UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CURSO DE CIÊNICA DA COMPUTAÇÃO

EDUARDO GEBER DE MELO ALBUQUERQUE

TRACER PROPAGATION: Interpretable Node Embedding

Recife
2024

EDUARDO GEBER DE MELO ALBUQUERQUE

TRACER PROPAGATION: Interpretable Node Embedding

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Bacharelado em Ciência da Computação.

Orientador: Ricardo Martins de Abreu Silva

Recife

2024

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Albuquerque, Eduardo Geber de Melo.
Tracer Propagation: Interpretable Node Embedding / Eduardo Geber de Melo Albuquerque. - Recife, 2024.
22 p. : il., tab.

Orientador(a): Ricardo Martins de Abreu Silva
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2024.
Inclui referências.

1. Análise de Grafos. 2. Imersão de Nós. 3. Detecção de Comunidades. 4. Análise de Dados. 5. Aprendizagem de Máquina. I. Silva, Ricardo Martins de Abreu. (Orientação). II. Título.

000 CDD (22.ed.)

EDUARDO GEBER DE MELO ALBUQUERQUE

TRACER PROPAGATION: Interpretable Node Embedding

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Bacharelado em Ciência da Computação.

Aprovado em: 25/03/2024

BANCA EXAMINADORA

Prof. Dr. Ricardo Martins de Abreu Silva (Orientador)

Universidade Federal de Pernambuco

Prof. Dr. Silvio de Barros Melo (Examinador Interno)

Universidade Federal de Pernambuco

Tracer Propagation: Interpretable Node Embedding

Eduardo Geber de Melo Albuquerque[1,*] and Ricardo Martins de Abreu e Silva[1]

[1]Informatics Center, Federal University of Pernambuco, Brazil
[*]Corresponding author: egma@cin.ufpe.br

Abstract

Graph analytics, crucial in various domains from social networks to biological systems, has seen a shift towards embedding graph nodes into low-dimensional spaces followed by applying standard machine learning techniques. This paradigm aims to preserve topological node similarity and global network structure in the latent embedding space. We introduce Tracer Propagation, a novel node embedding algorithm which generates interpretable embeddings by propagating continuous values (*tracers*) across the network, a mechanism inspired by the community detection algorithm Label Propagation. We evaluate Tracer Propagation's performance primarily in community detection tasks, utilizing K-means on resulting embeddings and comparing against ground-truth communities as well as standard algorithms like Label Propagation and Leiden. Our experiments show promising results on small graphs, demonstrating Tracer Propagation's effectiveness in capturing community structures and topological similarities. Beyond being only a node embedding algorithm, Tracer Propagation's interpretability — perhaps its most significant feature — enables defining novel node similarity measures, which can be be fed into traditional optimization-based node embedding algorithms and potentially enhance their performance. As a bonus, we introduce Principal Component Selection (PCS), a simple algorithm for dimensionality reduction promoting interpretability and reducing dataset redundancy.

Keywords: Graph Analysis, Node Embedding, Community Detection, Data Analysis, Machine Learning

Resumo

A análise de grafos, crucial em diversos domínios, desde redes sociais até sistemas biológicos, tem passado por uma mudança em direção ao mapeamento (imersão) de nós do grafo para espaços de baixa dimensão, seguido pela aplicação de técnicas de aprendizado de máquina. Esse paradigma visa preservar a similaridade topológica entre os nós e a estrutura global da rede no espaço latente de imersão. Introduzimos o Tracer Propagation, um novo algoritmo de imersão de nós que gera vetores interpretáveis ao propagar valores contínuos (marcadores) pela rede, um mecanismo inspirado no algoritmo de detecção de comunidades Label Propagation. Avaliamos o desempenho do Tracer Propagation principalmente em tarefas de detecção de comunidades, utilizando K-means nos vetores resultantes e comparando com comunidades de referência, bem como com algoritmos padrão como Label Propagation e Leiden. Nossos experimentos mostram resultados promissores em grafos pequenos, demonstrando a eficácia do Tracer Propagation em capturar estruturas comunitárias e similaridades topológicas. Além de ser apenas um algoritmo de incorporação de nós, a interpretabilidade do Tracer Propagation — talvez sua característica mais significativa — permite a definição de novas medidas de similaridade entre nós, que podem ser integradas a algoritmos tradicionais de imersão baseados em otimização e potencialmente melhorar seu desempenho. Como bônus, introduzimos o Principal Component Selection (PCS), um algoritmo simples de redução de dimensionalidade que promove a interpretabilidade e reduz a redundância do conjunto de dados.

