EPITA

GRADUATE SCHOOL OF COMPUTER SCIENCE

VERY LARGE GRAPHS

Fast Very Large Graph Diameter Estimations Using Quotient Graph

Yossra ANTARI - Axen GEORGET - Alex VAN VLIET

yossra.antari@epita.fr - axen.georget@epita.fr - alex.van-vliet@epita.fr

supervised by

Robert ERRA - Alexandre LETOIS - Mark ANGOUSTURES

May 2020

Table of Contents

1 Introduction

Since the beginning of the computers a few decades ago, everything is more and more connected. The computers, the roads, the cities, the people themselves, etc. This ultra-connected world can be interpreted as complex networks (or graphs) with each node corresponding to something linked in some way to something else. For instance, those links can represent roads between cities, connections between people in a social network, etc. In graph theory, we call these nodes vertices and these links edges.

Many statistics can be computed in these networks. But the problem is often that the computation of these statistics can require a significant amount of computing power. Depending on the graph size it can take hours just to compute simple statistics.

The diameter of a graph is one of those statistics. It is the greatest shortest path between two of its vertices. The notion of diameter of graphs can be used for example to model the quality of service constraints in data networks. Considering the graph of network-connected computers, the diameter would be the longest path data could take from one computer to another. Obviously, reducing the diameter of the graph means that the longest path is shorter, meaning that the connection is faster, and the quality of service is higher. However, as it is imaginable, there are a lot of computers. One graph, which will be detailed later, representing the IP addresses has more than 2 250 000 nodes. Even listing each pair of nodes would represent 5 062 500 000 000 computations, meaning that if 1000 pairs were listed every millisecond, it would take more than 58 days to show them all. As the useful theoretical properties of the diameter in the design of such networks have been pointed out, it seems logical that efficient algorithms for calculating the diameter of a graph can be very useful.

Several methods to approximate the diameter of a graph have been showcased in several studies. For example, the very common double sweep. This paper presents our research which was about having a more precise computation of the diameter of a graph using its quotient graph and the double sweep algorithm.

2 Graphs Used

The tests in this report were done on the following graphs:

Graph	Vertices	Edges	Real Diameter	File Size
simple	10	21	4	84 B
Slashdot0811	77 360	469 180	12	5.1 MB
roadNet-CA	1 957 027	2 760 388	865	41 MB
inet	1 694 616	11 094 209		154.3 MB
ip	2 250 046	19 393 724		254.4 MB
p2p	5 380 491	142 038 351		2.19 GB
web	39 252 879	781 439 892		13.47 GB

The vertices and edges in this table are representing the vertices and edges of the largest connected component of the graph. The diameter of the simple graph was computed but we could not compute the real diameter of the other graphs, it was taking way too much time. The diameters for the Slashdot0811 and roadNet-CA graphs was available on the SNAP website but there were some mistakes. The real diameters of those can be found on their respective research papers.

Simple

This graph is a hand-made connected-graph and does not represent anything specific. It just has a test purpose and since it is a small graph it is easy and fast to compute different statistics on it.

Slashdot0811

This graph is extracted from the Slashdot Zoo introduced in 2002 by Slashdot, a technology-related news website. The graph contains links between the users of the website. This graph is provided by the Stanford University's project called Stanford Network Analysis Project (SNAP) and was collected in November 2008.

roadNet-CA

This graph simply represents the road network of the California State (United-States of America). The intersections and endpoints are represented by the vertices and the roads by the edges. It is also provided by the SNAP project.

inet

This graph represents the links between IP addresses. It was collected by the Skitter project between January 2005 and May 2006 by probes all around the world. Those probes were executing traceroute to collect the data. This graph can be found in Clemence Magnien and Matthieu Latapy's directory gathering massive graphs.

ip

This graph represents the packages that went through a router. In the context of the MetroSec project, between March the 7th 2006 at 08:10 am and March 15th, 2006 at 02:22 pm, all the headers which were going through a specific router were saved. From those headers, the destination IP address and the sender IP address were extracted to make this graph. Again this graph can be found in Magnien and Latapy's directory.

p2p

This graph is an exchange graph representing the peer-to-peer file exchange systems between users. The capture of the exchanges was made between May 8th, 2004, and May 10th, 2004. Once again this graph can be found in Magnien and Latapy's directory.

web

Finally, this graph is representing the links between a set of web pages. To get this graph a crawl of all the pages in the .uk domain was made July the 11th 2005 at 00:51 and July the 30th 2005 at 23:24. Again this graph can be found in the Magnien and Latapy's directory.

