# GRAPH CLUSTERING ALGORITHMS IN GRAPHBLAS

An Undergraduate Research Scholars Thesis

by

CAMERON QUILICI

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

| | |
|---|---|
| Approved by<br>Faculty Research Advisor: | Dr. Tim Davis |

May 2024

| | |
|---|---|
| Majors: | Computer Science<br>Applied Mathematics |

# RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Cameron Quilici, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Faculty Research Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

# TABLE OF CONTENTS

Page

# ABSTRACT

Graph Clustering Algorithms in GraphBLAS

Cameron Quilici
Department of Computer Science and Engineering
Texas A&M University

Faculty Research Advisor: Dr. Tim Davis
Department of Computer Science and Engineering
Texas A&M University

Graph theory has long served as a cornerstone for computational problems among various domains. Hence, the development of graph algorithms has proved to be one of the most pronounced focuses in the field of computer science. Among the most recent developments in this field is the emergence of linear algebra as a tool for addressing graph related problems. GraphBLAS, an open-source API specification, realizes this intrinsic connection by providing a framework for constructing graph algorithms in the language of linear algebra. Graph clustering is the process of determining natural groups of nodes with relatively high connectivity in a graph structure. The Peer Pressure and Markov Cluster algorithms are two unsupervised processes which capitalize on linear algebraic principles to efficiently identify clusters within graphs. This paper aims to walk through the development and implementation of both algorithms using the SuiteSparse:GraphBLAS C API, with the additional goal of fostering intuition for crafting graph algorithms from a linear algebraic perspective. Our implementations will be added to the LAGraph repository, a collection of algorithms implemented using GraphBLAS. Additionally, we provide a suite of metrics which can be used to quantitatively measure the quality of a graph clustering. We demonstrate that our quality metrics surpass the speed of existing implementations and our clustering algorithms yield reasonable clusterings efficiently, even on large graphs.

# DEDICATION

*To my mother, for giving me the opportunity to receive an education.*

# ACKNOWLEDGMENTS

### Contributors

I would like to thank my faculty advisor, Dr. Tim Davis, for his guidance and support throughout my research. My success in this project would not have been possible without him.

I would also like to thank my wonderful girlfriend, Addison, for providing me with encouragement and love in addition to constantly helping me stay hopeful, even during particularly stressful periods.

Finally, thanks to Gábor Szárnyas for granting me permission to adopt the style of his "Introduction to GraphBLAS" slideshow presentation. Additionally, I am thankful for his generosity in sharing the source slides with me.

The source code for the SuiteSparse:GraphBLAS C API as well as the LAGraph repository utilized in order to develop the programs contained in this document was provided by Dr. Tim Davis, along with several additional contributors.

All other work conducted for the thesis was completed by the student independently.

### Funding Sources

Undergraduate research was supported by Dr. Tim Davis at Texas A&M University.

# NOMENCLATURE

| | |
|---|---|
| BLAS | Basic Linear Algebra Subroutine |
| API | Application Programming Interface |
| PPC | Peer Pressure Clustering |
| MCL | Markov Cluster Algorithm |
| $\mathbf{A} \in D^{n\times n}$ | $n \times n$ adjacency matrix with elements over domain $D$ |
| $\mathbf{v} \in D^{n}$ | length $n$ column vector with entries over domain $D$ |
| $A_{ij}$ | the entry in row $i$, column $j$ of the matrix $\mathbf{A}$ |
| $v_i$ | the $i^{\text{th}}$ entry of the vector (row or column) $\mathbf{v}$ |
| $\mathbf{A}^T$ | the transpose of $\mathbf{A}$ |
| $tr(\mathbf{A})$ | the trace of $\mathbf{A}$ |
| $G = (V, E)$ | the graph $G$ with vertex set $V$ and edge set $E$ |
| $\mathbb{R}$ | the set of real numbers |
| $\mathbb{R}_{\geq 0}$ | the set of non-negative real numbers |
| $\mathbb{Z}$ | the set of integers |
| $\mathbb{N}$ | the set of natural numbers |
| $\mathbb{B}$ | the set $\{0, 1\}$ (booleans) |

# 1. INTRODUCTION

## 1.1 Mathematical Background

### *1.1.1 Graph Theory*

A *graph* is a pair $G = (V, E)$ such that $E \subseteq [V]^2$, where $[V]^2 = \{\{u, v\} \mid u, v \in V, u \neq v\}$. That is, elements of $E$ are $2$-*element subsets* of $V$, with the exception of self-loops. The elements of $V$ are called the *vertices* and the elements of $E$ are called the *edges* of the graph $G$. An edge $e \in E$ can be denoted as $\{x, y\}$ or more commonly $xy$, denoting an edge between vertex $x$ and $y$.

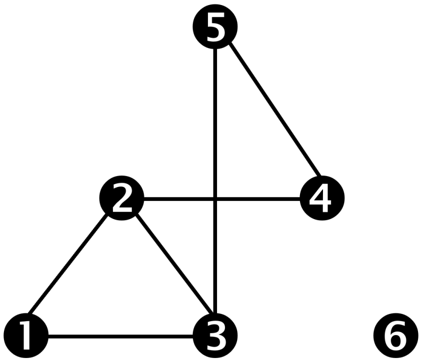


**Figure 1:** The graph $G_1$ on the set of vertices $V = \{1, \ldots, 6\}$ with $E = \{\{1,3\}, \{1,2\}, \{2,3\}, \{2,4\}, \{3,5\}, \{4,5\}\}$.

The *vertex* and *edge sets* of a particular graph are denoted $V$ and $E$, respectively. The *order* of a graph is denoted as $|G|$ and refers to the number of vertices of a graph. For example, the graph in Figure 1 has the property $|G_1| = |V| = 6$. A vertex $v$ is *incident* with an edge $e$ if $v \in e$ and we say $e$ is an edge *at* $v$. There is also the notion of a *non-edge*, which is a possible edge in $G$ which does not exist. For instance, in $G_1$, $\{6, 4\}$ is a non-edge. Furthermore, the set of all edges $e \in E$ at a vertex $v$ is denoted $E(v)$. The *degree* $d_G(v)$ (or simply $d(v)$ when the reference of a particular graph is clear) of a vertex $v$ is equal to $|E(v)|$, the number of edges at $v$. For instance, $d_{G_1}(2) = 3$ and $d_{G_1}(6) = 1$ [1].

Let $G = (V, E)$ and $G' = (V', E')$. Then $G \cup G' = (V \cup V', E \cup E')$ and $G \cap G' = (V \cap V', E \cap E')$. When $G \cap G' = \emptyset$, $G$ and $G'$ are *disjoint*. When $V' \subseteq V$ and $E' \subseteq E$, $G'$ is a *subgraph* of $G$, denoted $G' \subseteq G$.

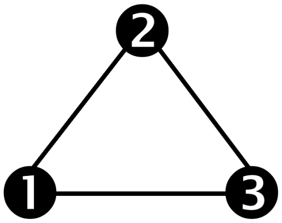


**Figure 2:** A subgraph $G_1' \subseteq G_1$.

There is also the notion of *directed graphs*, a pair $G = (V, E)$ where the set of *arcs* (or directed edges) $E$ is defined as $E = \{(u, v) \mid u, v \in V \times V\}$. In this way, edges between vertices have direction, and self-loops are possible.

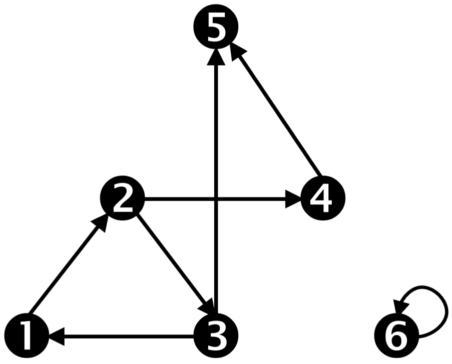


**Figure 3:** The directed graph $G_2$ on the set of vertices $V = \{1, \ldots, 6\}$ with $E = \{(1,2), (2,3), (3,1), (3,5), (2,4), (2,3), (6,6)\}$.

In a directed graph, each vertex has an *in-degree* and an *out-degree*, the number of edges coming into the vertex and the number of edges leaving the vertex, respectively. We denote $d^+(v)$ as the out-degree and $d^-(v)$ as the in-degree of vertex $v$. In $G_2$, $d^+(5) = 0$ and $d^-(5) = 2$.

### 1.1.2 Linear Algebraic Formulation

All finite graphs can be expressed as an *adjacency matrix*. The adjacency matrix $\mathbf{A} \in \mathbb{B}^{n \times n}$ of an unweighted graph $G$ is defined by

$$A_{ij} := \begin{cases} 1 & \text{if } v_i v_j \in E \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

When displaying adjacency matrices, we will not write any implicit zeros, i.e., if $v_i v_j \notin E$, then $A_{ij}$ is simply blank (see Figure 4 below). Further, the *dimension* of $\mathbf{A}$ representing a graph $G$ is necessarily $|G| \times |G|$ for finite graphs.

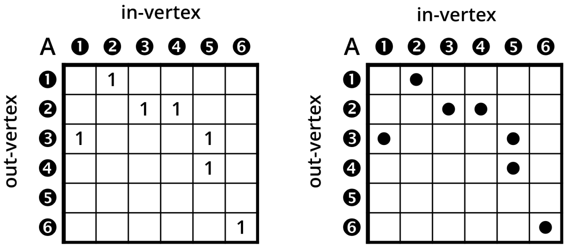


**Figure 4:** Two equivalent visual representation of the $6 \times 6$ adjacency matrix $\mathbf{A}$ representing $G_2$ from Figure 3. Oftentimes when the edges are unweighted, the representation depicted on the right is more clear.

Given this representation, there is an intrinsic relationship between linear algebra and graph algorithms. One can learn many properties of a particular graph just from an adjacency matrix alone. For instance, let $\mathbf{A} \in \mathbb{N}^{n \times n}$. One can quickly compute the out-degree of the $i^{\text{th}}$ vertex by computing $\sum_{j=1}^{n} A_{ij}$ and compute the in-degree of the $j^{\text{th}}$ vertex by computing $\sum_{i=1}^{n} A_{ij}$. Consider the matrix $\mathbf{A}$ from our working example and we can clearly see that the out-degree of vertex 2 is 2. Next, consider the squaring via standard matrix multiplication of $\mathbf{A}$ as shown in Figure 5. The resulting matrix $\mathbf{A}^2 \in \mathbb{N}^{n \times n}$ has the property that $A_{ij} = m$ if and only if there exist $m$ paths of length 2 from vertex $i$ to vertex $j$. The figure indicates that $A_{25} = 2$ and indeed, there are 2 paths of length 2 from vertex 2 to vertex 5 (highlighted in green). In fact, it can be verified that $\mathbf{A}^k \in \mathbb{N}^{n \times n}$ gives the number of paths of length $k \in \mathbb{N}$ from vertex $i$ to vertex $j$. While some properties of graphs can be realized via the standard matrix multiplication procedure (with addition and multiplication), this process is often too restrictive in the context of graph algorithms. For instance, if the domain of the entries of an adjacency matrix are not a subset of $\mathbb{R}$, then "addition" and "multiplication" may not be well-defined operations. Furthermore, for certain graph algorithms, one may want to consider, say, the *minimum* of two entries in a matrix-matrix/vector-matrix "multiply" rather than their *product*. One way to achieve this is to use a broader definition of matrix and vector multiplication using *semirings*.

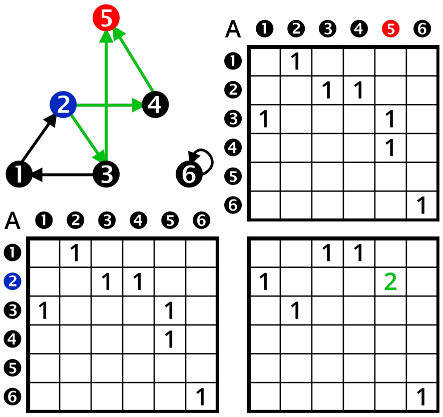


**Figure 5:** Matrix squaring relation to paths of length 2 in a graph.

### *1.1.3 Semirings*

A *semiring* is an algebraic structure $\langle D, \oplus, \otimes, 0\rangle$, with $D$ a non-empty set, on which we have defined operations of "addition" and "multiplication" which satisfy the following properties [2]:

(1) $\langle D, \oplus\rangle$ is a commutative monoid with identity element 0;

(2) $\langle D, \otimes\rangle$ is a monoid with identity element $1 \neq 0$;

(3) $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)\ \forall a, b, c \in D$;

(4) $0 \otimes a = 0 = a \otimes 0\ \forall a \in D$.

In the context of GraphBLAS, the requirement that $\langle D, \otimes \rangle$ is a monoid (point 2 above) is omitted. That is, $\langle D, \otimes \rangle$ can be any closed binary operator. A semiring that meets these weaker requirements is referred to as a *GraphBLAS semiring*.

