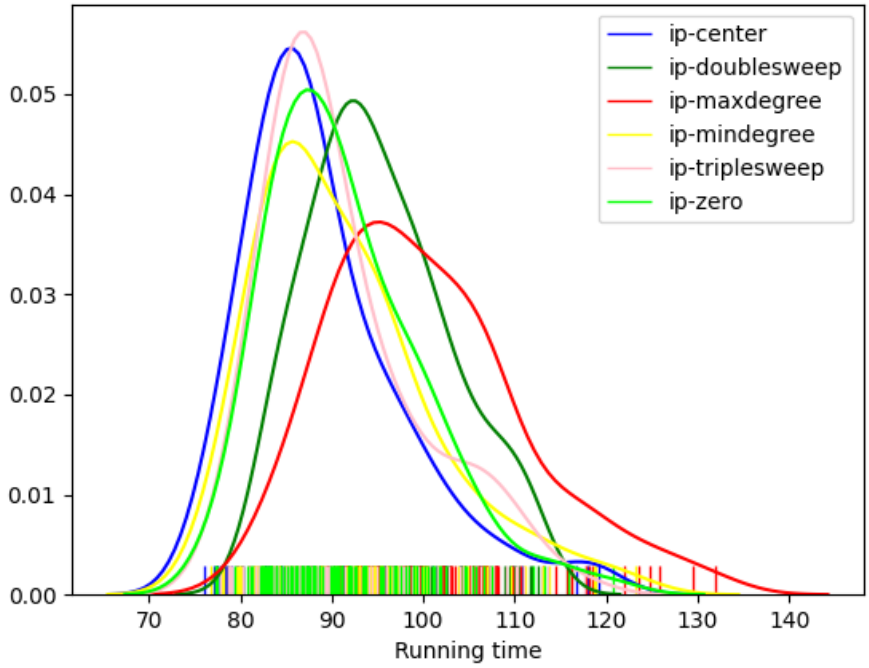

Figure 15: partially conclusive

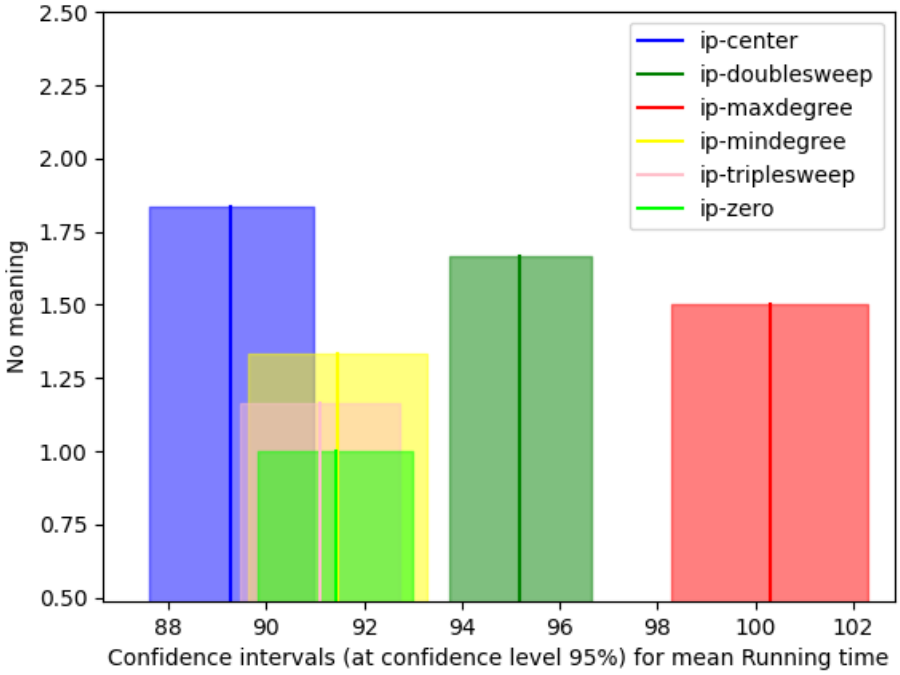

Figure 16: conclusive

5.3 p2p

On the graph "p2p", the Leiden algorithm was run 21 times for each prepared version of the graph. The computer used was the ASUS TUF Gaming A15 (first computer, see 4.1). Because the sample size is so small, the results are considered statistically inconclusive, and the results can only be found in the annex. (see section 8)