Palavras-Chave: Análise de Grafos, Imersão de Nós, Detecção de Comunidades, Análise de Dados, Aprendizagem de Máquina

1. Introduction

1.1 Node embedding

Graph analytics relates to performing analysis on data structured as graphs, a representation that occurs naturally in fields such as brain networks in brain imaging, molecular networks in drug discovery, protein-protein interaction networks in genetics, social networks in social media, bank-asset networks in finance, and publication networks in scientific collaborations. Traditional approaches for graph analytics relied on using the adjacency matrix to extract information directly from the graph topology, by using tools such as path analysis, connectivity analysis, community analysis, and centrality analysis. However, these approaches suffer from high computational requirements and usually do not generalize well when used outside of the context they were originally designed for (Xu, 2021).

In face of these issues, a recent trend that has shown remarkable performance is to first embed the nodes of the graph into a latent low-dimensional space such as Euclidean space, and to then use standard machine learning algorithms to perform tasks such as link prediction, node classification, community detection, among others. Besides lending themselves nicely to machine learning algorithms, node embeddings can also be used to compute node similarity in the original graph with standard similarity measures, such as dot product and cosine distance. The ultimate goal of this type of technique is that the global network structure as represented in the original graph can be translated into similarity among data points in the latent space.

Since DeepWalk (Perozzi et al., 2014) introduced and popularized this new graph analytics paradigm, most of the advanced node embedding algorithms work by solving an unsupervised optimization problem, usually using neural networks, where the vector embeddings are automatically learned so as to approximate the corresponding node similarities as computed from the original graph structure. Specifically, if $\text{Sim}(u_i, u_j)$ is a chosen similarity measure between nodes of the graph and z_i, a vector, is the node embedding of u_i, then the goal is to have $\text{Sim}(u_i, u_j) \approx z_j^T z_i$ for all i, j. In this sense, there are at least three types of node similarity used in the literature:

(1) **First-order similarity**. This relates to neighborhood-level proximity between nodes, defined simply as the weight of the edge between two neighbors (or 0 if the nodes are not neighbors). That is, denoting $s_{ij}^{(1)}$ as the first order-similarity from node i to node j, then $s_{ij}^{(1)} = A[i, j]$, where A is the adjacency matrix of the graph. Targeting only the preservation of first-order similarity in the latent space is insufficient for preserving global network structure (Xu, 2021).

(2) **Second-order similarity**. The second-order similarity from node i to node j, $s_{ij}^{(2)}$, is a measure of how similar node i's neighborhood is to node j's neighborhood. That is, let the first-order proximity of node i be $s_i^{(1)} = [A[i, 1], A[i, 2], \ldots, A[i, n]] = [s_{i1}^{(1)}, s_{i2}^{(1)}, \ldots, s_{in}^{(1)}]$, that is, $s_i^{(1)}$ is node i's weights to all other nodes. Then we could define $s_{ij}^{(2)}$ as the cosine similarity between $s_i^{(1)}$ and $s_j^{(1)}$ (Cai et al., 2018). Setting the objective function of the embedding algorithm to preserve second-order similarity often leads to good results, so that higher-order similarity, defined subsequently, is not used as often (Xu, 2021).

(3) **Higher-order similarity**. k-th order similarity, $s_{ij}^{(k)}$, can be generalized directly as the similarity between $s_i^{(k-1)}$ and $s_j^{(k-1)}$. Some authors also define higher-order similarity in terms of other metrics as well, such as Katz Index, Rooted PageRank, Adamic Adar, etc (Cai et al., 2018).