Importantly, matrix multiplication can be performed on various semirings. Let $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{n \times n}$. Using the conventional semiring such that $\oplus := +$ and $\otimes := \cdot$, matrix multiplication is defined by

$$\begin{aligned} \mathbf{C} &= \mathbf{AB} \\ C_{ij} &= \sum_{k=1}^{n} A_{ik} \cdot B_{kj} \end{aligned} \tag{2}$$

for each $i, j \in \{0, 1, \ldots, n\}$. This formula can be generalized with the notion of semirings. Let $\langle D, \oplus, \otimes, 0 \rangle$ be a GraphBLAS semiring and $\mathbf{A}, \mathbf{B}, \mathbf{C} \in D^{n \times n}$, then the matrix "multiplication" of $\mathbf{A}$ and $\mathbf{B}$ is defined as

$$\begin{aligned} \mathbf{C} &= \mathbf{A} \oplus . \otimes \mathbf{B} \\ C_{ij} &= \bigoplus_{k=1}^{n} A_{ik} \otimes B_{kj}. \end{aligned} \tag{3}$$

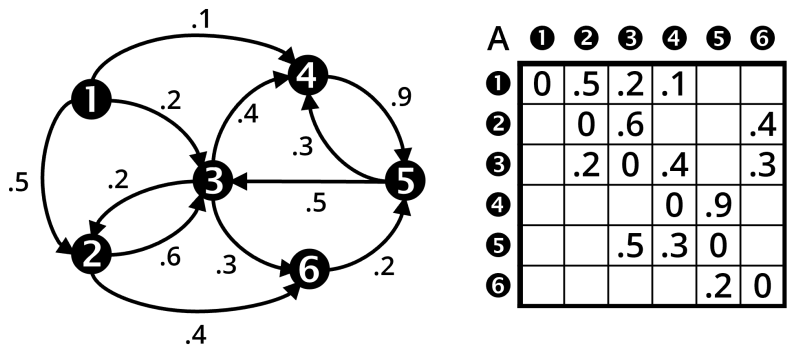
A matrix is *sparse* if most of its entries are zero. Most adjacency matrices are sparse since graphs often have relatively few connections per node. Let $\mathcal{H}_i$ denote the collection of column indices of nonzero entries in row $i$ of matrix $\mathbf{A}$ and let $\mathcal{K}_j$ denote the collection of row indices of nonzero entries in column $j$ of matrix $\mathbf{B}$. Then we can further generalize Equation 3 to sparse matrix multiplication as follows:

$$\begin{aligned} \mathbf{C} &= \mathbf{A} \oplus . \otimes \mathbf{B} \\ C_{ij} &= \bigoplus_{k \in \mathcal{H}_i \cap \mathcal{K}_j} A_{ik} \otimes B_{kj}. \end{aligned} \tag{4}$$

That is, the $\otimes$ operator is only applied where an entry is present in both $\mathbf{A}$ and $\mathbf{B}$.

### *1.1.4 Motivating Example*

With this generalized concept of matrix multiplication, we can now begin to realize the linear algebraic formulation of some important graph algorithms. As a motivating example, consider a linear algebraic formulation of the Bellman-Ford algorithm for the single-source shortest path (SSSP) problem. Given a graph $G = (V, E)$, the SSSP problem involves finding the shortest paths from some vertex $v \in V$ to all other vertices in $V$.



| A | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ |
|---|---|---|---|---|---|---|
| ❶ | 0 | .5 | .2 | .1 |  |  |
| ❷ |  | 0 | .6 |  |  | .4 |
| ❸ |  | .2 | 0 | .4 |  | .3 |
| ❹ |  |  |  | 0 | .9 |  |
| ❺ |  |  | .5 | .3 | 0 |  |
| ❻ |  |  |  |  | .2 | 0 |

**Figure 6:** The weighted directed graph $G_3$ (left) and its weighted adjacency matrix representation $\mathbf{A}$ (right). Note, when $A_{ij}$ is blank, the entry is implicitly zero and takes on the value of the respective semiring's additive identity, in this case $+\infty$. However, $A_{ij} = 0$ (on the diagonal) indicates that each vertex can reach itself with distance 0.

Developing linear algebraic formulations for graph algorithms involves making connections between matrix and graph operations and then expanding on these ideas via semirings. For SSSP, we would like to find the *shortest cumulative* paths from one vertex to all other vertices. Let $G = (V, E)$ be a weighted directed graph and $\mathbf{A} \in \mathbb{R}^{n \times n}_{\geq 0}$ be its adjacency matrix representation. Further, let $v_s \in V$ be the source vertex and $\mathbf{v}^{(0)}$ be the row vector of dimension $1 \times n$ such that the $s^{\text{th}}$ entry in $\mathbf{v}$, $v_s$, is equal to $1$ and all other entries are equal to 0. If we modify the matrix $\mathbf{A}$ to be unweighted such that $A_{ij} = 1$ if and only if there is an edge between $v_i$ and $v_j$, then with standard vector-matrix multiplication, one can see that the operation

$$\mathbf{v}^{(1)} = \mathbf{v}^{(0)}\mathbf{A} \tag{5}$$

gives precisely the vertices "one hop" away from $v_s$. That is, $v_i^{(1)} = 1$ if and only if there exists a path of length 1 from $v_s$ to $v_i$. Moreover, the operation $\mathbf{v}^{(2)} = \mathbf{v}^{(1)}\mathbf{A}$ gives the number of paths of length 2 from vertex $s$ to every other vertex, so on and so forth. For SSSP, we instead want to capture the *minimum cumulative* path from $v_s$ to all other vertices. Hence, we will use the so-called “min-plus” semiring where $D = \mathbb{R} \cup \{+\infty\}$, $\oplus := \min$, $\otimes := +$, and the additive identity is $+\infty$.

Now, setting $v_s$ equal to 0 and all other entries to $+\infty$ (the identity with respect to the min binary operator), we can modify Equation 5 to obtain

$$\begin{aligned}\mathbf{v}^{(1)} &= \mathbf{v}^{(0)} \oplus . \otimes \mathbf{A} \\ &= \mathbf{v}^{(0)} \min . + \mathbf{A}\end{aligned}$$

and we get that $\mathbf{v}^{(1)}$ holds the minimum cumulative weights from $v_s$ to vertices one hop away from it. Continuing these operations $n-1$ times, we obtain

$$\begin{aligned}\mathbf{v}^{(1)} &= \mathbf{v}^{(0)} \min . + \mathbf{A} \\ \mathbf{v}^{(2)} &= \mathbf{v}^{(1)} \min . + \mathbf{A} \\ &\vdots \\ \mathbf{v}^{(n-1)} &= \mathbf{v}^{(n-2)} \min . + \mathbf{A} \qquad (6)\end{aligned}$$

and $\mathbf{v}^{(n-1)}$ holds the shortest path lengths from $v_s$ to all vertices $n-1$ hops away from $v_s$. In other words, $\mathbf{v}^{(n-1)}$ holds the shortest path from vertex $s$ to every other vertex in the graph. Thus, we have solved the SSSP problem using only vector-matrix multiplication over a non-conventional semiring.

Figure 7 gives an example of this algorithm. The blue vertex represents the source vertex, the purple vertices represent those vertices on the frontier, and the gray vertices represent those vertices which have already been reached from the source. Moreover, the green highlights the values (edges) which are involved in the vector-matrix multiplication at each iteration.

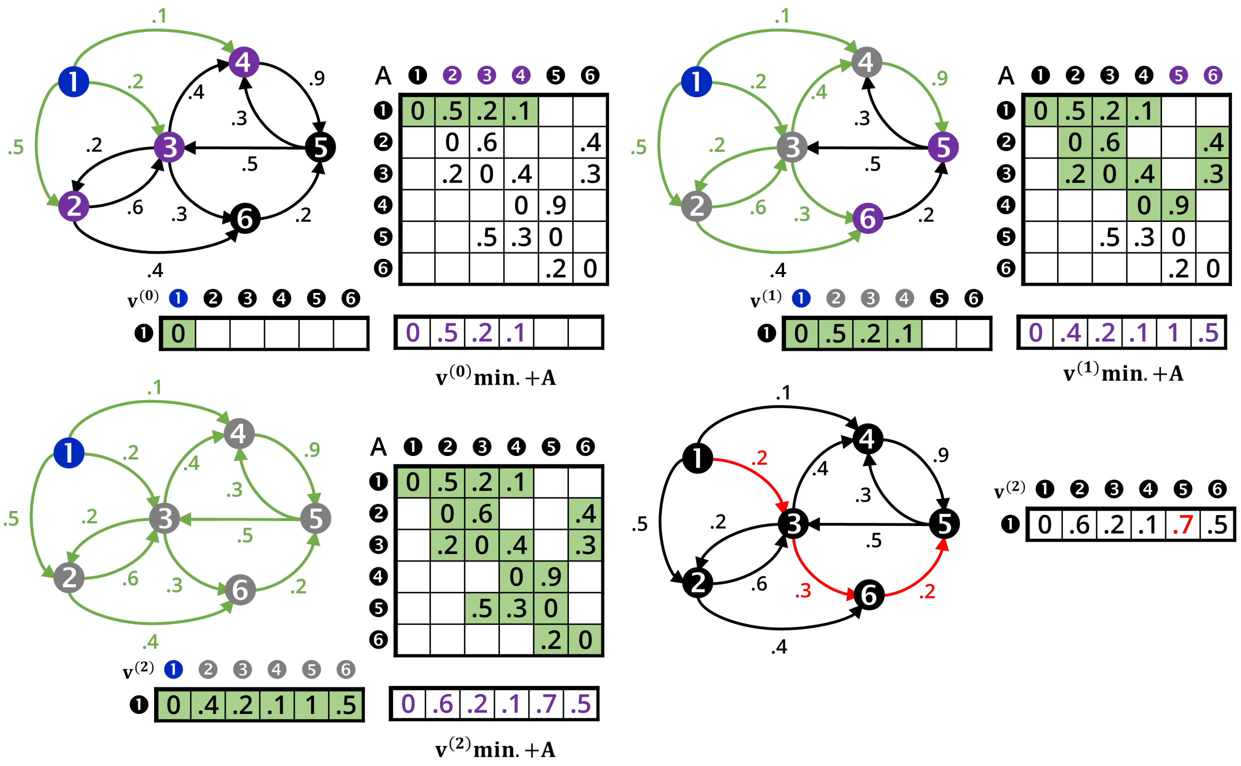

A
$\mathbf{v}^{(0)}$
$\mathbf{v}^{(0)}$min. +A
$\mathbf{v}^{(1)}$
$\mathbf{v}^{(1)}$min. +A
$\mathbf{v}^{(2)}$
$\mathbf{v}^{(2)}$min. +A

**Figure 7:** Example of the SSSP linear algebraic algorithm on $G_3$ from Figure 7. There are three iterations before a steady state is reached and the diagram is read from left to right, top to bottom. The bottom right figure demonstrates the shortest path found from vertex 1 to 6.

## 1.2 Graph Clustering

Since graphs mostly always represent networks of connections, a natural question to ask is how to best group their vertices into clusters such that there are more connections (edges) within clusters than there are between clusters. That is, the task of graph clustering is to find highly *intra*-connected groups of vertices. This process is highly applicable to many field such as social network analysis, machine learning, and biological pathway analysis.

Let $G$ be a graph such that $|G| = n$. A *clustering* (or *partition*) $\mathcal{C}_G$ is a collection of $k$ disjoint subgraphs of $G$ such that $1 \leq k \leq n$. That is,

$$\mathcal{C}_G = \{C_1, C_2, \ldots, C_k\} \tag{7}$$

where $\bigcap_{1 \leq i \leq k} C_i = \emptyset$, $\bigcup_{1 \leq i \leq k} C_i = V$, $C_1, \ldots, C_k \subseteq G$, and the vertices $v \in C_i$ are similar or connected in some predefined way [3]. Oftentimes, each individual $C_i$ is called a *cluster* or a *community*. Note that $|\mathcal{C}_G| \leq |G|$ for any graph. An edge between two vertices within the same cluster is called an *intra-cluster* edge and an edge between two vertices not within the same cluster is called an *inter-cluster* edge. Moreover, a *possible* intra/inter-cluster edge which does *not exist* is called an intra/inter-cluster *non-edge*, respectively. We denote $E_{intra}, E_{inter} \subseteq E$ as $E_{intra} = \{(i,j) \in E, C_i = C_j\}$ and $E_{inter} = \{(i,j) \in E, C_i \neq C_j\}$ to be the sets of intra-cluster and inter-cluster edges, respectively. Furthermore, we denote $N_{intra} = \{(i,j) \notin E, C_i = C_j\}$ and $N_{inter} = \{(i,j) \notin E, C_i \neq C_j\}$ to be the sets of intra-cluster and inter-cluster non-edges, respectively.

There are many possible clusterings of a graph and, as mentioned before, what makes a particular clustering better than another is based on some metric of "similarity" or "connectedness." Later in this paper (see Results), we will discuss in depth some of the quality metrics used in order to quantitatively identify clusterings as "good" or "bad."

In this paper, we present two clustering algorithms with their SuiteSparse:GraphBLAS implementations.

### *1.2.1 Peer Pressure Clustering*

The Peer Pressure Clustering (PPC) algorithm gets its name from the fact that a vertex's cluster assignment is propagated outwards towards its neighbors. That is, any given vertex will be put in the same cluster as the majority of its neighbors; it will be pressured by its peers. The algorithm was first proposed by A. Shah in his PhD thesis [4]. The algorithm begins by designating each vertex to its own cluster and then iteratively refines the cluster assignments of neighboring vertices via a weighted voting process. The weight of a particular vertex's vote can either be proportional to its edge weight, or proportional to its edge weight normalized by its out-degree. The latter is favorable when there are many vertices with very high out-degrees, so that these vertices do not dominate the influence on neighboring vertices. After each round of voting, each vertex counts its total received votes from all other clusters and then joins the cluster from which it received the maximum number of votes. If there is a tie between two clusters, the one with the minimum index is chosen, which makes the process deterministic. The algorithm terminates when the clustering reaches a steady-state, i.e., when no vertex changes clusters between subsequent iterations. Unfortunately, convergence is not always guaranteed, especially on graphs which have very little natural community structure. As a result of this, the algorithm can be modified to terminate whenever the percentage of total cluster assignments between subsequent iterations falls below some predefined small threshold.

### *1.2.2 Markov Clustering*

The Markov Cluster Algorithm (MCL) was first proposed by S. Dongen [5] in his PhD thesis and works by simulating random walks within a graph to explore its community structure. The fundamental idea is that a random walk, when initiated within a cluster, is more likely to remain within that cluster due to the higher density of connections as compared to inter-cluster connections. Given an adjacency matrix representation $\mathbf{A} \in \mathbb{R}_{\geq 0}^{n \times n}$ of a graph $G = (V, E)$ with $n$ vertices, the algorithm first normalizes each column of $\mathbf{A}$ (so that its sum is 1), ensuring that they are *stochastic*. Call this normalized version of $\mathbf{A}$ the *transfer matrix* and denote it as $\mathbf{T}$. Then $T_{ij}$ denotes the probability of a transition from vertex $i$ to vertex $j$. The *expansion* phase raises $\mathbf{T}$ to the $e^{\text{th}}$ power (expansion parameter) to simulate random walks of length $e$ across the graph. Now, $T_{ij}$ gives the probability of a 2-hop walk from vertex $i$ to $j$. Subsequently, the *inflation* phase raises each element of $\mathbf{T}$ to the power of $r$ (inflation parameter). This step increases the contrast between small differences in probabilities. Finally, the columns are again normalized and the process is repeated until convergence. Once a steady state transfer matrix is reached, rows with at least one positive value correspond to an *attractor* vertex which attracts the vertices corresponding to the column indices of the positive entries within the row.

