Clustering-based Graph Numbering using Execution Traces for Cache Misses Reduction in Graph Analysis Applications

Regis Audran MOGO WAFO[1*], **Thomas MESSI NGUELE**[1,2,3*], **Armel Jacques NZEKON NZEKO'O**[1,3*], **Xaviera Youh DJAM**[1*]

[1]University of Yaounde I, FS, Computer Science Department, Cameroon
[2]University of Ebolowa, HITLC, Computer Engineering Department, Cameroon
[3]Sorbonne Université, IRD, UMI 209 UMMISCO, F-93143, Bondy, France

*E-mail : {audran.mogo, thomas.messi, armel.nzekon, xaviera.kimbi}@facsciences-uy1.cm

DOI : 10.46298/arima.13538
Submitted on May 3, 2024 - Published on February 24, 2025
Volume : **43** - Year : **2025**

Editors : *Mathieu Roche, Nabil Gmati, Clémentin Tayou Djamegni*

Abstract

Social graph analysis is generally based on a local exploration of the underlying graph. That is, the analysis of a node of the graph is often done after having analyzed nodes located in its vicinity. However, over the time, networks are bound to grow with the addition of new members, which inevitably leads to the enlargement of the corresponding graphs. At this level, we therefore have a problem because more the size of the graph increases, more the execution time of graph analysis applications also increases. This is due to the very large number of nodes that will need to be treated. Some recent work in-faces this problem by exploiting the properties of social networks. One of this properties is the community structure that is used to renumber the nodes of the graph, in order to reduce cache misses. Reducing cache misses in an application usually allows to reduce the execution time of this application. In this paper, we argue that combining existing graph ordering with a new numbering that exploit execution traces analysis can allow to improve cache misses reduction and hence execution time reduction. The idea is to build graph numbering using execution traces of graph analysis applications and then combine it with an existing graph numbering (such as Cn-order, NumBaCo, Rabbit and G-order). To build this new ordering, we define a new distance and then used it to analyse execution traces with well known clustering algorithms K-means (for Kmeans-order) and hierarchical clustering (for cl-hier-order). We then combined these two numbering with the existing ones (previously mentioned). Experimental results did on three user machines show that: clustering based numbering allow to improve existing numbering. In fact, the combination between clustering based numbering and existing numbering allows to get better results. The gap (in term of cache misses reduction) between our proposal and the existing numbering can reach 12.89%. This is for example the case with the 4-cores user machine: with one thread, our proposal got the best cache misses reduction through the combination NumBaCo_K-means with 65.51% compared to 52.62% of cache misses reduction gotten with Cn-order (an existing numbering).

Keywords: Cache misses reduction, Execution trace, Clustering, Multi-cores architecture

I INTRODUCTION

Actors and their interconnections in social networks are modeled with graphs where nodes are actors and links are their interconnections. The advent of computers and communication networks allows to analyze data (see [2]) on a large scale and has lead to a shift from the study of individual properties of nodes in small specific graphs with tens or hundreds of nodes, to the analysis of macroscopic and statistical properties of large graphs also called complex networks, consisting of millions and even billions of nodes [2]. Social networks welcome new members every day. This contributes to enlarging the size of the corresponding graphs. It should be noted that the larger the graph, the longer the execution of the analysis applications takes time [5]. This usually results from a large number of [7] cache misses, themselves caused by many memory accesses in search of certain nodes during the execution of the graph analysis application.

Graph application consists in analyzing various nodes; process a node usually involves the analysis of the other ones located in its vicinity. Reducing cache misses in an application allows to reduce the execution time of this application. In order to reduce cache misses, the main idea is therefore to renumber the graph such away that the nodes likely to be processed together become close in the memory. Recent numbering algorithms guided by this idea have been proposed, such as Cn-order[8–10], Gorder [6], Rabbit order [5] and NumBaCo [7]. They stood out for their efficiency compared to other existing algorithms (see [8–10]).

In this paper, the execution traces analysis are used to build these new numberings, and also to combine them with one of the recent numbering algorithms in order to have better results. Machine learning techniques like k-means and hierarchical clustering are also used to analyze the execution traces and to construct these numbering. K-means consists of grouping nodes into k groups and hierarchical clustering consists of making a series of groupings, by aggregating the closest nodes at each step. The gotten numbering (based on these clustering) was then combined to the existing numbering (based on graph structure).

Contribution. We show in this paper that one can take into account graph structure (community) and information during application execution (execution traces) in order to reduce cache misses and hence to reduce execution time. In detail, our contribution is structured as follows:

- Firstly, one distance is defined between two nodes in an execution traces file.
- Two numbering algorithms based on clustering are then built: one based on the K-means algorithm (1) and another one based on Hierarchical clustering(2).
- These new numbering are then combined with existing ones Cn-order[8], Gorder[6], Rabbit-order[5] and NumBaCo[7].
- Experiments are conducted on three multi-core machines with PageRank and Astro-ph[4].

Paper organization. The remainder of this paper is structured as follows. Section II presents recent works on graph ordering. Section III recalls the cache management problem. Section IV introduces our main contribution. Experimental results are shown in section V. Section VI is devoted to the synthesis of our contribution together with future work.

II RELATED WORK

For several years, graph reordering algorithms have attracted the attention of researchers and this attention is constantly increasing. The most recent graph reordering include NumBaCo Messi Nguélé, Tchuente, and Méhaut [7], Gorder Wei, Yu, Lu, and Lin [6], Rabbit Order Arai, Shiokawa, Yamamuro, Onizuka, and Iwamura [5] and Cn-order Messi Nguélé, Tchuente, and

Méhaut [8]. Table1, gives a summary of the existing numbering methods with their advantages and limits.

The main idea of **NumBaCo** is to exploit the community structure of the graph nodes to improve graph analysis applications performances through cache misses reduction (see [7]).

In the case of **G-order**, Wei, Yu, Lu, and Lin [6]) offered an order that allows nodes in the direct neighborhood to be close in memory.

Rabbit-order proposed by Arai, Shiokawa, Yamamuro, Onizuka, and Iwamura [5]) proposes a numbering method based on two approaches, namely:

- Scheduling based on the community structure of the graphs in the real world: they try to match the hierarchical communities of the graphs in the real world with the hierarchies located at the level of the caches of the central processor unit;
- An incremental aggregation in parallel: this one aggregates in an incremental way the vertices of the same community in parallel, which has the consequence of reducing the number of vertices to be processed.

Cn-order proposed by Messi Nguélé, Tchuente, and Méhaut [8] is based on a fusion of the advantages of previous models such as:

- grouping in memory nodes appearing frequently in direct neighborhood (based on G-order)
- grouping in memory nodes belonging to the same community or sub-community (based on Numbaco and Rabbit-order).

None of these methods does exploit former executions of the target application. In this paper, we argue that exploiting former execution of the target applications (for example, exploiting execution traces analysis) can allow to improve existing performances.

Table 1: Comparison of existing numbering methods

Reference	**Year**	**Techniques**	**Specifications**	**Limits**
[7]	2015	NumBaCo	Exploits community structure of the graph in order to renumber it	Doesn't take into account nodes degrees heterogeneity Doesn't exploit former executions of target applications
[6]	2016	Gorder	Consider nodes sibling nodes	Does not take into account communities Doesn't exploit former executions of target applications
[5]	2016	Rabbit-order	Based on community structure + is parallel	Doesn't take into account nodes degrees heterogeneity Doesn't exploit former executions of target applications
[8]	2017	Cn-order	Combines advantages of the others	Doesn't exploit former executions of target applications

III CACHE MANAGEMENT PROBLEM

When a processor needs to access data during the execution of a program, it first checks the corresponding entry in the cache. If the entry is found in the cache, there is cache hit and the data is read or written. If the entry is not found, there is a cache miss. There are three main categories of cache misses which include - compulsory misses caused by the first reference to data, - conflict misses, caused by data that have the same address in the cache (due to the mapping), - capacity misses, caused by the fact that all the data used by the program cannot fit in the cache. Hereafter, we are interested in the last category. In common processors, cache memory is managed automatically by the hardware (pre-fetching, data replacement). The only way for the user to improve memory locality or to control and limit cache misses is the way he organizes the data structure used by its programs.

