A Distributed and Incremental Algorithm for Large-Scale Graph Clustering

Wissem Inoubli, Sabeur Aridhi, Haithem × Mezni, Mondher Maddouri, Engelbert Mephu Nguifo

A Distributed and Incremental Algorithm for Large-Scale Graph Clustering

Wissem Inoubli[a], Sabeur Aridhi[b], Haithem Mezni[c], Mondher Maddouri[d], Engelbert Mephu Nguifo[e]

[a]*University of Tallinn, Tallinn, Estonia and University of Tunis El-Manar, Tunis, Tunisia*
[b]*University of Lorraine, CNRS, LORIA / Inria NGE, France*
[c]*SMART Lab (Tunisia), Taibah University, KSA*
[d]*College Of Business, University of Jeddah, Saudi Arabia*
[e]*University Clermont Auvergne, CNRS, Clermont Auvergne INP, LIMOS, 63000 Clermont-Ferrand, France*

Abstract

Graph clustering is one of the key techniques to understand structures that are presented in networks. In addition to clusters, bridges and outliers detection is also a critical task as it plays an important role in the analysis of networks. Recently, several graph clustering methods are developed and used in multiple application domains such as biological network analysis, recommendation systems and community detection. Most of these algorithms are based on the structural clustering algorithm. Yet, this kind of algorithm is based on the structural similarity. This latter requires to parse all graph ' edges in order to compute the structural similarity. However, the height needs of similarity computing make this algorithm more adequate for small graphs, without significant support to deal with large-scale networks. In this paper, we propose a novel distributed graph clustering algorithm based on structural graph clustering. The experimental results show the efficiency in terms of running time of the proposed algorithm in large networks compared to existing structural graph clustering methods.

Keywords: Graph processing, Structural graph clustering, Big Graph Analysis, Community detection, Outliers detection, hubs detection

1. Introduction

Recently, the graph model has risen as one of the most used data models in several applications such as, social networks [31], road maps [8], pattern mining [18] and bioinformatics [45]. For instance, a recent ranking shows that the popularity of graph databases model increased up to around 500% in the last years [24]. It has been shown that the graph model is an optimum data model that allows to represent easily many relationships and facilitates the exploration of data. Taking social networks as an example, the graph model organizes data elements into a set of vertices representing the members, and a list of edges to materialize the relationships between vertices. In addition, the last years featured a big data explosion especially in graph-based social networks. As an example, Facebook in 2013 had over 874 million monthly users [5]. This proliferation of a huge amount of data and the massiveness of graphs introduce additional factors to the renewed popularity of graph analytics [24]. Consequently, new applications and use cases have been mentioned in the literature. One important task in graph mining is graph clustering. Graph clustering helps identifying tightly connected regions within a graph [47, 20]. It has been used to solve various problems such as discovering communities in social networks and detecting protein complexes in Protein–Protein Interaction (PPI) networks [45]. It is important to mention that in some applications, the used graph could be large and dynamic, which are two challenging features. The first one is about the graph size that makes its processing beyond the capacity of a single machine. For example, *Friendster* which is one of the used datasets in social networks, groups more than 65 millions vertices and around 2 billions edges. According to an experimental study [41], the needed main memory after loading this dataset is about 264 GB while the memory of a single high-performance computer is generally much less than 256 GB [41]. The second feature i.e., dynamicity, is about the fact that the structure of graphs change over time in terms of vertices and edges [38]. In this context and according to Facebook, 400 new users sign up for the website every 60 seconds and 20 million friend requests are sent every 20 minutes in 2020 [41] In such situation, the considered graph could be partitioned into several sub-graphs and the computation is performed in a parallel/distributed way [13, 2]. In this context, graph clustering algorithms are used to ensure the partitioning of the graph into several parts [1, 46].

In this paper, we propose DISCAN, a novel graph clustering algorithm in the context of large and dynamic graphs. The main contributions of this work are summarized as follows:

- We propose a distributed graph clustering algorithm of both large and distributed graphs (DSCAN). The proposed method is exact and allows discovering the same clusters that are discovered by SCAN, the reference algorithm for structural graph clustering [44].
- We propose DISCAN (Distributed and Incremental SCAN), an incremental implementation of DSCAN, and we experimentally evaluate its efficiency.
- We conduct an extensive experimental study with large

graphs, to evaluate the scalability of DISCAN. We compare DISCAN with four existing structural graph clustering algorithms.

The rest of the paper is organized as follows. After introducing graph clustering in Section 1, we present a brief overview of related works in Section 2. In Section 3, we present the basic concepts related to structural graph clustering. In Section 4, we present our proposed algorithms for large and dynamic graph clustering. In Section 5, we provide the experimental results. The last section is devoted to the conclusions and the future directions.

2. Related work

Several graph clustering algorithms have been proposed in the literature. Taking as examples, modularity-based approaches [28] that represent an optimization solution of the modularity measure for each partitioning schema (generated randomly or according to a heuristic function) [28]. The Louvain method [4] represents one of the clustering algorithms based on modularity, that initially generates a random clustering. After that, it starts to change every time a vertex from a cluster to another until getting a maximum modularity. Despite the fact that it gives very connected clusters in the larger graphs, the modularity measure cannot capture the small clusters [27]. Graph partitioning [6] and min-cut [15] are other methods used for the graph clustering which consist of splitting a graph into sub-graphs while optimizing the cut edge during the partitioning. The spectral clustering [42] is based on the graph density. It represents an input graph with a matrix and transforms this matrix so that to apply the basic clustering algorithm, like k-means [21].
A sampling-based distributed graph clustering method has been proposed in [37]. The proposed algorithm is based on the density of the graph to produce a set of sparse sub-graphs. Graph embedding has also been used in the graph clustering. In [19], the authors discussed the use of the graph embedding technique to combine the structural and attributed similarity over the graph clustering. The above methods provide, as output, a list of clusters which are not really sufficient to understand the graph behavior. To address this issue, the Structural Clustering Algorithm for Networks (SCAN) was proposed in [44] aiming, not only to identify the clusters in a graph, but also to provide additional information like hubs (vertices between one or more clusters) and outliers (vertices that do not belong to any cluster). These additional pieces of information can be used to detect vertices that can be considered as noise and also vertices that can be considered as bridges between clusters. The functional principle of SCAN is based on graph topology. It consists of grouping vertices that share the maximum number of neighbors. Moreover, it computes the similarity between all the edges of the graph in order to perform the clustering. The similarity computation step in SCAN is linear according to the number of edges, which degrades SCAN performance especially in case of large graphs. Structural graph clustering is one of the most effective clustering methods for differentiating the various types of vertices in a graph. In the literature, several works have been proposed for the structural graph clustering to overcome the drawback of SCAN. In [34], Shiokawa et al. proposed an extension of the basic SCAN algorithm, namely SCAN++. The proposed algorithm aims to introduce a new data structure of directly two-hop-away reachable node set (DTAR). This new data structure is the set of two-hop-away nodes from a given node that are likely to be in the same cluster as the given node. SCAN++ could save many structural similarity operations, since it avoids several computations of structural similarity by vertices that are shared between the neighbors of a vertex and its two-hop-away vertices.

In the same way, the authors in [9] suppose that the identification of core vertices represents an essential and expensive task in SCAN. Based on this assumption, they proposed a pruning method for identifying the core vertices after a pruning step, which aims to avoid a high number of structural similarity computations. To improve the performance and robustness of the basic SCAN, an algorithm named LinkSCAN* has been proposed in [29]. LinkSCAN* is based on a sampling method, which is applied on the edges of a given graph. This sampling aims to reduce the number of structural similarity operations that should be executed. However, LinkSCAN* provides approximate results.

Other works have proposed parallel and distributed implementations of SCAN algorithm [39, 30, 10, 36, 40, 35, 26, 49]. In [39], the authors proposed an approach based on openMP library [7]. The author's aims were to ensure a parallel computation of the structural similarity and to show the impact of parallelism on the response time. Their method was proven to be faster than the basic SCAN algorithm. Other works have focused on the problem of dynamic graph clustering. In [30] and [10, 12], the authors have extended SCAN algorithm to deal with the addition or removal of nodes and edges. Authors in [36], used the graphical processing unit (GPU) whose purpose is to parallelize the processing and to benefit from the high number of processing slots in the GPU which increase the degree of parallelism. Moreover, distributed approaches are proposed in order to overcome the scalability of structural similarity computing, these algorithms [49, 26, 35] are based on big data framework, but they do not support the already partitioned graphs.
On the other hand, an index approach is proposed in [40] where the authors have emphasized the impact of the two parameters μ and ϵ on the running time of the structural graph clustering. Using the index approach a significant improvement has been shown in terms of clustering running time. Again, the two sensitive parameters μ and ϵ have been taken into consideration by the authors in [43]. The main goal of the latter consists of estimate the best μ and ϵ values in order to enhance the accuracy and efficiency of the clustering results.
Most of the above presented works suffer from three major problems: (1) they do not deal with big graphs, (2) they do not consider already distributed/partitioned graphs and (3) they do not consider dynamic graphs.

Through Table 2, we have summarized the discussed approaches according to some features.
As shown in Table. 2, most proposed algorithms allow parallel

Table 1: Comparative study on existing graph clustering methods

Approach	Parallel	Distributed	Processing model	Graph partitioning	Main contributions
SCAN [44]	No	No	Sequential	No	Basic implementation of structural graph clustering
SCAN++ [34]	No	No	Sequential	No	Reducing the number of similarity computations
AnySCAN [48]	Yes	No	Parallel	No	Parallelizing SCAN
pSCAN [9]	Yes	No	Parallel	No	Reducing the number of similarity computations
Index-based SCAN [40]	No	No	Sequential	No	Interactive SCAN
ppSCAN [11]	Yes	No	Parallel	No	Parallel version of pSCAN
SCAN based on GPU [36]	Yes	No	Parallel	No	A GPU-based version of SCAN
pm-SCAN [33]	Yes	No	Parallel	Yes	Graph partitioning to reduce the I/O costs
DSCAN [35]	Yes	Yes	OpenMp and MPI	No	Distributed version of SCAN based on openMP and MPI framework

processing but could not deal with very large graphs. In addition, these approaches are unable to process large and dynamic graphs. Also in some applications, like social network, graphs are distributed by nature. Thereby, using the discussed algorithms, it should aggregate in one machine all partitions of a distributed graph in order to run the graph clustering. Based on this limits, in this work, we tackle the problem of large and dynamic graph clustering in a distributed setting.

