VLG Project 2: Clustering and Reordering

Mamoune El Habbari[1], Yassir Ramdani[2].

[1]mamoune.el-habbari@epita.fr, *SCIA 2021, EPITA.*
[2]yassir.ramdani@epita.fr, *SCIA 2021, EPITA.*

May 30, 2020

Abstract

In graph theory, clustering is a very important subject that finds its usefulness in computer science, especially image segmentation and complex network analysis but also in other fields like biology and medicine. Oftentimes, this clustering is resource intensive and takes a substantial amount of time to compute. Our goal is to measure the extent of how much a reordering of the graph impacts not only the computing time, but also the modularity. First, we run a clustering algorithm (in our case here Louvain) a first time on different very large (sparse) graphs and then we run it a second time after having reordered the graph using a BFS traversal. Now the most important thing to keep in mind is that the reordering is radically different depending on which point is chosen when doing the BFS traversal. That is why we experimented with several different key nodes of the graph, namely the center, peripheral nodes and the ones with maximum/minimum degree.

Introduction

A little bit of vocabulary

First, let's start by defining the terms that we are going to be using throughout the report.

- **Modularity:** a metric that helps us evaluate how effective the clustering was. The closer it is to 1, the better the clustering is.
- **Distance:** the shortest path between two nodes.
- **Eccentricity:** the maximal distance between a node and all the others.
- **Center:** node with a minimal eccentricity.
- **Peripheral nodes:** nodes that have a maximal eccentricity i.e. the furthest from the center.
- **Community:** a network is said to have community structure if the nodes of the network can be easily grouped into (potentially overlapping) sets of nodes such that each set of nodes is densely connected internally.[4]

Testing environment

The performance of the different algorithms was measured on a MacBook Pro with an Intel i9 chip (8 cores) and 16 GB of RAM, running macOS Catalina 10.15.4.

During all the tests, the computer was running only 1 terminal window (with our program), 1 Finder window and Sublime Text 3.2.2 (build 3211).

The program records all the execution stats (the graph name, Louvain execution duration, modularity, communi-

ties number, reorder id (if any), node type (if a reordering was applied)) in a *'records.csv'* file.

Each value is the mean of 3 executions of our program.

graph	duration	modularity	communities_number	reorder_id	node_type
main_comp/inet	43.506756	0.721962	206	27589	min_degree
main_comp/inet	43.554630	0.721962	206	28862	min_degree
main_comp/inet	43.730381	0.721962	206	35299	min_degree
main_comp/inet	43.910637	0.721962	206	37048	min_degree

Figure 2: Preview of *'records.csv'*

igraph

We rely heavily in our project on igraph. The latter is an open-source library written in C that provide a wide range of methods useful for creating and manipulating graphs. It's fast, reliable and capable of handling very large datasets efficiently. Furthermore, it allowed us to free ourselves from low-level programming and significantly improve our productivity.

The Code

The program

All the steps to compile the project are described in the *'README.md'*. Once compiled, you can use the program as following:

```
sh$ ./vlg graph_path [vertex_reorder_id
                      vertex_type]
```

Where '*vertex_reorder_id*' is the id of the vertex used as the reordering root and '*vertex_type*' is the type of this node (center, peripheral, min_degree, max_degree, ...)

Note: *'vertex_reorder_id'* and *'vertex_type'* are optional. If they are not given, the program will run Louvain without reordering.

```
yassram →  my-graph git:(master) × ./vlg main_comp/inet
[info] Loading main_comp/inet
[info] Loaded main_comp/inet
[info] Applying Louvain
[result] louvain time: 53.836859
[result] modularity: 0.710001
[result] number of communities: 446
[info] Saving stats in records.csv
```

Figure 1: The output of executing Louvain on 'ip graph' without reordering

The program records all the execution stats (the graph name, Louvain execution duration, modularity, communities number, reorder id (if any), node type (if a reordering was applied)) in a *'records.csv'* file.

Implementation

We chose to implement our code in C to profit from the computational speed that it offers.