In previous work ([7], [8], [10]), it was shown that the problem of reorganizing a graph through renumbering for cache misses is NP-Complete. Then some heuristics based on the graph structure are provided to solve this problem. In this paper we will show that combining solution based graphs structure (community) with the information about the execution of an application (execution traces) in order to reduce cache misses.

Modeling Cache misses in Graph Application. In this paragraph, we recall cache misses modeling already presented at [7–10]. We only consider capacity cache misses caused by the fact that the data used during the execution of a program cannot fit entirely in the memory cache. Let:

- D_c: The size of the cache memory,
- N : The Set of nodes
- b : The function which gives the number of the block to which a node x belongs.

$b(x) = x$ **Div** D_c (Div being the integer division).

When a node "x"(which is therefore in the cache) is manipulated by the processor and the processor tries to access another node "y", two situations are possible:

- if the two nodes are in the same memory block b(x) = b(y), then y is found also in cache
- if x and y are not in the same memory block, we say that there is a cache miss.

The cache miss can therefore be modeled with the function σ defined by:

$\sigma : N \text{ x } N \rightarrow 0, 1$

$$(x, y) \longmapsto \sigma(x, y) = \begin{cases} 0, \ if \ \ b(x) - b(y) = 0 \\ 1, else \end{cases}$$

This cache miss model can be used to count cache misses in a graph analysis application to give a formal definition of the problem of reducing cache misses.

Illustration example of Cache misses counting in Graph Application. Consider figure 1 with the graph at the left and the memory representation at the right. The graph has 14 nodes, the main memory has a size 4*4 and the cache memory has a size 4. Let 0 and 1 be the accessing nodes in a program. 0 has the neighbors {4, 11} while 1 has {5,10,13} as neighbors. Accessing to a node usually implies to access to its neighbors (in a graph application). So the access sequence is S = {**0**,4,11,**1**,5,10,13}. Graph nodes are stored consecutively (from 0 to 13) in the main memory. With this simple memory representation, we will get 7 cache misses when trying to access to nodes 0 and 1 (that is sequence S). In fact, trying to access to node 0 will cause the first cache miss, and the block memory containing nodes 0,1, 2 and 3 will be stored in the cache memory. But the next node to be accessed is node 4, witch is not yet in cache memory. So a second cache miss will happen and the nodes 4, 5, 6 and 7 will be stored in the cache memory. The same phenomenon will happen with nodes 11, 1, 5, 10 and 13 with a cache miss every time.

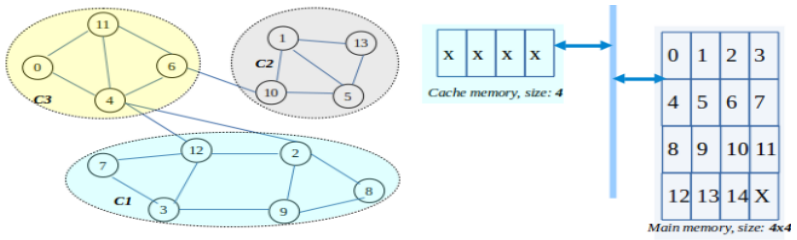
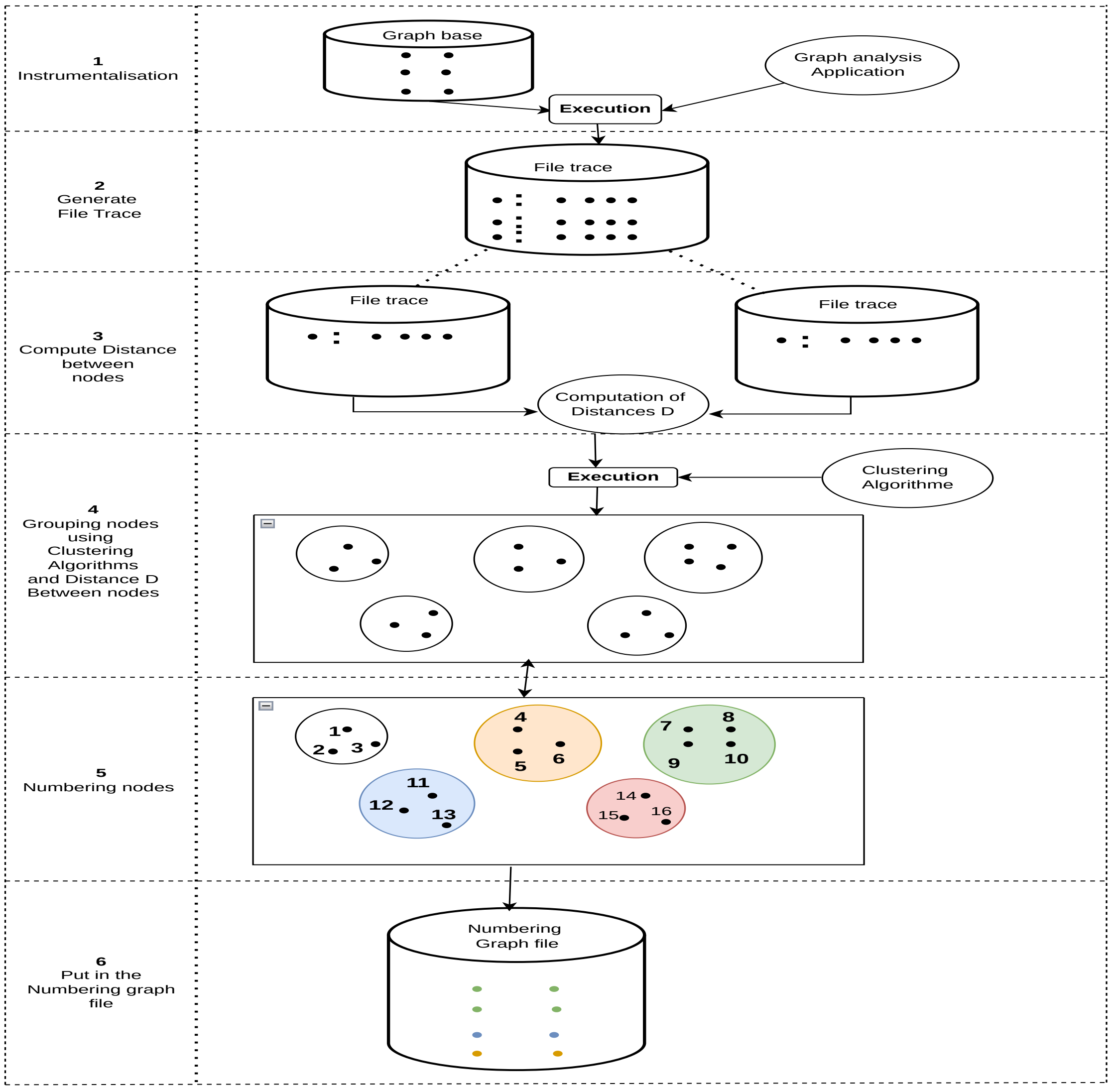


Figure 1: Cache misses counting illustration (see [7])

IV NUMBERING BASED ON CLUSTERING

The idea here is to build a graph numbering by analyzing execution traces of graph applications with clustering algorithms. Figure 2 presents the complete principle of numbering using execution traces analyses with six steps: - Instrumentalization, - Generate file trace, - Compute Distance between nodes, - Grouping nodes using clustering Algorithms and distance D, - Numbering nodes, - Put nodes in the new Graph.