3 Preprocessing

The graphs we used for this research were not in the same format and we thus had to process a few of them to have everything in the same format. This preprocessing consists of two operations, the format cleaning and the extraction of the largest connected component.

Considering the base format of the graphs we used, here is a summary of the preprocessing operations made on them:

Graph	**Format Cleaning**	**Largest Connected Component Extraction**
simple		
Slashdot0811		x
roadNet-CA		x
inet	x	x
ip	x	x
p2p	x	x
web	x	x

3.1 Format Cleaning

Considering the program only needs the list of edges we can and need to remove the useless information in the files. For instance, the inet graph is including the degree of each node along with the number of nodes in the graph. To remove that the following bash command has been used:

```
$ tail -n +(number_of_lines_to_remove + 1) file > file
```

The "number_of_lines_to_remove" corresponds to the number of nodes + 1 so we remove the first line and all the lines describing the degrees. For instance, if there is a graph with 50 vertices the "number_of_lines_to_remove" will be equal to 51.

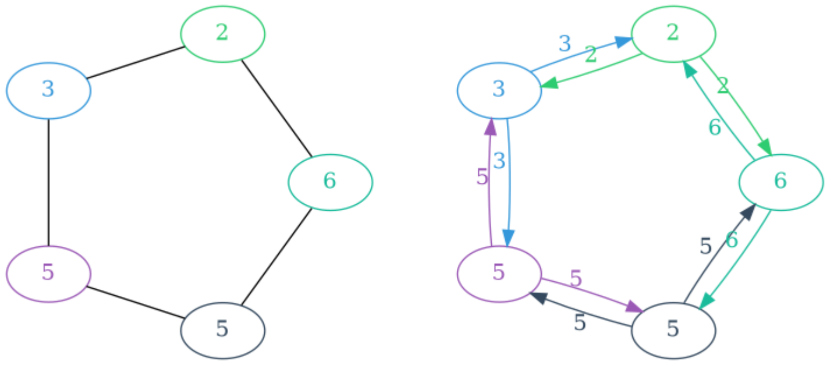
3.2 Largest Connected Component Extraction

The goal of all the algorithms implemented during this research is to find the diameter of a graph. When dealing with a disconnected graph, there are two ways to compute the diameter. The first one would be to say that because one node cannot be reached, the diameter is ∞. However, this does not provide any information because it makes sense to have disconnections in a real-world graph, and there are simple algorithms to detect such disconnections accurately. The second one, more interesting, is to consider the diameter of the graph to be the diameter of its largest connected component. Because we are dealing with very large graphs, it is more time-efficient to start by extracting this component and saving it to a file. A small program has been written to do so. It basically consists of computing the components, selecting the largest one of them, and exporting the resulting graph. During this stage, multiple edges connecting the same vertices are also merged together and loops are removed.

4 Algorithms implementations

4.1 First implementation

At first, we tried to calculate the graph diameter by following these steps: given a graph G, we run the Louvain algorithm which will return a list of communities. Using the communities, we create the quotient graph G_c. We weighed the quotient graph with the estimations of the components diameters after the multiple sweep (see below).



(a) Graph communities, the number inside of the vertices represent the diameter value of the component

(b) After multiple sweep, weighed graph

Figure 1: Schema enlightening our first naive implementation

4.2 Second implementation

In the *PROJET-VERSION-LONGUE-VLG-2020*, the following algorithm was explained:

- Given a graph G, we run the Louvain or Leiden algorithm which will return us a list of communities.
- Using the communities, we create the quotient graph G_c.
  1. A vertex from G_c is a community from G
  2. In G_c, there is an edge from a vertex i to a vertex j iff, in G, there is a vertex in the community C_i which is connected to a vertex from the community C_j.
- We calculate the exact (if possible) diameter value of quotient graph G_c.
- We take the communities of this diametric path : C_i and C_j.
- We choose the smallest community C_i of the diametric path.
- Every vertices of C_i are used to be candidate to start a BFS for the double-sweep instead of a random vertex.