## 1.3 GraphBLAS and LAGraph Overview

### *1.3.1 GraphBLAS*

The GraphBLAS standard formalizes the notion of graph algorithms as linear algebraic operations by providing a set of well-defined matrix and vector operations based on semirings [6]. In other words, the standard aims to provide a consistent set of “building blocks" which can be used to create graph algorithms in the language of linear algebra.

SuiteSparse:GraphBLAS is the first reference implementation of the GraphBLAS standard [7] and provides a set of methods which can be used to modify the objects defined in the C API Specification. In particular, this implementation focuses on sparse matrix operations. As we have noted, most adjacency matrix representations of graphs are sparse. Much work has already been done to optimize sparse matrix operations, and SuiteSparse:GraphBLAS realizes significant speedup by employing such work while abstracting away technical details from the programmer. In turn, users are able to implement a wide range of graph algorithms very efficiently while using simple user-level code.

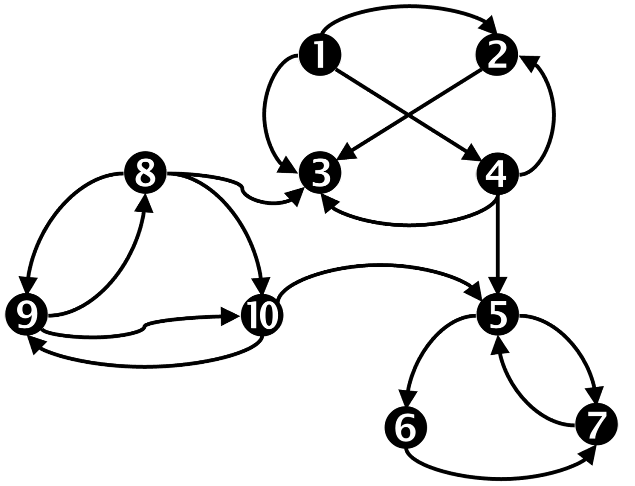
### *1.3.2 LAGraph*

The LAGraph repository is a community effort which aims to provide a centralized collection of graph algorithms implemented using GraphBLAS [8]. Not only does this allow researchers to methodically evaluate the coverage of graph algorithms using linear algebra, but it also acts as a resource for programmers and researchers in the field. As of March 2024, LAGraph includes many novel graph algorithms such as Page Rank, SSSP, and triangle counting. Additionally, the repository includes many more “experimental" graph algorithms (codes which are still under development) such as coarsening, matching, Fast Graphlet Transform, and many more.

While great progress has been made in this effort, there are still *many* more algorithms left to be implemented in the language of linear algebra. The ultimate goal of our project is to utilize SuiteSparse:GraphBLAS to develop the Peer Pressure and Markov Clustering algorithms, along with a few graph clustering quality metrics, and then add the implementations to the LAGraph repository.

# 2. METHODS

In this section, we present a comprehensive breakdown of the Peer Pressure and Markov clustering algorithms, utilizing the SuiteSparse:GraphBLAS C API for our exposition. Our objective is to guide the reader through each implementation phase. This approach is designed to provide a clear and detailed understanding of the algorithms' overarching linear algebraic structure while also providing the corresponding SuiteSparse:GraphBLAS code. The following implementations are written in the C programming language. Therefore, some prior knowledge of the language is expected of the reader.



| A | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ |  | 1 | 1 | 1 |  |  |  |  |  |  |
| ❷ |  |  | 1 |  |  |  |  |  |  |  |
| ❸ |  |  |  |  |  |  |  |  |  |  |
| ❹ |  | 1 | 1 |  | 1 |  |  |  |  |  |
| ❺ |  |  |  |  |  | 1 | 1 |  |  |  |
| ❻ |  |  |  |  |  |  | 1 |  |  |  |
| ❼ |  |  |  |  | 1 |  |  |  |  |  |
| ❽ |  |  | 1 |  |  |  |  |  | 1 | 1 |
| ❾ |  |  |  |  |  |  |  | 1 |  | 1 |
| ❿ |  |  |  |  | 1 |  |  |  | 1 |  |

**Figure 8:** The unweighted directed graph (left) and its adjacency matrix representation $\mathbf{A}$ (right). Note, despite being an unweighted graph, the domain of $\mathbf{A}$ is $\mathbb{R}$ and an edge is represented as a 1.

The directed graph $G$ and adjacency matrix $\mathbf{A} \in \mathbb{R}^{10\times 10}$ in Figure 8 will be our **working example** as we illustrate the implementations of these clustering algorithms using GraphBLAS. For the purposes of these algorithms, an edge is represented by a real-valued weight of 1. It is important to note that while our working example is a directed unweighted graph, both algorithms described in this chapter work on undirected and/or weighted graphs.

In order to better explain some codes, we will sometimes reference particular parts of a line of code using a "footnote" of the form `#1.#2`, both inline (the code) and in the body of the paper. The `#1` references the corresponding figure and the #2 is the specific footnote index within the figure.

## 2.1 Basics of SuiteSparse:GraphBLAS

In this section, we introduce some of the functions used in the following sections as well as insight into the basic structure of our implementations within the LAGraph repository.

The SuiteSparse:GraphBLAS API provides users a collection of various methods which are used to interface with each of the following objects: `GrB_Matrix`, `GrB_Vector`, `GrB_Type` specifies values stored in a GraphBLAS matrix/vector, `GrB_UnaryOp` specifies a unary operator, `GrB_IndexUnaryOp` specifies a type of unary operator that also operates on the *index* of a value, `GrB_BinaryOp` specifies a binary operator, `GrB_Monoid` specifies a monoid (an associative and commutative binary operator), `GrB_Semiring` specifies a GraphBLAS semiring, `GrB_Descriptor` specifies certain parameters which can be passed to a GraphBLAS method to modify its behavior, and `GrB_Scalar` specifies some scalar value [7]. Many of these objects, such as `GrB_Semiring`, provide support for user-defined objects which allows for maximum expressivity and ease-of-use in the graph algorithm design process.

In this chapter, we will detail the GraphBLAS methods used in our work, explaining each as we come across them. For brevity, we will not include the declaration and initialization of all objects used; the reader may assume that any objects passed to a method are either built-in or already exist. The following code generalizes the process of creating and initializing GraphBLAS objects.

```
GrB_Matrix M = NULL ;
GrB_Vector v = NULL ;

// GrB_Index used to store matrix dimensions, equivalent to uint64_t
GrB_Index n = 10 ;
GrB_Matrix_new(&M, GrB_INT64, n, n) ;
GrB_Vector_new(&v, GrB_INT64, n) ;
```

**Figure 9:** Declaring GraphBLAS objects in SuiteSparse:GraphBLAS. This creates an `n`-by-`n` matrix `M` and a vector `v` of length `n`, where `M` and `v` have entries of type `int_64t`.

### *2.1.1 LAGraph Basics*

As previously mentioned, LAGraph is a repository which allows users to easily implement graph algorithms on top of the GraphBLAS framework. As such, LAGraph both provides supplemental data structures (which are not a part of the GraphBLAS standard) and sets forth some conventions for creating graph algorithms using GraphBLAS. The most important data structure that LAGraph provides, and one that we use throughout this paper, is the `LAGraph_Graph`. This data structure holds important information about a graph, such as its adjacency matrix `A`, its type (directed, undirected), and other cached properties such as `A^T`.

```c
struct LAGraph_Graph_struct
{
    GrB_Matrix  A ;             // the adjacency matrix of the graph
    LAGraph_Kind kind ;         // the kind of graph

    // cached properties of the graph
    GrB_Matrix AT ;             // the transpose of A, with the same type
    GrB_Vector out_degree ;
    GrB_Vector in_degree ;
    LAGraph_Boolean is_symmetric_structure ;
    int64_t nself_edges ;       // number of entries on the diagonal of A
    GrB_Scalar emin ;           // minimum edge weight
    LAGraph_State emin_state ;  // VALUE, BOUND, or UNKNOWN
    GrB_Scalar emax ;           // maximum edge weight
    LAGraph_State emax_state ;  // VALUE, BOUND, or UNKNOWN
} ;

typedef struct LAGraph_Graph_struct *LAGraph_Graph ;
```

**Figure 10:** Code representing the data structure `LAGraph_Graph`.

Furthermore, LAGraph sets forth a convention for algorithm function headers, described by the code shown in Figure 11 [9].

```
int graph_algorithm
(
    // outputs
    TYPE *out1, TYPE *out2, ...
    // input/output
    TYPE inout,
    // inputs
    TYPE input1, TYPE input2, ...
    // error message holder
    char *msg
) ;
```

**Figure 11:** General function header for an LAGraph algorithm.

## 2.2 Peer Pressure Implementation

The heart of the Peer Pressure algorithm lies in the notion of vertices "voting" for their immediate neighbors to be in the cluster in which they reside. In order to capture this idea with a linear algebraic formulation, consider the following. Suppose $G = (V, E)$ is a graph with $n$ vertices and let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be its adjacency matrix representation. Furthermore, suppose $\mathcal{C}_G$ is some clustering of $G$. Define $\mathbf{C} \in \mathbb{B}^{N\times N}$ to be the cluster matrix where $C_{ij} = 1$ if and only if $v_j \in C_i$ and define $\mathbf{T} \in \mathbb{R}^{n\times n}$ to be the tally matrix where $T_{ij} = k$ implies there are $k$ votes for $v_j$ to be included in $C_i$. This formulation was originally posed by E. Robinson in Chapter 6 of "Graph Algorithms in the Language of Linear Algebra" [10, 11].

With this formulation, the voting phase can be easily represented as $\mathbf{T} = \mathbf{C} + .\ \mathrm{second}\mathbf{A}$. To see this intuitively, recall Equation 4 which describes a sparse matrix-matrix multiply

$$\begin{aligned} \mathbf{T} &= \mathbf{C} + .\ \mathrm{second}\mathbf{A} \\ T_{ij} &= \sum_{k\in\mathcal{H}_i\cap\mathcal{K}_j} \mathrm{second}(C_{ik}, A_{kj}) \\ &= \sum_{k\in\mathcal{H}_i\cap\mathcal{K}_j} A_{kj}, \end{aligned} \tag{8}$$

recalling that $\mathcal{H}_i$ denotes the collection of column indices of nonzero entries in row $i$ and $\mathcal{K}_j$ denotes the collection of row indices of nonzero entries in column $j$. Note that the "second" binary operator works for this operation since all entries in $\mathbf{C}$ are 1, so this is effectively the same as using the traditional plus-times semiring. In fact, using the plus-second semiring is actually faster, since values of $\mathbf{C}$ need not be accessed. For a particular vertex $v_j$, Equation 8 captures the notion of tallying up all votes from neighbors in cluster $C_i$ and letting that number the the overall vote for $v_j$ to gain membership into cluster $C_i$.

The function header for the PPC algorithm implementation is shown in Figure 12.

```
int LAGr_PeerPressureClustering(
    // output:
    GrB_Vector *c_f,      // output cluster vector
    // input:
    bool normalize,       // if true, normalize the input graph via out-degree
    bool make_undirected, // if true, make G undirected which generally leads
    ↪  to a coarser partitioning
    double thresh,        // convergence threshold
    int max_iter,         // maximum number of iterations
    LAGraph_Graph G,      // input graph
    char *msg
) ;
```

**Figure 12:** Function header for the Peer Pressure Algorithm LAGraph implementation.

In order to ensure that each vertex has some initial desire to stay in its own cluster, it is desirable to ensure that each vertex has a self-loop, i.e., each vertex gets to vote for itself to remain in its current cluster. This can be achieved with the following code:

```
1  // ones := vector of length n of all 1
2  GrB_Vector_assign(ones, NULL, NULL, 1, GrB_ALL, n, NULL);
3  GrB_Matrix_diag(&I, ones, 0);
4  GrB_Matrix_eWiseAdd(A, A 13.1, NULL, GrB_PLUS_FP64, A, I, GrB_DESC_SC 13.2);
```

**Figure 13:** Adding self-edges to a graph.

In line 2 of Figure 13, the `GrB_Vector_assign` method assigns the floating point value 1 to every index (`GrB_ALL`) for $n$ entries. This then allows us to create the identity matrix $\mathbf{I}_n$ via a call to the `GrB_Matrix_diag` which sets the diagonal of the input matrix to the entries in `ones`. Line 4 captures the idea of adding a self-loop to each vertex by performing an element-wise addition of the elements of $\mathbf{A}$ and $\mathbf{I}_n$ *only* where the values of $\mathbf{A}$ are empty, which prevents any modifications to a vertex which already has a self-loop. This idea of limiting where a computation takes place is called *masking* and is extremely important for many GraphBLAS operations as it can speed up certain computations. In line 4, the matrix `A` 13.1 is passed as the mask and the descriptor `GrB_DESC_SC` 13.2 indicates that the mask should be complemented and should be structural instead of valued, i.e., perform the operation only where $\mathbf{A}$ has no existing entries. Again, in this case the mask is used in order to prevent adding 1 to a diagonal element (self-edge) which already existed.

$$
\begin{array}{c|cccccccccc|}
\mathsf{A} & ❶ & ❷ & ❸ & ❹ & ❺ & ❻ & ❼ & ❽ & ❾ & ❿ \\ \hline
❶ & & 1 & 1 & 1 & & & & & & \\
❷ & & & 1 & & & & & & & \\
❸ & & & & & & & & & & \\
❹ & & 1 & 1 & & 1 & & & & & \\
❺ & & & & & & 1 & 1 & & & \\
❻ & & & & & & & 1 & & & \\
❼ & & & & & 1 & & & & & \\
❽ & & & 1 & & & & & & 1 & 1 \\
❾ & & & & & & & & 1 & & 1 \\
❿ & & & & & 1 & & & & 1 & \\ \hline
\end{array}
\oplus
\begin{array}{c|cccccccccc|}
\mathsf{I} & ❶ & ❷ & ❸ & ❹ & ❺ & ❻ & ❼ & ❽ & ❾ & ❿ \\ \hline
❶ & 1 & & & & & & & & & \\
❷ & & 1 & & & & & & & & \\
❸ & & & 1 & & & & & & & \\
❹ & & & & 1 & & & & & & \\
❺ & & & & & 1 & & & & & \\
❻ & & & & & & 1 & & & & \\
❼ & & & & & & & 1 & & & \\
❽ & & & & & & & & 1 & & \\
❾ & & & & & & & & & 1 & \\
❿ & & & & & & & & & & 1 \\ \hline
\end{array}
=
\begin{array}{c|cccccccccc|}
\mathsf{A} & ❶ & ❷ & ❸ & ❹ & ❺ & ❻ & ❼ & ❽ & ❾ & ❿ \\ \hline
❶ & 1 & 1 & 1 & 1 & & & & & & \\
❷ & & 1 & 1 & & & & & & & \\
❸ & & & 1 & & & & & & & \\
❹ & & 1 & 1 & 1 & 1 & & & & & \\
❺ & & & & & 1 & 1 & 1 & & & \\
❻ & & & & & & 1 & 1 & & & \\
❼ & & & & & 1 & & 1 & & & \\
❽ & & & 1 & & & & & 1 & 1 & 1 \\
❾ & & & & & & & & 1 & 1 & 1 \\
❿ & & & & & 1 & & & & 1 & 1 \\ \hline
\end{array}
$$

**Figure 14:** Illustration of the `GrB_Matrix_eWiseAdd` from the code snippet shown in Figure 13. In GraphBLAS notation, this operation may be expressed as $\mathbf{A}\langle\overline{\mathbf{A}}\rangle = \mathbf{A} + \mathbf{I}$.