3. Background

3.1. Structural graph clustering: Basic concepts

Graph clustering consists in dividing a graph into several partitions or sub-graphs. As with other clustering techniques, we must use one or more metrics to measure the similarity between two vertices or partitions in the graph. In the structural clustering technique, the graph structure or topology is splitted into a set of sub-graphs that are relatively distant and a set of vertices that are strongly connected. As a well-known algorithm for structural graph clustering, SCAN algorithm uses the structural similarity between vertices to perform the clustering. In addition, it provides other pieces of information like outliers and bridges. In what follows, we first present an overview of graphs and graph clustering with the SCAN algorithm.

Consider a graph $G = \{V, E\}$, where V is a set of vertices, and E is a set of edges between vertices. Each of those elements can represent a real property in real-word applications. Let u and v be two vertices in V. We denote by (u,v) an edge between u and v; u (resp. v) is said to be a neighbor of v (resp. u).

In the following, we extend some basic definitions of structural graph clustering, which will be used in our proposed algorithm.

Definition 3.1. *(Structural neighborhood) The structural neighborhood of a vertex v,is denoted by N(v), and represents all the neighbors of a given vertex $v \in V$, including the vertex v:*

$$N(v) = \{u \in V | (v, u) \in E\} \cup \{v\} \quad (1)$$

Table 2: Summary of symbols.

Symbol	Description
$N(v)$	The neighbors of vertex v
$N_\epsilon(v)$	The strong neighbors of v
V_c	The core vertices
$N_\epsilon(V_c)$	The border vertices of vertex V
o	An outlier vertex
b	A bridge vertex
$\mathbb{P}$	Partitions of G
P_i	A Partition i
$V^f_{P_i}$	The frontier vertices of P_i
$\mathbb{C}$	A set of clusters
$\mathbb{B}$	A set of borders
$\mathbb{O}$	A set of outliers
C_i	The i^{th} cluster
U	A set of updates (adding/removal of vertices/edges)
V_A	The affected vertices
V_{AD}	The directly affected vertices
V_{ID}	The indirectly affected vertices
V_{AB}	The affected bridges
E_A	A list of affected edges
V_{new}	A new Vertex in U
V_{old}	An old vertex in G
V_{TBD}	Vertices to be deleted
E_{TBD}	Edges to be deleted
$C_{affected}$	A list of affected clusters

Definition 3.2. *(Structural similarity) The structural similarity between each pair of vertices (u, v) in E represents a number of shared structural neighbors between u and v. It is defined by $\sigma(u, v)$.*

$$\sigma(u, v) = \frac{|N(u) \cap N(v)|}{\sqrt{|N(u)| \,.\, |N(v)|}} \quad (2)$$

After calculating the structural similarity with Eq. (2), SCAN uses two parameters to detect the core vertices in a given graph G.

Definition 3.3. *(ϵ-neighborhood) Each vertex has a set of structural neighbors, like it is mentioned in Definition 3.1. To group one vertex and its neighbors in the same cluster, they must have a strong connection (denoted by ϵ-neighborhood) between them.*

SCAN uses a ϵ threshold and Eq. (3) to filter, for each vertex, its strongest connections. The ϵ-neighborhood is defined as follows.

$$N_\epsilon(u) = \{N(u) | \sigma(u, v) \geq \epsilon\} \quad (3)$$

The ϵ threshold ($0 < \epsilon \leqslant 1$) shows to what extent two vertices u and v are connected based on the shared structural neighbors. In addition, it represents a metric with which the most important vertices, also called *core* vertices, are detected.

Definition 3.4. *(Core) Core vertices detection is a fundamental step in SCAN algorithm. It consists of finding the dominant vertices in a given graph G. This step allows to build the set of clusters or a clustering mapping. A core vertex v is a vertex which has a sufficient number of neighbors strongly connected with it $N_\epsilon(v)$. We use μ as a minimum number of strong connected neighbors (see Definition 3.3). A core vertex is modeled as follows:*

$$V_c = \{v \mid |N_\epsilon(v)| \geq \mu\} \quad (4)$$

Definition 3.5. *(Border) Let v_c be a core vertex. v_c has two lists of structural neighbors: (1) weak connected neighbors to v_c, also called noise vertices ($N(V_c) \setminus N_\epsilon(V_c)$), and (2) strong connected neighbors called reachable structured neighbors. In our work, reachable structured neighbors are called border vertices. $N_\epsilon(V_c)$ represents the border vertices of a core vertex V_c.*

Once the nodes and their borders are determined, it is straightforward to start a clustering step. To do so, we use the following definition:

Definition 3.6. *(Cluster) A cluster C ($|C| \geq 1$) is a nonempty subset of vertices, where its construction is based on the set of core vertices and their border vertices.*

The clusters' building algorithm consists of the following main steps: first, it randomly chooses a core from the cores' list and creates a cluster C. Then, it pushes the core and its borders into the same cluster. At the same time, the algorithm checks if the list of borders has a core vertex. Then, it inserts borders into the same cluster and it applies this step recursively until adding all the borders of the connected cores. Finally, the algorithm chooses other core vertices and executes the previous steps until checking all core vertices.

Among the fundamental information returned by SCAN, compared to other graph clustering algorithms, we mention bridge and outlier information, defined as below:

Definition 3.7. *(Bridge) A bridge b is a vertex $v \in V$ that does not belong to any cluster after performing the clustering step, and it has at least two links (edges) with two different clusters.*

Definition 3.8. *(Outlier) An outlier o is a vertex $v \in V$ that does not belong to any cluster after the clustering step, and it is not a bridge.*

The clustering step aggregates the core vertices and their borders into a set of clusters. However, some vertices do not have strong connections with a core vertex, which does not give the possibility to join any cluster. In this context, SCAN algorithm classifies those vertices into two families: bridges and outliers. A vertex v, that is not part of any cluster and has at least two neighbors in different clusters, is called bridge. Otherwise it is considered as an outlier.

Algorithm 1: Basic SCAN algorithm

```
   Input  : A Graph G and parameters (μ ,ϵ)
   Output: ℂ, 𝔹, 𝕆
 1 foreach (u, v) ∈ E do
 2 |   Compute σ(u, v)
 3 end
 4 Core ← ∅
 5 foreach u ∈ V do
 6 |   if (|N_ϵ(u)| ≥ μ) then
 7 |   |   Core = Core ∪ {u}
 8 |   end
 9 end
10 Cluster ← ∅
11 foreach unprocessed core vertex u ∈ Core do
12 |   Cluster ← {u}
13 |   Mark u as processed
14 |   foreach unprocessed border of vertex v ∈N_ϵ(u) do
15 |   |   Cluster←Cluster ∪ {v}
16 |   |   Mark v as processed
17 |   end
18 end
19
```

3.2. Running example

In this section, we explain through a running example, how SCAN algorithm works. Consider a graph G presented in Fig. 1 and the parameters $\epsilon = 0.7$ and $\mu = 3$. In the first step, Algorithm 18 (lines 1-3) use Eq. (2) to compute the structural similarity for each edge $e \in \mathbb{E}$. Then, Eq. (3) (lines 5-8) is used to define the core vertices (see gray vertices in Fig. 1). After that, we proceed to the clustering step, then we apply Definition 3.6 (lines 11-16) to build the clustering schema. In the example we have four core vertices: 0, 2, 9 and 10. Each core has a list of border vertices (the neighbors that have strong connections with a core). In the example, the vertex number 2 is a core. This latter has the vertices 1, 4, 5, 3 and 0 as the list of borders, since they have strong connections with the core. The core and its borders build a cluster, and if a border is a core we join all its borders into the cluster. Like in our example, the vertex 2 is a core. Hence, we join all its borders 1, 3, 5, 4, including 0 as being a core vertex. In this case, if the vertex 3 is not a border of vertex 2 and it is a border of vertex 0, then vertex 3 should belong to the same cluster of vertex 2 (which is a core vertex), since it is a reachable border of vertex 2. The last step of SCAN algorithm consists in defining the bridges and the outliers. As shown in Fig. 1, we have two clusters. The first one is composed of vertices 0, 1, 2, 3, 4, and 5, whereas the second cluster is composed of nodes 8, 9, 10, and 11. The remaining vertices

(6 and 7) must be categorized as outliers and bridges according to the Definition 3.7. In our example, vertex 7 has two connections with two different clusters, and vertex 6 has only one connection with one cluster, which makes vertex 7 a bridge and vertex 6 an outlier.

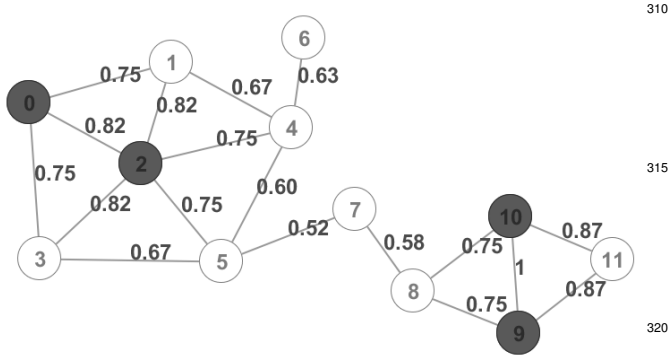


Figure 1: Running example of SCAN ($\epsilon = 0.7$ and $\mu = 3$).