At first, we implemented this exact algorithm but after a few tries, we quickly saw that it was taking way too much time to execute it. So we changed the last part of it: instead of iterating through every vertex of C_i we decided to iterate through only a few of them.

Also, we are not using the smallest community from the diametric path, but the first one. This is done because if it is on the outside of the quotient graph, it would make sense to be outside of the original graph as well.

5 The Code

The programs used in this experiment were implemented in the C programming language. This is mainly for performance reasons: C offers low-level memory management and thus allows to optimize it quite well. The igraph library is used to load and perform operations on graphs.

The version-control system git is used and the project is hosted in the GitLab online repository manager which makes the teamwork easier. A Continuous Integration pipeline was also set up using the GitLab CI/CD system. This is used to check that everything which is pushed compiles well and is quite useful to quickly detect problems in the code.

In order to build the code, we use the cross-platform build-system generator CMake. This is a very powerful tool that helps building projects of different sizes. In our case, even though the program is not very big it seemed very important to have a modular build system like CMake.

The program consists of three different sub-programs. They all have a different purpose and helped us during this research.

extract

This sub-program takes a graph file in parameter and will perform the extraction of the Largest Connected Component. The new graph is printed in the standard output stream and some information about the graph and the process are printed using the standard error stream. For instance, we use it for the inet graph this way:

```
$ ./extract inet.txt > inet.clean.txt
```

graph

This sub-program is the main program that helped us during this experiment. It takes several options in parameter along with a graph file. The main purpose of it is to execute the implemented algorithms and print a lot of information about it. This was useful to implement the algorithms efficiently and test them easily. Here is the list of the different options it can accept:

Option Name	Option Description
–help	Print the help message
–dot-original	Print the original graph in dot format
–dot-colored	Print the original graph in dot format, with the clusters grouped together and colored
–dot-quotient	Print the quotient graph in dot format
–use-louvain	Use Louvain instead of Leiden for communities computation
–print-membership	Print the membership of each vertex from the original graph
–quotient-try-all	Try all the start vertices in the selected community

For instance we can execute this sub-program with the simple graph:

```
$ ./graph simple.clean.txt
--------------------------------------------------
GENERAL INFORMATION:
Name: simple.clean.txt
Vertices: 10
Edges: 21
Directed: 0

--------------------------------------------------
DOUBLE SWEEP ALGORITHM:
Diameter (double sweep): 4

--------------------------------------------------
QUOTIENT STARTING DOUBLE SWEEP ALGORITHM:
Running Leiden (resolution: 0.023810, beta: 0.010000)
Clusters: 2
Modularity: 0.379819
Counts: 6 4
Diameters: 2 1
Name: quotient
Vertices: 2
Edges: 1
Directed: 0
Quotient diameter: 1
Quotient longest path: 0 1
Diameter (double sweep from starting community, n: 1): 4
Diameter (double sweep from starting community, n: 2): 4
Diameter (double sweep from starting community, n: 3): 4
Diameter (double sweep from starting community, n: 4): 4
Diameter (double sweep from starting community, n: 5): 4
Diameter (double sweep from starting community, n: 6): 4
Diameter (double sweep from starting community, n: 7): 4
Diameter (double sweep from starting community, n: 8): 4
Diameter (double sweep from starting community, n: 9): 4
```