6 Discussion

From the previous section, we make several observations:

- With the inet graph, the only conclusion at confidence level 95% on the **cluster number** variation (figures 1, 2) is that the average cluster number when reordering from a node obtained with the "doublesweep" method is higher than 303, while the one when using the "maxdegree" method is lower than 302. However, this difference, while observed, is considered insignificant, especially considering that Leiden is an iterative algorithm, so those minor differences could disappear with more iterations (by default, igraph's Leiden does two iterations). On the matter of **modularity** variation (figures 3, 4), the results are inconclusive. Finally, the variations of **running time** (figures 5, 6) for the Leiden algorithm are very clear even at confidence level 99.999% between reordered and shuffled graphs. This is the main result of our study. We are confident at 99.999% that the Leiden algorithm average running time in our experimental conditions for a shuffled graph is higher than 47 seconds, while the average time on reordered versions of the graph is, with same certainty, lower than 34 seconds. With a minimal gain of 13 seconds, that represents a minimal time reduction of 27%. Considering that the reordering takes an average of 7.5 seconds for this graph, a final lower boundary for the total time reduction is 11%. Again, those are all lower boundaries with confidence level 99.999%. It also appears that there is a performance difference depending on the root node chosen for the BFS reordering (figures 7, 8). For example, reordering from a node with the maximum degree appears to be slightly worse than reordering from a central node, with a minimal difference at confidence level 95% of 0.35 seconds between both real averages. This represents a minimal average time reduction of only 1% between the two methods, even though they are the ones giving the most different results; it seems that the choice of root node for the BFS has a low impact on the time reduction for the inet graph. A very rough estimate of a ranking for the performance on each version of the graph could be, from slower to faster: shuffled, maxdegree, mindegree/triplesweep/zero/doublesweep, center; but the data we have is not sufficient to conclude on that matter. We will see how this ranking compares with the one observed on the ip graph.

- With the ip graph, we cannot conclude at confidence level 95% on the **cluster number** variation (figures 9, 10). On the matter of **modularity** variation (figures 11, 12), the results are also inconclusive. At confidence level 95%, the variations of **running time** (figures 13, 14) for the Leiden algorithm between reordered and shuffle graphs is clear, with a minimal reduction of

35 seconds between average Leiden time for shuffled graph and maxdegree reordered graph (-25%), and 46 seconds between shuffled and center reordered graph (-33%). A very rough estimate of a ranking for the performance on each version of the graph could be, from slower to faster: shuffled, maxdegree, doublesweep/mindegree/zero/triplesweep, center. This correlates positively with the observations made on the inet graph: the non-reordered graph gives the worst performance, then comes the one reordered from a node with the maximum degree, then all the other reordered versions, with the best version being the graph reordered from the center node (figures 15, 16). These observations are only estimates, and not definitive results; further testing is required, especially with several other graphs.

- With the p2p graph, the inconclusiveness might be explained by the greater size and most probably by the lack of tests, resulting from the computational difficulty to obtain them. Indeed, since the run-time was much longer on p2p, only 21 tests on each prepared graph could be done. We cannot conclude without more tests.

Depending on the choice of reordering method and bound (upper or lower) for the confidence intervals, here are the maxima and minima in percent for the average time reduction between the shuffled graph and any reordered graph, no matter the method:

Average time reduction	Minimum	Maximum
inet	30%	33%
ip	26%	38%

On the inet graph, since all reordering methods gave average run time between 32 seconds and 34 seconds (and with a very low variance) (figure 8, the differences between the minimum and maximum average time reduction are small (3%). We conclude that the choice of root node for reordering does not matter on that graph, as long as the graph is reordered. The average running time when reordering from a node with maximum degree is less than 1 seconds longer than when reordering from a central node.

On the ip graph, the difference between minimum and maximum average running times is higher. This is due to a higher variance on the sampled times for each reordered graph, but also to a more notable difference in acceleration between graphs reordered with different nodes : the average running time when reordering from a node with maximum degree is roughly 10 seconds longer (~100s) than when reordering from a central node (~90s) (figure 16). While it was also true in the case of the inet graph, the contrast is greater in the case of the ip graph, even when putting its values to scale with the shorter average running time of inet. This difference in behavior between the two graphs is not explained; it could be linked to the structure of the graph, or simply to its size.