4. DISCAN: A Distributed and Incremental Algorithm for Large-Scale Graph Clustering

In this section, we introduce DISCAN: a new distributed and incremental algorithm for structural graph clustering. Our proposed approach is based on a master/slave architecture and is implemented on top of BLADYG framework [3]. This latter is a distributed block-centring graph processing framework in which the slaves are responsible for the execution of a specific computation and the master machine coordinates between all the slaves. The input data must be divided into sets of chunks (sub-graphs in our case). Each chunck/partition is assigned to a worker, which performs a specific computation. The master machine orchestrates the workers' execution. In the following, we present the main three steps of DISCAN: (1) graph partitioning, (2) distributed graph clustering named DSCAN [22] and (3) incremental graph clustering.

4.1. Distributed graph partitioning

The used distributed graph processing framework uses by default a random graph partitioning and provides the possibility to implement a personalized graph partitioning method. Motivated by the fact that the random graph partitioning method can affect the performance of the proposed algorithm and the graph partitioning problem is an NP-hard problems [14], we adapted an existing edge partitioning method [25] in a distributed manner as a preliminary step for our distributed clustering algorithm.

In this step, we split the input graph G into several small partitions $P_1, P_2, ..P_n$ and affect the partitions to the worker machines. To ensure the consistency property while dividing the input graph, we must identify a list of cuts edges in order to have a global view of G. As depicted in Algorithm 2, the master machine first divides equitably an input graph file into sub-files according to the number of edges and the number of partitions, and sends the sub-files to all the workers lines(2-5). Each worker gets a list of edges and vertices from its sub-file. Thereafter, it sends the list of vertices to all workers, in order to determine the frontier vertices so that to get the cuts edges. When each worker receives a list of vertices from its neighbor workers, it determines the vertices that belong to the current worker (partition). Consequently, these vertices are considered as frontier vertices in their partitions lines(6-9). For example, consider the partitions P_1 and P_2, which are assigned to two workers. Each partition has its set of edges: $P_1 = \{(1,2),(1,3),(1,4)\}$ and $P_2 = \{(1,5),(1,6),(5,7)\}$. When the workers receive the sets of vertices from other workers, they check the intersection between their vertices and the received ones. The obtained list of vertices is considered as a set of frontier vertices. According to our example, vertex 1 is considered as a frontier vertex in P_1, and vertices 5 and 6 are considered as frontier vertices in P_2. In the last step, once each worker has determined the frontier vertices, it fixes the cuts edges, i.e. edges that have a frontier vertex (lines 10-18).

Algorithm 2: Distributed graph partitioning

Input : Graph file GF as a text file, parameter (NP number of partitions)
Output: $\mathbb{P}$ set of partitions

1 *BLADYG initialization according to the NP parameter*
2 *Master machine: split GF into a set of sub-files* $\mathbb{GF}$
3 **foreach** $GF_i \in \mathbb{GF}$ **do**
4 | *Assign* GF_i *to worker* W_i
5 **end**
/* In parallel */
6 **foreach** *Worker* $W_i \in \mathbb{W}$ **do**
7 | *Get list of vertices and edges from* GF_i
8 | *Find the list of frontier vertices from the neighbor workers*
9 **end**
/* In parallel */
10 **foreach** *Worker* $W_i \in \mathbb{W}$ **do**
11 | **foreach** *Edge* $\in \mathbb{E}$ **do**
12 | | *(a,b)=Edge* ;
13 | | *Let* P_i^f *the frontier vertices*
14 | | **if** $(a \in P_i^f \cup b \in P_i^f)$ **then**
15 | | | *Set E as a cut edge*
16 | | **end**
17 | **end**
18 **end**

4.2. Initial distributed graph clustering

Initial graph clustering (DSCAN) has two main steps: (1) local clustering step and (2) merging step.

Step 1: Local clustering. As presented in Algorithm 3, the input graph G is splitted into multiple sub-graphs/partitions ($\mathbb{P}$), each one is assigned to a worker machine. The partitioning step, as mentioned in Section 4.1, is performed according to the α parameter, which refers to the number of worker machines (line 1). To overcome the loss of information during the partitioning step (edges connecting nodes in different partitions),

Algorithm 3: DSCAN

Input : Graph G , parameters (μ, ϵ, α)
Output: $\mathbb{C}lusters, \mathbb{B}ridges, \mathbb{O}utliers$

```
/* Divide G into subgraphs 𝔾 = {G1G2..Gα} according to
   parameter α                                              */
1  ℙ ⟵ Partition (G,α)
   /* In parallel                                           */
2  foreach Pi ∈ ℙ do
3  |  Assign Pi to Wi
4  end
   /* In parallel                                           */
   /* Step 1:  Local clustering                             */
5  foreach Worker Wi ∈ 𝕎 do
6  |  Let Pi the current partition
7  |  Find the frontier vertices in Pi and duplicate them into neighbor partitions
8  end
   /* In parallel                                           */
9  foreach Worker Wi ∈ 𝕎 do
10 |  Compute the structural similarity of a partition Pi using V^f list
11 |  Retrive local Cores and Borders in Pi
12 |  Build local clusters in Pi
13 |  Find local Bridges and Outliers in Pi
14 end
   /* Step 2:  Merging                                      */
15 All workers exchange theirs local clusters between them; using Worker2Worker
   message
16 foreach Worker Wi ∈ 𝕎 do
17 |  if (C1 ∩ C2 ∩ .. ∩ Cα = 𝕍; and ∃Vi ∈ ℂore then
18 |  |  C ← Merge(C1, C2,..Cα)
19 |  |  Send C to the master
20 |  end
21 |  else
22 |  |  Send local clusters to master
23 |  end
24 |  for Vi IN 𝕆utliers do
25 |  |  if (Vi ∈ ℂore ∪ 𝔹order ∪ 𝔹ridges ) then
26 |  |  |  Remove Vi from the list of 𝕆utliers
27 |  |  end
28 |  end
29 |  for Vi IN 𝕆utliers do
30 |  |  Nb_Connections ← 0
31 |  |  for Ci IN ℂlusters do
32 |  |  |  if (N(Vi) ∩ Ci ≠ ∅) then
33 |  |  |  |  Nb_Connections + +
34 |  |  |  end
35 |  |  end
36 |  |  if ( Nb_Connections ≥ 2 ) then
37 |  |  |  Add Vi to 𝔹ridges
38 |  |  |  Remove Vi from 𝕆utliers
39 |  |  end
40 |  end
41 end
42 Send Clusers,Bridges,Outliers to the master using Worker2Master message
```

frontier vertices are duplicated into neighboring partitions (line 5-8). Subsequently, for each partition P_i, a local clustering is performed on each worker machine (lines 9-14). Fig. 2 shows a demonstrative example of the duplication step. The demonstrative example describes how the graph consistency will be ensured during the partitioning step.

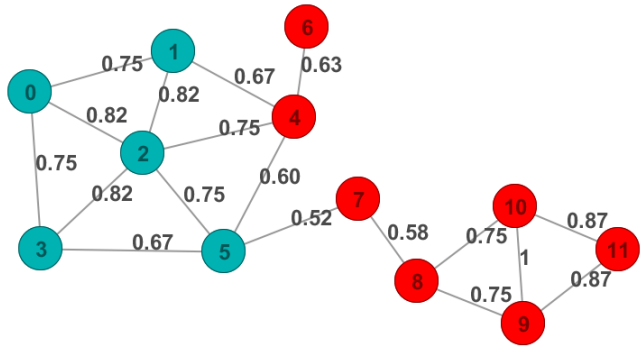


Figure 2: Illustrative example of partitioning step

Assume that a graph G is partitioned into two partitions P_1 (vertices in blue) and P_2 (vertices in red), like it is depicted in Fig. 2, and each partition has a set of vertices connected with other partitions. We call them frontier vertices of a given partition P_i, and they are denoted by $V^f_{P_i}$. For example, $V^f_{P_1}$={1, 2, 5} and $V^f_{P_2}$={4, 7}. Each $v \in V^f_P$ has a set of internal and external neighbors. For example, vertex 4 is a vertex of the partition P_2. It has the vertex 6 as internal and the vertices 1, 2 and 5 as external neighbors. Computing the structural similarity of a frontier vertex u considers that all the neighbors of u belong to the same partition. Thereby, we duplicate all frontier vertices in partition P to all neighbor partitions and we call them external vertices. For example in our running example, P_1 has frontier vertices V^f_{P1}={1,2 , 5} and $P2$ is the neighboring partition. Thus, we must duplicate $V^f_{P_1}$ into P_2 to ensure the accuracy of structural similarity of $V^f_{P_2}$ (see Fig. 3).

After that, when we apply a local clustering on P_1 and P_2, we will find several conflicts such as the vertex $v \in V^f_{P1}$ is a core vertex in P_1, and an outlier in P_2. These conflicts should be avoided in the merging step.

Step 2: Merging. The distribution of similarity computation and the local clustering step can improve the response time of DISCAN, compared to the basic sequential algorithm. However, we should take into consideration the exactness of the returned results, compared to those of the basic SCAN. To ensure the same results as in the basic SCAN, we defined a set of scenarios regarding the merging step. These scenarios will repetitively be applied to every two partitions of G, until combining all the partitions (see Algorithm 3, lines 16-37). For each pair of partitions P_i and P_j, a merging function is executed to combine the local results from P_i and P_j and store them in global variables like clusters, borders, bridges and outliers. Algorithm 3 also achieves several scenarios (Lemmas 4.1, 4.2 and 4.3) to solve the encountered conflicts, mentioned below:

Lemma 4.1. ***(Merging local clusters)*** *Let* $\mathbb{C}_1$ *and* $\mathbb{C}_2$ *two sets*

of local clusters in different partitions P_1 and P_2, respectively. $\exists c_1 \in \mathbb{C}_1$ *and* $\exists c_2 \in \mathbb{C}_2$, $Core(c_1) \cap Core(c_2) \neq \emptyset$.

Let c_i be a cluster that groups a set of border and core vertices. If c_i shares at least one core vertex c with another cluster c_j, then c has a set of borders in c_i and c_j, and all the vertices in c_i and c_j are reachable from c. Hence, c_i and c_j should be merged into the same cluster.