The program can be divided in several distinct steps:

1. Main component extraction
2. Load freshly computed main component
3. Reordering
4. Clustering
5. Recording results in a csv file

First, we wrote a quick program that take a graph as input and outputs the main component as another graph. Then, we started by loading the main component of each graph in memory using the interface provided by igraph.

At the reordering step, we are running a BFS traversal on specific points that were determined beforehand and retrieved the order. This order was used in conjunction with *igraph_permute_vertices* to obtain a newly reordered graph.

When clustering, we merely called *igraph_community_multilevel* to run Louvain on the graph. We recorded how long it took for the algorithm to complete and we extracted the number of community as well as the modularity. All this data is then written to a CSV file which would allow us to perform some stats on it.

Algorithms

Connected components decomposition

We used the method *igraph_decompose* to decompose the graph in connected components. Indeed, it makes little sense to run Louvain on parts of a graph that are not connected because we specifically want to highlight communities inside a connected network.

The time complexity of this algorithm is in $O(|V|+|E|)$ where V is the number of vertices and E the number of edges.

The algorithm is very simple: with a basic BFS traversal it adds any node that it has not encountered yet to a component of nodes. When every node of the connected component has been reached, the whole graph is copied and the vertices that do not belong to the component are removed. A vector of connected graphs is then returned.

Clustering (Louvain)

The igraph library implements the Louvain algorithm in its *igraph_community_multilevel* method. This clustering algorithm was created by Blondel, an applied mathematics professor at UCLouvain. It can handle graphs with up to 100 million nodes and billions of edges and it sports an $O(n \cdot \log(n))$ complexity.

The algorithm runs in two steps. The first one is the modularity optimization where in each iteration, it tries to maximize the modularity until it no longer can. This step is where all the computational effort is spent. The second step is where each small community is merged into a single node.

As stated above, a modularity of 1 means that the graph is fully modular and that the partitioning into communities is optimal while a modularity of -1/2 means that the communities are not dense at all.

Today, Louvain is one of the most efficient methods of community detection and consequently is very widely used.

Reordering (BFS)

Reordering a graph consist in swapping the vertices' indexes of the graph resulting an isomorphic graph.
Graph reordering is considered an optimisation of many large graph algorithms by enhancing the distribution of the graph data in memory to take benefit of the cache and avoid memory jumps.
This applies very well to reduce the Louvain computation time as well.

However choosing the order is decisive on the efficiency of the reordering. *A bad reordering can make the computation time even worse.*
Since we are dealing with large graphs, the reordering algorithm complexity should be taken into consideration.

In this project we used BFS[1], a simple but fast algorithm ($O(|V|+|E|)$) to traverse a graph, starting from a given node (the root of the new reordered graph) and renumber the nodes by following the order of the traversal.

[1] BFS: Breadth First Search

Experiment

The objective of this experiment is to evaluate the efficiency of different graph reordering in reducing the computing time of Louvain on a graph.

Since we use BFS for reordering, we note that the new graph vertices' order depends on the chosen root node in the BFS traversal. Through this project we have tried different kind of *particular* vertices (central, peripheral, min degree, max degree) in order to evaluate their reorder efficiency in reducing the computing time of Louvain.

Note that these particular nodes are considered as known (through an initial analysis of the graph) therefore their computation time is not taken into consideration.

The process

The experimental process of each measurement is quite simple.

1. Extract main component from the graph.
2. Reorder the graph by applying BFS algorithm with the given node as root.
3. Execute Louvain algorithm by measuring computing duration.
4. Save the graph name, reordering node type and reordering node id (if given) ,Louvain duration, modularity value and communities number in *'records.csv'*.