1 Instrumentalisation
Graph base
Graph analysis Application
Execution
2 Generate File Trace
File trace
3 Compute Distance between nodes
File trace
File trace
Computation of Distances D
4 Grouping nodes using Clustering Algorithms and Distance D Between nodes
Execution
Clustering Algorithme
5 Numbering nodes
6 Put in the Numbering graph file
Numbering Graph file

Figure 2: Principle of numbering by execution traces analyses

In this section, execution traces are presented first in section 4.1, and then our proposed distance is presented in section 4.2. This distance allows us to build our new numbering algorithm presented in section 4.3.

4.1 Execution traces

The execution traces allow to capture the behavior of the application during its execution. Since it is a graph analysis application, we are interested to keep the way nodes are accessed during execution. To obtain these traces, we add in the graph analysis program some code that allow to print at each line of a file, a node and the list of other nodes directly encountered during the execution program.

The goal of our work is to analyze this execution traces file in other to build a numbering that will allow "closer nodes" to have "closer numbering". To carry out this goal, we choose to analyze these execution traces with clustering algorithms such as K-means and Hierarchical clustering. Those algorithms use with a distance that measures the closeness between nodes of the execution traces file.

The next section presents the approach used to compute the distance between nodes in execution traces file.

4.2 Distance between two nodes

Alex Groce (see [3]) defines the distance between two execution traces x and y of the same program as follows: $\bar{D}(x,y) = \sum_{i=0}^{n} \Delta(i)$, $where\ \Delta(i) = \begin{cases} 0,\ if\ \ val_i^x = val_i^y \\ 1,\ if\ val_i^x \neq val_i^y \end{cases}$

val_i^x and val_i^y are the value of x and y at the time i.

Inspired by this, we define a metric between two nodes a and b encountered in the execution traces file.

Proximity D between two nodes. Let $\Gamma(a)$ and $\Gamma(b)$ the set of nodes encountered directly after a and b respectively. The proximity D between a and b is defined as follows:

$$D(a,b) = |\Gamma(a) \cup \Gamma(b)| - |\Gamma(a) \cap \Gamma(b)|$$
$$= (|\Gamma(a)| + |\Gamma(b)| - |\Gamma(a) \cap \Gamma(b)|) - |\Gamma(a) \cap \Gamma(b)|$$
$$= |\Gamma(a)| + |\Gamma(b)| - 2|\Gamma(a) \cap \Gamma(b)|$$

The proximity D between two nodes defined before is the well known symmetric differences that has the following properties. Let N be the set of nodes:

1. $\forall a,b \in N, D(a,b) \geq 0.$
   This is because $|\Gamma(a)| + |\Gamma(b)| \geq 2|\Gamma(a) \cap \Gamma(b)|$
2. $\forall a,b \in N, D(a,b) = 0 \iff a = b.$
   In fact, if $a = b$, we have $|\Gamma(a)| + |\Gamma(b)| = 2|\Gamma(a) \cap \Gamma(b)|$ then $D(a,b) = 0$
   In the other side, $\begin{cases} D(a,b) = 0 & \Longrightarrow |\Gamma(a)| + |\Gamma(b)| - 2|\Gamma(a) \cap \Gamma(b)| = 0 \\ & \Longrightarrow |\Gamma(a)| + |\Gamma(b)| = 2|\Gamma(a) \cap \Gamma(b)| \\ & \Longrightarrow a = b \end{cases}$
3. $\forall a,b \in N, D(a,b) = D(b,a)$
   This is because, $|\Gamma(a)| + |\Gamma(b)| = |\Gamma(b)| + |\Gamma(a)|$ and $|\Gamma(a) \cap \Gamma(b)| = |\Gamma(b) \cap \Gamma(a)|$
4. $\forall a,b,c \in N, D(a,c) \leq D(a,b) + D(b,c)$

For this last case, we should show that:

$|\Gamma(a)|+|\Gamma(c)|-2|\Gamma(a)\cap\Gamma(c)| \leq |\Gamma(a)|+|\Gamma(b)|-2|\Gamma(a)\cap\Gamma(b)|+|\Gamma(b)|+|\Gamma(c)|-2|\Gamma(b)\cap\Gamma(c)|$

This is shown on the Annex (see A).

4.3 Proposed Algorithms

In this section, we present numbering based on execution trace analysis with clustering and the combinations between these algorithms and the previous existing numbering.

4.3.1 Kmeans-order and Cl-hier-order

Algorithm 1 presents clustering based on K-means. At line 1, the structure for the representation of our execution traces on the file is built. At line 2, a clustering with K-means is performed, with the parameters "K" for the number of clusters, "Ite" for the number of iterations and calculating the proximity of the nodes with our distance "D". At line 3, the nodes are renumbered in such away that the ones which are close (with small distance) and are in the same cluster have consecutive numbers.

Algorithm 1 : **Kmeans-order**

```
Input: $G = (V, E), K, Ite, File\_trace, D$
Output: $G'$

1: $Trace\_kmeans \leftarrow Build\_Trace\_Kmeans(File\_trace)$
2: $Clusters\_kmeans \leftarrow Build\_Clusters\_Kmeans(Trace\_kmeans, K, Ite, D)$
3: $G' \leftarrow Renumbering(G, Clusters\_kmeans)$
4: Return $G'$
```

Algorithm 2 presents clustering based on Hierarchical Clustering. It has the same structure as the Algorithm 1. But at line 2, the cluster computation is performed through the hierarchical clustering algorithm. At line 3, it is the renumbering based on these clusters.

Algorithm 2 : **Cl-hier-order**

```
Input: $G = (V, E), File, D$
Output: $G'$

1: $Trace\_Cl - hier \leftarrow Build\_Trace\_Cl\_hier(File\_trace)$
2: $Hiearachies \leftarrow Buil\_hierar(G, Trace\_Cl\_hier, D)$
3: $G' \leftarrow Renumbering(G, Hiearachies)$
4: Return $G'$
```

4.3.2 Combination between Kmeans-order and Cl-hier-order

Algorithm 3 presents the combination kmeans-order with Hierarchical Clustering. At line 1, the K-means algorithm gives the first clusters, in order to reduce the size of the data set to be processed. At line 2 now, the hierarchical clustering algorithm computes the cluster, using the clusters provided by K-means. In line 3, the renumbering is done based on clusters in the hierarchy where the first level of hierarchy is composed of clusters produced by K-means .

Algorithm 3 : **kmeans-order_Cl-hier-order**

Input: $G = (V, E), K, File\_trace, D$

Output: G'

```
1: $Clusters\_kmeans \leftarrow Build\_clusters\_Kmeans(K, G, File\_trace, D)$
2: $Hiearachies \leftarrow Buil\_hier(G, Clusters\_kmeans, D)$
3: $G' \leftarrow Renumbering(G, Hiearachies, Clusters\_kmeans)$
4: Return $G'$
```

4.3.3 *Combination between existing numbering with Kmeans-order and Cl-hier-order*

In this part, We consider existing numbering as: Cn-order, NumBaCo, Rabbit-order and Gorder.

Algorithm 4 present the combinations Exist-Numbering_Kmeans-order. At line 1 we first produce a new graph renumbered by one of the existing numbering. Then on line 2 we extract the Traces, with the new graph previously obtained. In line 3 we build our structure for the representation of the traces which will be used by K-means. At line 4 we proceed to the construction of the clusters using K-means. At line 5, we renumber the nodes in such away that the ones which are close (with small distance) and are in the same cluster have consecutive numbers.