Lemma 4.2. ***(Outlier to Bridge)***
Let $\mathbb{C}_1$ and $\mathbb{C}_2$ be two sets of local clusters in partitions P_1 and P_2, respectively. $\exists$ two clusters c_1 and c_2 that belong to the sets of clusters $\mathbb{C}_1$ and $\mathbb{C}_2$. In addition, $\exists$ o an outlier in both partitions P_1 and P_2, with $N(o) \cap c_1 \neq \emptyset$ and $N(o) \cap c_2 \neq \emptyset$.

If c_i and c_j ($i \neq j$) share an outlier o, this means that o is weakly connected with the two clusters c_i and c_j. Hence, according to the Definition 3.7, o should be changed to a bridge vertex.

Lemma 4.3. ***(Bridge to Outlier)*** *Let c_1 and c_2 two local clusters in the partitions P_1 and P_2, respectively.*
$\exists$ a bridge b according to only two clusters c_1 and c_2, when c_1 and c_2 two clusters that will be merged into one cluster, b should be changed into an outlier.

Let c_i and c_j two clusters that have a set of vertices (borders or cores), and a set of bridges with other local clusters, and $\exists$ b a bridge vertex according to the two clusters c_i and c_j only, where $i \neq j$. In the merging step and according to Lemma 4.1, if one or several clusters share at least a core vertex, then they will be merged into a single cluster. In this case, $(c_i,c_j) \Rightarrow C$ which makes b be weakly connected with only one cluster C, then according to Definition 3.7, b should change its status from bridge to outlier.

For instance looking at lines 16 to 22, we have focused on the shared cores between two clusters and the case when they share at least one core. According to Lemma 4.1, we should merge them into one single cluster. Subsequently, in lines 23-25, we verify for each outlier if it does not belong to some sets of cores, borders or bridges. In this case, we must remove it from the list of outliers. Otherwise, a vertex v should be changed as a bridge if it is classified as an outlier in the two clusters c_i and c_j that are not in the same partition, and if it has two connections with different clusters in the merging step.

Illustrative example

Fig. 3 shows a demonstrative example of a graph clustering using DISCAN. In this example, we use the same parameters (ϵ and μ) of the running example in Section 3, and two partitions ($P1$ and $P2$). In the first step of DISCAN, the input graph G is divided into two partitions P_1 and P_2 as it presented in Fig. 3, where the blue vertices represent the first partition and the red vertices represent the second partition. In the second step, DISCAN duplicates the frontier vertices in each pair of adjacent partitions. For example, the blue vertices 1, 2 and 5 are duplicated in partition P_2 since they represent frontier vertices in their partition. In the same way, the vertices 4 and 7 are duplicated in the partition P_2. In the third step, all the workers perform the similarity computation, check the status for each vertex and build the local clusters, as it is shown in Fig 3. The last step of DISCAN consists of combining all local results returned by each worker. As shown in Fig. 3, there are some conflicts in terms of node status. For example, vertex 2 is a core vertex in P_1 whereas it is a noise (outlier) vertex in P_2. In the same way, vertex 4 is a border in P_1 and a noise in P_2. Then, after building the local clusters, P_1 has one cluster (1, 2, 3, 4, and 5) in which vertex 7 is a noise vertex in P_2. As for P_2, it groups the vertices 8, 9, 10, and 11 as a cluster and the remaining vertices (4, 5, 1, 2, 7, and 6) are considered as noise vertices including vertex 7. This latter is a bridge according to basic SCAN (see running example in Section 3)). In the merging step, DISCAN considers that the vertex 7 has two weak connections with two different clusters. Thus, vertex 7 is marked as a bridge.

4.3. Distributed graph maintenance

Consider a large and dynamic graph G, which will undergo several changes over time such as adding or deleting edges or/and vertices. When a clustering will be performed, the classic approaches re-run the clustering from scratch, which is obviously very expensive especially in large graphs. In this work, we propose an incremental and distributed algorithm for large and dynamic graph clustering. The main idea of our approach is to determine in each update the vertices and the edges concerned by this update, and according to that, we apply the new updates on the old clustering results.
Assume that $G = (V, E)$ is going to undergo several updates U over time. U can be adding/deleting vertices or edges from/to G. These changes can affect the similarity values and some vertex status (border, core, etc.). Consequently, the clustering schema may be affected in several situations.

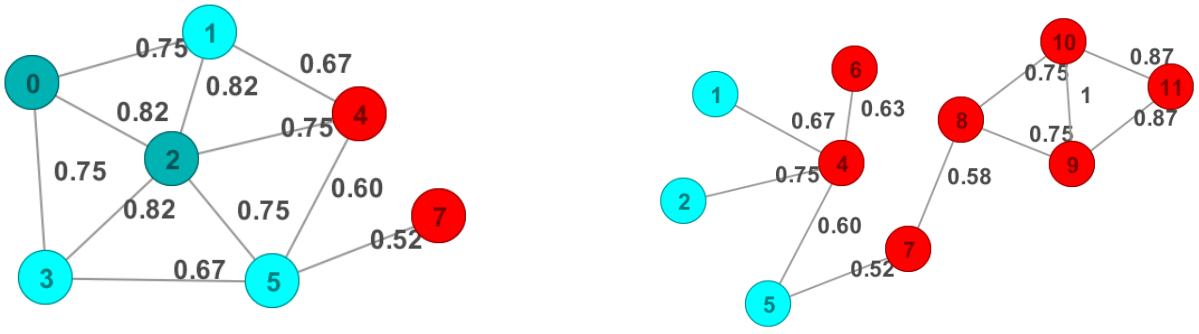
Definition 4.1. *Each vertex $u_i \in U$ generates modifications of several structural similarities, which is defined as the affected edges E_A.These vertices also change the status of several vertices and are noted as affected vertices V_A.*

Definition 4.2. *(Affected edges) When adding a vertex v_{new} associated with an existing vertex v_{old}, new edge (v_{new}, V_{old}) will be created and a set of edges denoted E_A will be marked as affected edges and should be updated.*

Note that $e = (u,v) \in E_A$, where $u \in V_{old}$ and $v \in N(V_{old})$.

In the case of removing a vertex V_{TBD}, a set of edges noted as E_{TBD} should be deleted from G. Therefore, a V_c of vertices is concerned by this update. Here $V_c = (u, v) \in E_{TBD} \setminus V_{TBD}$ and, according to V_c, the affected edges are $E_A = E_i \subset G$ and $E_i = (u, v) / \{u,v\} \in V_c$.

Also when adding or deleting an edge (v_i, v_j), the list of neighbors of both vertices v_i and v_j will be changed which can produce several other edges. The affected edges update their structural similarities. E_A is a set of edges $(u, v) \in E$ and $v_i = u$ or $v_j = v$



(a) Partition 1

(b) Partition 2

Figure 3: The partitioned graph G used in the running example in Section 3

Definition 4.3. *(Affected Vertices) For each update on G, some structural similarities can be changed. Thus, some vertices change their status (border, core vertices). These vertices are defined as affected vertices V_A. V_A can be divided into two sub-sets: (i) directly affected V_{AD}, which update their status depending directly on E_A, (ii) indirectly affected V_{ID} which depend on the status of V_{AD}. For example, when a core vertex changes his status to outlier or border, then all these border vertices will become affected indirectly according to this update. Let E_A=(v_1, v_2) be following an update V_{AD} =$V_{AD} \cup v_1 \cup v_2$, while V_{AI} are all neighbors of V_{AD}. Other vertices can be affected indirectly with several updates, and are dependent on the changed clusters. These vertices are defined the affected bridges and noted as V_{AB}. V_{AB} is the list of outliers or bridges which are connected with affected clusters described in the Definition 4.4.*

Definition 4.4. *(Affected clusters)*

The affected vertices in several situations can affect the clustering schema. The Affected (concerned) clusters represent the sub-set of clusters which contain one or more affected vertices, denoted by $C_{affected}$.
$C_{affected}$=$c \in \mathbb{C}$ such that $\exists v \in (v_{AD} \cup v_{ID})$ and $v \in c$.

Lemma 4.4. ***Change of structural similarity of affected edges*** *E_A are mainly depend on the Definition 3.1. The similarity value is based on the neighbors of both vertices of an edge. Therefore, each update (adding/deleting an edge/vertex) can change the list of neighbors of some vertices. As a result, some edges must change their structural similarity.*

Lemma 4.5. ***Change in the status of affected vertex*** *According to the Definition 3.4 and the Definition 3.3, for some updates in the structural similarity, when the similarity value exceeds or decreases the threshold ϵ the status of this edge maybe changed to strong or weakly connection. Thereby, this change can affect the status of each vertex $v \in V_{AD}$. Consequently, according to 3.5 the V_{DA} can change other vertices status indirectly which are defined as indirectly affected vertices V_{IA}. Rationally, those changes can affect the mapping of the clusters, and several clusters may be changed. Thereby, according to Definition 3.7, $\exists$ a vertex $v \in \mathbb{B}$ where v is only depends on two clusters c_1 and c_2. When c_1 and c_2 will be merged into one cluster c_{new}, v will be weakly connected with only c_{new}. In this case and according to Definition 3.7 v will be an outlier vertex. In the same way, when we have a cluster c and v is an outlier to c with two weak connections related to v_1 and v_2, after some updates c becomes splited into two clusters c_1 and c_2 when v_1 and v_2 are respectively in c_1 and c_2. In this case, v becomes a bridge between the new clusters c_1 and c_2 according to the connection of v with v_1 and v_2.*

Fig. 4 shows an example of a graph G with 3 partitions for which we deleted vertex 3 (yellow vertex). Then, all affected edges E_A (red edges) change their structural similarities since vertices 0, 2 and 5 (old neighbors of the deleted vertex 3) lost the vertex 3 as a neighbor, and all the edges associated with vertex 3 are removed from G. This new update affects only a sub-set of vertices in G. The example shows that vertex 2 ($v_2 \in V_A$) lost its core status, whereas vertex 9 keeps its core status ($v_9 \notin V_A$). Thus, E_A can change several vertices' status directly (V_{AD}) or indirectly (V_{AI}). Like it is shown in our example, vertex 2 $\in V_{AD}$ has lost a strong connection with the removed vertex which changed to an outlier in this case. In the same way, V_{AD} can change the status of V_{AI}. Vertices 0, 1, 4 and 5 were considered as border vertices according to vertex 2 (in the initial graph), and in our case, when the vertex 2 lost the core status, as a result V_{AI} are changed to outliers. According to Definition 4.4, V_{AD} and V_{AI} can change the old clustering schema and our example confirm that. Partition 1 of G groups two clusters. The first one is c_1 = { 12, 13, 14} and the second one is c_2 ={0, 1, 2, 3, 4, 5} (4 and 5 are external vertices). Thus, c_1 and c_2 are affected because c_1 contains the vertex 12 as an affected vertex, and in the c_2, all its vertices are affected because of the removed vertex. These clusters can affect in the last step other vertices (bridge or outlier vertices). Like it is depicted in Fig. 4 c_2 should be removed since it does not have any core vertex, and the vertices 6 and 7 were considered as bridges with c_2. Thereby, these bridges are susceptible to change their status.