1.2 Community detection

A problem closely related to node embedding is community detection. A community is loosely defined as a set of nodes of the graph which are strongly connected to each other but weakly connected to the rest of the graph, though there is no universally-

accepted formal definition of a community (Fortunato & Hric, 2016). If nodes of a graph correspond to entities in the real world — for example, the graph could be a scientific collaboration network, where nodes are scientists and edges represent paper coauthoring —, it is expected that, if a group of nodes is a community, then the corresponding real-world entities probably share similar properties or play similar roles. As an example, Figure 1 shows Zachary's Karate Club graph (Zachary, 1977), which is known to have 2 communities.

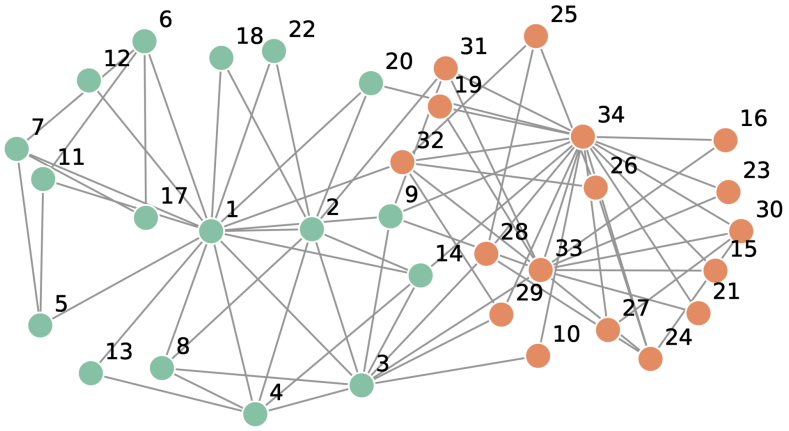


Figure 1: The Karate Club graph, a famous benchmark in the community detection literature, exhibits community structure and has two ground-truth communities. Source: wikipedia.org

There are several different scores one can define to measure how "community-like" a group of nodes is. Let us name a few. Let G be a directed graph with n nodes and m edges and let C be a subgraph of G with n_c nodes. An edge (u, v) of G is called *internal* to C if both u and v are in C; it is called a *boundary edge* if only one of u or v is in C. Let m_c be the number of internal edges of C and b_c be the number of boundary edges of C. Then, we can define the *intra-cluster density* of C, $\delta_{\text{int}}(C)$, as the ratio between the number of internal edges of C and the maximum possible number of internal edges, i.e.

$$\delta_{\text{int}}(C) = \frac{m_c}{n_c(n_c - 1)} \tag{1}$$

If C is a community, we expect $\delta_{\text{int}}(C)$ to be considerably larger than $\delta(G) = \delta_{\text{int}}(G)$, the *link density* of G. Similarly, the *inter-cluster edge density* of C, $\delta_{\text{ext}}(C)$, also called *cut ratio*, is the ratio between the number of boundary edges of C and the maximum possible number of boundary edges:

$$\delta_{\text{ext}}(C) = \frac{b_c}{2n_c(n - n_c)} \tag{2}$$

If C is a community, we expect $\delta_{\text{ext}}(C)$ to be considerably smaller than $\delta(G)$.

The *conductance* of C is the proportion of boundary edges of C relative to all edges incident on C:

$$\text{conductance}(C) = \frac{b_c}{m_c + b_c} \tag{3}$$

The smaller this value is, the more "community-like" C is.

Once a community detection algorithm has generated a partition of the nodes defining the sets of communities, it is also possible to compare how similar this partition is to a *ground-truth* community partition (i.e. a community assignment obtained from a source other than the graph topology alone, such as additional information or labels from the context where the data were collected); or one could compare the community partitions outputted by two different algorithms. This can be done with the Normalized Mutual Information (NMI), which ranges from 0 to 1 and indicates how similar two partitions are (Ana & Jain, 2003).

Several more such "community quality" measures can be defined; see Fortunato (2010) and Fortunato & Hric (2016).

1.3 The interplay between node embedding and community detection

Comparing the characterization of node similarity from Section 1.1 and the measures of "community-likeness" of Section 1.2, we see a node is expected to be much more similar to other nodes of its community than to nodes of other communities. Indeed, any node embedding algorithm that seeks to preserve node similarity can be used as a subroutine of a community detection algorithm: one simply obtains the embeddings of each node and then clusters the embeddings using an algorithm such as K-means in order to derive the communities. On the other way around, detecting communities can also help in obtaining better node embeddings; see Cavallari et al. (2017) for an example.