benchmark

This sub-program was the one used to try the implemented algorithms and measure their performances. It only takes one graph file as an argument and executes the three algorithms multiple times. For each algorithm, it is run for at least 3 times and at least 60 seconds. For instance, on a big graph like p2p, the program will run each algorithm 3 times because it is quite long to run them. But on a small graph like simple, it will run each algorithm as many times as possible for 60 seconds. After that, it will print different information about the performances of the algorithm. Here is an example on the roadNet-CA graph:

```
$ ./benchmark roadNet-CA.clean.txt
--------------------------------------------------
FUNCTION: normal_double_sweep

Read file ... start try number 0 ... diameter: 856
Read file ... start try number 1 ... diameter: 856
Read file ... start try number 2 ... diameter: 856

- Tries: 15
- Total elapsed: (cpu: 62.504656s | sys: 0.147254s | real: 62.651910s)
- Total elapsed (without file loading): (cpu: 39.876954s | sys: 0.709931
   s | real: 39.966518s)
- Loading time: 22.685392s
- Average time per try: (cpu: 2.658464s | sys: 0.047329s | real:
   2.664435s)
- Average try per second: 0.375314 try/s
- Average diameter: 856.000000


--------------------------------------------------
FUNCTION: quotient_starting_double_sweep_leiden

Read file ... start try number 0 ... (try 1: 865 | try 2: 865) diameter:
    865
Read file ... start try number 1 ... (try 1: 865 | try 2: 865) diameter:
    865
Read file ... start try number 2 ... (try 1: 865 | try 2: 865) diameter:
    865

- Tries: 3
- Total elapsed: (cpu: 63.829022s | sys: 0.157763s | real: 63.986785s)
- Total elapsed (without file loading): (cpu: 59.287689s | sys: 0.286678
   s | real: 59.434314s)
- Loading time: 4.552471s
- Average time per try: (cpu: 19.762563s | sys: 0.095559s | real:
   19.811438s)
```

```
- Average try per second: 0.050476 try/s
- Average diameter: 865.000000


--------------------------------------------------
FUNCTION: quotient_starting_double_sweep_louvain

Read file ... start try number 0 ... (try 1: 865 | try 2: 865) diameter:
    865
Read file ... start try number 1 ... (try 1: 865 | try 2: 865) diameter:
    865
Read file ... start try number 2 ... (try 1: 865 | try 2: 865) diameter:
    865

- Tries: 3
- Total elapsed: (cpu: 105.400347s | sys: 0.402835s | real: 105.803182s)
- Total elapsed (without file loading): (cpu: 100.912969s | sys:
   0.954204s | real: 101.305715s)
- Loading time: 4.497467s
- Average time per try: (cpu: 33.637656s | sys: 0.318068s | real:
   33.768572s)
- Average try per second: 0.029613 try/s
- Average diameter: 865.000000
```

6 Analysis

6.1 Working Environment

The tests were carried out in the following working environment:

- OS: macOS Catalina 10.15.4
- CPU: 2,7GHz Quad-Core Intel Core i7
- Language: C
- Compilator: Apple clang version 11.0.0
- Compilations flags: -Ofast -march=native

6.2 First implementation

After multiple tests, our results were wrong. We never had the right diameter result even for small graphs where we calculated a ground truth beforehand. We concluded that our approach was not the right one and we decided to follow the algorithm proposed by Robert Erra in his instruction more closely.

