MRAttractor: Detecting Communities from Large-Scale Graphs

Nguyen Vo, Kyumin Lee, Thanh Tran
Computer Science Department, Worcester Polytechnic Institute
Worcester, MA, USA
Email: {nkvo,kmlee,tdtran}@wpi.edu

***Abstract*—Detecting groups of users, who have similar opinions, interests, or social behavior, has become an important task for many applications. A recent study showed that dynamic distance based Attractor, a community detection algorithm, outperformed other community detection algorithms such as Spectral clustering, Louvain and Infomap, achieving higher Normalized Mutual Information (NMI) and Adjusted Rand Index (ARI). However, Attractor often takes long time to detect communities, requiring many iterations. To overcome the drawback and handle large-scale graphs, in this paper we propose MRAttractor, an advanced version of Attractor to be runnable on a MapReduce framework. In particular, we (i) apply a sliding window technique to reduce the running time, keeping the same community detection quality; (ii) design and implement the Attractor algorithm for a MapReduce framework; and (iii) evaluate MRAttractor's performance on synthetic and real-world datasets. Experimental results show that our algorithm significantly reduced running time and was able to handle large-scale graphs.**

I. Introduction

A community can be viewed as a group of vertices which are densely connected, when compared to the rest of a network. Detecting organizational groups of these vertices paves the way for understanding the underlying structure of complex networks. As a result, there are numerous algorithms proposed in the last decade such as [1], [2], [3], [4]. Recently, Shao et al. [5] proposed an algorithm called Attractor which utilized the viewpoint of dynamic distance of linked nodes in a graph to find high-quality communities. Instead of optimizing a specific objective, Attractor relied on three types of interactions to dynamically change the distance between vertices. Despite of its superiority over multiple baselines such as Louvain, Ncut and Infomap, this algorithm takes many iterations to converge edge distances, or it even may not converge in some cases [6].

Furthermore, there has been growing interest to process large-scale graphs such as online social networks (e.g., Facebook friendship network and Twitter follower network), and to extract communities. However, most existing community detection algorithms such as Attractor was not be able to handle the large graphs or was designed for a single machine.

In this paper, we were interested in improving Attractor, so that it can handle large-scale graphs, producing high quality communities as quick as possible with ensuring edge distances converged. Initially, we considered to design, improve, and implement Attractor on well-known graph processing frameworks such as Graphx, Pregel and Pegasus [7], [8], [9], [10], [11]. But these graph processing frameworks are not well suited for communication between unconnected nodes [7] because they share a similar vertex-centric paradigm where only connected vertices can directly communicate with each other. Therefore, we decided to design and implement Attractor on top of a well-known MapReduce framework, Hadoop system [12] which can take advantage of distributed computing power.

However, we faced the following key challenges when we began designing our MRAttractor, an advanced version of Attractor for Hadoop: (i) how to compute dynamic interactions in a distributed computing environment, where a partial graph was loaded to each slave node; (ii) how to force edge distances to converge with minimum overhead of network communication and disk I/O, when edge distances are fluctuated over time or convergence takes long time in some datasets [6]; and (iii) how to mitigate the skewness issue of parallel computing (i.e., a task in a slave node takes longer running time than tasks in the other slave nodes). Especially, researchers observed that the original Attractor takes long time in some datasets due to fluctuated edge distances [6]. If we just implement Attractor for Hadoop, we will face the same problem which will cause large overhead in the Hadoop system.

By overcoming and resolving these challenges, in this paper, we propose MRAttractor which consists of three main components: (i) Jaccard distance initialization; (ii) dynamic interaction computation by proposing our graph partition algorithms and applying a sliding window technique; and (iii) community extraction.

Our contributions in this paper are as follows:

- We applied a sliding window technique to ensure edges converged, reduce running time, and still achieve the same quality of extracted communities compared with Attractor.
- We designed and implemented MRAttractor, an improved version of Attractor, to reduce running time and handle large-scale graphs.
- We evaluated performance of MRAttractor in both synthetic and real-life datasets. Our results showed that MRAttractor was able to handle large-scale graphs, and significantly reduced running time.
- We publicly shared our source code and datasets available at http://bit.ly/mrattractor for the research community.

Notations	Meaning
$G = (V, E)$	The undirected graph inputted to MRATTRACTOR
n, m	$n = \|V\|$ and $m = \|E\|$ of the input graph
$d(u, v)$	Distance of edge (u, v)
$\Phi(u)$	u's neighbors, $\Phi(u) = \{v \| v \in V, (u, v) \in E\}$
$\Gamma(u)$	u's neighbors and associated distances $\Gamma(u) = \{(v, d(u, v)) \| u, v \in V, (u, v) \in E\}$
$(u, \Gamma(u))$	The star graph with center vertex u.
$deg(u)$	degree of vertex u, $deg(u) = \|\Phi(u)\| = \|\Gamma(u)\|$
$\langle k; v \rangle$	A key-value pair where k is key and v is value.
$DI(u, v)$, $CI(u, v)$ and $EI(u, v)$	Direct, common and exclusive interaction between linked nodes u and v respectively
$P(\cdot)$	Hash function for graph partition
$\triangle(u, v, c)$	Triangle, three edges $(u, v), (u, c), (v, c) \in E$.
$\wedge(u, v, x)$	Wedge, where $(u, v), (v, x) \in E$, $(u, x) \notin E$.
p	The number of partitions of graph $G(V, E)$.
$\mathbf{w}$	the sliding window vector of an edge (u, v)
λ, τ and γ	Cohesive parameter, threshold of sliding window, upper-bound of non-converged edges respectively.

TABLE I
THE NOTATIONS USED IN THIS PAPER

II. RELATED WORK

Community detection has been studied for a long time to unveil hierarchical structure and hidden modules of complex networks. To detect communities, many different algorithms were proposed [13], [14] categorized into (i) statistical inference based methods [15], (ii) optimization-based methods in which they are often designed to optimize a specific objective such as modularity [3], normalized cut [2] and betweenness [13], and (iii) dynamical processes based algorithms [4]. To complement the existing approaches, [5] recently proposed an algorithm called *Attractor*, which is based on dynamic distance between linked nodes. This algorithm has been investigated and extended in [6], [16]. Despite *Attractor*'s high precision, it was less efficient, requiring many iterations to converge [6].

Other researchers focused on detecting communities from large-scale graphs. One direction was to design and implement algorithms for a MapReduce framework. Tsironis et al. [17] proposed a MapReduce spectral clustering algorithm by employing eigensolver and parallel k-means algorithm [18]. Louvain [19], and community detection algorithms based on Label propagation [20] or propinquity dynamics [21] were developed for Hadoop. Another direction was to design and implement community detection algorithms for other frameworks. For example, [22], [23] developed community detection algorithms based on vertex-centric paradigm of Pregel [8]. In particular, Saltz et al. [22] developed an algorithm to optimize Weighted Community Clustering metric. Ling et al. [23] proposed modularity-based algorithm called FastCD on top of GraphX. Another work [24] employed PMETIS to parallelize the first iteration of Louvain algorithm.

III. BACKGROUND: ATTRACTOR

In this section, we briefly summarize how Attractor [5] works as the background knowledge so that readers can follow how our MRAttractor works in the following section. Table I presents frequently used notations in the rest of this paper.

Attractor consists of three main steps. Firstly, it initializes Jaccard distance of directly linked nodes as follows:

$$d(u, v) = 1 - |N(u) \cap N(v)|/|N(u) \cup N(v)| \quad (1)$$

, where $N(u) = \Phi(u) + \{u\}$ and $N(v) = \Phi(v) + \{v\}$. $\Phi(u)$ and $\Phi(v)$ are u's neighbors and v's neighbors respectively.

Secondly, it dynamically changes edges' distance by computing direct linked interaction (DI), common interaction (CI) and exclusive interaction (EI). These interactions are called Dynamic Interactions. The idea behind dynamic interactions is that the more a pair of vertices interacts with each other, the more their distance is reduced (i.e., they attract each other).