On the two sufficiently tested graphs and when taking into account the time taken to reorder, the reordering results in a minimal average time reduction of 10%

As long as the graph is reordered, the root node choice for the BFS reordering seems to have only a small impact on the performance of the Leiden algorithm.

Our study's main weakness is the use of only three graphs, one of them with an insufficient number of tests. Further research should be made on a greater number of graphs, potentially random ones. Also, a specialized benchmarking environment should have been used to obtain more reliable results.

7 Conclusion

On the two sufficiently tested graphs —and in our experimental conditions— reordering the graph had no conclusive impact on the number of clusters or the modularity of the graphs computed by the Leiden algorithm. On those same graphs, the effects of graph reordering on the performance of the Leiden algorithm are similar to those observed with the Louvain algorithm. When accounting for the time it takes to reorder the graph, we find a minimum average time reduction of 10% on reordered versions of the inet graph. When accounting only for the Leiden algorithm running time, the maximum average time reduction at confidence level 95% reaches 38% on the ip graph, reordering with a BFS starting from a central node instead of keeping a shuffled graph. On the studied graphs, it appears that all of the proposed root node choices give comparable time reduction, with the central nodes giving the best, and the nodes with highest degree giving the worst.

According to the mentioned LSE article[3], the number of cache misses —and therefore the time reduction— change depending on the computer's architecture; the same experiments could be reproduced on different architectures and with more graphs for potentially new conclusions.

Bibliography

[1] V. A. Traag, L. Waltman, N. J. van Eck. From Louvain to Leiden: guaranteeing well-connected communities
https://www.nature.com/articles/s41598-019-41695-z

[2] Breadth-first search algorithm
https://en.wikipedia.org/wiki/Breadth-first_search

[3] Lionel Auroux, Marwann Burelle, Robert Erra. Reordering Very Large Graphs for Fun & Profit. International Symposium on Web AlGorithms, Jun 2015, Deauville, France. ffhal-01171295
https://hal.archives-ouvertes.fr/hal-01171295/document

[4] Vignesh Balaji, Brandon Lucia. When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs, Carnegie Mellon University
https://users.ece.cmu.edu/~vigneshb/papers/IISWC2018-final-preprint.pdf

[5] The documentation of the Python igraph library for its Leiden algorithm
https://igraph.org/python/doc/igraph.Graph-class.html#community_leiden

[6] Link to the graphs used
http://data.complexnetworks.fr/Diameter/

[7] Fast Computation of Empirically Tight Bounds for the Diameter of Massive Graphs Clemence Magnien, Matthieu Latapy, and Michel Habib
https://arxiv.org/abs/0904.2728
http://www-rp.lip6.fr/~magnien/Diameter *(dead link)*

[8] Basic notations in graph theory
https://en.wikipedia.org/wiki/Graph_theory#Definitions

[9] Python igraph library for graph manipulations
https://igraph.org/

[10] Python numpy library for multi-dimensional lists manipulation
https://numpy.org/

8 Annexe

8.1 Results for p2p

Means for graph p2p	Modularity	Number of clusters	Running time
No reordering	0.50	19.50	809.60
Center node	0.51	22.69	832.17
Node zero	0.51	17.81	688.49
Max degree node	0.51	19.75	686.78
Min degree node	0.51	2.38	790.72
Double Sweep	0.51	21.0	659.22
Triple Sweep	0.51	20.56	773.24