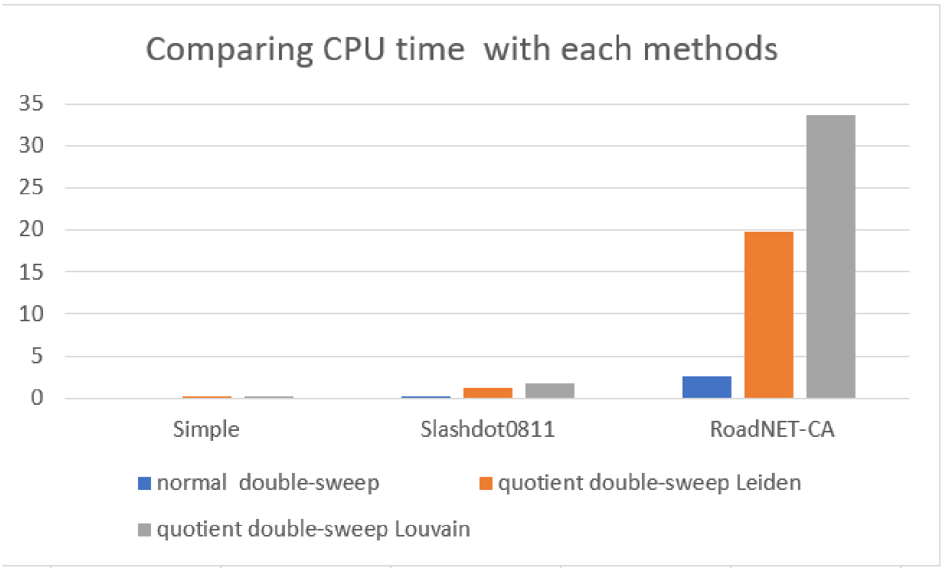
6.3 Second implementation

As mentioned in the Algorithms Implementations part before running the following final tests we used the naive implementation which was going through every vertex of the first community. This first implementation was taking way too much time even on relatively small graphs like inet. We thus tried to find which part was taking the most time and we figured that running the double sweep on every vertex of the first community not only was taking a lot of time but was also quite useless. Indeed, we found that running it on the first 3 vertices was giving the exact same results as if we run it on every vertex. That is why we decided to change the algorithm.

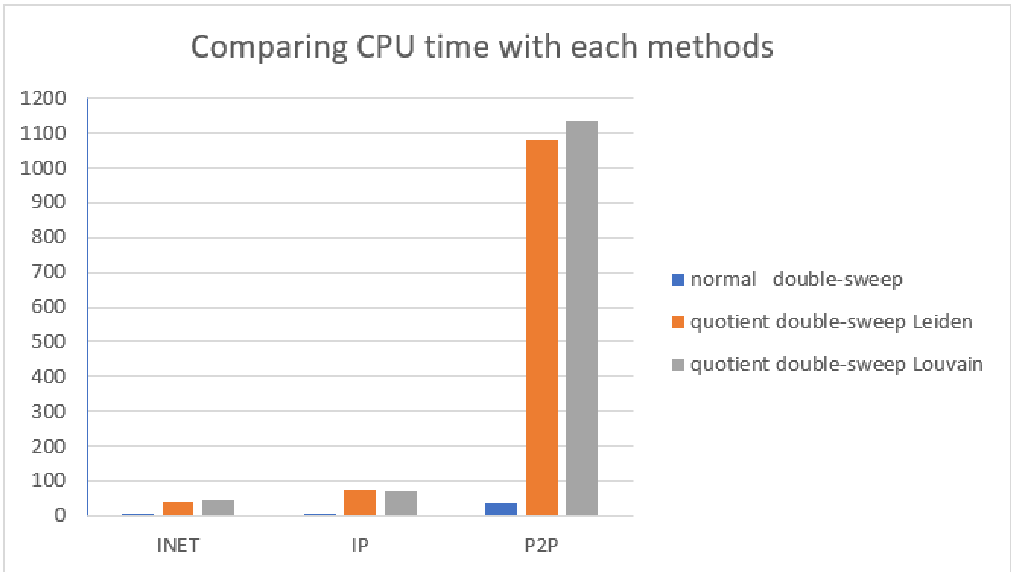
For the following results, we ran the tests in three different ways:

- **normal double-sweep**: running the double sweep algorithm on the original graph.
- **quotient double-sweep Louvain**: running the algorithm outlined in the previous section using Louvain algorithm to get the communities
- **quotient double-sweep Leiden**: running the algorithm outlined in the previous section using the Leiden algorithm to get the communities

Before showing you the results of our tests, we define "CPU time" as the only actual CPU time[1] used in executing the result. Below, you can find histograms representing a comparison between each method for each graph in term of CPU time (in seconds):



[1]**CPU time** is the amount of CPU time spent in user-mode code (outside the kernel) within the process. This is the only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.



The table below describes the diameter found using each method:

Graph	Normal double-sweep	Quotient double-sweep Leiden	Quotient double-sweep Louvain	Real Diameter
simple	4	4	4	4
Slashdot0811	12	12	12	12
roadNet-CA	856	865	865	865
inet	30	31	31	
ip	8	9	9	
p2p	9	9	9	
web	32			

When running the benchmark on the web graph the program was killed when trying to execute the Leiden algorithm on it. Giving the statistics of the normal double-sweep it clearly appears that the graph was not able to fit into the RAM of the computer and thus was loaded directly into the SWAP memory, which makes everything slower.