$DI(u, v)$ measures the direct influence of linked nodes and is defined based on $sin()$, the sine function, as follows:

$$DI(u, v) = sin(1 - d(u, v))/deg(u) + sin(1 - d(u, v))/deg(v) \quad (2)$$

$CI(u, v)$ measures influence from common neighbors c of u and v, denoted as $CN(u, v) = \Phi(u) \cap \Phi(v)$. Its main concept is if each $c \in CN(u, v)$ has a small $d(c, u)$ and small $d(c, v)$, u and v will be likely to be in a group.

$$CI(u, v) = \sum_{c \in CN(u,v)} CI_c(u, v) \quad (3)$$

where $CI_c(u, v)$ is equal to following expression:

$$\frac{(1-d(v,c)) \cdot sin(1-d(u,c))}{deg(u)} + \frac{(1-d(u,c)) \cdot sin(1-d(v,c))}{deg(v)} \quad (4)$$

$EI(u, v)$ measures influence from exclusive neighbors. Its main concept is that each exclusive neighbor x of v attracts v to move toward x. If x and u has high similarity, the movement of v to x will reduce $d(u, v)$. Otherwise, the distance will increase. The same concept applies to each exclusive neighbor y of u. EI of u and v is measured as follows:

$$EI(u, v) = \sum_{x \in EN(v)} EI_x(u, v) + \sum_{y \in EN(u)} EI_y(u, v) \quad (5)$$

, where EN(v) and EN(u) are sets of exclusive neighbors of v and u respectively. $EN(v) = \Phi(v) - (\Phi(u) \cap \Phi(v))$ and $EN(u) = \Phi(u) - (\Phi(u) \cap \Phi(v))$. $EI_x(u, v)$ is defined below:

$$EI_x(u, v) = \rho(x, u) \cdot sin(1 - d(v, x))/deg(v) \quad (6)$$

, where $\rho(x, u)$ is influence of vertex x on $d(u, v)$. Given cohesive parameter $\lambda \in [0; 1]$, $\rho(x, u)$ is computed based on $\vartheta(x, u)$, the similarity of unconnected nodes x and u:

$$\rho(x, u) = \begin{cases} \vartheta(x, u), & \text{if } \vartheta(x, u) \geq \lambda \\ \vartheta(x, u) - \lambda, & \text{otherwise} \end{cases} \quad (7)$$

We measure the similarity of x and u, $\vartheta(x, u)$, as follows:

$$\vartheta(x, u) = \frac{\sum_{c \in CN(x,u)} (1-d(x,c)+1-d(u,c))}{\sum_{k \in \Phi(x)} (1-d(x,k)) + \sum_{l \in \Phi(u)} (1-d(u,l))} \quad (8)$$

After computing DI, CI, EI for each edge $(u, v) \in E$, new distance $d(u, v)$ at timestamp $t + 1$ is updated as follows:

$$d^{t+1}(u, v) = d^t(u, v) - DI(u, v) - CI(u, v) - EI(u, v) \quad (9)$$

Attractor algorithm is looped until every edge distance converged (e.g., its distance becomes either 0 or 1). Thirdly, Attractor removes edges with distance 1 and finds connected communities with breath first search. Each connected component is an identified community.

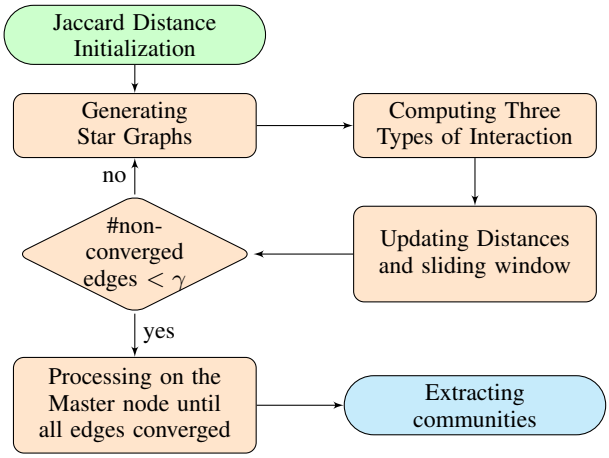


Fig. 1. Flowchart of three major components MRAttractor. (1) Jaccard Distance Initialization, (2) Dynamic Interactions and (3) Extracting communities

IV. MRATTRACTOR

In this section, we describe *MRAttractor*, our proposed distributed version of Attractor. It not only produces the same results with Attractor, but also significantly reduces the running time in both single machine and distributed system.

MRAttractor consists of three main components. The first one is to initialize Jaccard distance. The second component is to compute dynamic interactions and make edge distance converged, which consists of four phases such as generating star graphs, computing three types of interaction, updating distances, and running all edge convergence on the master node. The third component is to extract communities. Figure 1 shows three main components of MRAttractor. We explain each component in detail in the following subsections.

A. Jaccard Distance Initialization

For each vertex u, we find its neighbors $\Phi(u)$ and sort these neighbors increasingly based on their indexes. Then, for each edge (u, v) of graph $G(V, E)$, we can find common neighbors of u and v and compute Jaccard distance (See Eq.1) with complexity O($deg(u) + deg(v)$).

B. Computing Dynamic Interactions

After initializing all edge distances of $G(V, E)$, we move on to the second major component of MRATTRACTOR which consists of three MapReduce phases (i.e., generating star graphs, computing three types of Interactions, and updating distances based on sliding window), and running on the master node to make all edge distances converge.

Algorithm 1 MR1: Generating Star Graphs

Map: Input $\langle (u, v); d(u, v) \rangle$
1: emit $\langle u; v\ d(u, v) \rangle$; emit $\langle v; u\ d(u, v) \rangle$
Reduce: Input $\langle u; \Gamma(u) \rangle$
2: Sort $\Gamma(u)$ increasingly based on index of u's neighbors.
3: emit $\langle u;\ Sorted(\Gamma(u)) \rangle$

1) Generating Star Graphs: Algorithm 1 processes each edge (u, v) and its distance. Then, in reduce step, we sort $\Gamma(u)$ based on index of u's neighbors and output its star graph $\langle u; \Gamma(u) \rangle$. Note that a star graph is a tree of k nodes where center vertex has degree $k-1$, while other vertices have degree 1. Sorting helps us find common and exclusive neighbors of two linked nodes in linear time. Totally, there will be $n = |V|$ star graphs output from reduce instances of Algorithm 1.

2) Computing Three Types of Interactions: Direct Interaction (DI), Common Interaction (CI) and Exclusive Interaction (EI) are three interactions we need to compute. The hardest task is computing $EI(u, v)$ of edge (u, v) because $EI(u, v)$ depends on $\vartheta(x, u)$, the similarity of unconnected nodes x and u where $x \in EN(v)$ (see Eq.6 and Eq.7). Well-known large-scale graph processing frameworks [7], [8], [10] limitedly support direct communication between two unconnected nodes (e.g. vertex x and vertex u), leading to difficulties in computing $EI(u, v)$. Therefore, we propose DECGP, an algorithm to efficiently compute dynamic interactions of every edge (u, v) in $G(V, E)$. Our proposed DECGP algorithm was inspired by a graph partitioning algorithm introduced in [25], which showed that a graph partition algorithm helped mitigate skewness issue by evenly distributing workload to each reducer. The third challenge mentioned in Section I will be resolved by our graph partition algorithm.

Our graph partition algorithm uses a hash function $P(.)$, that maps each vertex to range $[0, p-1]$, to partition original graph $G(V, E)$ into p disjoint partitions $V_1, V_2, ..., V_p$ such that $V = V_1 \cup V_2 \cup ... \cup V_p$, and $V_i \cap V_j = \emptyset$. From these partitions, we form overlapping subgraphs $G_{ijk} = (V_{ijk}, E_{ijk})$ where $V_{ijk} = V_i \cup V_j \cup V_k$, $E_{ijk} = \{(u, v) \in E | u, v \in V_{ijk}\}$. An edge $(u, v) \in E$ is called an outer-edge if u and v are in different partition (i.e., $P(u) \neq P(v)$). Otherwise, it is an inner-edge. Intuitively, in each smaller-size subgraph G_{ijk}, we compute DI, CI and EI of edges $(u, v) \in E_{ijk}$. Let's take an example graph $G(V, E)$ of 12 vertices and 16 edges in Figure 2. By using hashing function $P(u) = u \mod p$ where $p = 4$, this graph is partitioned into 4 subgraphs G_{012}, G_{013}, G_{023} and G_{123}. The value on each edge is Jaccard distance. Although our algorithm shares similar methodology with [25], there are key differences as follows:

- We reduce complexity of finding subgraphs G_{ijk} that contain an edge (u, v) from O(p^3) to O(p^2).
- In our algorithm, we also include additional edges called **rear edges** in subgraphs G_{ijk} to compute exclusive interaction while in [25], subgraphs G_{ijk} only contain **main edges**. In each subgraph G_{ijk}, an edge (u, v) is called a main edge if $\{P(u), P(v)\} \in \{i, j, k\}$ otherwise it is called a rear-edge. In Figure 2, subgraphs G_{012}, G_{013}, G_{023} and G_{123} contain rear-edges denoted as dotted edges and main edges denoted as solid edges.
- In each subgraph G_{ijk}, we compute partial values of $DI(u, v)$, $CI(u, v)$ and $EI(u, v)$ of every main edges (u, v) instead of counting the number of triangles like [25].