Algorithm 4 describes the main steps of the proposed DISCAN. DISCAN provides real-time graph maintenance and a micro-batch clustering. Moreover, it performs the new clustering after several changes, in order to optimize the incremental clustering. We present below the main steps of DISCAN:

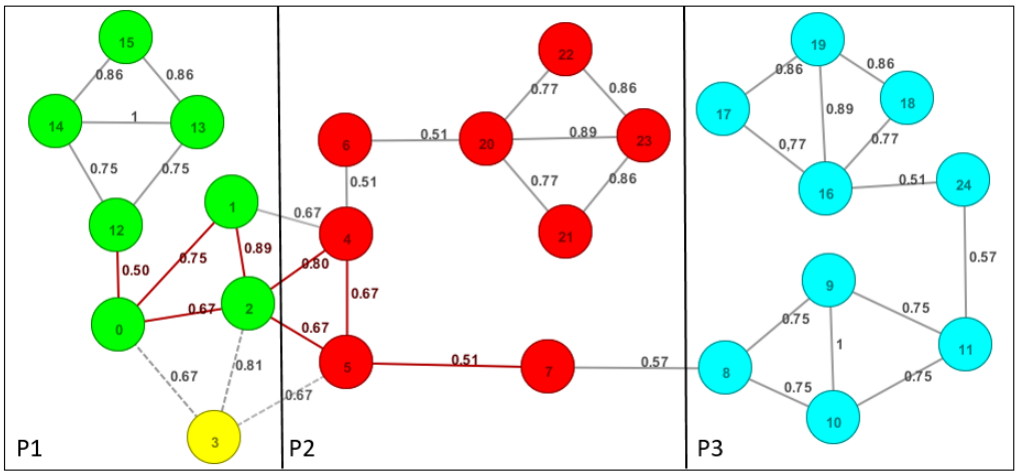


Figure 4: A partitioned graph G ($P1$, $P2$, $P3$), affected vertices, edges when deleting vertex 3

Step 1: Graph maintenance. In this step, DISCAN runs the graph maintenance in real time. In each update $u \in U$, it (i) updates the graph G structure, (ii) checks the affected edges E_A, and (iii) recomputes the similarities of E_A (lines 4-12 of the algorithm 4). Then, it gets V_A according to E_A in order to memorize them in a global variable. Like discussed in Definition 4.4, V_A can affect several clusters in the current partition or in other partitions. Thereby, each worker shares its V_A with all workers so that each one determines the affected clusters and saves them. Once the affected clusters are fixed, they affect some bridges or outliers vertices. So the affected bridges should be added to the list of affected vertices to be checked in the next step.

Step 2: Incremental local clustering. This step is performed in micro-batch processing mode, after a number of updates or using a window time. In both methods, DISCAN runs the same scenarios. These latter consist of two choices for a user who selects one of them according to the number of changes per seconds or according to other needs such as the graph size or graph evolution. Therefore, in each batch, DISCAN first checks the core vertices. The checking is performed only on the affected vertices V_A, where in general the $|V_A| \ll |V|$.

Secondly, DISCAN checks the border vertices like in the first clustering but using only E_A, since the border vertices are dependent on the strong connections with core vertices. For this reason, DISCAN checks the core vertices according to E_A only. After that, the updated cores and borders vertices will change the clustering schema depending of the affected vertices (cores, borders) like it is described in Definition 4.4. The affected clusters feat are five possible cases owing to each update of a graph: (1) split one cluster to small clusters, (2) remove an existing cluster, (3) build new clusters, (4) update the existing clusters and (5) merge two or more clusters. Consider set of cores and theirs borders, and the core vertices have strong connections between them. After the removal of a strong connection between two cores c_1 and c_2, then the clusters should be splited into sub-clusters depending on c_1 and c_2. For the case (2), each cluster is built on a list of cores and their borders. If one cluster does have any core vertex it should be removed. Sometimes, a new update makes an outlier vertex to a core vertex c_{new}. If this vertex has a strong connection with any other old core c, it joins the cluster of c, like in the case (4). If c_{new} does have any strong connections with any other core it builds a new cluster, like the case (3). In the last case (5), if there exists a new strong connection between two cores in two different clusters, then we merge these latter. In order to deal with all these cases, we remove all the affected clusters and re-build a new clustering in a same way of the first clustering presented in Definition 3.6 and using only the affected vertices (cores and borders vertices). The new clustering is performed only on the changed vertices which will be very faster compared to the first clustering.

Finally, DISCAN uses the new clusters and the remaining affected vertices to check if they represent new border vertices.

Step 3: Merge the new updates. After each batch, each worker starts to merge the new updates (see Algorithm 4 lines 19-23). The master machine gets all the affected vertices from all workers in order to facilitate the combination of the new changes. Here, some clusters should be verified. In each batch, the master keeps only the past unchanged clusters and requests all workers to get the changed clusters. Thereby, all workers combine the updated clusters eventually including new clusters. In the merging step, DISCAN uses the same scenarios as in the first clustering (see Lemma 4.1). If at most one cluster shares at least one core vertex, they can be merged into one cluster. In the next step, after getting the new clusters, the master machine requests all workers to get their borders vertices. Then, each worker filters only the affected vertices which will belong to its local bridges, and sends them to the master. The rest of affected vertices will be added to the global list of outliers. Finally, DISCAN initializes the global affected vertices and clusters to empty lists which will to be used in the next batch.

4.3.1. Time complexity analysis of DISCAN

The time complexity of DISCAN mainly depends on the affected edges and vertices in each new update. This complexity is very reduced compared to the initial complexity of DSCAN,

Algorithm 4: DISCAN

Input : Initial graph G as a text file, parameter (NP number of partitions), a new update U

Output: $\mathbb{C}, \mathbb{B}, \mathbb{O}$

```
/* Global affected vertices and clusters in
   each worker                                  */
```
$\mathbb{G}lobalAffectedVertices$=$\emptyset$;

$\mathbb{G}lobalAffectedClusters$=$\emptyset$;

```
/* Step 1:  Graph maintenance                   */
/* Update the inital graph in each new update
   and get the affected vertices and the
   affected clusters                            */
```
Master machine: send a new update u to all workers ;

```
/* In parallel                                  */
```
foreach $Worker_i$ $w_i \in \mathbb{W}$ **do**

  Update the current partition according to U ;

  Get E_A;

  Recompute the similarities of E_A;

  Get the immediately affected vertices V_I;

  Share the immediately affected vertices V_I *with all workers*;

  Get the indirectly affected vertices V_{Ind}, *and the affected clusters* C_A;

end

```
/* Step 2:  Local clustering schema
   maintenance                                  */
/* Update the old local clustering schema
   according to the global affected vertices
   */
```
foreach $Worker_i$ $w_i \in \mathbb{W}$ **do**

  Check Core vertices from GlobalAffectedVertices;

  Check Border vertices from GlobalAffectedVertices;

  Check affected clusters;

  Check affected bridges;

end

```
/* Step 3:  Merge the new updates               */
```
foreach $Worker_i$ $w_i \in \mathbb{W}$ **do**

  Merge all affected clusters from all workers;

  Check new Bridge vertices according to the new clusters;

  Check the remaining outlier vertices;

end

```
/* Reset the global affected vertices and
   clusters in each worker                      */
```
$\mathbb{G}lobalAffectedVertices$=$\emptyset$;

$\mathbb{G}lobalAffectedClusters$=$\emptyset$;

Send $\mathbb{C}$,$\mathbb{B}$,$\mathbb{O}$ *to master using Worker2Master message*;

since the number of affected edges $|E_A| \ll |E|$ and the number of affected vertices $|V_A| \ll |V|$. In DISCAN, the time complexity is based on three steps: (1) the *graph maintenance*, which takes $O(|E|/n)$ in the worst case and $O = 1$ in the best case, where n is the number of partitions; (2) the *similarity computation*, which takes $O(min(|N(u)|, |N(v)|).|\mathbb{E}_A|)$; and (3) the *clustering* step, which mainly depends on V_A. Thereby, the incremental clustering complexity is $O(|V_c \in V_A| - |(v_i, v_j)|_{i,j=1}^{|V_A|})$, with $v_i, v_j \in V_c$ and $\sigma(v_i, v_j) \geq \epsilon$.

5. Experiments

In this section, we evaluate the effectiveness and efficiency of our proposed algorithm for the structural clustering of large and dynamic distributed graphs.

5.1. Experimental protocol

We compared DISCAN with four existing structural graph clustering algorithms in the case of static graphs. Then, in a second set of experiments, we evaluated DISCAN performance on dynamic graphs. We also conducted experiments in order to study the impact of DISCAN features on the execution time such as the number of updates.