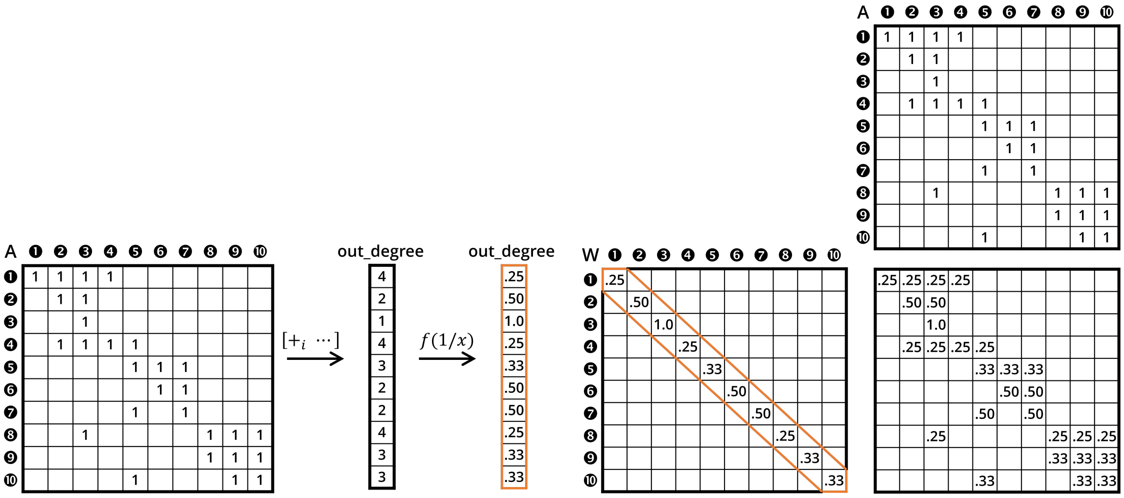
Depending on the desired outcome, it may be preferable to normalize the voting strength of vertices by their out-degree. This is analogous to normalizing each row of the cluster matrix $\mathbf{C}$. This can be achieved with the following code:

```
1      GrB_reduce(out_degree, NULL, NULL, GrB_PLUS_MONOID_INT64, A, NULL);
2      GrB_apply(w_temp, NULL, NULL, GrB_MINV_FP64, out_degree, NULL);
3      GrB_Matrix_diag(&W, w_temp, 0);
4      GrB_mxm(A, NULL, NULL, GrB_PLUS_TIMES_SEMIRING_FP64, W, A, NULL);
```

**Figure 15:** Assuring vertices have equal votes by normalizing weights via out-degrees.

Line 1 from Figure 15 uses the `GrB_reduce` method in order to sum up the rows of `A` using the `PLUS_MONOID` and places the result in the vector `out_degree`. Line 2 then applies the `GrB_MINV` unary operator to `out_degree`. That is, it divides each entry in the vector by 1. Then, line 3 constructs an $n \times n$ matrix `W` with `out_degree` as the main diagonal. Then, line

4 calls `GrB_mxm` to multiply `W` and `A` using the standard `PLUS_TIMES` semiring, which effectively normalizes each row by its sum. Figure 15 illustrates this process.



| A | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ | 1 | 1 | 1 | 1 | | | | | | |
| ❷ | | 1 | 1 | | | | | | | |
| ❸ | | | 1 | | | | | | | |
| ❹ | | 1 | 1 | 1 | 1 | | | | | |
| ❺ | | | | | 1 | 1 | 1 | | | |
| ❻ | | | | | | 1 | 1 | | | |
| ❼ | | | | | 1 | | 1 | | | |
| ❽ | | | 1 | | | | | 1 | 1 | 1 |
| ❾ | | | | | | | | 1 | 1 | 1 |
| ❿ | | | | | 1 | | | | 1 | 1 |

| out_degree | out_degree |
|---|---|
| 4 | .25 |
| 2 | .50 |
| 1 | 1.0 |
| 4 | .25 |
| 3 | .33 |
| 2 | .50 |
| 2 | .50 |
| 4 | .25 |
| 3 | .33 |
| 3 | .33 |

| W | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ | .25 | | | | | | | | | |
| ❷ | | .50 | | | | | | | | |
| ❸ | | | 1.0 | | | | | | | |
| ❹ | | | | .25 | | | | | | |
| ❺ | | | | | .33 | | | | | |
| ❻ | | | | | | .50 | | | | |
| ❼ | | | | | | | .50 | | | |
| ❽ | | | | | | | | .25 | | |
| ❾ | | | | | | | | | .33 | |
| ❿ | | | | | | | | | | .33 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| .25 | .25 | .25 | .25 | | | | | | |
| | .50 | .50 | | | | | | | |
| | | 1.0 | | | | | | | |
| | .25 | .25 | .25 | .25 | | | | | |
| | | | | .33 | .33 | .33 | | | |
| | | | | | .50 | .50 | | | |
| | | | | .50 | | .50 | | | |
| | | .25 | | | | | .25 | .25 | .25 |
| | | | | | | | .33 | .33 | .33 |
| | | | | .33 | | | | .33 | .33 |

**Figure 16:** Illustration of the normalization process described in Figure 15.

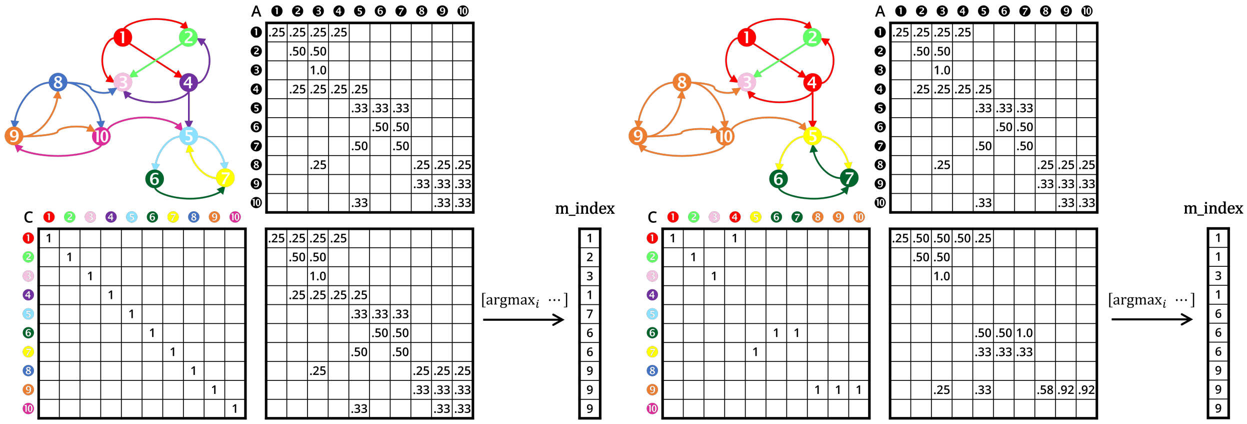
```
GrB_Index iter, num_changed = 0;
while (true)
{
    GrB_mxm(T, NULL, NULL, GxB_PLUS_SECOND_SEMIRING_FP64, C, A, NULL);
    // m_index(j) = argmax(T(:j))
    GrB_Index *m_index_values;
    LAGraph_Malloc((void **)&m_index_values, n, sizeof(GrB_INT64), msg);
    GrB_Vector_extractTuples_INT64(NULL, m_index_values, &n, m_index);
    GrB_Matrix_new(&C_temp, GrB_BOOL, n, n)
    GrB_extract(C_temp 13.4 , NULL, NULL, I 13.3, GrB_ALL 13.1, n,
    ↪  m_index_values 13.2, n, NULL);

    // If the percentage of vertices cluster assignments which have changed
    ↪  since last iteration is below some predefined amount, terminate.
    GrB_eWiseMult(CD, NULL, NULL, GrB_ONEB_BOOL, C, C_temp, NULL);
    GrB_reduce(&num_changed, NULL, GrB_PLUS_MONOID_INT64, CD, NULL);
    num_changed = n - num_changed;
    double percent_updated = num_changed * 1.0 / n;
    // (Pseudocode) Check if percentage falls beneath threshold and if so, set
    ↪  output and terminate.

    GrB_free (&C);
    C = C_temp;
    C_temp = NULL;
    iter++;
}
```

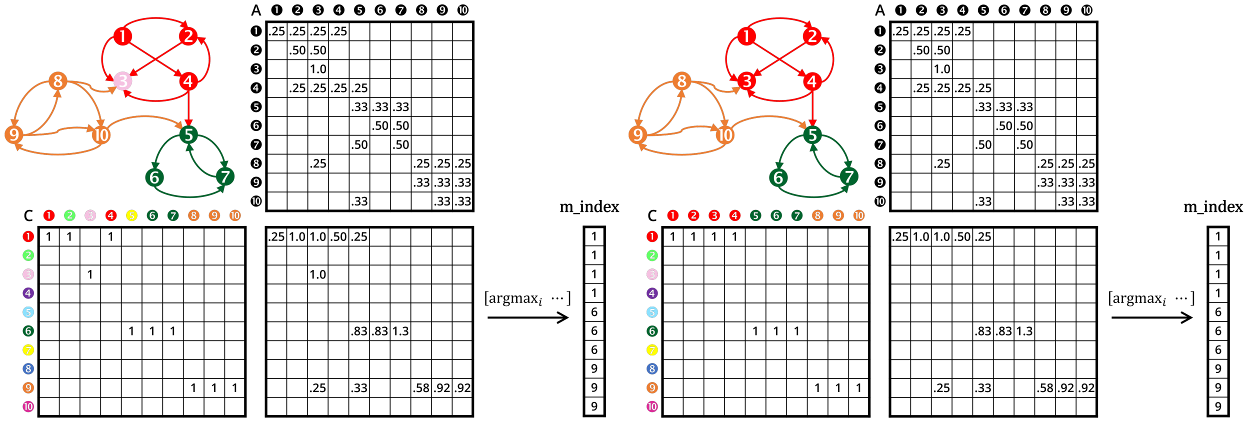
**Figure 17:** Main logic of the SuiteSparse:GraphBLAS implementation of the PPC algorithm.

The main algorithm logic which encapsulates the voting process is straightforward using SuiteSparse:GraphBLAS. The loop in Figure 17 runs until the percentage of vertices that have *changed* cluster assignments between subsequent iterations is below some predefined small threshold at which point the final clustering is achieved.



**(a)** Initial clustering and first iteration.

**(b)** Second iteration.



**(c)** Third iteration.

**(d)** Fourth iteration and final clustering.

**Figure 18:** Example of the PPC algorithm on the working example (Figure 8). **Note**, as computed in Figure 14, each vertex has a self-edge (despite not being shown in the above graphs).

Line 4 from Figure 17 is analogous to Equation 8 and represents the voting process. This call specifies to multiply `C` (cluster matrix) by `A` (normalized adjacency matrix) using the the `PLUS_SECOND` semiring and stores the result in `T` (the tally matrix). Note that `NULL` is passed to the second (mask) and third (accumulator) parameters indicating they are both unused. In general, passing `NULL` to a parameter of a GraphBLAS method indicates it is not used in the corresponding computation.

At this point (after line 4 from Figure 17), the tally matrix holds all votes for the current iteration. Consider our working example and Figure 18a. The third column in the resulting matrix $\mathbf{T}$ indicates that vertex 3 has .25 votes to be in cluster 1, .50 votes to be in cluster 2, 1.0 vote to be in cluster 3, .25 votes to be in cluster 4, and .25 votes to be in cluster 8. Of course, now we must find which cluster of these cast the *most* votes for each vertex. That is, we need to find the argmax over all columns of $\mathbf{T}$.

SuiteSparse:GraphBLAS does not yet have a built-in argmax function. However, such an operation is easily achievable through a thoughtful combination of GraphBLAS methods. The following code describes the argmax functionality and is placed at line 5 of the code in Figure 17.