These differences are described as follows:

(i) Reducing complexity of finding subgraphs G_{ijk}.

Given an edge (u, v), a graph partition algorithm of Siddharth et al. [25] takes $O(p^3)$ for finding subgraphs G_{ijk} that

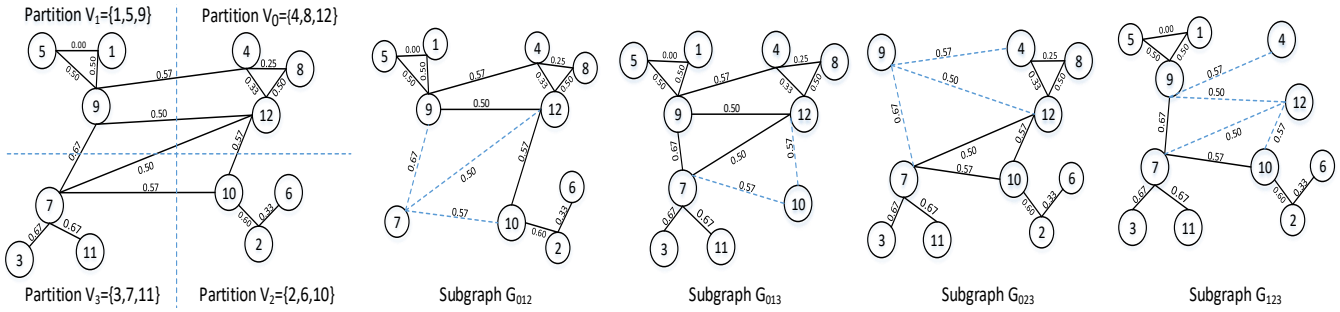


Fig. 2. Original graph $G(V, E)$ where $|V| = 12$, $|E| = 16$. Each edge is associated with Jaccard distance in Equation 1. This graph is partitioned into four subgraph G_{012}, G_{013}, G_{023} and G_{123}. The hash function $P(u) = u \mod p$ where $p = 4$. The solid lines are main edges and the dot lines are rear edges.

(u, v) belongs to. However, our proposed algorithm only takes $O(p^2)$ for the task (see Lemma IV.1 and associated proof).

Lemma IV.1. *For each edge (u, v) in original graph $G(V, E)$:*

1) *If edge (u, v) is an inner edge, there will be $\frac{(p-1)\cdot(p-2)}{2}$ distinct subgraphs G_{ijk} containing it.*
2) *If edge (u, v) is an outer edge, there will be $p - 2$ subgraphs G_{ijk} containing this edge.*

Proof. (1) If edge (u, v) is an inner edge (e.g., $P(u) = P(v) = i$), it will be emitted to G_{ijk} for each $j \in [0; p - 1]$, $k \in [0; p-1]$ and $j \neq i$, $i \neq k$, $k \neq j$. Therefore, there are $\frac{(p-1)\cdot(p-2)}{2}$ subgraphs G_{ijk} containing the inner edge (u, v).

(2) If edge (u, v) is an outer edge (e.g., $P(u) = i, P(v) = j$), it will be output to G_{ijk} where $i = P(u)$ and $j = P(v)$ for each $k \in [0; p - 1], k \neq i$ and $k \neq j$. So, there are $p - 2$ subgraphs G_{ijk} containing this edge. □

Due to Lemma IV.1, finding subgraphs G_{ijk} that contain an edge (u, v) can be implemented in quadric complexity as shown in Algorithm 2.

Algorithm 2 Finding G_{ijk} containing an edge (u, v)

```
function FindSubGraphs(u, v)
    S = ∅
    if P(u) = P(v) then
        for a ∈ [0; p − 1] and a ≠ P(u) do
            for b ∈ [a + 1; p − 1] and b ≠ P(u) do
                S = S ∪ {sorted(a, b, P(u))}
    else
        for a ∈ [0; p − 1] and a ≠ P(u) and a ≠ P(v) do
            S = S ∪ {sorted(a, P(u), P(v))}
    return S
```

(ii) Adding rear-edges to subgraphs G_{ijk}. What is the motivation of adding rear-edges?

Let's consider only main edges (u, v) of subgraphs G_{ijk}. Once again, an edge (u, v) in subgraph G_{ijk} is a main edge if $\{P(u), P(v)\} \in \{i, j, k\}$. Otherwise, it is a rear-edge. For each main edge (u, v) associated with its distance in each subgraph G_{ijk}, we can load vertices' degree into memory, and compute $DI(u, v)$ based on Eq.2 and $CI_c(u, v)$ for every common vertex c based on Eq.4. But how can we compute $EI(u, v)$? Let's look at a main edge (12, 10) of subgraph G_{012} in Figure 2 as an example. We can see that vertex 9 is an exclusive neighbor of vertex 12. $EI(10, 12)$ depends on $EI_9(10, 12)$ (See Equation 6). To compute $EI_9(10, 12)$, we need to measure similarity $\vartheta(9, 10)$ in Eq.8 and $\rho(9, 10)$ in Eq.7. To accurately compute $\vartheta(9, 10)$, we need both star graph $(9, \Gamma(9))$ and star graph $(10, \Gamma(10))$ to find common neighbors. But in subgraph G_{012}, without considering rear-edges, we are missing necessary edges $(9, 7)$ and $(10, 7)$ to compute $\vartheta(9, 10)$ since $P(7) = 3 \notin \{0, 1, 2\}$. Similar analysis to vertex 2, an exclusive neighbor of vertex 10, we are missing edge $(12, 7)$ to compute $\vartheta(12, 2)$. Motivated by these observations and to guarantee the correctness of MRAttractor, we added rear edges $(9, 7)$, $(10, 7)$ and $(12, 7)$, denoted as dotted edges, to subgraph G_{012} to correctly compute exclusive interactions. We resolved the first challenge mentioned in Section I by adding rear-edges.

Algorithm 3 MR2: DECGP - Computing 3 Types Interactions

```
Map: Input: ⟨u; Γ(u)⟩
S_T = ∅                                   ▷ Set of triples (i, j, k)
for each (v, d(u, v)) ∈ Γ(u) do
    S_T = S_T ∪ FindSubGraphs((u, v))
for each (i, j, k) ∈ S_T do
    emit ⟨(i, j, k); (u, Γ(u))⟩
Reduce: Input: ⟨(i, j, k); {(u, Γ(u))|P(u) ∈ {i, j, k}}⟩
Read value p, λ from Hadoop configuration object.
stars = ∅                                 ▷ An empty dictionary of star graphs
S_M = ∅                                   ▷ Set of main edges.
for each (u, Γ(u)) routed by key (i, j, k) do
    stars[u] = Γ(u)                       ▷ Insert to dictionary
    for each (v, d(u, v)) ∈ Γ(u) do
        if (P(u), P(v)) ∈ {i, j, k} then ▷ (u, v) is a main edge
            S_M = S_M ∪ {sorted(u, v), d(u, v)}
for each (u, v, d(u, v)) ∈ S_M and 0 < d(u, v) < 1 do
    S_I = 0
    S_I+ =ComputeDI(u, v, d(u, v), p)
    S_I+ =ComputeCI(u, v, d(u, v), Γ(u), Γ(v), p, {i, j, k})
    for (x, d(v, x)) ∈ Γ(v) and x ∈ EN(v) do
        if P(x) ∈ {i, j, k} then
            Γ(x) = stars[x]               ▷ Retrieve neighbors of vertex x
            S_I+ =ComputeEI(∧(u, v, x), Γ(u), Γ(x), p, λ)
    for (y, d(u, y)) ∈ Γ(u) and y ∈ EN(u) do
        if P(y) ∈ {i, j, k} then
            Γ(y) = stars[y]               ▷ Retrieve neighbors of vertex y
            S_I+ =ComputeEI(∧(v, u, y), Γ(v), Γ(y), p, λ)
    emit ⟨(u, v); S_I⟩
```