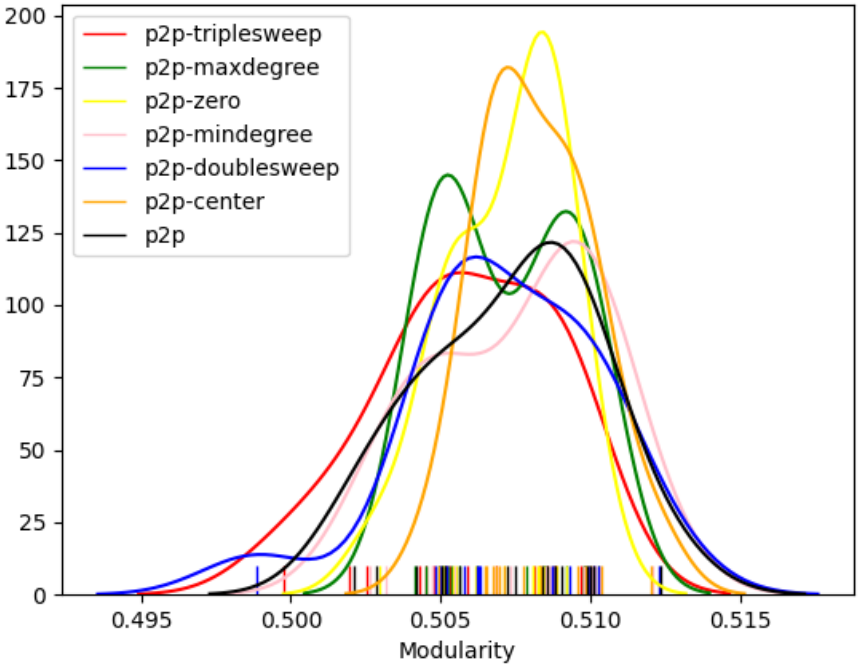

Figure 17: inconclusive

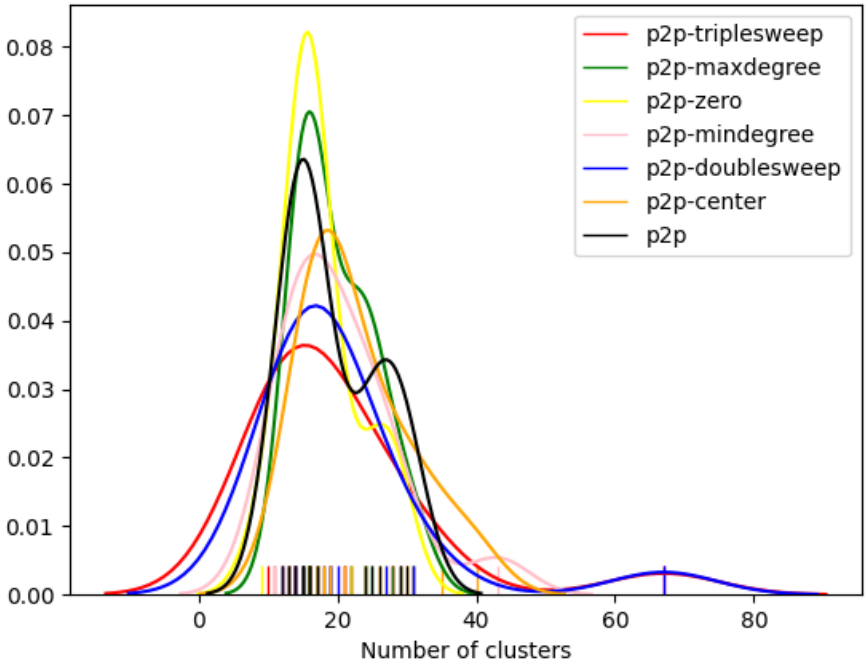

Figure 18: inconclusive

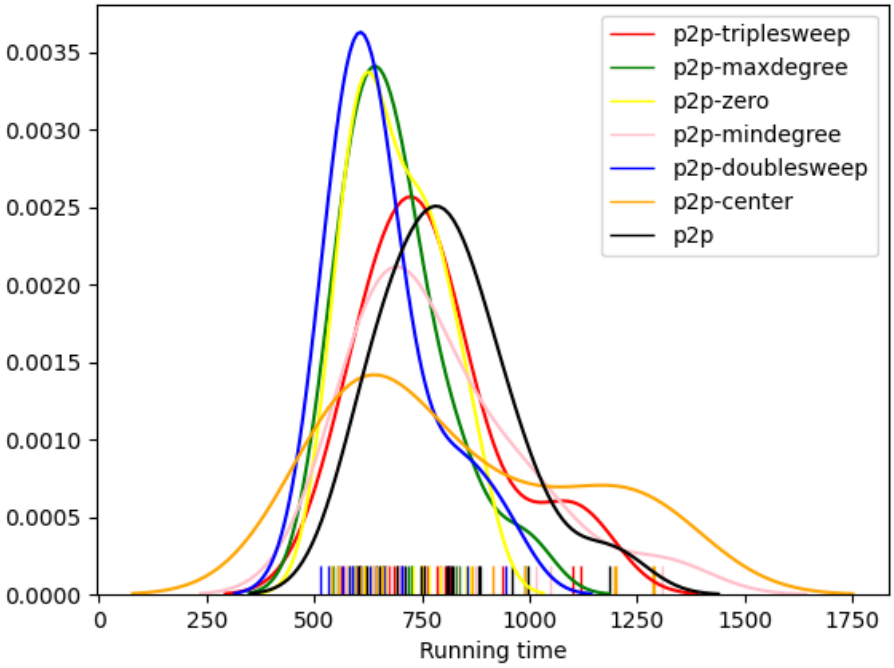

Figure 19: inconclusive

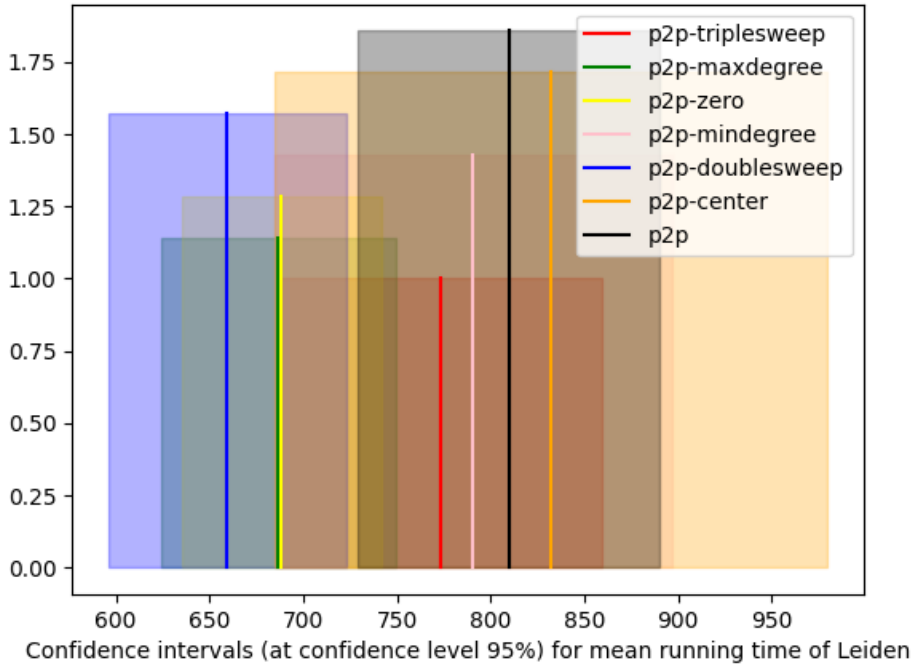

Figure 20: inconclusive

8.2 Code snippets

Creating the permutation list and reordering the graph once the bfs is computed:

```
# Get the list of nodes visited by the Breadth First Traversal (BFS)
# (the get_bfs function is detailed below)
visited = graph.get_bfs(node_choice)

# We create the permutation list from the bfs list, this gives
# us a list such as:    IF visited[i] = x THEN permutation_list[x] = i
# (the permutation_from_bfs function is shown below)
permutation_list = permutation_from_bfs(visited)

# We use permute_vertices from igraph to get the reordered graph
# Vertex k of the original graph will become vertex permutation[k]
reordered_graph = graph.permute_vertices(permutation_list)
```

Creating the permutation list from the list of nodes visited by the BFS:

```
def permutation_from_bfs(visited : list) -> list:
perm = np.empty(len(visited)) #same length
for i in range(len(visited)):
    perm.put(visited[i], i)
return perm.tolist()
```

Creating the list of nodes visited by a BFS starting from a chosen node:

```
def get_bfs(g: ig.Graph, root: str = "zero", center = -1) -> list:
    if root == 'zero':
        vid = 0
    elif root == 'center':
        if center == -1:
            print("Error: no center provided.")
            exit(1)
        else:
            vid = center
    elif root == 'maxdegree':
        max_d = 0
        for v in g.vs:
            if v.degree() > max_d:
                vid = v.index
                max_d = v.degree()
    elif root == 'mindegree': #same here
        min_d = sys.maxsize
        for v in g.vs:
            if v.degree() < min_d:
                vid = v.index
                min_d = v.degree()
    elif root == 'doublesweep':
        #last node visited by bfs from node 0
        extreme_1 = g.bfs(0)[0][-1]
        vid = extreme_1
    elif root == 'triplesweep':
        #last node visited by bfs from node 0
        extreme_1 = g.bfs(0)[0][-1]
        #again from previous last node
        extreme_2 = g.bfs(extreme_1)[0][-1]
        vid = extreme_2
    return g.bfs(vid)[0]
```