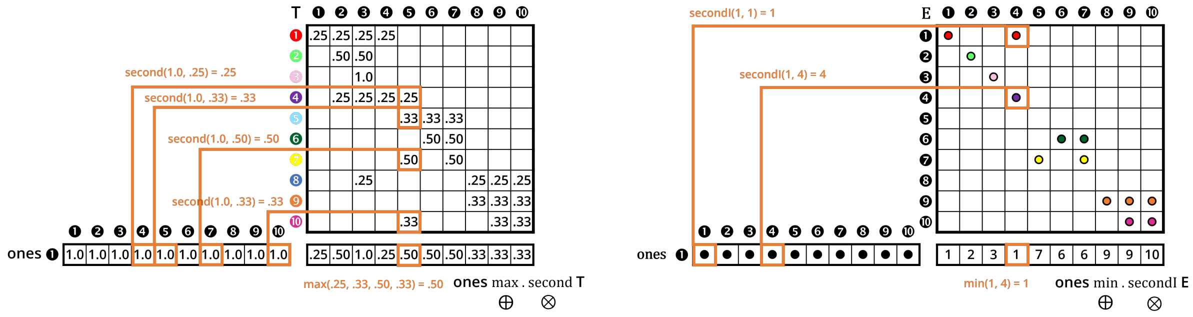
```
1  GrB_vxm(m, NULL, NULL, GrB_MAX_SECOND_SEMIRING_FP64, ones, T, NULL);
2  GrB_Matrix_diag(&D, m, 0);
3  GrB_mxm(E, NULL, NULL, GxB_ANY_EQ_FP64, T, D, NULL);
4  GrB_Matrix_select(E, NULL, NULL, GrB_VALUENE_BOOL, E, 0, NULL);
5  GrB_vxm(m_index, NULL, NULL, GxB_MIN_SECONDI_INT64, ones, E, NULL);
```

**Figure 19:** Argmax (over columns) code used in the PPC algorithm. [12]

Line 1 from Figure 19 finds the maximum *number* of votes among all clusters for each vertex. This is accomplished via a call to `GrB_vxm` with the `MAX_SECOND` semiring which performs a vector-matrix multiplication between `ones` and `T` where the binary operator `SECOND` is defined by $\text{second}(x, y) = y$ and the monoid `MAX` is defined by $z = \max(x, y)$. More formally, we have that

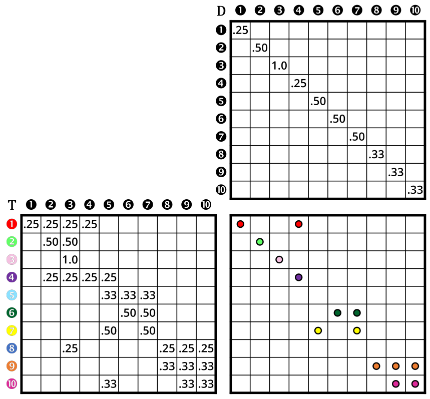
$$\begin{aligned} m_j &= \max_{1 \le k \le n} \{\text{second}(ones_k, T_{kj}\} \\ &= \max_{1 \le k \le n} \{T_{kj}\} \end{aligned} \tag{9}$$

for each index $j$ of the vector $\mathbf{m}$. For this operation, `ones` is a vector of length $n$ of all 1's, although this particular value is actually arbitrary since the `SECOND` binary operator simply takes the value present in `T`. Line 2 initializes the matrix `D` such that its diagonal is equal to `m`. Line 3 finds *which* cluster(s) cast the maximum vote for a particular vertex. This is accomplished by a matrix-matrix multiply between `T` and `D` using the `ANY_EQ` semiring. The `EQ` binary comparator is simply defined as $f(x, y) = 1 \iff x = y$ and the `ANY` monoid is defined by $z = f_{\text{any}}(x, y) = x$ or $y$ and gives GraphBLAS the freedom to choose either $x$ or $y$ arbitrarily. In this case, the matrix `D` is diagonal, so the power of the `ANY` monoid is not fully realized (as there will never be more than one operation in a dot product and thus there will never be an opportunity to choose one value arbitrarily). Line 4 employs the `GrB_select` function to select (keep) only values of `E` which are not equal (`GrB_VALUENE_BOOL`) to 0, i.e., it drops explicit zeros. Finally, line 5 performs a vector-matrix multiplication between `ones` and `E` using the `GxB_MIN_SECONDI` where the positional binary operation `SECONDI` is defined by $z = f(A_{ik}, B_{kj}) = k$ for some matrices $\mathbf{A}$ and $\mathbf{B}$ and the `MIN` monoid is defined by $z = \min(x, y)$. This line captures the minimum row *index* present in each column, i.e., if two clusters cast the same number of votes for a vertex, then the vertex will be subsumed by the cluster with the lowest index.



**(a)** Corresponds to line 1 in Figure 19.

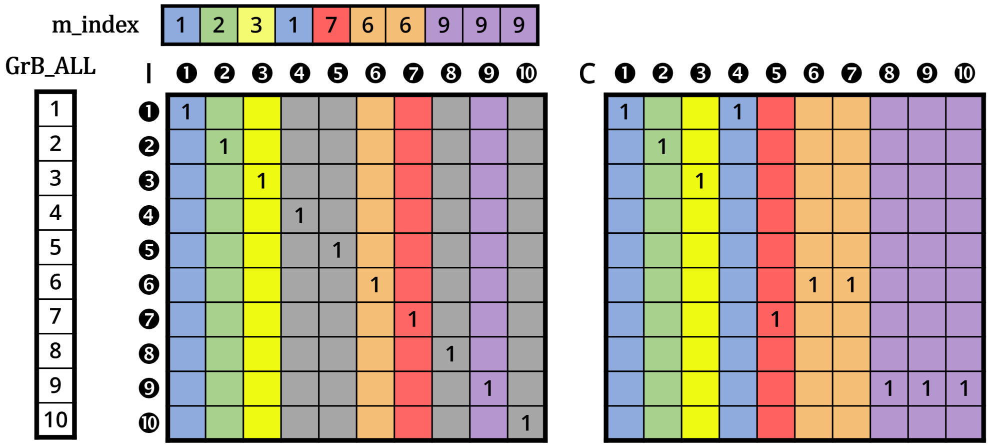
**(c)** Corresponds to line 5 in Figure 19.



**(b)** Corresponds to line 3 in Figure 19.

**Figure 20:** Example of the argmax functionality. The example above takes place between the first and second iterations in the working example (between Figure 18a and Figure 18b).

After the argmax code has identified the cluster which cast the most votes for each vertex, lines 6-10 of the code in Figure 17 assembles the new cluster matrix based on the values in `m_index`. That is, $m\_index_j = k$ implies that $C_{kj} = 1$. Lines 6-7 allocate the array `m_index_values` to hold the $(index, value)$ tuples extracted using the GraphBLAS method `GrB_Vector_extractTuples`. Note, `NULL` is passed to the first parameter as we need not extract the *indices*, only the *values* of `m_index`. Then, line 10 uses the `GrB_extract` function to extract all rows 13.1 and the column indices specified in `m_index_values` 13.2 from the identity matrix `I` 13.3, and places the result in `C_temp` 13.4.



| m_index | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 7 | 6 | 6 | 9 | 9 | 9 |

| GrB_ALL |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |
| 10 |

| I | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ | 1 | | | | | | | | | |
| ❷ | | 1 | | | | | | | | |
| ❸ | | | 1 | | | | | | | |
| ❹ | | | | 1 | | | | | | |
| ❺ | | | | | 1 | | | | | |
| ❻ | | | | | | 1 | | | | |
| ❼ | | | | | | | 1 | | | |
| ❽ | | | | | | | | 1 | | |
| ❾ | | | | | | | | | 1 | |
| ❿ | | | | | | | | | | 1 |

| C | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ | 1 | | | 1 | | | | | | |
| ❷ | | 1 | | | | | | | | |
| ❸ | | | 1 | | | | | | | |
| ❹ | | | | | | | | | | |
| ❺ | | | | | | | | | | |
| ❻ | | | | | | 1 | 1 | | | |
| ❼ | | | | | 1 | | | | | |
| ❽ | | | | | | | | | | |
| ❾ | | | | | | | | 1 | 1 | 1 |
| ❿ | | | | | | | | | | |

**Figure 21:** Example of submatrix extraction using `GrB_extract`. This follows from the working example and is the next step after the argmax procedure laid out in Figure 19.

Finally, lines 13-16 of Figure 17 count the number of vertices which have changed clusters between iterations. When this number falls below a certain threshold (line 17, passed by user), the algorithm terminates. Line 13 uses the `GrB_eWiseMult` method with the `GrB_ONEB_BOOL` binary operator, defined as $f(x, y) = 1$, in order to place a 1 in `CD` at the indices where `C` and `C_temp` intersect (are both 1). After the call, $(CD)_{ij} = 1$ if vertex $j$ *remained* in cluster $i$ between two subsequent iterations. Then, line 14 uses `GrB_reduce` with the typical additive monoid to sum up all entries in the matrix (that is, the total number of vertices which did not change clusters). Then, the number of vertices that *did* change clusters is simply the total number of vertices $n$ minus this number. When the ratio of this number and $n$ falls beneath the user-defined threshold `thresh`, the clustering is deemed stable and the algorithm terminates.

## 2.3 Markov Cluster Implementation

The Markov Cluster Algorithm (MCL) was originally formulated in the language of linear algebra, and therefore lends itself nicely to a GraphBLAS implementation. Furthermore, as the original algorithm is based on matrix multiplication of adjacency matrices over the traditional plus-times semiring, we need not devise our own formulation. In other words, MCL translates almost directly into GraphBLAS. The following code shows the function header for this algorithm.

```
int LAGr_MarkovClustering(
    // output:
    GrB_Vector *c_f,               // output cluster vector
    // input
    int e,                         // expansion coefficient
    int i,                         // inflation coefficient
    double pruning_threshold,      // threshold for pruning values
    double convergence_threshold, // MSE threshold for convergence
    int max_iter,                  // maximum iterations
    LAGraph_Graph G,               // input graph
    char *msg
) ;
```

**Figure 22:** Function header for the Markov Cluster Algorithm LAGraph implementation.

Suppose $G = (V, E)$ is our input graph with $n$ vertices and suppose $\mathbf{A} \in \mathbb{R}^{n \times n}$ is its adjacency matrix representation. As before, the first step is to add a self-edge to each vertex. This can be done via the same code used in Figure 13. Then, the main algorithm logic can begin, which is shown in Figure 23.

Lines 3-6 normalize the columns of the transfer matrix `T_temp`. This is nearly identical to the code in Figure 15, which normalized the *rows* of a matrix. Notice, in this call to `GrB_reduce`, the `GrB_DESC_T0` descriptor 23.1 is used to first transpose the input matrix, effectively reducing across the columns of `T_temp`.

Line 8 uses the `GrB_select` method to keep only the entries in `T_temp` whose values are greater than `pruning_threshold`, which is a user-passed parameter. Unlike the PPC algorithm, the matrix being acted on does not remain sparse throughout the algorithm since MCL squares the transfer matrix in the expansion step. Therefore, while `T` starts as a sparse matrix, it quickly becomes dense due to matrix squaring. Line 8 helps keep the transfer matrix as sparse as possible by dropping negligible entries. Of course, this ultimately leads to a different and perhaps less accurate clustering than with no pruning.

```
1  while (true)
2  {
3      GrB_reduce(w, NULL, NULL, GrB_PLUS_MONOID_FP32, T_temp, GrB_DESC_T0 23.1);
4      GrB_apply(w, NULL, NULL, GrB_MINV_FP32, w, NULL);
5      GrB_Matrix_diag(&D, w, 0);
6      GrB_mxm(T_temp, NULL, NULL, GrB_PLUS_TIMES_SEMIRING_FP32, T_temp, D,
       ↪  NULL);
7
8      GrB_select(T_temp, NULL, NULL, GrB_VALUEGT_FP32, T_temp,
       ↪  pruning_threshold, NULL);
9
10     // Compute MSE between subsequent iteration transfer matrices
11     GxB_Matrix_eWiseUnion(MSE, NULL, NULL, GrB_MINUS_FP32, T_temp, zero_FP32
       ↪  23.2, T, zero_FP32 23.3, NULL);
12     GrB_eWiseMult(MSE, NULL, NULL, GrB_TIMES_FP32, MSE, MSE, NULL);
13     GrB_reduce(&mse, NULL, GrB_PLUS_MONOID_FP32, MSE, NULL);
14     GrB_Matrix_nvals(&nvals, MSE);
15     mse /= nvals;
16
17     if (iter > max_iter || mse < convergence_threshold) break;
18
19     // Expansion step
20     for (int i = 0; i < e - 1; i++)
21     {
22         GrB_mxm(T_temp, NULL, NULL, GrB_PLUS_TIMES_SEMIRING_FP32, T_temp,
           ↪  T_temp, NULL);
23     }
24
25     // Inflation step
26     GrB_Matrix_apply_BinaryOp2nd_FP32(T_temp, NULL, NULL, GxB_POW_FP32,
       ↪  T_temp, (double)i, NULL);
27
28     iter++;
29 }
```

**Figure 23:** Main algorithm logic of Markov Cluster Algorithm LAGraph implementation.

Lines 11-15 compute the mean squared error (MSE) of two subsequent iterations of the transfer matrix. Since MCL is not always guaranteed to converge, the MSE between iterations gives an idea of how stable the transfer matrix is at a given point. Given two matrices, $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$

where $\mathbf{A}$ has $p \leq n^2$ nonzero entries and $\mathbf{B}$ has $q \leq n^2$ nonzero entries, the MSE is defined as

$$\text{MSE} = \frac{1}{k}\sum_{i,j}(A_{ij} - B_{ij})^2, \tag{10}$$

where $k = \max\{p, q\}$.[1] Line 11 uses `GxB_Matrix_eWiseUnion` in order to calculate the element-wise subtraction `T_temp - T`. Note that `eWiseUnion` is not the same method as `eWiseAdd`, which we have used before. With this method, the additional parameters $\alpha$ 23.2 and $\beta$ 23.3 are passed which define the inputs to the binary operator (in this case `MINUS_FP32`) when entries are present in one of `T` or `T_temp` but not the other. More formally, this specifies that [12]

for all entries $(i, j)$ in `T` $\cap$ `T_temp`
    `MSE(i, j)` = `T(i, j) - T_temp(i, j)`

for all entries $(i, j)$ in `T` $\setminus$ `T_temp`
    `MSE(i, j)` = `T(i, j) -` $\beta$