Studied graphs

Graph	Vertices	Edges	Radius	Diameter
inet	1.7m	11m	16 ~18	31 ~36
ip	2.2m	19m	~6	~12
p2p	5.8m	142m	~6	9 ~12

inet

internet topology graph (inet) obtained from traceroutes ran daily in 2005 by Skitter from several scattered sources to almost one million destinations, leading to 1 719 037 vertices and 11 095 298 edges[1].
link: `http://data.complexnetworks.fr/Diameter/inet.gz`

ip

a traffic graph (ip) obtained from MetroSec which captured each ip packet header routed by a given router during 24 hours, two ip addresses being linked if they appear in a packet as sender and destination, leading to 2 250 498 vertices and 19 394 216 edges[1].

link: `http://data.complexnetworks.fr/Diameter/ip.gz`

p2p

a peer-to-peer graph (p2p) in which two peers are linked if one of them provided a file to the other in a measurement conducted on a large eDonkey server for a period of 47 hours in 2004, leading to 5 792 297 vertices and 142 038 401 edges[1].
link: `http://data.complexnetworks.fr/Diameter/p2p.gz`

Results

Raw data

For each graph, we ran our program thrice on each significant node and we recorded the data in an csv file. The following values that are presented below are the average duration of Louvain throughout those three runs.

ip

node type	node id	Louvain duration	mod. value	commu. nº
-	-	48.005950	0.318904	158
max	1	70.369512	0.320195	80
ctr	86	72.078341	0.322209	37
min	722	111.129705	0.320939	100
min	825	64.245353	0.319881	97
min	858	108.850174	0.318975	82
min	860	103.980690	0.320939	100
prp	192554	70.772786	0.320195	80
prp	1042594	83.010455	0.320422	88
prp	1806604	71.162140	0.320195	80

inet

node type	node id	Louvain duration	mod. value	commu. nº
-	-	41.749084	0.710001	446
max	2	48.481363	0.733372	167
min	27589	43.179610	0.721962	206
min	28862	42.670842	0.721962	206
min	35299	42.783597	0.721962	206
min	37048	43.582502	0.721962	206
min	41200	43.386387	0.721962	206
ctr	165547	44.233944	0.725383	265
ctr	208405	43.021787	0.726560	256
ctr	208406	59.309776	0.727648	178
prp	1511903	39.711643	0.729594	203
prp	1717718	48.935015	0.733372	167

p2p

node type	node id	Louvain duration	mod. value	commu. nº
-	-	1322.774658	0.497106	9
max	1	850.277160	0.491474	8
min	4027	997.246603	0.495739	9
min	4092	998.226420	0.495739	9
min	4333	996.975586	0.495739	9
min	4342	1006.326436	0.495739	9
min	4345	997.185079	0.495739	9
ctr	17444	1384.289551	0.499244	9
ctr	18632	959.997457	0.499837	9
ctr	30438	971.335347	0.496781	9
ctr	89448	1008.757609	0.492102	9
ctr	129670	1131.921834	0.485594	8
prp	135496	710.882080	0.491444	8
ctr	1432588	781.486938	0.495592	8
prp	1975095	661.536438	0.491460	8
prp	2034294	851.619955	0.491467	8
prp	2296169	853.465088	0.491467	8
prp	2593126	710.503499	0.491444	8
ctr	2778934	1203.224650	0.495889	8
ctr	2824085	479.412394	0.499039	8
ctr	4638343	686.150818	0.495804	7
prp	4979458	803.496623	0.491448	8
prp	5554437	830.570028	0.491468	8
prp	5664968	849.717306	0.491467	8

Observations

inet

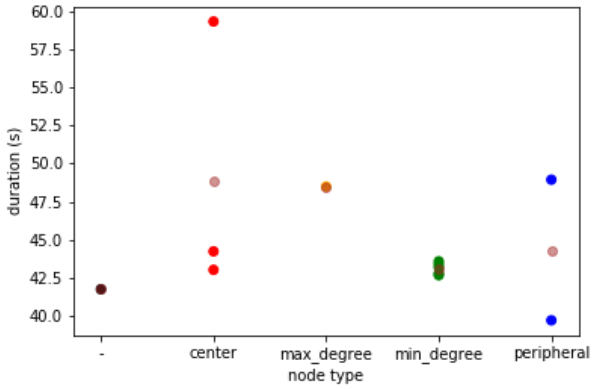

Figure 3: Louvain execution time for each node type for inet graph.

We can see that the only reordering that enhances Louvain computing duration is peripheral one (and only one of the two nodes). Note that there is a very high difference in execution time between two center nodes.