- Basic SCAN,
- pSCAN: a pruning SCAN algorithm,
- AnySCAN: a parallel implementation of basic SCAN using OpenMP library,
- ppSCAN: a pruning and parallel SCAN implementation.

The above mentioned algorithms are implemented with *C* language. Thus, we used the GCC/GNU compiler to build their binary versions. The compared algorithms could be divided into two categories: (i) centralized algorithms such as SCAN and pSCAN, and (ii) parallel algorithms such as AnySCAN and ppSCAN. To run both centralized and parallel algorithms, we used a *T3.2xlarge* virtual machine on Amazon EC2. This machine is equipped with an 8 vCPU Intel Skylake CPUs at 2.5 GHz and 32 GB of main memory on a Ubuntu 16.04 server distribution. In order to evaluate DISCAN, we used a cluster of 10 machines, each of them is equipped with a 4Ghz CPU, 32 GB of main memory, and operating with Linux Ubuntu 16.04. Details on the configuration and the deployment of DISCAN are available following this link: `https://github.com/inoubliwissem/remote-master`.

5.2. Experimental data

For all test cases with static graphs, we used real-world graphs (see Table 3) obtained from the Stanford Network Analysis Project (SNAP) [1]. Then, to simulate dynamic changes in the graphs, we generated a set of new edges for each static graph ($G2, G3, G4$ and $G5$). The new updates are grouped as a set of batches. In each one, we generated five batches with different sizes that vary from 2000 to 10000 new edges.

[1] https://snap.stanford.edu/data/

Table 3: Graph datasets

Graph	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$
G1: California road network	1 965 206	2 766 607	1.4
G2: Youtube	1 134 890	2 987 624	2.63
G3: Orkut	3 072 441	117 185 083	38.14
G4: LiveJournal	3 997 962	34 681 189	8.67
G5: Friendster	65 608 366	1 806 067 135	27.52
G6: web-ClueWeb09-50m^2	428 136 613	446 534 058	1.04

5.3. *BLADYG framework*

BLADYG is a distributed and parallel graph processing framework that runs on a commodity hardware. The architecture of BLADYG is based on a master/slave architecture. BLADYG starts by reading the input graph from many different sources, which can be local or distributed files, such as Hadoop Distributed File System (HDFS) and Amazon Simple Storage Service (Amazon S3). The communication model used by BLADYG is the message passing technique, which consists in sending messages explicitly from one component to another, in order to get or send useful data during the graph processing. In the same way, BLADYG defines two types of messages: (i) worker-to-worker messages, and (ii) master-to-worker messages. BLADYG allows its users to implement their own partitioning techniques.

5.4. *Experimental results*

Speedup of initial distributed graph clustering. DISCAN is designed to deal with dynamic graphs. However, it starts by doing an initial clustering of the input graph. Then, it deals with incremental changes in the graph. We evaluated the speedup of the first step of DISCAN and we compared it with the basic SCAN and its variants presented in Section 5.1. The compared algorithms use different graph representations. In fact, AnySCAN and SCAN implementations use the adjacency list representation [16], whereas both pSCAN and ppSCAN use the Compressed Sparse Row (CSR) format [17]. In our proposed algorithm, we used an edge list format, in which each line represents one edge of the graph. The incompatibility of graph representations poses an additional transformation cost while evaluating the studied methods. For example, the transformation of the live journal dataset from edge list to adjacency list takes around 100 seconds using one machine equipped with an 8 vCPU and 32 GB of main memory.

As shown in Fig. 5, our approach is slower than the other algorithms, in the case of small graphs ($G1$, $G2$, and $G3$). Also, there was a very large gap between DISCAN and the other algorithms, especially with pSCAN and ppSCAN. This gap is reduced when the graph size increases (case of $G4$ dataset). The plots bars, in Fig. 5, show a gap of $12x$ between DISCAN and basic SCAN with the $G1$ dataset and $2x$ only with the $G4$ dataset. We notice that the gap between DISCAN and ppSCAN depends mainly on the size of the used dataset. For example, with the $G1$ dataset, the gap between DISCAN and ppSCAN reached $20x$, while it is reduced to $11x$ with the $G4$ dataset. That can be explained by the pruning step of pSCAN,

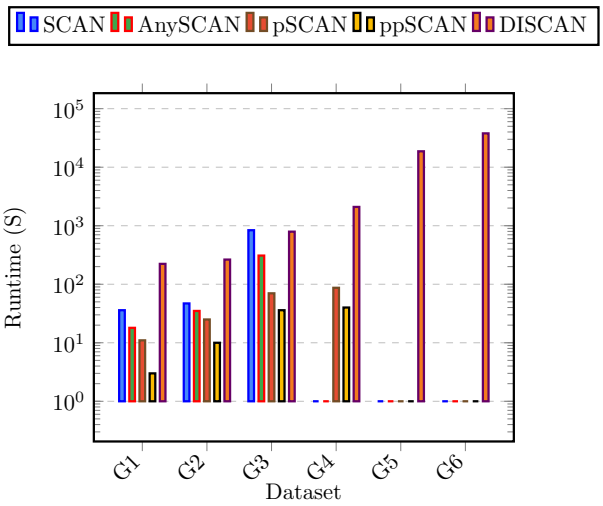


Figure 5: Impact of the graph size on the processing time of SCAN variants and DISCAN

which exempts several similarity calculations during the clustering step. It is important to mention that DISCAN is a distributed implementation of SCAN, which is not the case for the other studied algorithms, as they are centralized. This leads to additional costs related to data distribution, synchronization and communication. Fig. 5 also shows that with modest hardware configurations, only DISCAN can scale with large graphs like the $G5$ and $G6$ datasets.

Imapct of the graph partitioning on DISCAN. In our vision, the partitioning step has a direct impact on the response time of DISCAN. For this reason, we randomly generated four partitioning schemas, and for each one, we got the number of cut-edges as follows: 17.6M, 19.2M, 22.3M and 24.6M for the partitions $P1$, $P2$, $P3$ and $P4$, respectively. Then, we run DISCAN on all the partitions with the same configuration (10 machines, ϵ=0.5 and μ=3). As shown in Fig. 6, there is a very clear impact

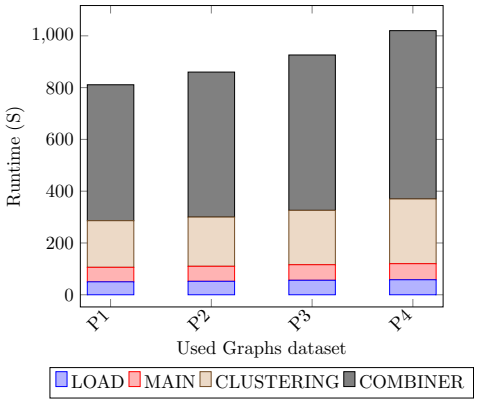


Figure 6: Impact of the partitioning step on DISCAN computation time (ϵ=0.5, μ=3).

of the graph partitioning on the response time of DISCAN, as this latter rises from nearly $800s$ to $1000s$ with $P1$ and $P2$. Furthermore, the elapsed time of each DISCAN step varies from one partitioning to another. This disparity is noticed mostly

during the merging step and slightly in the clustering step. This is probably explained by the number of vertices, which are duplicated due to the number of cuts-edges produced by the graph partitioning. This number would affect the amount of similarity computation operations during the clustering step, and it increases the communication cost during the merging step.

SpeedUp of incremental clustering. The main goal of this test series is to compare DISCAN with other algorithms. Since there is no incremental algorithm for the structural graph clustering, we compared the speedup of DISCAN with the results produced with pSCAN and ppSCAN, the fastest existing algorithm (see Section 5.1). Initially, we started by running all the algorithms on the used graphs. Then, we periodically added new batch of updates of different sizes. Fig. 7 shows the running time of the tested algorithms with different graphs. In all the used graphs, DISCAN is outperformed by the other algorithms, except for pSCAN which does not support the $G5$ graph. Despite that pSCAN and ppSCAN re-execute the clustering from scratch, they are faster than DISCAN in the case of $G2$, and have a close time in the case of $G3$. In the case of $G4$, DISCAN initially features a better behavior than pSCAN and ppSCAN. In fact, ppSCAN and pSCAN are faster than DISCAN, but the gap in running time begins to narrow between $G2$ and $G4$. However, when we add a new batch to the initial graphs, DISCAN becomes faster than pSCAN and ppSCAN. In the case of $G3$, Fig. 7 shows a gap of $2x$ between DISCAN and pSCAN and almost close running time to DISCAN and ppSCAN. Furthermore, the gap between DISCAN and pSCAN starts to increase, to finally reach $3x$, and 10% between DISCAN and ppSCAN.

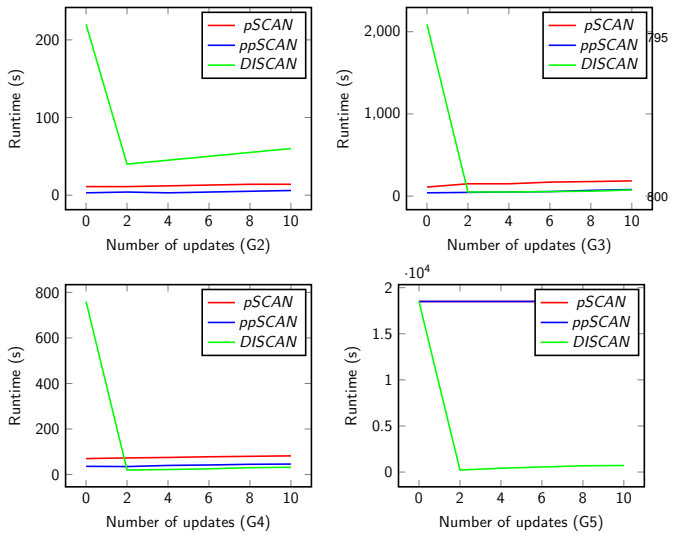


Figure 7: Impact of the number of updates on the processing time of the pSCAN, ppSCAN and DISCAN (ϵ=0.5, μ=3).