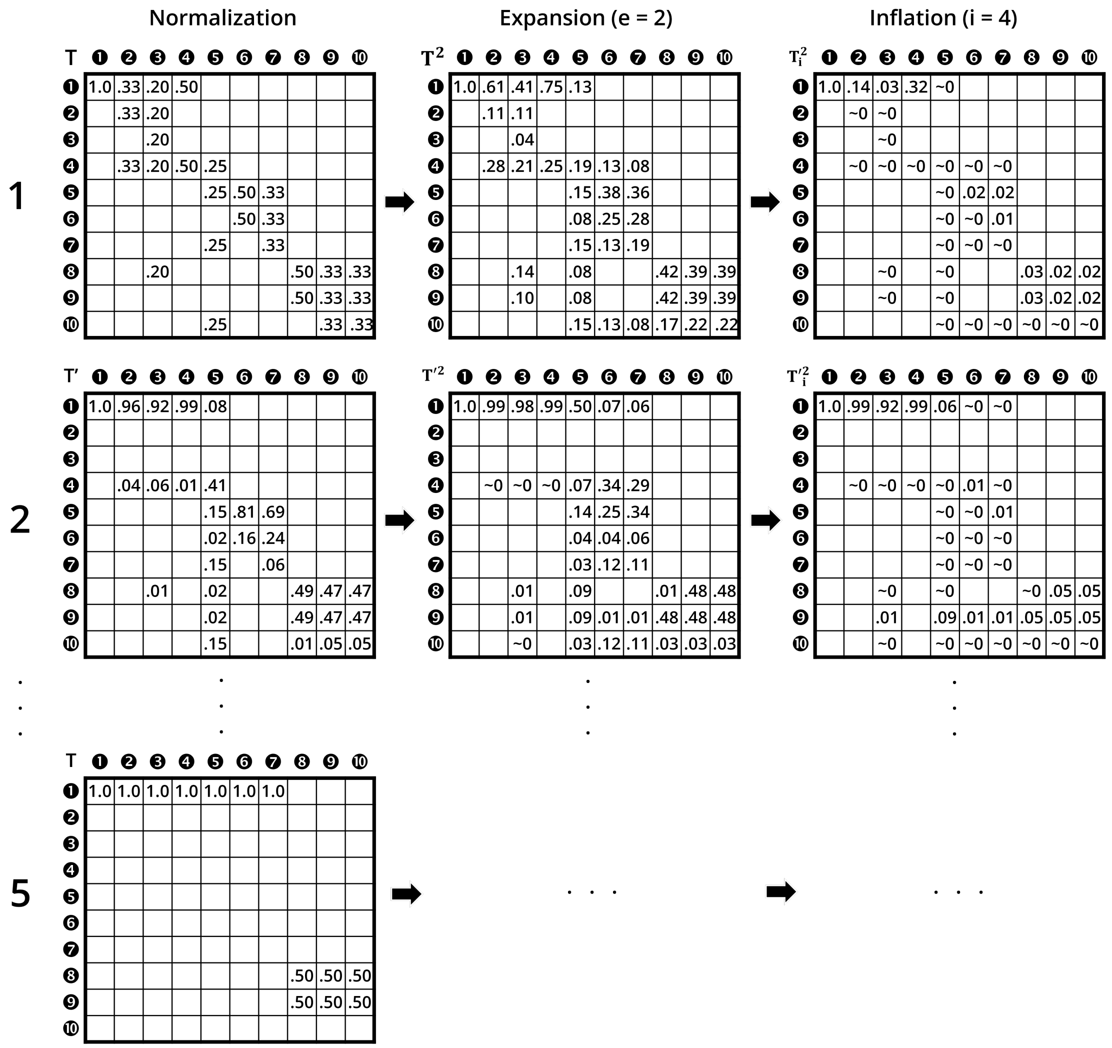
for all entries $(i, j)$ in `T_temp` $\setminus$ `T`
    `MSE(i, j)` = $\alpha$ `- T_temp(i, j).`

In this case, $\alpha$ and $\beta$ are defined as `zero_FP32` which is a user defined `GrB_Scalar` which simply holds the value $0.0$. Line 12 uses `GrB_eWiseMult` to perform element-wise multiplication of `MSE` with itself, effectively squaring each entry. Then, line 13 uses `GrB_reduce` with the `PLUS_MONOID` to sum up all entries in `MSE` which is subsequently divided by the total number of nonzero entries (`nvals`) in `MSE` to obtain the MSE. Line 17 then checks if this MSE falls below the user-defined threshold or if the algorithm has reached the maximum number of iterations and in both cases, the loop terminates.

Lines 20-23 represent the expansion step, and simply uses `GrB_mxm` to multiply `T_temp` by itself `e - 1` times, effectively raising it to the `e`th power. Finally, line 26 represents the inflation step and raises each entry in the transfer matrix to the `i`th power using the `GrB_Matrix_apply_BinaryOp2nd` method. This method applies the `POW_FP32` binary operator, defined by $f(x, y) = x^y$, to each element of the matrix `T_temp` where the input scalar `i` is taken as the second argument of the operator ($y$) and the elements of `T_temp` are taken as the first argument ($x$).

---

[1] The use of $k$ in this context is not standard in the calculation of MSE. For our purposes, it makes sense to consider only the values which are nonzero, since we are dealing with sparse matrices.



| T | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ | 1.0 | .33 | .20 | .50 | | | | | | |
| ❷ | | .33 | .20 | | | | | | | |
| ❸ | | | .20 | | | | | | | |
| ❹ | | .33 | .20 | .50 | .25 | | | | | |
| ❺ | | | | | .25 | .50 | .33 | | | |
| ❻ | | | | | | .50 | .33 | | | |
| ❼ | | | | | .25 | | .33 | | | |
| ❽ | | | .20 | | | | | .50 | .33 | .33 |
| ❾ | | | | | | | | .50 | .33 | .33 |
| ❿ | | | | | .25 | | | | .33 | .33 |

| $T^2$ | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ | 1.0 | .61 | .41 | .75 | .13 | | | | | |
| ❷ | | .11 | .11 | | | | | | | |
| ❸ | | | .04 | | | | | | | |
| ❹ | | .28 | .21 | .25 | .19 | .13 | .08 | | | |
| ❺ | | | | | .15 | .38 | .36 | | | |
| ❻ | | | | | .08 | .25 | .28 | | | |
| ❼ | | | | | .15 | .13 | .19 | | | |
| ❽ | | | .14 | | .08 | | | .42 | .39 | .39 |
| ❾ | | | .10 | | .08 | | | .42 | .39 | .39 |
| ❿ | | | | | .15 | .13 | .08 | .17 | .22 | .22 |

| $T_i^2$ | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ | 1.0 | .14 | .03 | .32 | ~0 | | | | | |
| ❷ | | ~0 | ~0 | | | | | | | |
| ❸ | | | ~0 | | | | | | | |
| ❹ | | ~0 | ~0 | ~0 | ~0 | ~0 | ~0 | | | |
| ❺ | | | | | ~0 | .02 | .02 | | | |
| ❻ | | | | | ~0 | ~0 | .01 | | | |
| ❼ | | | | | ~0 | ~0 | ~0 | | | |
| ❽ | | | ~0 | | ~0 | | | .03 | .02 | .02 |
| ❾ | | | ~0 | | ~0 | | | .03 | .02 | .02 |
| ❿ | | | | | ~0 | ~0 | ~0 | ~0 | ~0 | ~0 |

| T′ | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ | 1.0 | .96 | .92 | .99 | .08 | | | | | |
| ❷ | | | | | | | | | | |
| ❸ | | | | | | | | | | |
| ❹ | | .04 | .06 | .01 | .41 | | | | | |
| ❺ | | | | | .15 | .81 | .69 | | | |
| ❻ | | | | | .02 | .16 | .24 | | | |
| ❼ | | | | | .15 | | .06 | | | |
| ❽ | | | .01 | | .02 | | | .49 | .47 | .47 |
| ❾ | | | | | .02 | | | .49 | .47 | .47 |
| ❿ | | | | | .15 | | | .01 | .05 | .05 |

| $T'^2$ | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ | 1.0 | .99 | .98 | .99 | .50 | .07 | .06 | | | |
| ❷ | | | | | | | | | | |
| ❸ | | | | | | | | | | |
| ❹ | | ~0 | ~0 | ~0 | .07 | .34 | .29 | | | |
| ❺ | | | | | .14 | .25 | .34 | | | |
| ❻ | | | | | .04 | .04 | .06 | | | |
| ❼ | | | | | .03 | .12 | .11 | | | |
| ❽ | | | .01 | | .09 | | | .01 | .48 | .48 |
| ❾ | | | .01 | | .09 | .01 | .01 | .48 | .48 | .48 |
| ❿ | | | ~0 | | .03 | .12 | .11 | .03 | .03 | .03 |

| $T_i'^2$ | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ | 1.0 | .99 | .92 | .99 | .06 | ~0 | ~0 | | | |
| ❷ | | | | | | | | | | |
| ❸ | | | | | | | | | | |
| ❹ | | ~0 | ~0 | ~0 | ~0 | .01 | ~0 | | | |
| ❺ | | | | | ~0 | ~0 | .01 | | | |
| ❻ | | | | | ~0 | ~0 | ~0 | | | |
| ❼ | | | | | ~0 | ~0 | ~0 | | | |
| ❽ | | | ~0 | | ~0 | | | ~0 | .05 | .05 |
| ❾ | | | .01 | | .09 | .01 | .01 | .05 | .05 | .05 |
| ❿ | | | ~0 | | ~0 | ~0 | ~0 | ~0 | ~0 | ~0 |

| T | ❶ | ❷ | ❸ | ❹ | ❺ | ❻ | ❼ | ❽ | ❾ | ❿ |
|---|---|---|---|---|---|---|---|---|---|---|
| ❶ | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | | | |
| ❷ | | | | | | | | | | |
| ❸ | | | | | | | | | | |
| ❹ | | | | | | | | | | |
| ❺ | | | | | | | | | | |
| ❻ | | | | | | | | | | |
| ❼ | | | | | | | | | | |
| ❽ | | | | | | | | .50 | .50 | .50 |
| ❾ | | | | | | | | .50 | .50 | .50 |
| ❿ | | | | | | | | | | |

**Figure 24:** Example of MCL on our working example.

Upon convergence, the steady-state version of the $\mathbf{T}$ may be interpreted as follows. Let an *attractor* vertex be any vertex such that its corresponding row in $\mathbf{T}$ has at least one positive value. Each attractor *attracts* the vertices within the row of the attractor which also have positive values (including itself). Such vertices correspond to column indices of $\mathbf{T}$. To compute the final cluster vector, we can take the argmax over the columns of $\mathbf{T}$ and place them into a vector $\mathbf{v}$ with the property $\mathbf{v}_i = k$ if and only if vertex $i$ is in cluster $k$. When two attractors attract the same vertices with the same strength, the minimum index vertex is taken as the attractor. For example, in the steady-state transfer matrix (bottom left) in Figure 24, the attractors are vertices $1$ and $8$ (vertices $8$ and $9$ attract the same vertices with the same strength, so $\min\{8, 9\} = 8$ is taken as the attractor) which attract vertices $\{2, 3, 4, 5, 6, 7\}$, $\{8, 9, 10\}$, respectively. Then, the final clustering obtained is $\mathcal{C} = \{\{1, 2, 3, 4, 5, 6, 7\}, \{8, 9, 10\}\}$. As shown previously, this computation can be performed using SuiteSparse:GraphBLAS as shown in Figures 19 and 20. At this point, the final cluster vector is returned and the algorithm is complete.

When the pruning threshold is high, some columns in the steady-state transfer matrix are empty, i.e., they are not attracted to any other vertex. If the algorithm terminates and there are fewer entries in $\mathbf{T}$ than there were to begin with, the vertices associated with those columns which are missing a positive entry are assigned arbitrarily to the cluster associated with their index.

# 3. RESULTS

In this chapter, we survey the results obtained while testing our implementations. In doing so, we will discuss both the algorithmic *efficiency* (which will showcase the power SuiteSparse:GraphBLAS API) of the programs as well as the *quality* of the clusterings which they produce.

## 3.1 Quality Metrics

As mentioned in Chapter 1, it is difficult to qualitatively say what makes a particular clustering $\mathcal{C}$ of a graph $G = (V, E)$ "good." This is largely due to the fact that the very definition of a cluster/community is often heavily dependent on the type of network being analyzed. For instance, the types of communities which emerge in a large-scale network of proteins will almost certainly be vastly different than those which emerge in an Instagram social network. However, many *quality metrics* (or *functions*) have been proposed to quantitatively define the quality of a cluster.

### *3.1.1 Quality Metrics: Performance, Coverage, and Modularity*

Despite the context-dependent nature of cluster quality, intuition alone suggests that any good cluster should have more intra-cluster edges than inter-cluster edges. In other words, a good clustering should have many edges connecting vertices within the same cluster and relatively few edges connecting vertices in different clusters. In this section, we will discuss three basic quality metrics which build on this very principle: *coverage*, *performance*, and *modularity*.

The *coverage* of a clustering $\mathcal{C}$ is the ratio of the number of intra-cluster edges and the number of total edges and is formally defined as

$$\text{Cov}(\mathcal{C}) = \frac{|E_{intra}|}{|E|}. \tag{11}$$

The *performance*[1] of $\mathcal{C}$ is the ratio of the number of intra-cluster edges plus the number of

---

[1] Note, despite the name, "performance" has no relation to the *algorithmic* performance (i.e., runtime and efficiency) of a graph clustering algorithm. Rather, it is a measure of the quality of the clustering itself.

inter-cluster non-edges and the number of total possible edges in $G$ and is formally defined as [13]

$$\text{Perf}(\mathcal{C}) = \frac{|E_{intra}| + |N_{inter}|}{n(n-1)/2}. \tag{12}$$

Note, if $G$ is directed, the denominator is $n(n-1)$.