Algorithm 4 : **Exist-Numbering_Kmeans-order**

Input: $G = (V, E), K, D, Ite$

Output: G''

```
1: $G' \leftarrow Buil\_New\_Graph(Existing - numbering, G)$
2: $File\_Trace' \leftarrow Build\_File\_Trace'\_Kmeans(G')$
3: $Trace'\_kmeans \leftarrow Build\_Trace'\_Kmeans(File\_Trace')$
4: $Clusters'\_kmeans \leftarrow Build\_Clusters'\_Kmeans(G', Trace'\_kmeans, K, Ite, D)$
5: $G'' \leftarrow Renumbering(G', Clusters'\_kmeans)$
6: Return $G''$
```

Algorithm 5 present the combination exist-numbering_Cl-hier-order. It is the same process with algorithm 4, but here we permute K-means-order by Cl_hier-order.

Algorithm 5 : **Exist-Numbering_Cl-hier-order**

Input: $G = (V, E), D$

Output: G''

```
1: $G' \leftarrow Buil\_New\_Graph(Existing\_Numbering, G)$
2: $File\_Trace' \leftarrow Build\_File\_Trace'(G')$
3: $Trace'\_Cl\_hier \leftarrow Build\_Trace'\_Cl\_hier(File\_Trace')$
4: $Hiearachies' \leftarrow Build\_hierar'(G', Trace'\_Cl\_hier, D)$
5: $G'' \leftarrow Renumbering(G', Hiearachies')$
6: Return $G''$
```

4.4 Complexity

The complexity in the presented algorithms depends on the initial clustering algorithms used. For k-means, the complexity depends on the number of iterations and the number of clusters to build. More the the number of iterations and clusters is high, more higher complexity. For Hierarchical Clustering, the complexity depends on the number of the graph nodes and the number of groupings carried out at each hierarchy. If we have many grouping in the first hierarchy, we can finish quickly the program, but if we have one grouping in each hierarchy, the complexity becomes very high.

For every case, the time for re-numbering the graph is $O(N)$, where N is the number of graph nodes.

V EXPERIMENTAL EVALUATION

The experiments were done on three user machines:

- A two-cores machine with 1.10GHz, 6GB of Ram, L2 of 4MB KB and L1 of 24KB;
- A 4-cores with 1.70GHz, 4GB of Ram ,L3 of 3MB KB, L2 of 256 KB and L1 of 32KB.
- A 12-cores machine with 4.70GHz, 16GB of Ram ,L3 of 12 MB, L2 of 2048 KB and L1 of 48KB.

For this evaluation, we present results got with the well known graph analysis application, Pagerank [1]. We used a Posix thread implementation proposed by *Nikos Katirtzis*[1]. This implementation uses adjacency list representation. We use Astro-ph dataset [4] that has n=16,046 nodes and m=242,502 edges (that is almost **2.46 MB** with the formula presented at Messi Nguélé and Méhaut [10], the graph size is $G\_space = 8m + 40N\ bytes$).

In this section, we compare previously proposed graph numbering (Cn-order, Rabbit, Gorder, NumBaCo) with numbering based clustering (through execution traces analysis). We also analyse the case where we fuse existing numbering with clustering based ones. We do this in three ways: cache reference reduction (section 5.1), cache-misses reduction (section 5.2), execution time reduction (section 5.3). This task was done with three tables:

- Table 2 that compares cache references, cache misses and execution time on the two-cores machine described above;
- Table 3 that does the same but on the 4-cores machine described above;
- Table 4 that does the same again but on the 12-cores machine described above.

In each table, the color signification is as follows:

- The four best performances are colored. The two best performance are in bold.
- The color **blue** is for the time, the color **red** for the cache misses and the color **orange** for the cache references.
- In the heuristic column, new proposed ones are **bold**. We put the k-means and cl-hier (which are the base of our proposed orders) in **green**.

[1] `https://github.com/nikos912000/parallel-pagerank`

.

Heuristic	**Time** (ms)	**Cache misses**(th)	**Cache references**(th)
Astro-ph 1th	976.84	1893.65	69646.05
Gorder 1th	**894.19 (8.46%)**	1581.28 (16.50%)	37224.29 (46.55%)
NumBaCo 1th	896.67 (8.21%)	**1532.83 (19.05%)**	40092.06 (42.43%)
Rabbit 1th	907.24 (7.13%)	1719.17 (9.21%)	41608.28 (40.26%)
Cn-o 1th	896.52 (8.22%)	1706.46 (9.89%)	37070.80 (46.77%)
k-means-o 1th	941.79 (3.59%)	1835.72 (3.06%)	48939.62 (29.73%)
cl-h-o 1th	932.98 (4.49%)	1841.19 (2.77%)	48887.31 (29.81%)
k-means-o_cl-h-o 1th	904.55 (7.40%)	1993.18 (-5.26%)	38025.62 (45.40%)
Cn-o_k-means-o 1th	924.48 (5.36%)	1555.45 (17.86%)	45363.86 (34.87%)
Cn-o_cl-h-o 1th	899.82 (7.89%)	1636.62 (13.57%)	35628.96 (48.84%)
Gorder_k-means-o 1th	909.10 (6.93%)	**1479.79 (21.86%)**	45959.29 (34.01%)
Gorder_cl-h-o 1th	**887.13 (9.18%)**	1564.93 (17.36%)	**35345.70 (49.25%)**
NumBaCo_k-means-o 1th	925.58 (5.25%)	1742.74 (7.97%)	45953.94 (34.02%)
NumBaCo_cl-h-o 1th	907.86 (7.06%)	1715.44 (9.41%)	35864.36 (48.50%)
Rabbit_k-means-o 1th	919.32 (5.89%)	1728.48 (8.72%)	46236.76 (33.61%)
Rabbit_cl-h-o 1th	896.14 (8.26%)	1533.67 (19.01%)	**35251.46 (49.38%)**
Astro-ph 2th	764.15	2470.65	76210.12
Gorder 2th	791.41 (-2.79%)	2310.96 (6.46%)	43455.21 (37.61%)
NumBaCo 2th	690.50 (7.54%)	2532.54 (-2.51%)	47035.25 (32.47%)
Rabbit 2th	658.66 (10.80%)	2689.75 (-8.87%)	49065.78 (29.55%)
Cn-o 2th	688.03 (7.79%)	2427.47 (1.75%)	43511.17 (37.53%))
k-means-o 2th	649.27 (11.76%)	2605.58 (-5.46%)	56326.53 (19.12%)
cl-h-o 2th	644.14 (12.29%)	2387.23 (3.38%)	55760.22 (19.94%)
k-means-o_cl-h-o 2th	646.42 (12.05%)	2600.44 (-5.25%)	44760.02 (35.73%)
Cn-o_k-means-o 2th	656.77 (10.99%)	2349.04 (4.92%)	52293.08 (24.92%)
Cn-o_cl-h-o 2th	**637.06 (13.01%)**	2331.34 (5.64%)	**42258.31 (39.32%)**
Gorder_k-means-o 2th	663.23 (10.33%)	**2293.60 (7.17%)**	53396.72 (23.33%)
Gorder_cl-h-o 2th	**635.1 (13.21%)**	**2195.94 (11.12%)**	**41894.20 (39.85%)**
NumBaCo_k-means-o 2th	644.51 (12.25%)	2587.76 (-4.74%)	53264.53 (23.52%)
NumBaCo_cl-h-o 2th	702.30 (6.33%)	2620.71 (-6.07%)	43166.25 (38.02%)
Rabbit_k-means-o 2th	661.03 (10.56%)	2444.63 (1.05%)	53051.03 (23.83%)
Rabbit_cl-h-o 2th	684.06 (8.20%)	2538.87 (-2.76%)	42306.57 (39.25%)

Table 2: Graph Ordering Comparison with Pagerank, 2 threads, Astro-ph with a 2-cores in term of time (in milliseconds, ms), cache misses (in thousands, th) and cache references (in thousands, th)

5.1 Cache-References Reduction

Cache references correspond to the number of time the program (Pagerank in this case) looks for data at last level cache memory (L2 for two-cores and L3 for 4-core and 12 cores). So when the data required by the processor is not present at the first level cache memory (L1 for two-cores, L1 and L2 for 4-cores and 12 cores), it is looked at the last level cache memory and this causes a cache reference. This means that, the number of cache references increases with the increasing of first level cache memory misses. In other words, the reduction of cache misses at the first level cache memory will allow the reduction of cache references.