(iii) Computing DI, CI and EI of main edges. By adding rear edges to G_{ijk}, we are now able to compute DI, CI and EI of main edges in subgraph G_{ijk}. Algorithm 3 shows the

pseudocode of DECGP to compute DI,CI and EI of every main edge in G_{ijk}. It is a MapReduce algorithm consisting of a map function and a reduce function. In particular, each map instance handles a star graph $(u, \Gamma(u))$ output from Algorithm 1. It firstly finds distinct subgraphs G_{ijk}, represented as a sorted triple (i, j, k), which contains at least one of the edges $(u, v, d(u, v))$ where $(v, d(u, v)) \in \Gamma(u)$ by applying function FINDSUBGRAPHS. Secondly, it emits star graph $(u, \Gamma(u))$ to subgraph G_{ijk} (See lines [2-5] of DECGP). By emitting the whole star graph $(u, \Gamma(u))$ to subgraph G_{ijk}, we ensure that there are always enough edges to compute similarity of unconnected nodes.

Moving on to each reduce instance of Algorithm 3. Each reduce instance receives a list of star graph $(u, \Gamma(u))$ routed by the sorted key (i, j, k) of subgraph G_{ijk}, which can be viewed as an adjacent list representation of subgraph G_{ijk}. From lines [8-13] in Algorithm 3, we find set S_M of main edges $(u, v, d(u, v)) \in E_{ijk}$, which are used for computing DI, CI and EI in Lines [14-26]. After computing DI, CI and EI of every non-converged main edge in G_{ijk}, each reduce instance will emit key-value pairs where key is a main edge (u, v) and value is the aggregated DI, CI and EI of edge (u, v). Three main functions COMPUTEDI, COMPUTECI and COMPUTEEI are explained below.

Computing DI. Algorithm 4 shows how we compute Direct Interaction. $DI(u, v)$ of each main edge (u, v) is computed based on Eq.2. However, this edge (u, v) will be re-computed in multiple subgraphs G_{ijk} based on Lemma IV.1 so we need to scale $DI(u, v)$ down to guarantee correctness. In particular, if (u, v) is an inner edge, we scale $DI(u, v)$ down $\frac{(p-1)\cdot(p-2)}{2}$ times. Otherwise, we scale it down $p - 2$ times.

Algorithm 4 Direct interaction of the edge (u, v)

```
1: function ComputeDI(u, v, d(u, v), p)
2:     DI = sin(1−d(u,v))/deg(u) + sin(1−d(u,v))/deg(v)
3:     if P(u) = P(v) then
4:         DI = 2 · DI/((p − 1) · (p − 2))
5:     else
6:         DI = DI/(p − 2)
       return DI
```

Computing CI. Algorithm 5 shows how we compute common interaction $CI(u, v)$ of main edge (u, v). The input of this function is the main edge (u, v), its distance $d(u, v)$, u's neighbors $\Gamma(u)$, v's neighbors $\Gamma(v)$, the number of partitions p and key of subgraph G_{ijk}. For each common neighbor c of u and v, we only consider vertex $P(c) \in \{i, j, k\}$ because we only care about main edges of G_{ijk}. We compute $CI_c(u, v)$ based on Eq.4. However, we need to scale down $CI_c(u, v)$ since three vertices u,v and c will form a triangle $\triangle(u, v, c)$ and Siddharth et al., [25] pointed out that $\triangle(u, v, c)$ can be repeated in several subgraphs G_{ijk} in Lemma IV.2. In particular, if three vertices of $\triangle(u, v, c)$ are in a same partition V_i, we scale $CI_c(u, v)$ down $\frac{(p-1)\cdot(p-2)}{2}$ times. If $\triangle(u, v, c)$ has two nodes in same partition and the third node belongs to a different partition, $CI_c(u, v)$ is scaled down $p - 2$ times.

Algorithm 5 Common interaction of the edge (u, v)

```
1: function ComputeCI(u, v, d(u, v), Γ(u), Γ(v), p, {i, j, k})
2:     CI = 0
3:     for c ∈ CN(u, v) and P(c) ∈ {i, j, k} and (c, d(u, c)) ∈
       Γ(u) and (c, d(v, c)) ∈ Γ(v) do
4:         w1 = 1 − d(u, c) ; w2 = 1 − d(v, c)
5:         CIc(u, v) = w2·sin(w1)/deg(u) + w1·sin(w2)/deg(v)
6:         if P(u) = P(v) = P(c) then
7:             CIc(u, v) = 2 · CIc(u, v)/((p − 1) · (p − 2))
8:         else if P(u) = P(v) | P(u) = P(c) | P(v) = P(c) then
9:             CIc(u, v) = CIc(u, v)/(p − 2)
10:        CI = CI + CIc(u, v)
11:    return CI
```

Lemma IV.2. *Given $\triangle(u, v, c)$ of original graph $G(V, E)$:*

1) *if three nodes are in a same partition, $\triangle(u, v, c)$ will also appear in $\frac{(p-1)\cdot(p-2)}{2}$ different subgraphs G_{ijk}.*
2) *if two nodes are in a same partition and one node belongs to a different partition, $\triangle(u, v, c)$ will appear in $p - 2$ different subgraphs G_{ijk}.*
3) *if three nodes are in three different partition, there is only one subgraph containing $\triangle(u, v, c)$.*

Proof. (1) When u, v and c are in a same partition $P(u) = P(v) = P(c) = i$, $\triangle(u, v, c)$ will appear in subgraphs G_{ijk} where $j \in [0; p - 1]$, $k \in [0; p - 1]$ and $j \neq k, k \neq i, i \neq j$. Totally, there are $\frac{(p-1)\cdot(p-2)}{2}$ different subgraphs G_{ijk}.

(2) Without loss of generality, we assume that $P(u) = P(v) = i$ and $P(c) = j$. $\triangle(u, v, c)$ will appear in subgraphs G_{ijk} where $k \in [0; p - 1]$ and $k \neq i$ and $k \neq j$. Therefore, there are $p - 2$ distinct subgraphs G_{ijk} containing $\triangle(u, v, c)$.

(3) Finally, when three nodes are in three different partitions $P(u) = i$, $P(v) = j$ and $P(c) = k$, there is only one subgraph G_{ijk} that contains this $\triangle(u, v, c)$. □

Algorithm 6 Exclusive interaction of vertex x on edge (u, v)

```
1: function ComputeEI(∧(u, v, x), Γ(u), Γ(x), p, λ)
2:     EIx(u, v) = 0
3:     Compute ϑ(x, u) based on Γ(u) and Γ(x)
4:     ρ(x, u) = ϑ(x, u)
5:     if ϑ(x, u) < λ then
6:         ρ(x, u) = ϑ(x, u) − λ
7:     EIx(u, v) = ρ(x, u) · sin(1 − d(v, x))/deg(v)
8:     if P(u) = P(v) = P(x) then
9:         EIx(u, v) = 2 · EIx(u, v)/((p − 1) · (p − 2))
10:    else if P(u) = P(v) | P(u) = P(x) | P(v) = P(x) then
11:        EIx(u, v) = EIx(u, v)/(p − 2)
12:    return EIx(u, v)
```

Computing EI. For each main edge (u, v) of G_{ijk}, we sequentially process each exclusive neighbor $x \in EN(v)$ (See Lines [18-21] of Algorithm 3) and each exclusive neighbor $y \in EN(u)$ (See Lines [22-25] of Algorithm 3). Once again, we only pay attention to vertex x and vertex y such that $P(x) \in \{i, j, k\}$ and $P(y) \in \{i, j, k\}$ since main edges are our target.

Before describing details of the computation, we present definition of a wedge or a two-hop path in Definition 1.

Definition 1 (Wedge)**.**
In $G(V, E)$, three nodes u, v and x form a wedge denoted as $\wedge(u, v, x)$ if $(u, v) \in E, (v, x) \in E$ and there is no edge between u and x. A wedge can be called two-hop path [26].