Among the most popular quality metrics is *modularity*, denoted as $Q$ [14]. This metric quantifies the strength of division of a graph into communities by comparing the actual density of intra-cluster edges to the density one would expect to find if the edges of the graph were distributed at random (while preserving node degrees) according to some null model. The function is defined formally as

$$Q = \frac{1}{2 \cdot |E|} \sum_{ij} (A_{ij} - P_{ij}) \cdot \delta(v_i, v_j). \tag{13}$$

Here, $\mathbf{A}$ is the adjacency matrix, $\mathbf{P}$ represents the null model where $P_{ij}$ denotes the number of expected edges between $v_i$ and $v_j$, and the delta function is defined as $\delta(v_i, v_j) = 1$ if $v_i$ and $v_j$ are in the same cluster and 0 otherwise. For large-scale networks, the model used for $\mathbf{P}$ is called the *configuration model*. This particular model often chosen as it preserves the degree distribution of the original network while randomizing the actual connections between vertices. The configuration model yields an expected number of $P_{ij} = k_i k_j / 2|E|$ edges between $v_i$ and $v_j$ [13]. Now, we may replace Equation 13 with

$$Q = \frac{1}{2 \cdot |E|} \sum_{ij} \left( A_{ij} - \frac{k_i k_j}{2 \cdot |E|} \right) \cdot \delta(v_i, v_j) \tag{14}$$

where $k_i$ and $k_j$ are the degrees of $v_i$ and $v_j$, respectively. Finally, the only pairs of vertices that contribute to the total modularity are those belonging to the same cluster, therefore these vertices can be grouped together and Equation 14 may be rewritten as [14]

$$Q = \sum_{c=1}^{n_c} \left[ \frac{L_c}{|E|} - \gamma \left( \frac{d_c}{2 \cdot |E|} \right) \right] \tag{15}$$

where $n_c$ denotes the total number of clusters, $L_c$ denotes the number of intra-cluster edges in cluster $c$, $d_c$ denotes the sum of the degrees of all vertices in cluster $c$, and $\gamma$ denotes the resolution parameter. The resolution parameter effectively scales the importance of the null model, but it is common to simply use $\gamma = 1$. When $G$ is directed, $d_c = d_c^+ \cdot d_c^-$. The range of $Q$ is between $-1$ and $1$. When $Q$ is positive, this indicates that the number of intra-cluster edges in each community is on average greater than what would be expected in a random edge distribution. Conversely, $Q \approx 0$ indicates that there is little difference in the intra-cluster density distribution between the actual graph and a random model.

### *3.1.2 Linear Algebraic Formulation*

These quality metrics have straightforward computations and as such, there are some existing implementations. For instance, NetworkX's `partition_quality` function calculates the coverage and performance of a graph in $O(C^2 + L)$ time, where $C$ is the number of communities and $L$ is the number of links [15]. However, a GraphBLAS implementation of these metrics will be important in order to: (1) speed up computations as graphs get large, (2) provide a new approach in addition the sequential methods which already exist, and (3) provide a framework for future contributors to the LAGraph repository as additional graph clustering algorithms are added.

In what follows, let $G = (V, E)$ be a directed graph with $n$ vertices and $\mathcal{C}$ be a clustering of $G$. Note that $G$ need not be directed, and this is only assumed for the sake of explanation. In the following sections, we will explain any modifications to computations which are caused by $G$ being undirected. Furthermore, let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be the adjacency matrix for $G$ and $\mathbf{C} \in \mathbb{B}^{n \times n}$ be the sparse clustering matrix where $C_{ij} = 1 \iff v_j \in C_i$. For the purposes of quality metrics, self-edges are removed since it makes little sense to count a self-loop towards intra-cluster density, therefore assume the diagonal entries of $\mathbf{A}$ are not present.

### *3.1.3 Coverage Metric Implementation*

The linear algebraic formulation for calculating coverage and performance is straightforward and efficient. The matrix product $\mathbf{CA}$ has the property that the value present in entry $(CA)_{ij}$ is equal to the number of incoming edges from vertices in $C_i$ to $v_j$. Next, the product $\mathbf{CAC}^T$ has the property that the value present in entry $(CAC^T)_{ij}$ is equal to the number of edges between cluster $i$ and $j$. Therefore, the entries on the diagonal of this product (where $i = j$) give the number of intra-cluster edges in cluster $i = j$. Then, the total number of intra-cluster edges in $G$ is given by $tr(\mathbf{CAC}^T)$ and then, rewriting Equation 3.1, we obtain

$$Cov(\mathcal{C}) = \frac{tr(\mathbf{CAC}^T)}{|E|}. \tag{16}$$

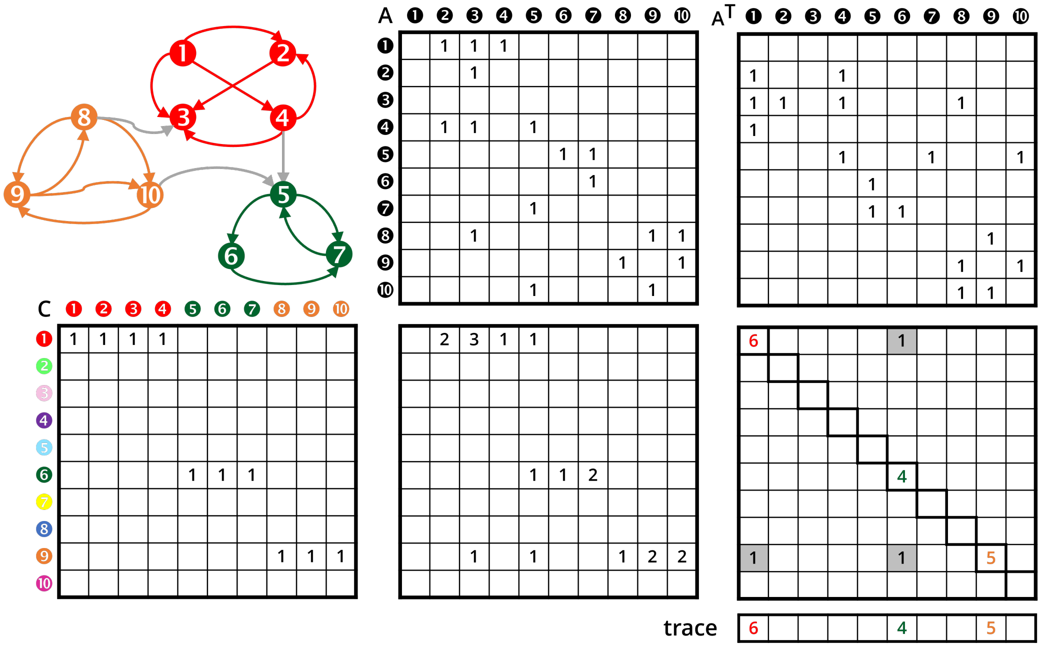
Note, when $G$ is undirected, each edge is counted twice so Equation 16 is divided by 2.

Of course, this formulation is wonderfully and simply expressed using SuiteSparse:GraphBLAS methods as shown in Figure 25.

```
1  GrB_Matrix_select(A, NULL, NULL, GrB_OFFDIAG, A, 0, NULL);
2  GrB_mxm(CA, NULL, NULL, GrB_PLUS_TIMES_SEMIRING_INT64, C, A, NULL);
3  GrB_mxm(CA, NULL, NULL, GrB_PLUS_TIMES_SEMIRING_INT64, CA, C 25.2, GrB_DESC_T1
   ↪  25.1);
4  GxB_Vector_diag(trace, CA, 0, NULL);
5  GrB_Vector_reduce_INT64(&n_intraEdges, NULL, GrB_PLUS_MONOID_INT64, trace,
   ↪  NULL);
6  // If undirected, divide coverage by 2.
7  double coverage = (double)n_intraEdges / nedges;
```

**Figure 25:** Calculating coverage in SuiteSparse:GraphBLAS.

Line 1 calls the `GrB_Matrix_select` method with the `GrB_IndexUnaryOp` `GrB_OFFDIAG` which selects only the entries in `A` off of the 0th diagonal (the main diagonal). This operation is analogous to removing the self-edges in $G$. Lines 2-3 calculate $\mathbf{CAC}^T$ using the `GrB_mxm` method with the traditional semiring. Notice, line 3 uses the `GrB_DESC_T1` 25.1 descriptor in order to transpose the first (0-based) input `C` 25.2. Line 4 uses `GxB_Vector_diag` to extract the 0th diagonal from `CA` (which now holds the product $\mathbf{CAC}^T$) and stores it in the vector `trace`. The call to `GrB_Vector_reduce_INT64` in line 5 sums up all the elements of `trace` using the standard monoid and stores the value in `n_intraEdges`. Now, coverage can be computed directly by dividing the total number of intra-cluster edges (`n_intraEdges`) by the total number of edges (`nedges`).



| A | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 |  | 1 | 1 | 1 |  |  |  |  |  |  |
| 2 |  |  | 1 |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |
| 4 |  | 1 | 1 |  | 1 |  |  |  |  |  |
| 5 |  |  |  |  |  | 1 | 1 |  |  |  |
| 6 |  |  |  |  |  |  | 1 |  |  |  |
| 7 |  |  |  |  | 1 |  |  |  |  |  |
| 8 |  |  | 1 |  |  |  |  |  | 1 | 1 |
| 9 |  |  |  |  |  |  |  | 1 |  | 1 |
| 10 |  |  |  |  | 1 |  |  |  | 1 |  |

| $A^T$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 |  |  |  |  |  |  |  |  |  |  |
| 2 | 1 |  |  | 1 |  |  |  |  |  |  |
| 3 | 1 | 1 |  | 1 |  |  |  | 1 |  |  |
| 4 | 1 |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  | 1 |  |  | 1 |  |  | 1 |
| 6 |  |  |  |  | 1 |  |  |  |  |  |
| 7 |  |  |  |  | 1 | 1 |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  | 1 |  |
| 9 |  |  |  |  |  |  |  | 1 |  | 1 |
| 10 |  |  |  |  |  |  |  | 1 | 1 |  |

| C | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  | 1 | 1 | 1 |  |  |  |
| 7 |  |  |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  | 1 | 1 | 1 |
| 10 |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  | 2 | 3 | 1 | 1 |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  | 1 | 1 | 2 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  | 1 |  | 1 |  |  | 1 | 2 | 2 |
|  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|
| 6 |  |  |  |  | 1 |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  | 4 |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
| 1 |  |  |  |  | 1 |  |  | 5 |  |
|  |  |  |  |  |  |  |  |  |  |

| trace |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
|  | 6 |  |  |  |  | 4 |  |  | 5 |  |

**Figure 26:** Calculating the coverage of the final clustering from Figure 18d.

### *3.1.4 Performance Metric Implementation*

In order to compute performance, the number of inter-cluster non-edges ($|N_{inter}|$) must be calculated. The most obvious solution would be to construct the matrix $\mathbf{A}'$ to be the complement of $\mathbf{A}$, and then $tr(\mathbf{C}\mathbf{A}'\mathbf{C}^T)$ would equal $|N_{inter}|$. However, given $\mathbf{A}$ is sparse, $\mathbf{A}'$ is incredibly dense, leading to a very expensive computation of $\mathbf{C}\mathbf{A}'\mathbf{C}^T$. This method is not practical for very large graphs. Instead, we calculate $|N_{inter}|$ directly using the values previously computed. Note that $|E| = |E_{intra}| + |E_{inter}|$ and $E_{intra} \cap E_{inter} = \emptyset$ which implies that $|E_{inter}| = |E| - |E_{intra}|$. Recall that there are $n(n-1)$ possible edges in $G$ and by the same logic, for each cluster $C_i \in \mathcal{C}$, there are $|C_i|(|C_i| - 1)$ possible intra-cluster edges. Taking the sum over all clusters, we obtain

$$\begin{aligned} K &= \sum_{i=1}^{n_c} \left(|C_i|^2 - |C_i|\right) \\ &= \sum_{i=1}^{n_c} |C_i|^2 - \sum_{i=1}^{n_c} |C_i| \\ &= \left(\sum_{i=1}^{n_c} |C_i|^2\right) - n, \end{aligned} \tag{17}$$

the number of possible intra-cluster edges. Furthermore, we have that $E_{inter} \cap N_{inter} = \emptyset$ and $|E_{inter}| + |N_{inter}| = n(n-1) - K$. Therefore, we have

$$\begin{aligned} |E_{inter}| + |E_{inter}| &= n(n-1) - K \\ \iff \qquad |N_{inter}| &= n(n-1) - K - |E_{inter}| \\ &= n(n-1) - K - (|E| - |E_{intra}|). \end{aligned} \tag{18}$$

Note that when $G$ is undirected, each edge is counted twice so there are $n(n-1)/2$ possible edges and $K/2$ possible intra-cluster edges, and hence Equation 18 must be adjusted accordingly. In order to compute this in SuiteSparse:GraphBLAS, we only need to compute $K$ in addition to our computations in Figure 25. The code in Figure 27 shows how to do so.

```
1  GrB_Matrix_reduce_INT64(k, NULL, NULL, GrB_PLUS_MONOID_INT64, C, NULL);
2  GrB_Vector_apply_BinaryOp2nd_INT64(k, NULL, NULL, GxB_POW_INT64, k, 2, NULL);
3  GrB_Vector_reduce_INT64(&sum_k2, NULL, GrB_PLUS_MONOID_INT64, k, NULL);
4  // Lines 1-5 from Figure 14
5  GrB_Index n_interEdges, n_interNonEdges;
6  n_interEdges = nedges - n_intraEdges;
7  n_interNonEdges = n * (n - 1) - (sum_k2 - n) - n_interEdges;
8  double performance = (double)(i_intraEdges + n_interNonEdges) / (n * (n - 1));
```

**Figure 27:** Calculating performance in SuiteSparse:GraphBLAS.

Again, if the graph is undirected, all edge counts must be divided by 2. Line 1 from Figure 27 uses the `GrB_Matrix_reduce` method in order to reduce (using the standard addition monoid) across all rows of `C` and place the results in the vector `k`. Lines 3-4 are equivalent to the summation in Equation 3.8. First, the method `GrB_apply_BinaryOp2nd` applies the `GxB_POW` binary operator (defined as $f(x, y) = x^y$) where a scalar, in this case 2, is bound to the second input of the operator $f$ effectively squaring each element in `k`. Line 3 then sums up all the values of `k` and places the corresponding value in `sum_k2`, which is analogous to the variable $K$ in Equation 17. Lines 5-8 directly apply the equation derived in Equation 18 to obtain the performance.