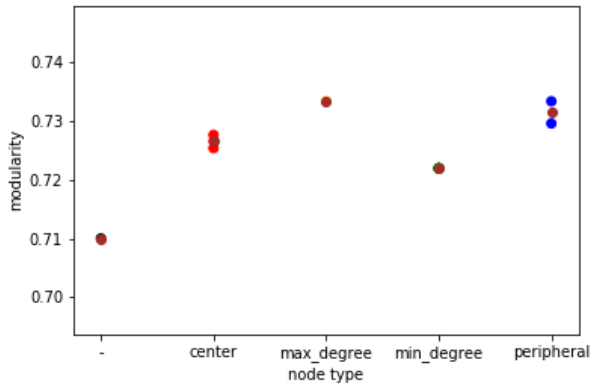

Figure 4: Modularity value for each node type for inet graph.

The value of the modularity has increased after every single reordering but it is not that much greater than the value of base Louvain.

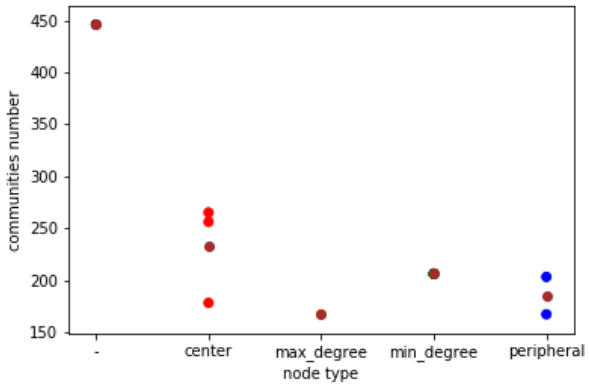

Figure 5: Communities number for each node type for inet graph.

There is significantly less communities after reordering, regardless of the node type that we chose. There is almost a 100 community difference between two center nodes.

ip

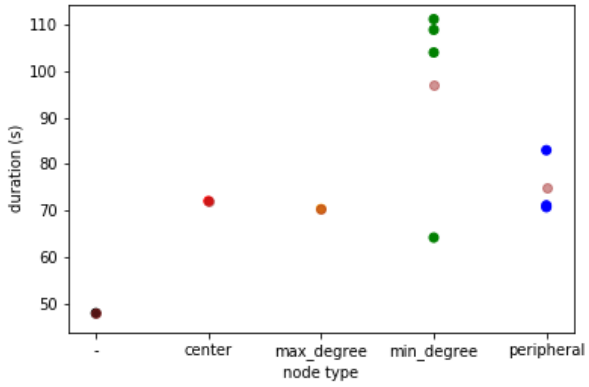

Figure 6: Louvain execution time for each node type for ip graph.

All reorderings increase Louvain execution time.

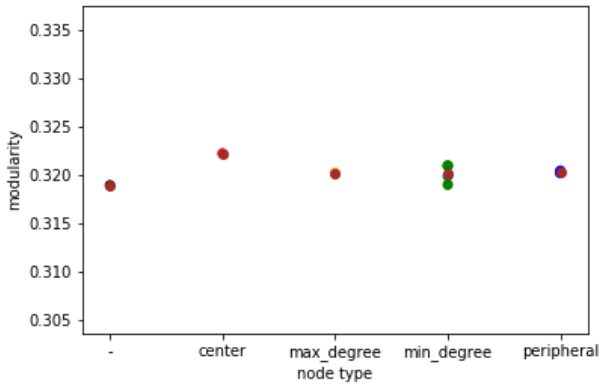

Figure 7: Modularity value for each node type for ip graph.

The modularity value is still very close to the one of base Louvain.

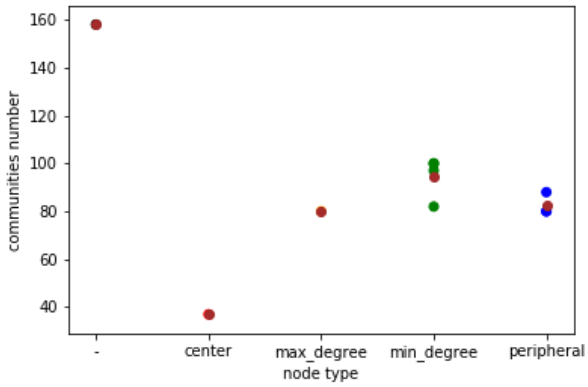

Figure 8: Communities number for each node type for ip graph.