Thus, it is reasonable to conjecture that the performance of a node embedding algorithm on downstream machine learning tasks such as node classification and link prediction is directly related to the algorithm's performance as a community detector. Our goal in this paper will then be to propose a node embedding algorithm called Tracer Propagation and first evaluate and tune its performance on the task of community detection. Moreover, the algorithm will generate highly interpretable node embeddings: because the embeddings will *not* be automatically learned as the solution to an optimization problem, we will be able to interpret each component of the embedding vector of a node as a measure of similarity to some other node of the graph. Thanks to this interpretability, the embedding vectors generated by Tracer Propagation can also be used to define node similarity functions for consumption by standard, optimization-based node embedding algorithms, which, we conjecture, could lead these algorithms to perform better than if they used second-order proximity as defined in Section 1.1. In this paper, we developed the Tracer Propagation algorithm and evaluated its performance as a community detector (when used as a subroutine as explained in the previous paragraph), and we leave it to

future work to test the two aforementioned conjectures.

Tracer Propagation is inspired on a popular and effective community detection algorithm called Label Propagation (Cordasco & Gargano, 2011; Raghavan et al., 2007). Label Propagation works by first assigning each node with a unique label. Then, on each subsequent iteration, the label of a node is updated to the label that is most common among its neighbors; ties are broken randomly. On convergence, nodes with the same label are assigned the same community. This way, discrete values — the labels — propagate across the network and determine communities. In order to convert this idea into an algorithm that generates vectorial node embeddings, we will replace labels with continuous values, which propagate by traversing edges and decaying on each traversal. Details are explained in the next section.

2. Tracer Propagation

2.1 Intuition

Our goal is to convert the main idea behind the Label Propagation algorithm into the continuous setting, so that a community detection algorithm may become a node embedding algorithm. We begin by assigning a real value, such as 1, to a source node. This value will propagate through the network in an iterative fashion: at each iteration, the value at any given node is transmitted to its neighboring nodes, decaying as it traverses an edge, and respecting edge directions (we regard a simple edge as two directed edges for this matter). The weight of an edge represents how strongly the value is transmitted through the edge: the higher the weight, the less the value will decay on its traversal (we assume that edges of an unweighted graph all have weight 1). The more edges the value traverses, the more it will decay, which should cause the algorithm to converge. Then, we expect, nodes that are topologically closer and more similar to the source — from whence the value originally stemmed — should end up with a higher magnitude of the value than nodes farther from the source. Repeating this process with r different sources sampled from the graph, each node will end up associated with an r-dimensional vector, representing the magnitude of the value of each source that managed to reach the respective node. This should already be a good beginning towards creating an embedding for each node.

This procedure is analogous to the technique of *isotopic tracing* in chemistry, also called *isotopic labeling* (The Editors of Encyclopædia Britannica, 2018). In this technique, an atom of a chemical element (the *reactant*) is replaced with one of its isotopes. The isotope contains a different neutron count from the original atom, which makes it detectable by using appropriate measuring instruments; however, it still behaves in mostly the same way as the original atom. In this manner, as the altered reactant undergoes a chemical reaction, its pathway and distribution throughout the whole reaction can be detected, which yields much insight about the workings of the phenomenon and the role of the reactant in it. In sum, with this technique, one is able to follow and measure the trace

of a reactant as it participates in a chemical process, thanks to the artificially-introduced isotope working as a *tracer* of the reactant.

In our algorithm, the value first assigned to a source node also works as a *tracer*: as it propagates through the network, we will be able to detect the "influence" of the source over each other node, as measured by the magnitude of the tracer that reached each node. This should give us an idea of the *global* and *long-term* influence the source exerts over other nodes, beyond the *local* and *short-term* influence information contained in the adjacency matrix, since the tracer propagates over time and can reach nodes beyond the immediate neighborhood of the source. Intuitively, we can think of influence as an "asymmetric similarity" measure. Let us put the idea into symbols.