In Table 2, with the user machine two-cores, we can see that:

- With one thread, k-means and cl-hier orders (the base of our new proposed orders) reduce the cache references more than base line (without numbering). But they don't do more than existing ordering (Gorder, NumBaCo, Rabbit and Cn-order).
  We can see an improvement when combining k-means and cl-hier with existing ordering. And some of these combination give very good results. For example, the two combinations (Rabbit_cl-hier-order) and (Gorder_cl-hier-order) produce the best cache-references reduction 49.38% and 49.25% respectively while the existing ordering Cn-order produces the third performance with 46.77%.

- Like the previous observation, With two threads, k-means and cl-hier orders reduce the cache references more than base line. And also, they don't do more than existing ordering. It is with the combination that we have an improvement. In fact, the combinations Gorder_cl-hier-order and Cn-order_cl-hier-order give the best results with 39.85% and 39.32% respectively. The best existing order comes at the fifth position behind other combination with 37.61%.

In Table 3 with the user machine 4-cores and Table 4 with the user machine 12-cores we have quite the same observations, that is: k-means and cl-hier orders improve the base line (without any numbering), do worst than the existing numbering, but their combination with these last one can allow some improvements and even give better results.

According to these observations, we can say our newly proposed heuristic is more efficient when it is combined with existing numbering. It allows to reduce cache references and is in some cases better than existing numbering. So, as expected, since it reduces cache references, it helps to reduce cache misses for lower levels of memory cache.

Heuristic	**Time** (ms)	**Cache misses** (th)	**Cache references** (th)
Astro-ph 1th	983.97	1404.34	28672.87
Gorder 1th	**900.62 (8.47%)**	751.31 (46.50%)	16978.87 (40.78%)
NumBaCo 1th	905.25 (8.00%)	896.31 (36.18%)	**15152.01 (47.16%)**
Rabbit 1th	914.73 (7.04%)	1086.81 (22.61%)	16533.93 (42.34%)
Cn-o 1th	**892.39 (9.31%)**	665.33 (52.62%)	**14698.59 (48.74%)**
k-means-o 1th	950.17 (3.44%)	1195.37 (14.88%)	22385.70 (21.93%)
cl-h-o 1th	959.13 (2.52%)	1658.89 (-18.13%)	22492.25 (21.56%)
k-means-o_cl-h-o 1th	917.94 (6.71%)	1028.04 (26.80%)	16742.54 (41.61%)
Cn-o_k-means-o 1th	929.78 (5.51%)	953.61 (32.10%)	20027.35 (30.15%)
Cn-o_cl-h-o 1th	903.06 (8.22%)	**525.40 (62.59%)**	16092.41 (43.88%)
Gorder_k-means-o 1th	934.33 (5.05%)	868.65 (38.15%)	21413.63 (25.32%)
Gorder_cl-h-o 1th	907.33 (7.79%)	808.25 (42.45%)	16195.74 (43.52%)
NumBaCo_k-means-o 1th	924.18 (6.08%)	**484.30 (65.51%)**	20254.35 (29.36%)
NumBaCo_cl-h-o 1th	911.60 (7.36%)	1079.61 (23.12%)	16107.63 (43.82%)
Rabbit_k-means-o 1th	933.37 (5.14%)	970.35 (30.90%)	20829.65 (27.35%)
Rabbit_cl-h-o 1th	916.99 (6.81%)	1142.33 (18.66%)	16220.99 (43.43%)
Astro-ph 2th	770.58	1784.53	30485.73
Gorder 2th	937.64 (-21.68%)	1493.62 (16.30%)	17951.56 (41.11%)
NumBaCo 2th	663.45 (13.90%)	1448.12 (18.85%)	**16865.39 (44.68%)**
Rabbit 2th	575.40 (25.33%)	1309.51 (26.62%)	18777.25 (38.41%)
Cn-o 2th	683.86 (11.25%)	1203.83 (32.54%)	**16333.35 (46.42%)**
k-means-o 2th	**570.09 (26.02%)**	1434.87 (19.59%)	24521.64 (19.56%)
cl-h-o 2th	573.90 (25.52%)	1591.81 (10.80%)	24592.08 (19.33%)
k-means-o_cl-h-o 2th	621.42 (19.36%)	3092.95 (-73.32%)	18886.29 (38.05%)
Cn-o_k-means-o 2th	609.62 (20.89%)	1334.22 (25.23%)	22032.80 (27.73%)
Cn-o_cl-h-o 2th	579.80 (24.76%)	**984.11 (44.85%)**	18069.54 (40.73%)
Gorder_k-means-o 2th	610.47 (20.78%)	1263.17 (29.22%)	23595.91 (22.60%)
Gorder_cl-h-o 2th	605.60 (21.41%)	1267.19 (28.99%)	18002.23 (40.95%)
NumBaCo_k-means-o 2th	**559.63 (27.38%)**	**1159.25 (35.04%)**	22427.91 (26.43%)
NumBaCo_cl-h-o 2th	681.36 (11.58%)	1351.63 (24.26%)	17676.53 (42.02%)
Rabbit_k-means-o 2th	601.76 (21.91%)	1306.93 (26.76%)	22889.38 (24.92%)
Rabbit_cl-h-o 2th	669.41 (13.13%)	1414.45 (20.74%)	17925.69 (41.20%)

Table 3: Graph Ordering Comparison with 2 Pagerank, 2 threads, Astro-ph with a 4-cores in term of time (in milliseconds, ms), cache misses (in thousands, th) and cache references (in thousands, th)

5.2 Cache-Misses Reduction

Cache misses correspond to the number of time the program looks for data at the last level cache memory (L2 for two-cores, L3 for 4-cores and 12-cores) and doesn't find it. When the data required by the processor is not present at last level cache memory, it is looked at central memory and this causes a cache misses.

In term of cache misses reduction, when looking at the three Tables 2, 3 and 4, results gotten with our heuristics compared to existing one have the same tendency to the one gotten in term of cache references and are even better.

In fact, with Table 2,

- With one thread, k-means and cl-hier orders improve only the base line (without numbering). But when combined with existing ordering, they improve them. For example, we can see that the combination Gorder_Kmean-order produces the best cache-misses reduction 21.86%, NumBaCo is the second with 19.05% and another combination (Rabbit_cl-hier-order) is the third with 19.01%.
- With two threads, the combinations (Gorder_cl-hier-order) and (Gorder_kmeans-order) are the first with 11.12% and 7.17% respectively. The existing numbering Gorder takes the third place with 6.46%. Many ordering do not do well than the base line (the percentage is negative).