Again, there is significantly less communities after reordering, regardless of the node type that we chose.

p2p

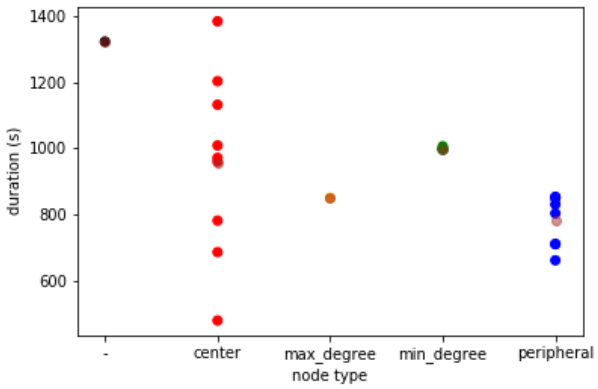

Figure 9: Louvain execution time for each node type for p2p graph.

Some center nodes give us very good optimization of Louvain execution time while others do not. We note that all peripherals, max degrees and min degrees enhance it.

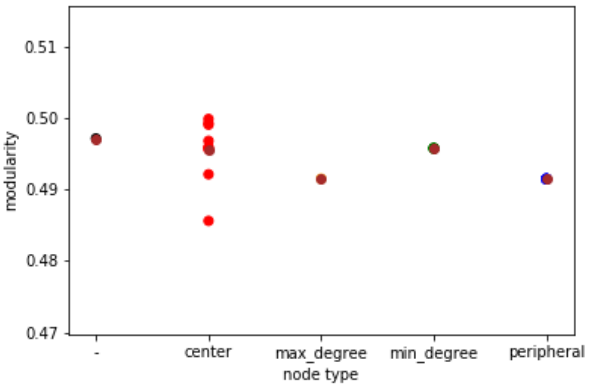

Figure 10: Modularity value for each node type for p2p graph.

The modularity values are still very close to each other although there is an observable disparity between the modularities of the center nodes.

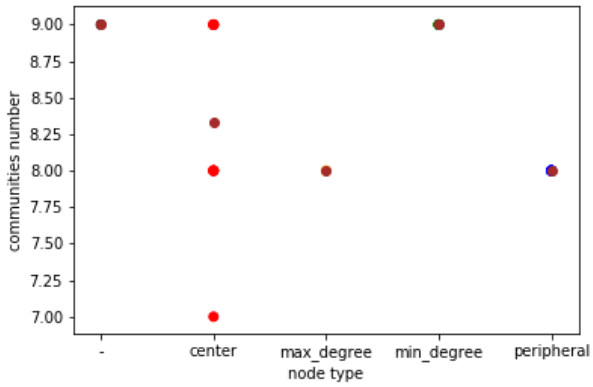

Figure 11: Communities number for each node type for p2p graph.

Only min degree nodes keep the same community number as without reordering. Other nodes decrease the number of communities.

Overall

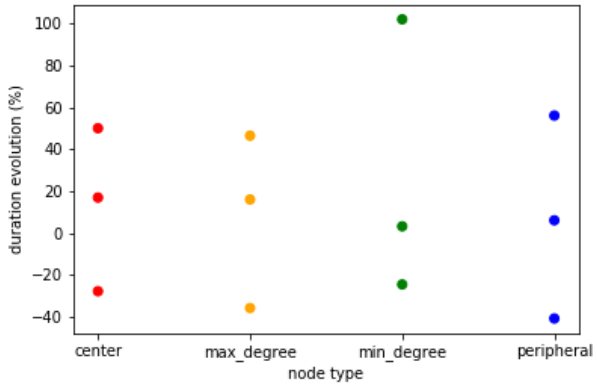

Figure 12: Louvain execution time mean evolution compared to computing without reordering for the 3 studied graphs. (Lower is better)

Overall, the evolution of the mean value of the execution time seems to be the same when you compare a reordering with a center node or with a max degree node. The peripheral nodes seem to give the best performance of the algorithm.