2.2 The long-term influence matrix

If $M \in \mathbb{R}^{m \times n}$ is a matrix, we use the notation $M[i, j]$ to refer to the number of M at line i and column j; $M[i, :]$ to refer to the i-th line of M, a row vector; and $M[:, j]$ to refer to the j-th column of M, a column vector. This is MATLAB notation. If all $1 \leq a_i \leq n$, then $M[:, [a_1, a_2, \ldots, a_r]]$ is the matrix obtained by concatenating the columns $a_1, a_2, \ldots, a_r$ of M side-by-side, in this order. Let G be a weighted directed graph with nonnegative weights (which is required for the above rationale to make sense), with loops allowed, but no multiple edges. Let $A \in \mathbb{R}^{n \times n}$ be the adjacency matrix of G; we call $W = A^T$ the *weight matrix* of G. If there is no edge from u to v in G, then $A[u, v] = W[v, u] = 0$. Suppose G has n nodes numbered 1 to n, and let $s_1 < s_2 < \ldots < s_r$ be r chosen source nodes from the graph. Let $B[s_j, j]$ be the tracer value supplied to source s_j, with $B \in \mathbb{R}^{n \times r}$ (for now, we assume $B[u, j] = 0$ if $u \neq s_j$, that is, no tracer is supplied to non-source nodes). Finally, let $T_k[u, j]$ be the amount of tracer that stemmed from the source node s_j and reached node u at iteration k of the algorithm. As we will see, the value of T_0 is arbitrary and can be set to 0. The *tracer propagation rule* is then

$$T_{k+1}[u, j] = \left(\sum_{v=1}^{n} W[u, v] \cdot T_k[v, j] \right) + B[u, j] \tag{4}$$

That is, each node receives from its incoming neighbors the weighted sum of their previous tracer values, which causes tracers to propagate through the network. Additionally, to compensate for the fact that tracers decay as they propagate, each source node s_j has its tracer replenished by the supply value $B[s_j, j]$ at each iteration. By requiring that tracer values decrease as they traverse an edge, if we didn't provide supply values at each iteration, the final state of the system could end up being the 0 matrix. B has a function similar to the bias in a neural network, hence the letter "B".

The propagation rule of Equation 4 can be readily cast to a matricial equation:

$$T_{k+1} = WT_k + B \tag{5}$$

This can be thought of as the "affine version" of a stochastic system with transition equation $v_{k+1} = Pv_k$, where P is a stochastic matrix, also called a transition matrix. In our case, it is easy to verify that

$$T_k = W^{k-1}T_0 + \left(\sum_{i=0}^{k-1} W^i\right) B \tag{6}$$

The summation in Equation 6 converges iff (if, and only if) $|\lambda_i| < 1$ for all complex eigenvalues λ_i of W, which is equivalent to $\rho(W) < 1$, where $\rho(W)$ is the *spectral radius* of W, that is, the highest magnitude of an eigenvalue of W (Kani et al., 2021). In this case, as k tends to infinity, W^{k-1} tends to 0 and $\left(\sum_{i=0}^{k-1} W^i\right)$ tends to $(I - W)^{-1}$, and we have

$$T = T_\infty = (I - W)^{-1}B \tag{7}$$

which doesn't depend on the initial state T_0 (as in the case of a stochastic system). We also note that, as expected, T is a stationary state of the system supplied by B:

$$\begin{aligned} WT + B &= W\left(\lim_{k\to\infty}\sum_{i=0}^{k-1} W^i\right)B + B \\ &= \left(\lim_{k\to\infty}\sum_{i=1}^{k} W^i + I\right)B \\ &= \left(\lim_{k\to\infty}\sum_{i=0}^{k} W^i\right)B \\ &= (I - W)^{-1}B \\ &= T \end{aligned}$$

That is, if we propagate the tracer values of T and then replenish source tracer values with B, the resulting tracer values will remain unchanged.