As we can see, three nodes u, v and x form a $\wedge(u, v, x)$. Similarly, three nodes v, u and y create $\wedge(v, u, y)$. Without loss of generality, we only explain how we compute $EI_x(u, v)$, the effect of exclusive neighbor $x \in EN(v)$ on distance of edge (u, v) based on $\wedge(u, v, x)$.

In function COMPUTEEI of Algorithm 6, we input $\wedge(u, v, x)$, u's neighbors, x's neighbors, the number of partitions p and cohesive parameters λ [5]. To begin with, we compute $\vartheta(x, u)$ (See Eq.8), the similarity of vertex u and x based on $\Gamma(u)$ and $\Gamma(x)$. Then, we derive $\rho(x, u)$ and compute $EI_x(u, v)$ in Eq.6. Computing $EI_x(u, v)$ in each subgraph G_{ijk} will face duplication problem since wedge $\wedge(u, v, x)$ can appear in other different subgraphs G_{ijk}. Therefore, we need to scale down $EI_x(u, v)$ appropriately. Lemma IV.3 shows the number of subgraphs G_{ijk} that a wedge $\wedge(u, v, x)$ can appear. In particular, if $\wedge(u, v, x)$ has three nodes in a same partition, we scale $EI_x(u, v)$ down $\frac{(p-1)\cdot(p-2)}{2}$ times. If the first two nodes in $\wedge(u, v, x)$ are in the same partition, but the third node is in another partition, we scale $EI_x(u, v)$ down $p - 2$ times (see Lines [8-11] of Algorithm 6).

Lemma IV.3. *For each wedge $\wedge(u, v, x)$:*

1) *If three nodes u, v and x are placed in the same partition, it will appear in $\frac{(p-1)\cdot(p-2)}{2}$ different subgraphs G_{ijk}.*
2) *If two nodes are in the same partition and the other one belongs to a different partition, it will appear in $p - 2$ different subgraphs G_{ijk}.*
3) *If each vertex is in different partitions, it will belong to only one subgraph G_{ijk}.*

Proof. (1) When three nodes are in a same partition, $P(u) = P(v) = P(w) = i$. Edges (u, v) and (v, x) are two inner edges and always appear together. Due to Lemma IV.1, inner edges will appear in $\frac{(p-1)\cdot(p-2)}{2}$ subgraphs G_{ijk}. Therefore, $\wedge(u, v, x)$ will appear in $\frac{(p-1)\cdot(p-2)}{2}$ different subgraphs G_{ijk}.

(2) Without loss of generality, we assume that $P(u) = P(v) = i$ and $P(x) = j$. For each subgraph G_{ijk} that outer edge (v, x) appears, we can see that inner edge (u, v) also appears, leading to the existence of $\wedge(u, v, x)$ in G_{ijk}. We know that the outer edge (v, x) will appear in $p - 2$ different subgraphs G_{ijk} based on Lemma IV.1. Therefore, $\wedge(u, v, x)$ will appear in $p - 2$ different subgraphs G_{ijk}.

(3) If each vertex are in different partitions, $P(u) = i$, $P(v) = j$, $P(x) = k$. Thus, only one G_{ijk} has this wedge. □

3) Updating edge distances based on sliding window: Next, we move on to updating distance of every edge (u, v) and its sliding window in original graph $G(V, E)$ based on $DI(u, v)$, $CI(u, v)$ and $EI(u, v)$ computed by Algorithm 3.

In Section I, we addressed three key challenges to design and implement MRAttractor. The second challenge was how to make all edge distances converged with minimum overhead of network communication and disk I/O. To overcome the second challenge, we use sliding window technique [27].

Algorithm 7 MR3: Updating Distances and Sliding Window

Map: Input: $\langle (u, v); S_I \rangle$, $\langle (u, v); d(u, v) \rangle$ and $\langle (u, v); \mathbf{w} \rangle$
emit $\langle (u, v); vl \rangle$ with vl is either S_I, $d(u, v)$ or $\mathbf{w}$
Reduce: Input: $\langle (u, v); values \rangle$
Read settings s the maximum size of sliding window and τ
$d^t(u, v) = 0$; $\Delta^t(u, v) = 0$; $\mathbf{w}^{t+1} = \emptyset$
for $vl \in values$ **do**
  if vl **is w then**
    $\mathbf{w}^{t+1} = \mathbf{w}$
  else if vl is $d(u, v)$ **then**
    $d^t(u, v) \leftarrow d(u, v)$
  else if vl is S_I **then**
    $\Delta^t(u, v) \leftarrow \Delta^t(u, v) + S_I$
if $d^t(u, v) = 0$ **or** $d^t(u, v) = 1$ **then**
  return ▷ This edge was converged. No need to process it
if $\Delta^t(u, v) \neq 0$ **then**
  $d^{t+1}(u, v) = d^t(u, v) - \Delta^t(u, v)$
  $index = (t + 1) \mod s$
  $\mathbf{w}^{t+1}[index] = -1$ ▷ Set position $index$ of vector $\mathbf{w}^{t+1}$
  if $d^{t+1}(u, v) > d^t(u, v)$ **then**
    $\mathbf{w}^{t+1}[index] = 1$
  x is the number of 1 in sliding window $\mathbf{w}^{t+1}$
  y is the number of -1 in sliding window $\mathbf{w}^{t+1}$
  if $t + 1 \geq s$ **then** ▷ $\mathbf{w}^{t+1}$ of edge (u, v) is full
    if $\mathbf{w}^{t+1}[index] = 1$ **and** $x \geq \tau \cdot s$ **then**
      $d^{t+1}(u, v) = 1$
    if $\mathbf{w}^{t+1}[index] = -1$ **and** $y \geq \tau \cdot s$ **then**
      $d^{t+1}(u, v) = 0$
  if $d^{t+1}(u, v) \geq 1$ **then**
    $d^{t+1}(u, v) = 1$
  if $d^{t+1}(u, v) \leq 0$ **then**
    $d^{t+1}(u, v) = 0$
  emit $\langle (u, v); d^{t+1}(u, v) \rangle$
  emit $\langle (u, v); \mathbf{w}^{t+1} \rangle$

Our sliding window model works as follows: For each edge (u, v), we use a binary vector $\mathbf{w}$, indicating the status of edge (u, v), $\mathbf{w}_i = 1$ means that $d(u, v)$ at iteration i^{th} increases and $\mathbf{w}_i = -1$ means $d(u, v)$ decreases. By using sliding window, we only keep the last s statuses of each edge (i.e. $\mathbf{w} \in \mathbf{R}^s$) to observe the increasing/decreasing trend of the edge (u, v) in the last s iterations which may be more reliable to reflect the convergence trend of an edge. Then, we predict its final distance (i.e. 0 or 1). To decide if an edge converges or not, we use a threshold $\tau \in [0; 1]$. In particular, if the last status of edge (u, v) is -1 and there are at least $\tau \times s$ negative values in vector $\mathbf{w}$, we decide that edge will eventually converge to 0 (e.g., when s=10, τ=0.6 and last status=-1, if at least 6 statuses in $\mathbf{w}$ are -1, edge distance will be set to 0). If the last status of edge (u, v) is 1 and there are at least $\tau \times s$ positive values in vector $\mathbf{w}$, we will set $d(u, v) = 1$. Otherwise, edge (u, v) still does not converge, and we continue to compute its dynamic interactions as shown in Figure 1.

Algorithm 7 shows our pseudocode to update edge distances. The map instances of Algorithm 7 process three types of input: (1) $\langle (u, v); S_I \rangle$ is the key-value pairs generated Dynamic Interactions. Recall that S_I is the aggregated sum

of DI, CI and EI of edge (u, v) output from reduce instances of Algorithm 3. (2) $\langle (u, v); d(u, v) \rangle$ is the edge (u, v) and its distance in previous iteration. At the first iteration, $d(u, v)$ is Jaccard distance. (3) $\langle (u, v); \mathbf{w} \rangle$ is sliding window of edge (u, v). At the first iteration, $\mathbf{w}$ is an empty vector. Note that, for each edge (u, v), there are only one pair $\langle (u, v); d(u, v) \rangle$, one pair $\langle (u, v); \mathbf{w} \rangle$ and multiple pairs $\langle (u, v); S_I \rangle$.

Each map instance of Algorithm 7 simply outputs a key-value pair where key is an edge (u, v) and value is either S_I, $d(u, v)$ or sliding window vector $\mathbf{w}$.