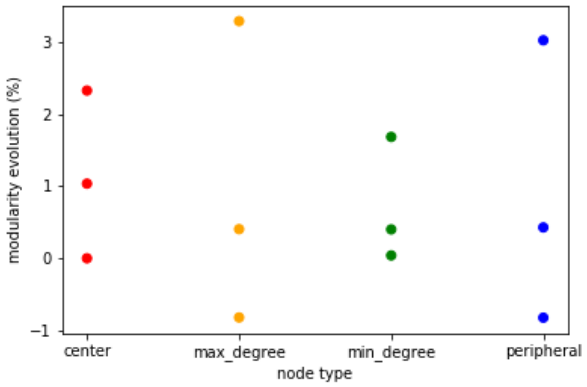

Figure 13: Modularity evolution compared to computing without reordering for the 3 studied graphs. (higher is better)

Interpretations

Louvain computing time

When we started the experiment, we expected to see a clear improvement on the performance of the algorithm after reordering, but this was only really apparent when we ran it on the biggest graph with more than 140 million edges (p2p). For the inet graph, only one of the peripheral nodes guarantees a better performance and it is really not enough for the trouble.

In terms of computation speed, it is not worth to run Louvain on reordered graphs that only have ten to twenty million edges. However for very large graphs it is getting interesting as it is almost 40% faster to run Louvain on a graph reordered by a peripheral node.

The peripheral node seem to really be the best node to choose when we consider the different kinds that we focused our attention on. The execution times between different peripheral nodes are very close to each other, which means that whatever peripheral node you chose, you are guaranteed to have acceptable performances.

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

Figure 14: Latency comparison key numbers[2]

One possible explanation for the impact of reordering on the performance of the Louvain algorithm is that the new graph takes full advantage of how the memory is represented in a computer so that the CPU caches are used at their maximum, decreasing by that cache misses and memory jumps.

Clustering efficiency: modularity

When taking into account the modularity after each reordering, we can observe that the value does not vary a lot. This makes sense because reordering a graph does not affect its structure so it is essential the same with only different labels on each node, thus the decomposition into communities should not be drastically different between a graph and a reordered version of it. The small difference can be explained by the difference of the indexes since Louvain uses a heuristic method that is based on modularity optimization[3].

The case of the central nodes

The experiment highlighted a very interesting aspect concerning the central nodes. Indeed, we ran Louvain on graphs reordered with several different ones and all these experiences points to the same exact conclusion: these set of nodes cannot be trusted. The computing times for different central nodes is wildly inconsistent and the same can be said for the value of the modularity or the number of communities. This can only mean one thing: being a center has no real significance and does not necessarily mean that the reordering is going to give a better time on Louvain. This special kind of node does not hold a real value in our case and there is no reason to use them.

Conclusion

Clustering very large graphs can be costly in time. Reordering proves to be a good solution to enhance Louvain computing time. Since we are dealing with millions of nodes reordering algorithm complexity should be as low as possible. A simple BFS traversal can be sufficiently efficient if the new graph's root is well chosen. After lot of measurements, we can confirm that reordering has an impact on the performance of the Louvain algorithm. It is quite visible that peripheral are giving the best overall results with modularity taken into consideration. We clearly need more results with different graphs to understand these results better.

References

[1] Clémence Magnien, Matthieu Latapy, Michel Habib. *Fast Computation of Empirically Tight Bounds for the Diameter of Massive Graphs* April 17th, 2009 https://arxiv.org/pdf/0904.2728.pdf

[2] Peter Norvig. *Teach Yourself Programming in Ten Years* http://norvig.com/21-days.html#answers

[3] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, Etienne Lefebvre, *Fast unfolding of communities in large networks* https://arxiv.org/abs/0803.0476.

[4] *Community structure* https://en.wikipedia.org/wiki/Community_structure.