As per our previous discussion, we call

$$L = (I - W)^{-1} \tag{8}$$

the *long-term influence matrix* of W. As already noted, if $T = LB$, then T is a stationary state supplied by B, that is, $T = WT + B$. This is true for any matrix B, which means that the set of stationary states of L is the (matricial) image of L. However, because L is invertible, any matrix T is a stationary state of L, supplied by $L^{-1}T$. Indeed,

$WT + L^{-1}T = (W + L^{-1})T = (W + I - W)T = T$. In particular, L is a stationary state of L, supplied by $L^{-1}L = I$, that is, $L = WL + I$. Because W has nonnegative entries by assumption, L has as well (since $L = \sum_{i=0}^{\infty} W^i$). This implies that WL has nonnegative entries, and so the diagonal entries of $L = WL + I$ are all ≥ 1. We have experimentally observed that this frequently causes the diagonal entries of L to be quite far from their neighboring values, which is detrimental to an embedding; for this reason, we redefine

$$L = (I - W)^{-1} - I \tag{9}$$

and the above discussion proves that L redefined this way continues to have nonnegative entries.

So far we have skipped through the quintessential assumption that the eigenvalues of W all have magnitude less than 1, which does not hold in general for the weight matrix of an arbitrary graph. To tackle this, we need to first *squash* the eigenvalues of W so that their magnitudes are bounded as desired. For this, we need to use a *matricial squash function* S such that the eigenvalues of $S(W)$ all have magnitude less than 1 for any real matrix W with nonnegative entries. Perhaps the first such function that comes to mind is

$$S(W) = \frac{1}{(a \cdot \rho(W) + b)} W, \tag{10}$$

where a and b are hyper-parameters of S and $a \geq 1$, $b \geq 0$, and $a > 1$ or $b > 0$. In this case, $\rho(S(W)) = \rho(W)/(a \cdot \rho(W) + b) < 1$, as desired. We call the function of Equation 10 the *linear squash function*.

If the eigenvalues of W are all real (e.g. W is symmetric), we can also use the following *logistic squash function*:

$$S(W) = 2b \left(\left(I + e^{-aW} \right)^{-1} - \frac{1}{2} I \right) \tag{11}$$

with $0 < b \leq 1$ and $a > 0$. This is the same as the sigmoid activation function sometimes used in neural networks (Aggarwal, 2018), except it is applied to matrices using the matrix exponential (which is *not* the same as applying the exponential function component-wise) (Hall, 2015). As long as the eigenvalues of W are all real, the eigenvalues of $S(W)$ are guaranteed to have magnitude less than b. As we will see, this non-linearity usually makes it a more effective function than the linear squash function.

The long-term influence matrix is then redefined (again) as

$$L = (I - S(W))^{-1} - I, \tag{12}$$

for some appropriate choice of squash function S. Because W is component-wise non-negative by assumption, it is easy to see that, if S is the linear squash function, then L as defined in Equation 12 is guaranteed to be component-wise nonnegative, which is required for the interpretation of $L[u, v]$ as the long-term influence exerted on u by v. If S is any non-linear squash function, we can still analyze S as if it were linear as a way of simplifying the model with an approximation. However, when S is non-linear, then L might in fact end up with some negative "artifacts", which are negative cells with a small magnitude (as we have verified experimentally). To work around this, we replace negative entries on L with 0, which amounts to component-wise applying ReLU to L: $L[i, j] := \max(L[i, j], 0)$.

There is only one last problem with this definition, which is the scale of L: because of squashing and inverting, the eigenvalues, and thus the entries, of L might not be comparable to (of the same order of magnitude as) those of W. To fix this, we simply multiply L by a value which will make $\rho(L) = \rho(W)$[1]:

$$\begin{aligned} L_0 &= (I - S(W))^{-1} - I \\ L &= \text{ReLU}\left(\frac{\rho(W)}{\rho(L_0)} \cdot L_0\right) \end{aligned} \tag{13}$$

Notice that $\rho(L_0) = (1-\rho(S(W)))^{-1} - 1$, whereas it is usually straightforward to compute $\rho(S(W))$ given that $\rho(W)$ is known.

Let us illustrate the mapping from W to L with a numerical example. Figure 2 presents visualizations of the weight matrix W and of the long-term influence matrix L of the unweighted Zachary's Karate Club graph (Zachary, 1977), a symmetric graph with 34 nodes and 78 edges. The color intensity represents the magnitude of the value in a cell, from 0 (white) to 1 (completely green), though some values in W slightly exceed 1. Both images use the same color scale, which is possible due to the rescaling of L that happens in Equation 13, which equates the scales of W and L. A drawing of the graph can be seen in Figure 1, and its node embedding, as generated by our algorithm, is plotted in Figure 3.