In Table 3 we remark that:

- With one thread, we can see that only k-means-order offers an improvement of the base line, but without exceeding the performances given by existing numbering. Despite this first observation, we also note that the combinations (NumBaCo_kmeans-order and Cn-order_Cl-hier-order) produce the best cache-misses reduction with respectively 65.51% and 62.59%. Cn-order takes the third place with 52.62%.
- With two threads, our heuristic k-means-order success to improve line base(without numbering) and two existing methods (NumBaCo and Gorder). The combination still keep the best reduction (Cn-order_Cl-hier-order and NumBaCo_Kmean-order) with 44.85% and 35.04% respectively, and Cn-order is still the third with 32.54%.

In Table 4 we remark that:

- With one thread, we can see that k-means-order produces the best cache-misses reduction with 12.98%. Rabbit takes the second place with 11.99% and then the combinations (Rabbit_kmeans-order and Rabbit_Cl-hier-order) take the third and fourth place with 10.96% and 8.66% respectively.
- With two threads, as seen with one thread, k-means-order reduces the cache misses of base line (without numbering) with 18.31% and outperforms all the existing numbering. Cl-hier-order also improves the base line with 11.84% and improves all the existing numbering. the combinations (Gorder_Cl-hier-order, Rabbit_Kmeans-order and kmeans_Cl-hier-order) take the second place with 15.83%, third place with 15.58% and fourth place with 13.11% respectively.

These observations show that our main heuristic sometimes improve the existing numbering (k-means-order at table 4 takes the first place with one and two threads). Even the combinations proposed allow to improve the existing numbering, this is the case for exeample with NumBaCo_kmeans-order which gives highest improvement with 65.51%.

Heuristic	**Time** (ms)	**Cache misses** (th)	**Cache references** (th)
Astro-ph 1th	257.92	354.88	25438.29
Gorder 1th	**242.55 (5.96%)**	328.57 (7.42%)	18993.89 (25.33%)
NumBaCo 1th	**243.88 (5.44%)**	325.98 (8.15%)	**18658.66 (26.65%)**
Rabbit 1th	247.99 (3.85%)	**312.33 (11.99%)**	20680.43 (18.70%)
Cn-o 1th	244.20 (5.32%)	343.56 (3.19%)	**18758.09 (26.26%)**)
k-means-o 1th	252.95 (1.93%)	**308.80 (12.98%)**	24612.55 (3.25%)
cl-h-o 1th	253.48 (1.72%)	336.83 (5.09%)	24871.53 (2.23%)
k-means-o_cl-h-o 1th	246.93 (4.26%)	349.06 (1.64%)	21086.16 (17.11%)
Cn-o_k-means-o 1th	250.35 (2.93%)	353.12 (0.50%)	23644.92 (7.05%)
Cn-o_cl-h-o 1th	245.33 (4.88%)	350.81 (1.15%)	20725.76 (18.53%)
Gorder_k-means-o 1th	248.49 (3.65%)	330.20 (6.96%)	23308.4 (8.37%)
Gorder_cl-h-o 1th	244.01 (5.39%)	341.64 (3.73%)	19556.72 (23.12%)
NumBaCo_k-means-o 1th	248.85 (3.52%)	328.35 (7.48%)	23660.38 (6.99%)
NumBaCo_cl-h-o 1th	245.23 (4.92%)	332.80 (6.22%)	19998.94 (21.38%)
Rabbit_k-means-o 1th	250.67 (2.81%)	315.99 (10.96%)	23834.96 (6.30%)
Rabbit_cl-h-o 1th	245.95 (4.64%)	324.16 (8.66%)	20460.49 (19.57%)
Astro-ph 2th	260.91	387.78	27610.67
Gorder 2th	251.33 (3.67%)	351.63 (9.32%)	22197.45 (19.61%)
NumBaCo 2th	247.55 (5.12%)	366.24 (5.55%)	**21263.48 (22.99%)**
Rabbit 2th	**240.09 (7.98%)**	342.30 (11.73%)	23255.84 (15.77%)
Cn-o 2th	248.11 (4.91%)	366.93 (5.38%)	**21027.98 (23.84%)**
k-means-o 2th	242.56 (7.03%)	**316.79 (18.31%)**	26830.44 (2.83%)
cl-h-o 2th	242.20 (7.17%)	341.88 (11.84%)	25998.54 (5.84%)
k-means-o_cl-h-o 2th	240.89 (7.67%)	336.94 (13.11%)	23450.35 (15.07%)
Cn-o_k-means-o 2th	242.16 (7.19%)	366.17 (5.57%)	25549.98 (7.46%)
Cn-o_cl-h-o 2th	242.85 (6.92%)	368.26 (5.03%)	23119.75 (16.27%)
Gorder_k-means-o 2th	241.69 (7.37%)	353.71 (8.79%)	25523.43 (7.56%)
Gorder_cl-h-o 2th	240.37 (7.87%)	**326.40 (15.83%)**	22840.31 (17.28%)
NumBaCo_k-means-o 2th	**239.21 (8.32%)**	340.44 (12.21%)	25157.05 (8.89%)
NumBaCo_cl-h-o 2th	243.54 (6.66%)	338.02 (12.83%)	22697.09 (17.80%)
Rabbit_k-means-o 2th	241.76 (7.34%)	327.36 (15.58%)	25357.19 (8.16%)
Rabbit_cl-h-o 2th	248.74 (4.67%)	374.88 (3.33%)	23251.75 (15.79%)

Table 4: Graph Ordering Comparison with Pagerank, 2 threads, Astro-ph with a 12-cores in term of time (in milliseconds, ms), cache misses (in thousands, th) and cache references (in thousands, th)

5.3 Execution Time Reduction

In this section, we analyse the execution time reduction with Tables 2, 3 and 4. Execution time reduction corresponds to the gain due to the numbering when executing Pagerank on astro-ph graph. The observations made here are seen as a tangible impact of the improvements made by cache references and cache misses reduction discussed on the previous section 5.1 and section 5.2.

In Table 2, we can see that:

- With one thread, K-means-order and cl-hier-order improve base line without improve existing numbering. It is their combination with this last that improve performances. For example, the combination Gorder_Cl-hier-order produces the best time reduction with 9.18% while the combination Rabbit_cl-hier-order is third with 8.26%. The existing numbering Gorder is the second with 8.46%.
- With two threads, K-means-order and Cl-hier-order succeed to improve base line as well as the existing numbers; Cl-hier-order even take the third place with 12.29%. The combinations give the best improvement, it is the case of Gorder_Cl-hier-order with 13.21%, which correspond at the first place on the cache reference and cache misses. At the Second place we have the combination Cn-order_Cl-hier-order with 13.01% corresponding at the second place in cache reference and fourth place in cache misses.

In Table 3, with user machine 4-cores, we can remark that:

- With one thread, on this architecture, K-means and Cl-hier-order improve the base line(without numbering) but the exiting methods Cn-order, and Gorder take the two first place with 9.31% and 8.47% respectively. In the third place we have the combination Cn-order_Cl-hier-order with 8.22%.
- With two threads, K-means and Cl-hier-order improve the base line(without numbering) as well as all existing numbering with 26.02% and 25.52% respectively. In addition the combination NumBaCo_kmeans-order takes the first place with 27.38%. Rabbit-order takes the fourth with 25.33%.

In Table 4, with user machine 12-cores, we can remark that:

- K-means and Cl-hier-order improve the base line (without numbering) but not the existing methods Gorder and NumBaCo that take the first two places with 5.96 % and 5.44% respectively. The combination Gorder_Cl-hier-order combination takes the third place with 5.39%.
- With two threads, K-means and Cl-hier-order improve the base line(without numbering) with 7.03% and 7.17% respectively. They even did better than all the existing ordering except Rabbit-order which takes the second place with 7.98%. The combinations (NumBaCo_kmeans-order, Gorder_Cl-hier-order and Gorder_kmeans-order) take the first place with 8.32%, the third place with 7.87% and the fourth place with 7.37% respectively.