Each reduce instance of Algorithm 7 receives $values$ routed by key (u, v) and performs two tasks with edge (u, v)- (1) computing its new distance and (2) updating its sliding window vector. To begin with, from Lines [3-10], we sum up all S_I values and store it into $\Delta^t(u, v)$. We can verify that $\Delta^t(u, v) = DI(u, v) + CI(u, v) + EI(u, v)$. After computing $\Delta^t(u, v)$, we can derive $d^{t+1}(u, v)$, distance of edge (u, v) at timestamp $t + 1$ based on Eq.9. Next, from Lines [15-25], we update sliding window vector $\mathbf{w}^{t+1}$. Due to modulo function, $index \in [0; s - 1]$. Finally, we emit pair $\langle (u, v); d^{t+1}(u, v) \rangle$ for new distance of edge (u, v) and pair $\langle (u, v); \mathbf{w}^{t+1} \rangle$ for new sliding window vector. These key-value pairs will act as new input for next iteration of MRATTRACTOR.

4) Running on Master node: After deriving new distance of all edges in original graph $G(V, E)$, we will check how many edges (u, v) are still non-converged $0 < d^{t+1}(u, v) < 1$. If the number of non-converged edges are smaller than a threshold γ we will continue our computation on Master node, which control slave nodes. There are two reasons why we do this. Firstly, Attractor algorithm suffers long-tail iterations because some edges converge slowly [6]. Secondly, after multiple iterations, the number of non-converged edges are very small which can be handle efficiently on single computer. A well-known problem of MapReduce Hadoop is the overhead of I/O operations [28]. Therefore, by running on single computer, we can avoid unnecessary overhead of Hadoop framework. In this work, we set $\gamma = 10,000$ for all testing networks. The second challenge mentioned in Section I is resolved by the sliding window model and running on the master node.

5) Complexity Analysis: We now show the correctness of computing dynamic interactions and analysis of DECGP's complexity since it is the most time consuming part.

Lemma IV.4. *For each edge (u, v) of $G(V, E)$, its $DI(u, v)$, $CI(u, v)$ and $EI(u, v)$ are computed correctly in each loop.*

Proof. In each G_{ijk}, we compute partial values of DI, CI and EI for every main edge with appropriate scaling as shown in Algorithms 3, 4, 5 and 6. After computing dynamic interactions, we aggregate DI, CI and EI for every edge of the original graph $G(V, E)$ in reduce instances of Algorithm 7. Since we apply scaling correctly, the aggregated values $\Delta^{t+1}(u, v)$ of every edge is exactly equal to $DI(u, v) + CI(u, v) + EI(u, v)$, leading to correctness of our computation. □

Lemma IV.5. *For each setting of p:*

1) The expected number of main edges in G_{ijk} is $O(\frac{m}{p^2})$.

Datasets	|V|	|E|	|classes|	AVD	CC
Karate	34	78	2	4.588	0.571
Football	115	613	12	10.661	0.403
Polbooks	105	441	3	8.400	0.488
Amazon	334,863	925,872	top5000	5.530	0.397
Collaboration	9,875	25,973	unknown	5.260	0.472
Friendship	58,228	214,078	unknown	7.353	0.172
Road	1,088,092	1,541,898	unknown	2.834	0.046

TABLE II
NETWORKS WITH LABELS AND NON-LABELS, AVERAGE DEGREE (AVD), AND AVERAGE CLUSTERING COEFFICIENT (CC).

2) The expected number of key-value pairs $\langle (u, v); S_I \rangle$ is $O(mp)$ for all reduce instances of Algorithm 3.

Proof. (1) The probability that $P(u) \in \{i, j, k\}$ is $O(\frac{3}{p})$. An edge (u, v) in G_{ijk} is a main edge if $P(u) \in \{i, j, k\}$ and $P(v) \in \{i, j, k\}$. Therefore, the likelihood that an edge appears between u and v is $\frac{9}{p^2}$, resulting in the expected number of main edges of G_{ijk} is $O(\frac{m}{p^2})$, where m is total number of edges in the original graph $G(V, E)$.

(2) For each main edge (u, v) of G_{ijk}, after computing partial value of DI, CI and EI, we emit one key-value pair $\langle (u, v); S_I \rangle$. Since the number of main edges is $O(\frac{m}{p^2})$ and the number of subgraphs G_{ijk} is $O(p^3)$, the expected number of key-value pairs $\langle (u, v); S_I \rangle$ is $O(p^3 \cdot \frac{m}{p^2}) = O(mp)$. □

Based on Lemma IV.5, we approximately estimate complexity of computing dynamic interactions in each subgraph G_{ijk}. For each main edge (u, v) of G_{ijk}, time complexity of computing direct interaction is O(1). Time complexity to compute common interaction is O($deg(u) + deg(v)$). Time complexity of computing exclusive interaction is about O($T \cdot (deg(u) + deg(v))$) where T is average number of exclusive neighbors of each node. Therefore, for all main edges of G_{ijk}, the total complexity is about $T \cdot \sum_{(u,v) \in E_{ijk}} (deg(u) + deg(v) \leq T \cdot \sum_{u \in V_{ijk}} deg^2(u)$. It has been proved in [29] that $\sum_{u \in V} deg^2(u) \leq m\big(\frac{2m}{n-1} + n - 2\big)$ for the original graph $G(V, E)$. Then, what is the upper bound of $\sum_{u \in V_{ijk}} deg^2(u)$ for the subgraph G_{ijk} which is smaller than $G(V, E)$? The expected number of vertices of G_{ijk} is $O(n/p)$, so we can roughly estimate the upper bound of $T \cdot \sum_{u \in V_{ijk}} deg^2(u)$ as $O\big(\frac{mnT}{p}\big(\frac{2m}{n-1} + n - 2\big)\big)$.

C. Extracting communities.

When each of all edge distances converges to either 0 or 1, we remove edges with distance equal to 1. Then, we find communities as connected components by running the breath first search on the master node.

V. EXPERIMENTS

In this section, we evaluate performance of our sliding window model on real-world datasets. Then, we evaluate performance of MRAttractor on both synthetic and real-world datasets. Our experiments mainly focus on evaluating the efficiency because Attractor [5] outperformed other community detection algorithms such as Spectral clustering, Louvain and Infomap, achieving higher NMI and ARI [30].

	Karate				Football				Polbooks				Amazon			
	Purity	NMI	ARI	#iters	Purity	NMI	ARI	#iters	Purity	NMI	ARI	#iters	Purity	NMI	ARI	#iters
Attractor	1.000	0.924	0.939	13	0.930	0.924	0.888	9	0.857	0.589	0.680	16	0.978	0.960	0.580	62
IAttractor	0.529	0.000	0.000	6	0.783	0.638	0.846	7	0.467	0.000	0.000	7	0.716	0.846	0.033	8
[0.5-10]	1.000	0.924	0.939	11	0.930	0.924	0.888	9	0.857	0.589	0.680	13	0.978	0.960	0.580	18 ↓70.8%
[0.7-10]	1.000	0.924	0.939	12	0.930	0.924	0.888	9	0.857	0.589	0.680	15	0.978	0.960	0.580	25 ↓67.0%

TABLE III
PERFORMANCE OF ATTRACTOR AND SLIDING WINDOW ON GRAPHS WITH LABELS.

	Friendship				Amazon				Collaboration				Road			
	modul	ncut	#coms	#iters	modul	ncut	#coms	#iters	modul	ncut	#coms	#iters	modul	ncut	#coms	#iters
Attractor	0.421	0.607	8044	323	0.741	0.398	23822	62	0.337	0.159	785	43	0.865	0.264	56967	37
0.5-10	0.347	0.606	7939	19 ↓35%	0.741	0.398	23800	18 ↓71%	0.337	0.158	784	17 ↓10%	0.865	0.264	56952	18 ↓16%
0.5-15	0.424	0.606	8022	23 ↓28%	0.741	0.398	23820	23 ↓68%	0.337	0.158	784	23 ↓3.5%	0.865	0.264	56966	22 ↓14%
0.7-10	0.416	0.606	7991	28 ↓36%	0.741	0.398	23802	25 ↓69%	0.337	0.158	784	21 ↓9.4%	0.865	0.264	56952	21 ↓16%
0.7-15	0.424	0.607	8034	35 ↓23%	0.741	0.398	23821	31 ↓67%	0.337	0.158	784	27 ↓3.3%	0.865	0.264	56966	25 ↓12%

TABLE IV
PERFORMANCE OF ATTRACTOR AND SLIDING WINDOW ON GRAPHS WITHOUT LABELS.