DISCAN scalability. Fig. 8 shows the scalability of DISCAN w.r.t. the number of workers, with $G2$ and $G3$ datasets, and for different update batches using default parameters (ϵ=0.5 and μ=3).

Overall, the number of workers affects the running time depending on the size of the used dataset and the number of updates. In Fig. 8 with $G2$ dataset, all curves have almost the same shape. The response time is low with 2 to 4 workers, but this is not always valid, because this improvement depends on the number of updates. The improvements are between 10% and 50%, respectively, for 2000 and 10000 updates. Afterwards, the curves grow until 8 workers, then comes down. In the case of $G3$ dataset, the scalability becomes very high. The response time decreases depending on the number of workers. This improvement is about 30% and 40% depending on the number of updates. This behavior can be explained by the number of affected vertices and clusters. When this number is small, the needed resources (e.g., workers) are small. Nevertheless, with many workers, the processing time of DISCAN becomes high due to the communication cost.

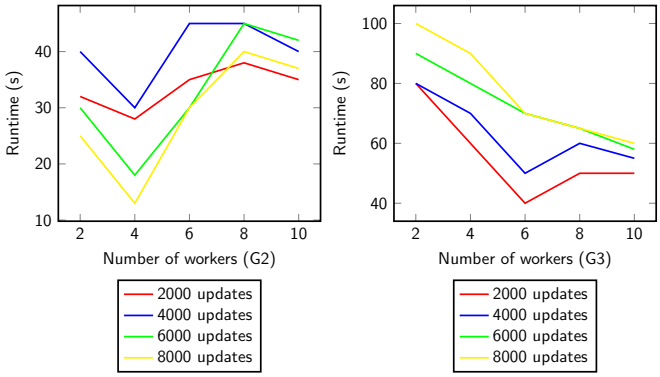


Figure 8: Impact of the workers' number on DISCAN processing time (ϵ=0.5, μ=3).

Exactness of DISCAN In this test series, we prove the exactness of DISCAN, by comparing its obtained results to those produced by ppSCAN. For this purpose, we used the information-theoretic metric and Normalized Mutual Information (NMI) [32]. The NMI is a normalization of the Mutual Information that shows how two clustering results are equals. The NMI value ranges from 0 (no mutual information) to 1 (perfect correlation). Table 4 shows the NMI score between DISCAN and ppSCAN results on all the experimented datasets, expect for the Friendster dataset, as this latter was not tested by ppSCAN. Like depicted in Table 4, DISCAN ensures the exactness feature.

Table 4: Exactness evaluation

Datasets	Youtube	LiveJournal	Orkut	Friendster
NMI Value	1.0	1.0	1.0	–

Impact of the batch size. The main goal of this experiment is to evaluate the impact of the batch size on the response time. Initially, we run DISCAN with different datasets (G2, G3, G4 and G5) and different batch sizes (between 1000 and 2000) on a cluster of 10 machines, and using the default parameters (ϵ=0.5, μ=3). Fig. 9 shows that the running time increases for

all curves. This is explained by the initial graph size, as we highlighted in Fig. 7, but also in function of the batch size. We can also notice that the increase depends on the size of the initial graph and the batch size. The increase rate is about 5%, 50% and 150%, respectively for G2, G4 and G5. That can mainly be explained by the graph size, since for each update DISCAN checks (in the graph maintenance step) all the affected vertices and edges, in order to update the affected similarities. Such checking depends on the graph size. On the other hand, in the reclustering step, DISCAN first performs the clustering on the affected vertices. Then, it merges all the affected clusters on the master machine. Thus, the running time spent on the merging step depends on the number of affected edges, because it must check all the affected clusters. The affected clusters' checking requires the communication between the master and other workers. In this way, when the batch is small, i.e. few affected vertices, the communication cost becomes acceptable.

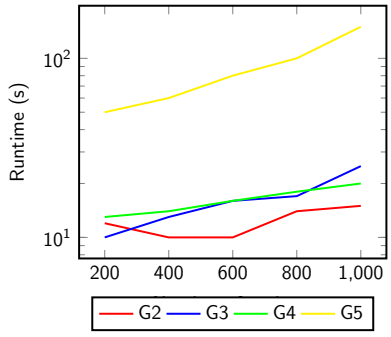


Figure 9: Impact of the batch size on the running time of DISCAN (ϵ=0.5, μ=3)

Impact of the vertex type (internal *vs* external). Each update on an existing graph may be performed on internal or frontier (external) vertices. In this experiment, we show the difference between the updates in internal and external vertices, in terms of running time. For $G2$, $G3$, $G4$ and $G5$, we generated several internal and external batches of adding new edges. After that, we ran DISCAN using those batches, in order to compare their computation times. As depicted in Fig. 10, the updates

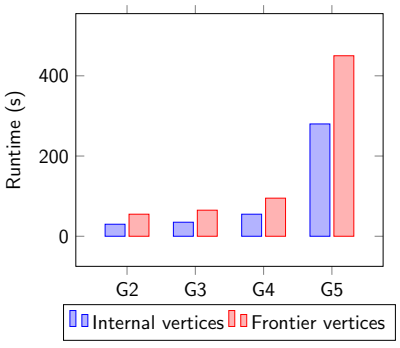


Figure 10: Impact of the vertices type (internal or external) on the processing time (ϵ=0.5, μ=3)

on the frontier vertices are very expensive with regard to the running time. This latter's gap trends to around 50% between internal and external vertices updates. That could be attributed to the graphs' maintenance step, which is performed by DISCAN. This latter is time-consuming in the case of external vertices, compared to internal vertices. In fact, for internal vertices, all maintenance operations are performed locally (on the same partition), whereas for external vertices, the graph maintenance is carried out on several partitions, which generates additional costs. These latter are mainly caused by the duplication of the frontier partitions and obviously the communication between all the concerned workers.

DISCAN steps. We noticed, from the previous experiment, that the response time depends on the type of vertices (internal or external). For external vertices, the algorithm's performance is slow, unlike the case of internal vertices.

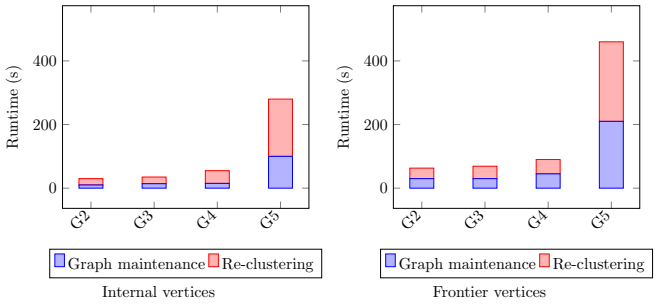


Figure 11: Performance of DISCAN's graph maintenance and graph reclustering steps (ϵ=0.5, μ=3).

Fig. 11 clearly shows a span of the difference between graph maintenance and reclustering steps for both internal and external vertices. The reclustering step takes about 30% of the global running time in the case of internal vertices, while this rate is about 50% in the case of frontier vertices. On the other hand, the difference between internal and external updates in terms of running time during the reclustering is about 3x. This is understandable, because the external updates, like the duplication of frontier vertices and the graph maintenance according to the neighboring partitions, need additional computation time. The additional treatment sometimes requires several message exchanges between workers.

Resources consumption. In this experiment, we study the resources consumption and communication overhead during the running of DISCAN. In order to get the resources consumption in the cluster of machines, a monitoring solution [23] is adopted for this purpose. The cluster's resource consumption is gathered for each machine at each second. Fig 12 shows the resources consumption in terms of CPU, memory and network traffic usage. The CPU usage shows a height CPU consumption during the running of DISCAN. 80% to 100% of the cluster CPU were occupied throughout 90% of the running time. Based on Fig 12 a huge memory was needed, and as it is depicted by the memory usage curve, the peak memory usage reaches 30GB by machine (around 252 GB as a total memory consumption). The traffic bandwidth sub-figure shows the amount of data exchanged between the machines in the cluster. It is obvious that a massive

usage of the communication is recorded during two periods, and a very small communication usage was noticed during the remaining periods. This can be explained by DISCAN behavior, where the graph maintenance and combining steps of DISCAN require a lot of communication, which decreases the performance of DISCAN. This also could be attributed to the use of communications between workers (Peer-To-Peer communication) in BLADYG framework, which is expensive compared to the master-slaves communication.

6. Conclusion

In this paper, we proposed DISCAN, a distributed algorithm for big graph clustering based on the structural similarity. DISCAN is build on top of a distributed and master/slaves architecture which makes it scalable and works on the community of modest machines. The proposed algorithm is able to deal with any graph size and is scalable with a large number of machines, in a parallel way. We have presented the main functions of DISCAN, starting from the partitioning to the combining of intermediate results, for each worker. Also, we have performed an extensive experimental study to validate and compare our proposed algorithm with other ones. The experiments have shown that DISCAN featured an horizontal scalability that is not guaranteed with other algorithms.

In the future, we will improve DISCAN's graph partitioning step. We plan to exploit the density feature during the graph partitioning, whereas for the dynamic graph clustering, we plan to make DISCAN support large and evolving graphs, in order to check all the snapshots of a graph. Having this in hand, we can assess and follow the evolution of each cluster, as well as other vertex types (border, outlier, and bridge).

Acknowledgment

We would like to express our sincere gratitude to the University of Clermont Auvergne and the LIMOS laboratory especially William Guyot, for providing human conditions, logistical and material support that helped us to the development and achievement of this project. They mainly allowed us to perform the experimental study of this work in a data center called GALACTICA (https://galactica.isima.fr). This work was partially supported by the CNRS-INRIA/FAPs project "TempoGraphs" (PRC2243).

References

[1] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov. Streaming graph partitioning: an experimental study. *Proceedings of the VLDB Endowment*, 11(11):1590–1603, 2018.