With these last observations we can say that the heuristics proposed in this article improve the performance of base line most of the time and we also note that the combinations succeed in improving the existing numbering even more.

5.4 Global Observations

When looking at all criterion together (time, cache misses, cache references), we can observe that, in many cases, the gains obtained on the memory caches are the causes of the gains obtained in the execution times. For example, this is the case on table2 with the combination Gorder_Cl-hier-order which correspond to the first place on the cache reference(39.85%) and the first place on cache misses(11.12%) and finally the best performance in execution time with 13.21%.

VI CONCLUSION

In this paper, we proposed a new numbering that exploit execution traces analysis in order to improve memory cache reduction and hence execution time reduction in graph application. To build this ordering, we defined a new distance and then we used it to analyse execution traces with well known clustering algorithms K-means (for Kmeans-order) and hierarchical clustering (for cl-hier-order). Even the results obtained with these clustering based ordering was already interesting, we did combinations between existing numbering and clustering based numbering in order to obtain better results. Experimental results did on three user machines show that:

- clustering based numbering allow to improve existing numbering.
- combination between clustering based numbering and existing numbering allows to get better results.

The gap between existing numbering results and results based on our proposal can sometime be more than 12%. For example, with user machine 4-cores, with one thread, we got the best cache misses reduction through the combination NumBaCo_K-means with 65.51% compared to 52.62% of cache misses reduction gotten with Cn-order (an existing numbering).

Our proposal has two main drawbacks. The first one is base on the fact that clustering such as hierarchical clustering takes long time to be proposed executed. This became a real challenge when someone wants to number a huge graph with our clustering based numbering. One can solve this problem by choosing a good clustering algorithm. The second drawback is related to our distance definition. In fact, our distance puts together nodes that have close execution traces lines bringing together nodes that are in the same line of execution trace. One can solve this problem by redefining a distance that will consider nodes that are in the same line of execution traces. Another solution can be based on considering execution traces as itemset and then measure the closeness of nodes through frequent itemset mining algorithm.

References

[1] L. Page, S. Brin, R. Motwani, and T. Winograd. “The PageRank citation ranking: bringing order to the web.” In: *Stanford InfoLab* (1999).

[2] M. E. Newman. “The structure and function of complex networks”. In: *SIAM* 45.2 (2003), pages 167–256.

[3] A. Groce. “Error explanation with distance metrics”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2004, pages 108–122.

[4] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014.

[5] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. “Rabbit order: Just-in-time parallel reordering for fast graph analysis”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2016, pages 22–31.

[6] H. Wei, J. X. Yu, C. Lu, and X. Lin. “Speedup graph processing by graph ordering”. In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pages 1813–1828.

[7] T. Messi Nguélé, M. Tchuente, and J.-F. Méhaut. “Social network ordering based on communities to reduce cache misses”. In: *Revue ARIMA* Volume 24 - 2016-2017 - Special issue CRI 2015 (May 2017).

[8] T. Messi Nguélé, M. Tchuente, and J. Méhaut. “Using Complex-Network Properties for Efficient Graph Analysis”. In: *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy*. 2017, pages 413–422.

[9] T. Messi Nguélé. “DSL for Social Network Analysis On Multicore Architecture”. PhD thesis. Université Grenoble Alpes; Université de Yaoundé I, Sept. 2018.

[10] T. Messi Nguélé and J.-F. Méhaut. “Applying Data Structure Succinctness to Graph Numbering For Efficient Graph Analysis”. In: *Revue Africaine de Recherche en Informatique et Mathématiques Appliquées* 32 (2022).

A ANNEX: FOURTH PROPERTIES OF D PROXIMITY

On this case we should show that:

$$|\Gamma(a)| + |\Gamma(c)| - 2|\Gamma(a) \cap \Gamma(c)| \leq |\Gamma(a)| + |\Gamma(b)| - 2|\Gamma(a) \cap \Gamma(b)| + |\Gamma(b)| + |\Gamma(c)| - 2|\Gamma(b) \cap \Gamma(c)|$$

Now we permute the elements of left and right and we also change the signe:

$|\Gamma(a)| + |\Gamma(b)| - 2|\Gamma(a) \cap \Gamma(b)| + |\Gamma(b)| + |\Gamma(c)| - 2|\Gamma(b) \cap \Gamma(c)| \geq |\Gamma(a)| + |\Gamma(c)| - 2|\Gamma(a) \cap \Gamma(c)|$

the $|\Gamma(a)|$ and the $|\Gamma(c)|$ will cancel out and th $|\Gamma(b)|$ will add up;

We also pass the factor $-2|\Gamma(a) \cap \Gamma(c)|$ on the other side of the inequality in order to have 0 on one side. Thus, we have:

$2|\Gamma(b)| - 2|\Gamma(a) \cap \Gamma(b)| - 2|\Gamma(b) \cap \Gamma(c)| + 2|\Gamma(a) \cap \Gamma(c)| \geq 0$

$\iff 2[|\Gamma(b)| - |\Gamma(a) \cap \Gamma(b)| - |\Gamma(b) \cap \Gamma(c)| + |\Gamma(a) \cap \Gamma(c)|] \geq 0$

$\iff |\Gamma(b)| - |\Gamma(a) \cap \Gamma(b)| - |\Gamma(b) \cap \Gamma(c)| + |\Gamma(a) \cap \Gamma(c)| \geq 0$

We group those who have the negative sign on one side and we have:

$|\Gamma(b)| + |\Gamma(a) \cap \Gamma(c)| - |\Gamma(a) \cap \Gamma(b)| - |\Gamma(b) \cap \Gamma(c)| \geq 0$

$\iff |\Gamma(b)| + |\Gamma(a) \cap \Gamma(c)| - [|\Gamma(a) \cap \Gamma(b)| + |\Gamma(b) \cap \Gamma(c)|] \geq 0$

So to show that $D(a, b) + D(b, c)$, it is enough to show that:

$|\Gamma(b)| + |\Gamma(a) \cap \Gamma(c)| - [|\Gamma(a) \cap \Gamma(b)| + |\Gamma(b) \cap \Gamma(c)|] \geq 0.$

Let us show that: $|\Gamma(b)| + |\Gamma(a) \cap \Gamma(c)| - [|\Gamma(a) \cap \Gamma(b)| + |\Gamma(b) \cap \Gamma(c)|] \geq 0$

$|\Gamma(b)| + |\Gamma(a) \cap \Gamma(c)| - [|\Gamma(a) \cap \Gamma(b)| + |\Gamma(b) \cap \Gamma(c)|] \geq 0$

$\iff |\Gamma(b)| + |\Gamma(a) \cap \Gamma(c)| \geq [|\Gamma(a) \cap \Gamma(b)| + |\Gamma(b) \cap \Gamma(c)|]$

With this inequality, the factors that can have different values are:

i^0) $|\Gamma(a) \cap \Gamma(c)| \leq |\Gamma(a)|$ If $|\Gamma(a)| < |\Gamma(c)|$ Or $\leq |\Gamma(c)|$ If $|\Gamma(c)| < |\Gamma(a)|$
ii^0) $|\Gamma(a) \cap \Gamma(b)| \leq |\Gamma(a)|$ If $|\Gamma(a)| < |\Gamma(b)|$ Or $\leq |\Gamma(b)|$ If $|\Gamma(b)| < |\Gamma(a)|$
iii^0) $|\Gamma(b) \cap \Gamma(c)| \leq |\Gamma(b)|$ If $|\Gamma(b)| < |\Gamma(c)|$ Or $\leq |\Gamma(c)|$ If $|\Gamma(c)| < |\Gamma(b)|$

For this inequality to no longer hold, the right side would have to be greater than the left side. The maximum cases for this to happen are:

1st Case : $|\Gamma(a) \cap \Gamma(b)| = |\Gamma(a)|$ and $|\Gamma(b) \cap \Gamma(c)| = |\Gamma(b)|$
2nd Case: $|\Gamma(a) \cap \Gamma(b)| = |\Gamma(a)|$ and $|\Gamma(b) \cap \Gamma(c)| = |\Gamma(c)|$
3rd Case: $|\Gamma(a) \cap \Gamma(b)| = |\Gamma(b)|$ and $|\Gamma(b) \cap \Gamma(c)| = |\Gamma(b)|$
4th Case: $|\Gamma(a) \cap \Gamma(b)| = |\Gamma(b)|$ and $|\Gamma(b) \cap \Gamma(c)| = |\Gamma(c)|$

From these different cases, we can determine the probable maximum value of $|\Gamma(a) \cap \Gamma(c)|$ contained in the left part of the inequality and see if the inequality will always be respected.