Nodes 0 and 33 are the most prominent in the graph. The graph captures members of a karate club, and edges represent members who interacted with each other outside the club. Due to a conflict that occurred between the administrator "John A" (node 0) and the instructor "Mr. Hi" (node 33), the club split into two groups, which can be detected as two communities that revolve around these two nodes. This can also be visualized in the plot of Figure 3. In the matrix visualization of Figure 2, this can be noticed from the fact that the diagonal values $L[0, 0]$ and $L[33, 33]$, which we call "self-influences", are the greatest in the matrix, as a result of nodes 0 and 33 having the most edges and thus

[1] At least until we apply ReLU.

influencing the network the most.

2.3 Choosing sources — the Principal Component Selection algorithm

Now that we have the long-term influence matrix L, how do we extract the node embeddings from it? Following the discussion in Section 2.1, the embedding of node u is simply the vector that entails the tracer amount of each source node $s_1 < s_2 < \ldots < s_r$ that reached u, that is, it's the vector $[L[u, s_1], L[u, s_2], \ldots, L[u, s_r]]$, which is merely a subsequence of $L[u, :]$. So it only remains to determine (1) how many sources to choose (i.e. determine the embedding dimension r) and (2) which sources to choose.

For one thing, we could simply choose r to be a desired dimensionality number, such as 128, and then randomly choose r different nodes from the graph. However, if we can afford it, we can do better. Our strategy will be to choose r nodes that are very prominent but, at the same time, very unrelated, so that our node embeddings are made up of components that explain well all nodes but are also not redundant components. For example, nodes 0 and 33 of the discussion above on the Karate graph fit well our requirements. The Principal Component Analysis (PCA) algorithm does something similar: out of a dataset, it builds up new components that explain the most variance while being the most uncorrelated to each other, as per Pearson's correlation coefficient (Jolliffe & Cadima, 2016). The only caveat in our problem is that we strictly need the new components we will use to be a *subset* of the components we already have, lest we might prematurely lose the influence interpretability of the component values. This then gives rise to the following Principal Component Selection (PCS) algorithm. Given a correlation threshold τ, the algorithm chooses a set of r components from the dataset such that all components are correlated to each other by less than τ and such that they have the most variance possible.

In Algorithm 1, $M \in \mathbb{R}^{m \times n}$ is a data matrix whose rows are observations and whose columns are components. It returns the indexes of r principal components $p_1 < p_2 < \ldots < p_r$ where each $1 \leq p_i \leq n$. Indexes start at 1.

Notice the algorithm only takes as input the dataset and the correlation threshold τ, but not the number r of sources, which is automatically returned as a result of including or excluding components. By choosing τ closer to 1, r will be greater; by choosing τ closer to 0, r will be smaller. Also note that we have chosen to include all components that are negatively correlated; if desired, this can be changed by replacing `corr[c,p]` in Line 8 of Algorithm 1 with `|corr[c,p]|`.

So, our strategy will be to first compute the long-term influence matrix L, and then run PCS on L with a sensible correlation threshold τ in order to obtain r source nodes (where the number r is also returned by PCS).

Algorithm 1 Principal Component Selection ($M \in \mathbb{R}^{m \times n}, 0 \leq \tau \leq 1$)

```
vars := variances(M)
sorted_by_var := argsort(vars, "decreasing")
principal_comps := [sorted_by_var[1]]
for i in [2..n] do
    c := sorted_by_var[i]
    include := true
    for p in principal_comps do
        if corr[c,p] > τ then
            include := false
            break
        end if
    end for
    if include then
        principal_comps.push(c)
    end if
end for
principal_comps := sort(principal_comps, "increasing")
return principal_comps
```