7 Discussion

7.1 Normal double-sweep vs quotient double-sweep

The results we found seem to be in adequation with our assumptions. Indeed, for the simple graph and Slashdot0811, all the methods found the real diameter. It starts to be interesting with the roadNet-CA where the double-sweep found a diameter of 856, and the quotient method 865, which is the ground truth. Regarding inet and ip, it can be seen that the diameter computed with the quotient is higher than the diameter computed with a normal double-sweep. As both methods find the diameter by computing one shortest path between two nodes, it is safe to assume that all the diameters computed are lower than the real diameter, and thus the quotient method is closer.

This means that during our tries, the method which used the quotient graph was three times out of six closer to the real diameter than the simple double-sweep and always as good.

7.2 Louvain vs Leiden

As enlightened in Traag, V.A., Waltman, L. van Eck, N.J. *From Louvain to Leiden: guaranteeing well-connected communities.* Sci Rep 9, 5233 (2019), Louvain is one of the most famous algorithms for uncovering community structure. This algorithm has a major defect that largely went unnoticed until now: the Louvain algorithm may yield arbitrarily badly connected communities[2].To address this problem, the Leiden algorithm was introduced. It has been proved that the Leiden algorithm yields communities that are guaranteed to be connected.

Accordingly, we decided that it would be interesting to also use the Leiden algorithm for our tests. As expected, in most cases Leiden was quicker than Louvain but we had exceptions where running Louvain gave a CPU time smaller than Leiden (e.g table below).

Graph	**Louvain**	**Leiden**
roadNet-CA	33,63s	19,76s
ip	70,43s	73,93s

[2]In the worst case, communities may even be disconnected, especially when running the algorithm iteratively. In our experimental analysis, we observe that up to 25% of the communities are badly connected and up to 16% are disconnected

8 Conclusion

This report presents the research work done with the goal of finding an efficient way to estimate a Very Large Graph's diameters. We saw that different algorithms were tried and that we can start to draw a conclusion. As it has previously been shown, the simple double-sweep algorithm gives good results and is very fast to compute. However, selecting the starting point for the double-sweep is crucial to have optimal results. We have shown that computing the quotient graph, its real diameter, and starting from a vertex in one of the communities from its extremities is a good way to do so. On the six graphs which were tried during this project, three of them had the same results, and three had better results. For the graphs where the ground truth was known, it was achieved in all the cases. However, it is important to be careful with this last piece of information as due to the complexity of finding the real diameter, this means that the graphs where the ground truth is available are smaller. In the end, this means that the quotient double-sweep algorithm seems more robust. The downside is that its computational time is higher, but still significantly lower than the exact computation. In a case where the communities of a graph are needed, or known, the cost of computing the communities can be removed from the algorithm and it becomes a lot quicker.

Even after finding some promising results, it feels that we only scratched the surface of what is possible with the quotient double-sweep algorithm. To have more complete results, it is important to try to run the algorithm using the Leiden communities computation with different parameters, which was not done here. It seems also interesting to try to optimize the algorithm by adding some boundaries as explained in the JMM AMS MMA DENVER presentation.

9 References

- Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. (2008) 'Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters', [online]. Available at: `https://arxiv.org/abs/0810.1355v1` (Accessed: May 2020)
- Matthieu Latapy and Clemence Magnien. (2007) 'Measuring Fundamental Properties of Real-World Complex Networks', [online]. Available at: `https://arxiv.org/abs/cs/0609115v2` (Accessed: May 2020)
- Traag, V.A., Waltman, L. van Eck, N.J. (2019) 'From Louvain to Leiden: guaranteeing well-connected communities'. [online]. Available at: `https://www.nature.com/articles/s41598-019-41695-z` (Accessed: May 2020)
- Stanford Network Analysis Platform (SNAP), `https://snap.stanford.edu/snap/` (Accessed: May 2020)
- Slashot0811 Graph, `https://snap.stanford.edu/data/soc-Slashdot0811.html` (Accessed: May 2020)
- roadNet-CA Graph: `https://snap.stanford.edu/data/roadNet-CA.html` (Accessed: May 2020)
- Clemence Magnien and Matthieu Latapy massive graphs directory: `http://data.complexnetworks.fr/Diameter/` (Accessed: May 2020)
- igraph C library: `https://igraph.org/c/` (Accessed: May 2020)
- time (Unix) Wikipedia page: `https://en.wikipedia.org/wiki/Time_%28Unix%29` (Accessed: May 2020)