1.0.1 1st case : $(|\Gamma(a) \cap \Gamma(b)| = |\Gamma(a)|$ *and* $|\Gamma(b) \cap \Gamma(c)| = |\Gamma(b)|)$

In this first case we have the following information:

$$\begin{aligned}(|\Gamma(a)| < |\Gamma(b)| \text{ and } |\Gamma(b)| < |\Gamma(c)|) &\implies |\Gamma(a)| \leq |\Gamma(b)| \leq |\Gamma(c)| \\ &\implies |\Gamma(a) \cap \Gamma(c)| = |\Gamma(a)|\end{aligned}$$

The inequality $|\Gamma(b)| + |\Gamma(a) \cap \Gamma(c)| \geq [|\Gamma(a) \cap \Gamma(b)| + |\Gamma(b) \cap \Gamma(c)|]$

thus becomes $|\Gamma(b)| + |\Gamma(a)| \geq |\Gamma(a)| + |\Gamma(b)|$

And the inequality is therefore verified.

1.0.2 *2nd case* : $(|\Gamma(a) \cap \Gamma(b)| = |\Gamma(a)|\ \ and\ \ |\Gamma(b) \cap \Gamma(c)| = |\Gamma(c)|)$

Here $|\Gamma(a)| < |\Gamma(b)|$ and $|\Gamma(c)| < |\Gamma(b)| \implies |\Gamma(a)| \leq |\Gamma(b)| \geq |\Gamma(c)|$

In this case, we cannot directly conclude. We can however say that: $\exists\ q1, q2 \subset N\ /\ \ |\Gamma(b)| = |\Gamma(a)| + |q1|$ and $|\Gamma(b)| = |\Gamma(c)| + |q2|$

Thereby

$|\Gamma(a)| = |\Gamma(b)| - |q1|\ \ and\ \ |\Gamma(c)| = |\Gamma(b)| - |q2| \implies |\Gamma(a) \cap \Gamma(c)| = |\Gamma(a)| + |\Gamma(c)| - |\Gamma(a) \cup \Gamma(b)|$

Or

$|\Gamma(a) \cup \Gamma(b)| = |\Gamma(b)|$ because $|\Gamma(b)| > |\Gamma(a)|$

The inequality can be written:

$|\Gamma(b)| + |\Gamma(a)| + |\Gamma(c)| - |\Gamma(b)| \geq |\Gamma(a)| + |\Gamma(c)|$

We note that the $|\Gamma(a)|$ and $|\Gamma(c)|$ will cancel on either side of the inequality and we will have:

$|\Gamma(b)| - |\Gamma(b)| \geq 0$;

The $|\Gamma(b)|$ will cancel out and we will therefore have a valid expression.

1.0.3 *3rd case* : $(|\Gamma(a) \cap \Gamma(b)| = |\Gamma(b)|\ \ and\ \ |\Gamma(b) \cap \Gamma(c)| = |\Gamma(b)|)$

For the 3^{rd} case, we have $|\Gamma(b)| < |\Gamma(a)|$ and $|\Gamma(b)| < |\Gamma(c)| \implies |\Gamma(a)| \geq |\Gamma(b)| \leq |\Gamma(c)|$

In this case, we cannot directly conclude. However we can say that: $\exists\ q3, q4 \subset N\ \ /\ \ |\Gamma(a)| = |\Gamma(b)| + |q3|$ and $|\Gamma(c)| = |\Gamma(b)| + |q4|$.

Thus, $\Gamma(a) \cap \Gamma(c) = (\Gamma(b) \cup q3) \cap (\Gamma(b) \cup q4) = \Gamma(b) \cup (q3 \cap q4)$

Hence, $|\Gamma(a) \cap \Gamma(c)| = |\Gamma(b) \cup (q3 \cap q4)| = |\Gamma(b)| + |q3 \cap q4| - |\Gamma(b) \cap (q3 \cap q4)|$

The inequality therefore becomes:

$|\Gamma(b)| + |\Gamma(b)| + |q3 \cap q4| - |\Gamma(b) \cap (q3 \cap q4)| \geq |\Gamma(b)| + |\Gamma(b)|$

The $|\Gamma(b)|$ will all cancel out and we will have:

$|q3 \cap q4| - |\Gamma(b) \cap (q3 \cap q4)| \geq 0 \quad \iff \quad |q3 \cap q4| \geq |\Gamma(b) \cap (q3 \cap q4)|$

But by definition, $|q3 \cap q4| \geq |\Gamma(b) \cap (q3 \cap q4)$

The result is therefore valid, which allows us to verify our 3^{rd} case.

1.0.4 *4th case*: $(|\Gamma(a) \cap \Gamma(b)| = |\Gamma(b)|\ \ and\ \ |\Gamma(b) \cap \Gamma(c)| = |\Gamma(c)|)$

For this last case, $|\Gamma(c)| < |\Gamma(b)|$ and $|\Gamma(b)| < |\Gamma(a)|$ $\implies |\Gamma(a)| \geq |\Gamma(b)| \geq |\Gamma(c)|$

$$\implies |\Gamma(a) \cap \Gamma(c)| = |\Gamma(c)|.$$

The inequality therefore becomes: $|\Gamma(b)| + |\Gamma(c)| \geq |\Gamma(b)| + |\Gamma(c)|$

This gives us a valid result.

All cases could be tested and verified.

B BIOGRAPHY

Regis Audran MOGO WAFO, Computer Science PhD Student in University of Yaounde 1

Thomas MESSI NGUÉLÉ, Computer Science Doctor/PhD, Senior Lecturer at University of Yaounde 1, Head of Department of Computer Engineering at HITLC of University of Ebolowa

Armel Jacques NZEKON NZEKO'O, Computer Science Doctor/PhD, Senior Lecturer at University of Yaounde 1

Xaviera YOUTH KIMBI, Computer Science Doctor/PhD, Senior Lecturer at University of Yaounde 1

C ACKNOWLEDGMENT

The successful completion of this research has been made possible through the generous support and collaboration of various scholarship stakeholders. We express our sincere gratitude to International Development Research Centre (IDRC), Swedish International Development Cooperation Agency (SIDA) and African Center for Technology Studies (ACTS) for awarding the Artificial Intelligence for Development (AI4D) scholarship program that funded this research. This scholarship not only provided financial support but also served as a source of motivation and encouragement throughout the project.

We are grateful for the support of all members of our work team namely High performance data science (HIPERDAS), who contributed to stimulating discussions and shared their insights. The collaborative environment fostered by our academic community has been integral to the development of this work. Lastly, we extend our thanks to all those who, directly or indirectly, played a role in the realization of this paper. Your support has been crucial, and we are grateful for the opportunities provided by the scholarship stakeholders and the broader academic community.