Datasets	|V|	|E|	|classes|	AVD	CC
1M	20,000	1,000,310	191	100.031	0.699
2M	40,000	1,994,815	387	99.741	0.692
4M	80,000	3,996,488	761	99.912	0.707
6M	120,000	5,994,313	1137	99.905	0.700
8M	160,000	7,977,113	1512	99.713	0.697

TABLE V
SYNTHETIC NETWORKS

Datasets	|V|	|E|	|classes|	AVD	CC
DBLP	317,080	1,049,866	top5000	6.622	0.632
Texas Road	1,379,917	1,921,660	unknown	2.785	0.047
Youtube	1,134,890	2,987,624	top5000	5.265	0.081
Flixster	2,523,386	7,918,801	unknown	6.276	0.083

TABLE VI
THE LARGE-SCALE REAL-LIFE NETWORKS.

A. Performance of Applying Sliding Window to Attractor

To evaluate performance of applying our sliding window to Attractor and whether the sliding window reduce the community detection quality, we utilized all the datasets employed in [5], including datasets with or without ground truth (knowing the number of true communities or not). Since these datasets were introduced in [5], we omit their descriptions in this paper. Table II presents statistics of the datasets. For labeled datasets, we report well-known measures - Purity, NMI and ARI [30], and # of iterations (#iters) to make algorithms converged. For unlabeled datasets, we report normalized cut (Ncut) [31], modularity [3], # of extracted communities (#coms), # of iterations (#iters), and running time reduction.

We compared Attractor with our sliding window (SAttractor) with the original Attractor and IAttractor [6], an improved version of Attractor to make edge distances converged quickly. We set $\lambda = 0.5$ for all methods, following [5]. For IAttractor, we used exactly same parameter settings in [6] such as enhanced factor $\delta = 1$ and Con_Co=0.99 (making edges converged when the proportion of converged edges was greater than Con_Co threshold). For sliding window, we set $\tau = \{0.5, 0.7\}$ since it is more reliable to reflect the converging trend of edges, and set $s = \{10, 15\}$.

Table III presents the results of Attractor, IAttractor and sliding window model with $s = 10$ and $\tau \in \{0.5, 0.7\}$ in labeled datasets. We only report results of $s = 10$ because these datasets require a few iterations to converge and sliding window only take effect after s iterations. The experimental results showed that (1) SAttractor achieved the same quality of extracted communities with Attractor; (2) The number of iterations took in SAttractor was always equal to or smaller than the number of iterations took in Attractor; and (3) IAttractor failed producing the same quality of communities. We investigated the reason why IAttractor performed poorly in terms of the quality, and found that factor δ increased dynamic interactions too much, and made most edges converged to zero.

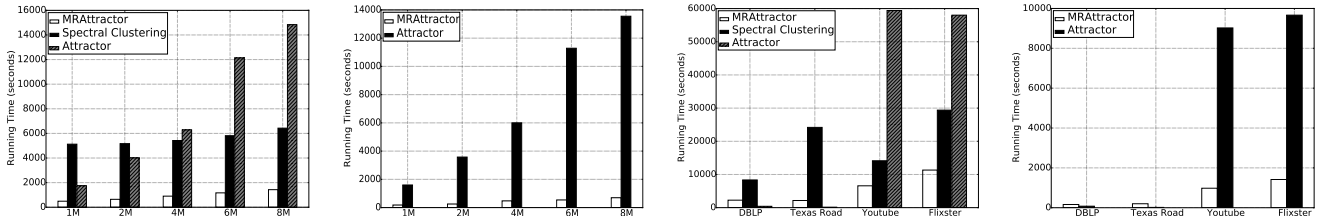
In small datasets such as Karate, Football and Polbooks, we might observe minor improvement by applying the sliding window in terms of running time. However, in Amazon dataset, we observed huge improvement by reducing up to 70.8% running time (reducing from 885 to 258 seconds).

Table IV shows experimental results in unlabeled datasets. Note that IAttractor failed producing the same quality of communities again, so we only report results of Attractor and SAttractor. When we keep track up only the last 10 statues of edges (i.e., $s = 10$), the quality of extracted communities in Friendship dataset was worse than the other sliding window settings. In other datasets, the quality was once again consistent with Attractor. When increasing s to 15 and 20, the quality of extracted communities in all datasets were consistent with those found by Attractor. Besides, #iterations was significantly reduced. In particular, #iterations in Friendship dataset decreased from 323 to only 23 when we set $s = 15$ and $\tau = 0.5$, achieving 28% running time reduction.

B. Performance of MRATTRACTOR

So far, we observed that our sliding window model in a single machine environment reduced number of iterations and the running time, while preserved the quality of extracted communities. Next, we turn to examine MRAttractor's performance in large-scale graphs. As we described in the previous section, MRAttractor has several enhancements including the sliding window against Attractor. In the following experiments, we used both synthetic and real-life datasets/networks.

1) Experimental settings: We created five synthetic graphs presented in Table V by using Fortunato's benchmarking tool [32] for community detection. This tool was also employed in the prior work such as [5], [17], [24]. According to Twitter and Facebook statistics [33], [34], users have 100s



(a) Running time on synthetic graphs (b) Running time of the 1st iteration (c) Running time on real-life graphs (d) Running time of the 1st iteration

Fig. 3. Running time of MRAttractor with single-machine Attractor and Spectral Clustering algorithm on simulated networks and real-life graphs.

followers or friends on average. Therefore, when we generated synthetic datasets, we set average degree to 100. Table VI shows statistics of large real-life graphs. **DBLP** is a graph of co-authorship where two authors are connected if they co-authored at least one paper. **Texas Road** is a road network where nodes are intersections and endpoints. Edges are roads connecting them in Texas. **Youtube** dataset is a snapshot of friendship graph on Youtube. If two users are friends, they will be connected. **Flixster** is the friendship network of movie site Flixster.com where two users are linked if they are friends.

We implemented MRATTRACTOR on the top of Hadoop and ran experiments on a cluster which consisted of one master node and five slave nodes, each of which had 8 cores. We set the number of reducers to 30 and used sliding window with $s = 15$ and $\tau = 0.5$ since these settings achieved the fastest running time while the quality of extracted communities was good in the previous experiment. The other parameters were $\lambda = 0.5$, $\gamma = 10000$, and the number of partitions $p = 20$. The hashing function $P(\cdot)$ was modulo function.

We compared MRATTRACTOR with two baselines:

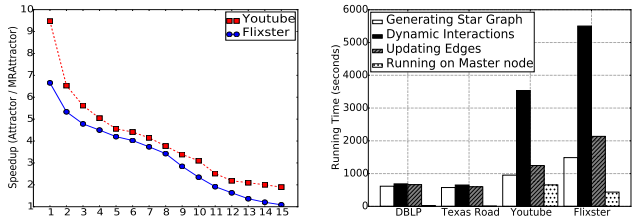
- **Attractor:** We implemented Attractor [5] in Java and ran it on a computer with 64GB RAM and Intel Xeon 8 cores 2.10GHz. The cohesive parameter was $\lambda = 0.5$.
- **Spectral Clustering for MapReduce:** It was a MapReduce version of the spectral clustering proposed in [17]. We set 20 iterations for finding eigenvectors and 30 iterations for K-means, following the same settings in [17].

We ran MRATTRACTOR and the baselines multiple times, and achieved consistent results.

2) Experimental Results: Figure 3 shows performance of MRAttractor and baselines in synthetic and real-life networks. In the synthetic networks (see Figure 3(a)), MRAttractor was significantly faster than Attractor and Spectral clustering. In particular, MRAttractor was 3.56∼10.39 times faster than Attractor, and 4.50∼10.45 times faster than Spectral clustering. Note that, we also generated synthetic graphs with average degree=200, and observed that MRAttractor was much faster than Attractor and Spectral clustering.