### *3.1.5 Modularity Metric Implementation*

Finally, we can calculate modularity in SuiteSparse:GraphBLAS using similar methods. Referring back to Equation 15, the components we must obtain in order to calculate modularity are $L_c$ and $d_c^+, d_c^-$. Notice that $L_c$ was computed using the code in Figure 25 (line 5). In order to compute the combined in and out degree of each cluster, we can follow a similar approach to calculating the number of intra-cluster edges within each cluster. Figure 28 provides the code to do so.

```
1  GrB_reduce(out_degree, NULL, NULL, GrB_PLUS_MONOID_INT64, A, NULL);
2  GrB_reduce(in_degree, NULL, NULL, GrB_PLUS_MONOID_INT64, A, GrB_DESC_T0);
3  GrB_mxv(k_out, NULL, NULL, GrB_PLUS_TIMES_SEMIRING_INT64, C, out_degree,
   ↪  NULL);
4  GrB_mxv(k_in, NULL, NULL, GrB_PLUS_TIMES_SEMIRING_INT64, C, in_degree, NULL);
```

**Figure 28:** Calculating the combined in/out-degrees of clusters in SuiteSparse:GraphBLAS.

Lines 1 and 2 use `GrB_reduce` with the `PLUS_MONOID` in order to calculate the out-degree and in-degree for each vertex and places the results in `out_degree` and `in_degree`, respectively. Then, lines 3 and 4 use a similar process as the one used to calculate the number of intra-cluster *within* each cluster by multiplying the cluster matrix `C` by the in/out-degree vectors using `GrB_mxv`. This gives new `GrB_Vectors` `k_in` and `k_out` which hold the combined out/in-degree for each cluster, respectively.

Finally, in order to calculate modularity according to equation 15, we extract the values from `k_out`, `k_in`, and `l` (the `GrB_Vector` which holds the number of intra-cluster edges for each cluster) using `GrB_Vector_extractTuples` and place them in respective arrays. Then, modularity can be calculated directly using a simple `for` loop, which is shown in Figure 29.

```c
// Extract actual values of l, k_out, and k_in for modularity calculations
GrB_Index *lX, *k_outX, *k_inX;
// allocate memory for arrays on heap using LAGraph_Malloc
GRB_TRY(GrB_Vector_extractTuples_INT64(NULL, lX, &nclusters, l));
GRB_TRY(GrB_Vector_extractTuples_INT64(NULL, k_outX, &nclusters, k_out));
GRB_TRY(GrB_Vector_extractTuples_INT64(NULL, k_inX, &nclusters, k_in));

GrB_Index m, out_degree_sum;
GRB_TRY(GrB_reduce(&out_degree_sum, NULL, GrB_PLUS_MONOID_INT64, out_degree,
↪  NULL));

m = out_degree_sum;
double norm = 1.0 / (m * m);

// Compute modularity
double mod = 0.0;
for (int c = 0; c < nclusters; c++)
{
    mod += (1.0 * lX[c] / nedges) - (gamma * ((k_outX[c] * k_inX[c]) * norm));
}
```

**Figure 29:** Calculating modularity in SuiteSparse:GraphBLAS.

### 3.2 Benchmarking Results

The following results were benchmarked on Texas A&M's `BACKSLASH` system, featuring an Intel Xeon E5-2695 v2 CPU with 24 cores at 2.40GHz, capable of turbo speeds up to 3.20GHz, and 60 MiB of L3 cache, designed for high-performance computing tasks. The sparse matrices used for benchmarking are all chosen from the SuiteSparse Sparse Matrix Collection [16].

Table 1 outlines the results of running our implemented algorithms on the com-Youtube, com-LiveJournal, and com-DBLP graphs, which represent different kinds of social networks. Additionally, our implementations were tested against the Community Detection using Label Propagation (CDLP) clustering algorithm, which is a part of the LAGraph repository. Table 2 outlines the results of running the algorithms on directed graphs. Table 4 summarizes the performance of our cluster quality metric implementations ran on clusterings of graphs of various sizes. The rows labelled $n$, $nvals$, and $nclusters$ describe the size of the graphs and clusterings. In particular, $n$ is the number of vertices, $nvals$ is the number of edges, and $nclusters$ is the number of clusters.

**Table 1:** Benchmarking results for undirected graphs.

| | com-Youtube | | | | com-LiveJournal | | | | com-DBLP | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | 1,134,890 | | | | 3,997,962 | | | | 317,080 | | | |
| $nvals$ | 2,987,624 | | | | 34,681,189 | | | | 1,049,866 | | | |
| | PPC3 | PPC4 | MCL | CDLP | PPC3 | PPC4 | MCL | CDLP | PPC3 | PPC4 | MCL | CDLP |
| Time (s) | 6.084 | 2.324 | 18.16 | 22.47 | 39.48 | 50.15 | 54.28 | 79.04 | 2.653 | 0.7592 | 1.596 | 6.006 |
| Cov | 0.7838 | 0.1046 | 0.3241 | 0.6941 | 0.7844 | 0.1649 | 0.1761 | 0.9562 | 0.6251 | 0.3622 | 0.5952 | 0.6438 |
| Perf | 0.9134 | 0.9999 | 0.9997 | 0.8203 | 0.9084 | 0.9999 | 0.9999 | 0.4022 | 0.9996 | 0.9999 | 0.9999 | 0.9970 |
| Mod | 0.6294 | 0.1045 | 0.3238 | 0.4857 | 0.6688 | 0.1648 | 0.1761 | 0.4677 | 0.6240 | 0.3620 | 0.5951 | 0.6393 |
| Avg. Size | 26.74 | 1.355 | 4.893 | 19.69 | 34.87 | 2.119 | 3.922 | 111.4 | 8.963 | 2.151 | 8.328 | 14.02 |

**Table 2:** Benchmarking results for directed graphs.

| | wiki-Topcats | | | | | | email-Eu-core | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | 1,791,489 | | | | | | 1,005 | | | | | |
| $nvals$ | 28,511,807 | | | | | | 25,571 | | | | | |
| | PPC1 | PPC2 | PPC3 | PPC4 | MCL | CDLP | PPC1 | PPC2 | PPC3 | PPC4 | MCL | CDLP |
| Time (s) | 15.204 | 15.90 | 14.73 | 29.29 | 20.93 | 37.37 | 0.0102 | 0.0153 | 0.0118 | 0.0182 | 0.0185 | 0.0648 |
| Cov | 0.7908 | 0.0779 | 0.9378 | 0.2744 | 0.1639 | 0.9387 | 0.9971 | 0.2899 | 0.9609 | 0.3235 | 0.2545 | 1.000 |
| Perf | 0.6454 | 0.9999 | 0.3195 | 0.9934 | 0.9985 | 0.3008 | 0.1419 | 0.9722 | 0.2636 | 0.9666 | 0.9524 | 0.0621 |
| Mod | 0.2212 | 0.0775 | 0.1260 | 0.1652 | 0.1630 | 0.1357 | 0.0000 | 0.2422 | 0.0792 | 0.2698 | 0.2126 | 0.0000 |
| Avg. Size | 37.44 | 1.795 | 569.4 | 2.223 | 10.20 | 755.9 | 23.92 | 2.512 | 43.69 | 3.073 | 4.975 | 50.25 |

**Table 3:** Description of column labels for Tables 1 and 2.

| | Keep Edge Weights as Is | Normalize Edge Weights via Out-Degree |
|---|---|---|
| Keep Directed | PPC1 | PPC2 |
| Make Undirected | PPC3 | PPC4 |

Table 1 compares the algorithms when run on undirected graphs, i.e., graphs with a symmetric adjacency matrix. PPC was run with the parameters `thresh = 0.0001` and `max_iter = 50` while MCL was run with the parameters `e = i = 2`, `pruning_threshold = 0.00025`, `convergence_threshold = 1.0e-8`, and `max_iter = 50`. The columns PPC3 and PPC4 differ in that PPC3 runs our implementation of the Peer Pressure algorithm with normalizing weights via out-degrees of vertices (as shown in the working example) while PPC4 does not perform this normalization. In general, our results show that when running PPC on an undirected graph, not normalizing edge weights via out-degree yields coarser partitions and therefore more reasonable clusterings. However, depending on the application, it may be more favorable to find finer clusterings in which case normalizing the weights may be beneficial. Our MCL implementation gives reasonable clusterings when compared to CDLP and PPC when run with its current configuration. Not shown in Table 1 is that MCL scales poorly as graphs get even larger due to the matrix squaring which is involved, which yields dense matrices not suitable for GraphBLAS.

Table 2 compares the algorithms when run on directed graphs. Both algorithms were run with the same parameters as described in the preceding paragraph. As summarized in Table 3, PPC1 considers the directed structure and does not normalize vertex weights via out-degree, PPC2 considers the directed structure and does normalize vertex weights via out-degree, PPC3 considers the underlying undirected structure and does not normalize vertex weights via out-degree, and PPC4 considers the underlying undirected structure and normalizes vertex weights via out-degree. The results indicate that most configurations give reasonable clusterings efficiently, however it is evident that particular configurations/algorithms have varying results depending on the input graph. And again, the best configuration to use will largely be dependent on the context of the graph clustering problem. For instance, it is interesting to note that the PPC3 configuration and CDLP give similar clustering results, however PPC3 runs in under half the time. Moreover, while MCL gives smaller clusters when run on directed graphs, this may be useful for particular problems.

**Table 4:** Quality metric runtime comparison between our implementations using SuiteSparse:GraphBLAS (GB) and NetworkX's implementations using Python (NX). The Speedup is the ratio of NetworkX's runtime and our runtime.

| | email-Enron | | | com-Amazon | | | com-Youtube | | | com-LiveJournal | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | 36,692 | | | 334,863 | | | 1,134,890 | | | 3,997,962 | | |
| $nvals$ | 183,831 | | | 925,872 | | | 2,987,624 | | | 34,681,189 | | |
| $nclusters$ | 2,178 | | | 38,662 | | | 42,438 | | | 114,636 | | |
| | GB | NX | Speedup | GB | NX | Speedup | GB | NX | Speedup | GB | NX | Speedup |
| Partition Quality (s) | 0.1167 | 0.4044 | 3.465 | 0.1167 | 91.46 | 783.7 | 0.4414 | 116.3 | 263.4 | 2.707 | 834.1 | 308.1 |
| Modularity (s) | 0.1119 | 0.1838 | 1.642 | 0.1119 | 1.941 | 17.34 | 0.4901 | 6.417 | 13.09 | 3.602 | 73.10 | 20.29 |

Table 4 compares our partition quality function (which computes both performance and coverage of a clustering) with NetworkX's Python implementation. As expected, our SuiteSparse implementation realizes significant speedup when compared to a sequential implementation. Of course, comparing an algorithm which is written in C to a Python counterpart is not necessarily a "fair fight." Nevertheless, our implementations using SuiteSparse:GraphBLAS prove to be extremely efficient and scalable and provide an alternative to NetworkX’s function.

# 4. CONCLUSION

We conclude that graph clustering algorithms are generally well-suited to a SuiteSparse:GraphBLAS implementation. Moreover, one can expect such an implementation to yield meaningful clusterings in a reasonable amount of time. Most importantly, our implementations use simple, user-level code which can be referenced by subsequent authors in creating graph clustering algorithms in the language of linear algebra.

In particular, our SuiteSparse:GraphBLAS implementation of the Peer Pressure clustering algorithm saw 3-4 times speedup when compared to the existing LAGraph CDLP algorithm, with comparable cluster quality. We have shown that different parameter tunings for the PPC algorithm can have an affect on the granularity of the clustering produced, which may be useful in certain applications. Our MCL implementation also realized approximately 2 times speedup when compared to CDLP algorithm, however we note that this algorithm is less suitable for a linear algebraic formulation as matrix squaring compromises the sparsity of subsequent transfer matrices. Finally, we effectively implemented some quality metrics which consistently outperform NetworkX's equivalent metrics by a factor greater than 10.

There is still *much* research to be conducted on the topic of graph clustering in SuiteSparse:GraphBLAS. For instance, there are lots of graph clustering algorithms such as Louvain community detection and spectral clustering which are perhaps suitable for a GraphBLAS implementation. Furthermore, we hope more work can be done in optimizing our implementations of PPC and MCL. In particular, allowing for more continuous control of granularity for PPC would be helpful in identifying a wider range of clusterings. Finally, there exist many other cluster quality metrics which could possibly be computed using GraphBLAS.

# REFERENCES

[1] R. Diestel, *Graph Theory: 5th edition*. Springer Graduate Texts in Mathematics, Springer-Verlag, © Reinhard Diestel, 2017.

[2] J. Berstel and D. Perrin, *Theory of Codes*. ISSN, Elsevier Science, 1985.

[3] S. E. Schaeffer, "Graph clustering," *Computer Science Review*, vol. 1, no. 1, pp. 27–64, 2007.

[4] V. B. Shah, *An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity*. PhD thesis, UNIVERSITY OF CALIFORNIA Santa Barbara, 2007.

[5] S. Dongen, "Graph clustering by flow simulation," *PhD thesis, Center for Math and Computer Science (CWI)*, 05 2000.

[6] B. Brock, A. Buluç, R. Kimmerer, J. Kitchen, M. Kumar, T. Mattson, S. McMillan, J. Moreira, M. Pelletier, and E. Welch, "The graphblas c api specification: Version 2.1.0." `https://graphblas.org/docs/GraphBLAS-2.1.0.pdf`, 2023. Retrieved 22 December 2023.

[7] T. A. Davis, "Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra," *ACM Trans. Math. Softw.*, vol. 45, dec 2019.

[8] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira, and C. Yang, "Lagraph: A community effort to collect graph algorithms built on top of the graphblas," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 276–284, 2019.

[9] G. Szárnyas, D. A. Bader, T. A. Davis, J. Kitchen, T. G. Mattson, S. McMillan, and E. Welch, "Lagraph: Linear algebra, network analysis libraries, and the study of graph algorithms," 2021.

[10] E. Robinson, *6. Complex Graph Algorithms*, pp. 59–84.

[11] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial and Applied Mathematics, 2011.

[12] T. A. Davis, *SuiteSparse:GraphBLAS User Guide*. Texas A&M University, Texas, Jan. 2024. Version 9.0.1, pp. 287.

[13] S. Fortunato, “Community detection in graphs,” *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010.

[14] M. E. J. Newman and M. Girvan, “Finding and evaluating community structure in networks,” *Phys. Rev. E*, vol. 69, p. 026113, Feb 2004.

[15] A. A. Hagberg, D. A. Schult, and P. J. Swart, “Exploring network structure, dynamics, and function using networkx,” in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught, and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008.

[16] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, dec 2011.