[2] S. Aridhi, L. d'Orazio, M. Maddouri, and E. M. Nguifo. Density-based data partitioning strategy to approximate large-scale subgraph mining. *Information Systems*, 48:213–223, 2015.

[3] S. Aridhi, A. Montresor, and Y. Velegrakis. Bladyg: A graph processing framework for large dynamic graphs. *Big Data Research*, 9:9–17, 2017.

[4] T. Aynaud and J.-L. Guillaume. Static community detection algorithms for evolving networks. In *8th International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, pages 513–519. IEEE, 2010.

[5] M. Baborska-Narozny, E. Stirling, and F. Stevenson. Exploring the relationship between a'facebook group'and face-to-face interactions in'weak-tie'residential communities. In *Proceedings of the 7th 2016 International Conference on Social Media & Society*, page 17. ACM, 2016.

[6] U. Brandes, M. Gaertler, and D. Wagner. Experiments on graph clustering algorithms. In *European Symposium on Algorithms*, pages 568–579. Springer, 2003.

[7] J. M. Bull. Measuring synchronisation and scheduling overheads in openmp. In *Proceedings of First European Workshop on OpenMP*, volume 8, page 49. Citeseer, 1999.

[8] L. Cao and J. Krumm. From gps traces to a routable road map. In *Proceedings of the 17th ACM international conference on advances in geographic information systems*, pages 3–12. ACM, 2009.

[9] L. Chang, W. Li, X. Lin, L. Qin, and W. Zhang. pscan: Fast and exact structural graph clustering. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 253–264. IEEE, 2016.

[10] L. Chang, W. Li, L. Qin, W. Zhang, and S. Yang. pscan: Fast and exact structural graph clustering. *IEEE Transactions on Knowledge and Data Engineering*, 29(2):387–401, 2017.

[11] Y. Che, S. Sun, and Q. Luo. Parallelizing pruning-based graph structural clustering. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018*, pages 77:1–77:10, 2018.

[12] Y. Chen, R.-H. Li, Q. Dai, Z. Li, S. Qiao, and R. Mao. Incremental structural clustering for dynamic networks. In *International Conference on Web Information Systems Engineering*, pages 123–134. Springer, 2017.

[13] W. Dhifli, S. Aridhi, and E. M. Nguifo. Mr-simlab: Scalable subgraph selection with label similarity for big data. *Information Systems*, 69:155–163, 2017.

[14] I. S. Dhillon. Co-clustering documents and words using bipartite spectral graph partitioning. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 269–274. ACM, 2001.

[15] C. H. Ding, X. He, H. Zha, M. Gu, and H. D. Simon. A min-max cut algorithm for graph partitioning and data clustering. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 107–114. IEEE, 2001.

[16] B. Doerr and D. Johannsen. Adjacency list matchings: an ideal genotype for cycle covers. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1203–1210. ACM, 2007.

[17] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills. Vectorized sparse matrix multiply for compressed row storage format. In *International Conference on Computational Science*, pages 99–106. Springer, 2005.

[18] P. Fournier-Viger, G. He, C. Cheng, J. Li, M. Zhou, J. C.-W. Lin, and U. Yun. A survey of pattern mining in dynamic graphs. *WIREs Data Mining and Knowledge Discovery*, n/a(n/a):e1372.

[19] P. Goyal and E. Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[20] S. Günnemann, B. Boden, and T. Seidl. Finding density-based subspace clusters in graphs with feature vectors. *Data mining and knowledge discovery*, 25(2):243–269, 2012.

[21] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[22] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and E. Nguifo. Un algorithme distribué pour le clustering de grands graphes. In *20ème édition de la conférence francophone" Extraction et gestion des connaissances"*, 2020.

[23] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and E. M. Nguifo. An experimental survey on big data frameworks. *Future Generation Computer Systems*, 86:546–564, 2018.

[24] A. P. Iyer, A. Panda, S. Venkataraman, M. Chowdhury, A. Akella, S. Shenker, and I. Stoica. Bridging the gap: towards approximate graph analytics. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, page 10. ACM, 2018.

[25] S. Ji, C. Bu, L. Li, and X. Wu. Local graph edge partitioning with a two-stage heuristic method. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 228–237. IEEE, 2019.

[26] J. Kim, M. Shin, J. Kim, C. Park, S. Lee, J. Woo, H. Kim, D. Seo, S. Yu, and S. Park. Cass: A distributed network clustering algorithm based on

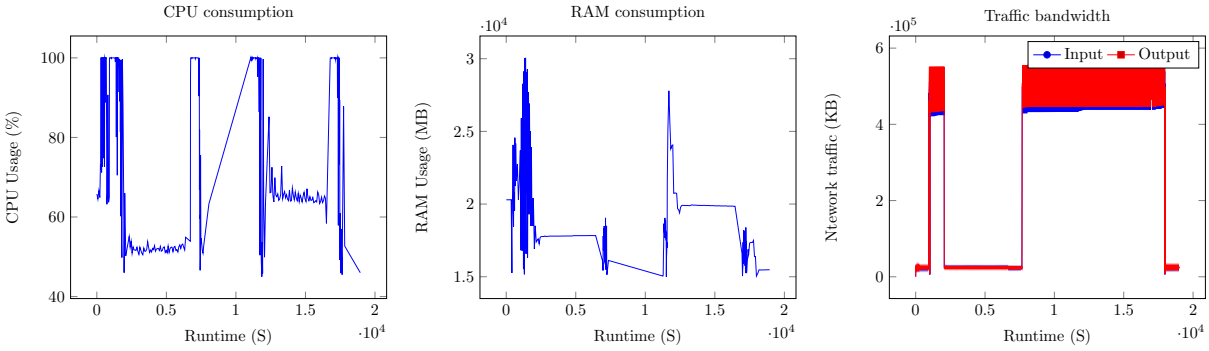


Figure 12: Resources consumption by DISCAN.

structure similarity for large-scale network. *PloS one*, 13(10):e0203670, 2018.

[27] Y. Kozawa, T. Amagasa, and H. Kitagawa. Gpu-accelerated graph clustering via parallel label propagation. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 567–576. ACM, 2017.

[28] D. LaSalle and G. Karypis. Multi-threaded modularity based graph clustering using the multilevel paradigm. *Journal of Parallel and Distributed Computing*, 76:66–80, 2015.

[29] S. Lim, S. Ryu, S. Kwon, K. Jung, and J.-G. Lee. Linkscan*: Overlapping community detection using the link-space transformation. In *2014 IEEE 30th International Conference on Data Engineering (ICDE)*, pages 292–303. IEEE, 2014.

[30] S. T. Mai, M. S. Dieu, I. Assent, J. Jacobsen, J. Kristensen, and M. Birk Scalable and interactive graph clustering algorithm on multicore cpus. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 349–360. IEEE, 2017.

[31] A. Said, R. A. Abbasi, O. Maqbool, A. Daud, and N. R. Aljohani. Cc-ga: A clustering coefficient based genetic algorithm for detecting communities in social networks. *Applied Soft Computing*, 63:59–70, 2018.

[32] H. Schütze, C. D. Manning, and P. Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[33] J. H. Seo and M. H. Kim. pm-scan: an i/o efficient structural clustering algorithm for large-scale graphs. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 2295–2298. ACM, 2017.

[34] H. Shiokawa, Y. Fujiwara, and M. Onizuka. Scan++: efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. *Proceedings of the VLDB Endowment*, 8(11):1178–1189, 2015.

[35] H. Shiokawa and T. Takahashi. Dscan: Distributed structural graph clustering for billion-edge graphs. In *International Conference on Database and Expert Systems Applications*, pages 38–54. Springer, 2020.

[36] T. R. Stovall, S. Kockara, and R. Avci. Gpuscan: Gpu-based parallel structural clustering algorithm for networks. *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3381–3393, 2015.

[37] H. Sun and L. Zanetti. Distributed graph clustering and sparsification. *ACM Trans. on Parallel Computing (TOPC)*, 6(3):1–23, 2019.

[38] S. Sun, W. Li, Y. Wang, W. Liao, and P. Yu. Continuous monitoring of maximum clique over dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[39] T. Takahashi, H. Shiokawa, and H. Kitagawa. Scan-xp: Parallel structural graph clustering algorithm on intel xeon phi coprocessors. In *Proceedings of the 2nd International Workshop on Network Data Analytics*, page 6. ACM, 2017.

[40] D. Wen, L. Qin, Y. Zhang, L. Chang, and X. Lin. Efficient structural graph clustering: an index-based approach. *Proceedings of the VLDB Endowment*, 11(3):243–255, 2017.

[41] T. Weng, X. Zhou, K. Li, P. Peng, and K. Li. Efficient distributed approaches to core maintenance on large dynamic graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[42] S. White and P. Smyth. A spectral clustering approach to finding communities in graphs. In *Proceedings of the 2005 SIAM international conference on data mining*, pages 274–285. SIAM, 2005.

[43] C. Wu, Y. Gu, and G. Yu. Dpscan: Structural graph clustering based on density peaks. In *International Conference on Database Systems for Advanced Applications*, pages 626–641. Springer, 2019.

[44] X. Xu, N. Yuruk, Z. Feng, and T. A. Schweiger. Scan: a structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 824–833. ACM, 2007.

[45] Y. Xu, V. Olman, and D. Xu. Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees. *Bioinformatics*, 18(4):536–545, 2002.

[46] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 555–564. ACM, 2017.

[47] K. R. Žalik and B. Žalik. Memetic algorithm using node entropy and partition entropy for community detection in networks. *Information Sciences*, 445:38–49, 2018.

[48] W. Zhao, G. Chen, and X. Xu. Anyscan: An efficient anytime framework with active learning for large-scale network clustering. In *2017 IEEE International Conference on Data Mining (ICDM)*, pages 665–674. IEEE, 2017.

[49] W. Zhao, V. Martha, and X. Xu. Pscan: a parallel structural clustering algorithm for big networks in mapreduce. In *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, pages 862–869. IEEE, 2013.