In real-life networks, MRAttractor performed worse than Attractor, when the datasets such as DBLP and Texas Road were small, and their average graph density was small as well. It makes sense because iterative MapReduce algorithms usually suffer overhead of network I/O and disk I/O [28]. However, when MRAttractor dealt with larger datasets such as Youtube and Flixster, it performed significantly faster than the baselines. In particular, it was 5.12∼9.02 times faster than Attractor (see Figure 3(c)). Attractor had 1,099 and 1,513 iterations to converge in Youtube and Flixster networks, respectively whereas MRAttractor only had 22 and 25 iterations in Youtube and Flixster, respectively.

In addition, Figures 3(b) and 3(d) shows running time of the first iteration of both Attractor and MRAttractor in synthetic and real-life networks. The first iteration takes the longest time among all the iterations. Again, MRAttractor was much faster than Attractor across all the synthetic networks, and Youtube and Flixster. As we can observe in 1M and 2M synthetic networks, even though graph sizes in 1M and 2M synthetic networks were relatively small, as long as their average graph density was high, MRAttractor would be faster than Attractor (unlike DBLP and Texas Road networks which had the same edge size but much smaller average graph density).



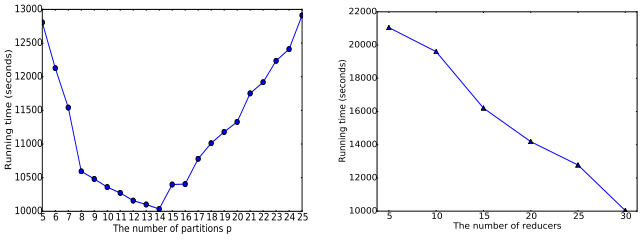
(a) Speedup in the first 15 iterations (b) Running time in each phase

Fig. 4. Speedup of MRAttractor over Attractor (left) and running time of MRAttractor (right).

Figure 4(a) shows how much speedup MRAttractor made over Attractor in Youtube and Flixster datasets in the first 15 iterations. MRAttractor was 9.5 and 6.8 times faster than Attractor in the first iteration. In later iterations, the speedup gradually decreased because many edges converged over time.

Next, we examine running time of each of the three MapReduce phases and remaining edge distance convergence on master node presented in Figure 1. Figure 4(b) shows details of running time in each phase. In small datasets such as DBLP and Texas Road, running time of MapReduce phases were quite similar. However, in larger networks, running time of computing dynamic interactions became dominant due to high complexity of this step. We also observed that running time of remaining edge convergence step increased as the graph size increased even though we set small $\gamma = 10,000$.

3) Number of Graph Partitions and Reducers: We are interested in measuring the sensitivity of the number of graph partitions p in terms of running time. Figure 5(a) shows how running time in Flixster graph (the largest real-life dataset in this paper) was changed, when we varied $p \in [5; 25]$. Given



(a) Effect of p on running time. (b) Varying the number of reducers

Fig. 5. Sensitivity of the number of graph partitions and reducers.

a small p, extracting communities was slow since each of subgraphs contained a large number of vertices and edges. As p increased, the running time decreased significantly until p was 14. When p was greater than 14, it took longer running time because too many subgraphs were generated, leading to performance deterioration.

Another interesting question is how the number of reducers affects performance of MRAttractor. To answer the question, we set $p = 14$, and varied the number of reducers on Flixster dataset. Figure 5(b), the running time decreased linearly as we increased the number of reducers in our Hadoop cluster.

Overall, MRAttractor not only produced the same quality of extracted communities like Attractor (we also ran other experiments to confirm MRAttractor correctly produced the same communities like Attractor), but also significantly faster than Attractor and a MapReduce version of Spectral Clustering in the large datasets or high graph density in small datasets.

VI. Conclusions and Future Work

In this paper, we have presented how we have designed and implemented MRAttractor, an advanced version of Attractor for a MapReduce framework, Hadoop. Our proposed framework has handled large-scale graphs and significantly outperformed Attractor, IAttractor and Spectral Clustering, reducing running time and producing the same quality of extracted communities. In the future, we will implement MRAttractor for Spark and focus on handling large-scale time-evolving networks. Our graph partitioning algorithm has an advantage over existing graph processing frameworks when tackling exclusive interactions. In Pregel [8], for example, a vertex cannot directly communicate with an exclusive neighbor x without knowing x's identifier. GraphX [7], even does not support communication of unconnected nodes. Thus, when designing algorithms that require exclusive interactions, our method can be customized to meet this demand.

Acknowledgment

This work was supported in part by NSF grants CNS-1553035 and CNS-1755536, and Google Faculty Research Award. Any opinions, findings and conclusions or recommendations expressed in this material are the author(s) and do not necessarily reflect those of the sponsors.

References

[1] M. E. Newman, "Detecting community structure in networks," *The European Physical Journal B-Condensed Matter and Complex Systems*, vol. 38, no. 2, pp. 321–330, 2004.

[2] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and computing*, vol. 17, no. 4, pp. 395–416, 2007.

[3] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, 2008.

[4] M. Rosvall and C. T. Bergstrom, "Maps of random walks on complex networks reveal community structure," *PNAS*, 2008.

[5] J. Shao, Z. Han, Q. Yang, and T. Zhou, "Community detection based on distance dynamics," in *KDD*, 2015.

[6] T. Meng, L. Cai, T. He, L. Chen, and Z. Deng, "An improved community detection algorithm based on the distance dynamics," in *Intelligent Networking and Collaborative Systems (INCoS)*, 2016.

[7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *USENIX OSDI*, 2014.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010.

[9] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *USENIX OSDI*, 2012.

[10] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *ICDM*, 2009.

[11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *USENIX OSDI*, 2012.

[12] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[13] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *PNAS*, 2002.

[14] S. Fortunato and D. Hric, "Community detection in networks: A user guide," *Physics Reports*, vol. 659, pp. 1–44, 2016.

[15] B. Karrer and M. E. Newman, "Stochastic blockmodels and community structure in networks," *Physical Review E*, 2011.

[16] N. Vo, K. Lee, C. Cao, T. Tran, and H. Choi, "Revealing and detecting malicious retweeter groups," in *ASONAM*, 2017.

[17] S. Tsironis, M. Sozio, M. Vazirgiannis, and L. Poltechnique, "Accurate spectral clustering for community detection in mapreduce," in *NIPS Workshops*, 2013.

[18] W. Zhao, H. Ma, and Q. He, "Parallel k-means clustering based on mapreduce," in *CLOUD*, 2009.

[19] Sotera, "Community detection and compression analytic for big graph data," https://github.com/Sotera/distributed-louvain-modularity, 2014.

[20] A. U. Bhat, "Scalable community detection using label propagation & map-reduce," http://www.akshaybhat.com/static/files/LPMR.pdf, 2012.

[21] Y. Zhang, J. Wang, Y. Wang, and L. Zhou, "Parallel community detection on large networks with propinquity dynamics," in *KDD*, 2009.

[22] M. Saltz, A. Prat-Pérez, and D. Dominguez-Sal, "Distributed community detection with the wcc metric," in *WWW*, 2015.

[23] X. Ling, J. Yang, D. Wang, J. Chen, and L. Li, "Fast community detection in large weighted networks using graphx in the cloud," in *HPCC*, 2016.

[24] C. Wickramaarachchi, M. Frincu, P. Small, and V. K. Prasanna, "Fast parallel algorithm for unfolding of communities in large graphs," in *HPEC*, 2014.

[25] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *WWW*, 2011.

[26] R. Gupta, T. Roughgarden, and C. Seshadhri, "Decompositions of triangle-dense graphs," *SIAM Journal on Computing*, 2016.

[27] J. H. Chang and W. S. Lee, "Finding recent frequent itemsets adaptively over online data streams," in *KDD*, 2003.

[28] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of mapreduce: An in-depth study," *VLDB Endowment*, 2010.

[29] D. de Caen, "An upper bound on the sum of squares of degrees in a graph," *Discrete Mathematics*, 1998.

[30] C. D. Manning, P. Raghavan, H. Schütze *et al.*, *Introduction to information retrieval*. Cambridge university press Cambridge, 2008.

[31] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Transactions on pattern analysis and machine intelligence*, 2000.

[32] A. Lancichinetti, S. Fortunato, and F. Radicchi, "Benchmark graphs for testing community detection algorithms," *Physical review E*, 2008.

[33] T. Telegraph, "Average twitter user is an an american woman with an iphone and 208 followers," http://bit.ly/1BNIQLC, 2012.

[34] A. Smith, "6 new facts about facebook," http://pewrsr.ch/1kENZcA, 2014.