2.4 Influence back-propagation

Even after obtaining the r sources using PCS, there is still a problem with L that prevents it from performing well as an embedding matrix, most notably in medium-sized and large graphs, which have greater diameters (i.e. greater longest shortest paths). This has to do with the distribution of influence values in L (remember that $L[u, v]$ is interpreted as the extent by which v influences u in the long run). To see this, let us turn our attention back to the matrix visualizations of Figure 2. We can see that, in general, if $W[u, v] = 1$, then $L[u, v]$ will have a moderate magnitude, while if $W[u, v] = 0$, then $L[u, v]$ will be quite close to 0, with few exceptions. That is, $L[u, v]$ is of reasonable magnitude if there is an edge from v to u, but is very small otherwise, even in cases were the length of the shortest path from v to u is 2. This is contrary to our initial design intention, where we intended for $L[u, v]$ to approach 0 slowly as u "moves away" from v, rather than vanishing as soon as there is no edge from v to u.

This *premature vanishing* happens because of the need to squash W before being able to propagate tracer values, so that the propagation rule of Equation 4 converges. The spectral radius of Karate Club graph's weight matrix, presented in Figure 2, is about 6.7. Using the linear squash function with parameters $a = 1$ and $b = 3$ in Equation 10, we will find that the non-zero entries of $S(W)$ are all $\frac{1}{6.7+3} \approx 0.1$, which are effectively the edge weights that will be used for tracer propagation in Equation 4. This results in tracer propagation not being able to reach much far from the source node before vanishing, as 0.1^2 is already very small. Bringing b closer to 0 in Equation 10 (of the linear squash function) would not help a lot, and, besides, making b too close to 0 could result in overflow and numerical stability problems. As a consequence, L can only be a reliable

source of long-term *local* (neighborhood-level) similarity, since even slightly more distant nodes will usually be regarded as barely influencing each other. If we are to obtain a good-quality embedding matrix, this issue must be solved, which means that tracer values must be more permissive with respect to the topological distance between nodes. We must somehow find a way of propagating tracers with higher edge weights, while still guaranteeing convergence.

Our proposed solution is to perform a single more tracer propagation pass on top of the values obtained from the long-term influence matrix, as follows. Suppose we have at our disposal a rough — and it can be *very* rough — measure of dissimilarity, or topological distance, between nodes of the graph, which we call the *separation* from a node v to a node u, $\texttt{sep}[u, v]$, in order to differentiate it from the well-established concept of node distance. For example, the separation could be simply the unweighted shortest path from v to u as obtained by BFS (breadth-first search), or it could use a more sophisticated technique. The only requirements are that $\texttt{sep}[u, u] = 0$ for all u, $\texttt{sep}[u, v] \geq 0$ for all u, v, and $\texttt{sep}[u, v] = \infty$ if there is no path from v to u. We then define the *linkage* from node v to node u as follows:

$$\texttt{linkage}[u, v] = \left(\frac{1}{\texttt{sep}[u, v] + 1} \right)^{\alpha} \tag{14}$$

Where $\alpha > 0$ is a parameter of the model, optionally used to strengthen ($0 < \alpha < 1$) or weaken ($\alpha > 1$) linkages. Note that $0 \leq \texttt{linkage}[u, v] \leq 1$ and that $\texttt{linkage}[u, u] = 1$ for all u, v. Linkages are used to compensate for the premature vanishing problem we discussed earlier, and $\texttt{linkage}[u, v]$ should, in general, be much greater than $L[u, v]$, most notably when (v, u) is not an edge.

The procedure is then the following. If the long-term influence from v to u is $L[u, v]$, we will define the *linked influence* from v to u, $F[u, v]$, to be

$$F[u, v] = \sum_{w=1}^{n} L[u, w] \cdot \texttt{linkage}[w, v] \tag{15}$$

That is, $F[u, v]$ is $L[u, v]$ plus the sum of the influences from all nodes $w \neq v$ to u, weighted by $\texttt{linkage}[w, v]$. This way, given $\texttt{linkage}[w, v]$ (which can be thought of as a kind of weight from v to w), we are propagating $L[u, w]$ (the influence that w exerts on u) — *back* to v, using the intermediate $\texttt{linkage}[w, v]$, in order to build up the final linked influence from v to u, $F[u, v]$. Clearly from Equation 15,

$$F = L \cdot \texttt{linkage} \tag{16}$$

As a rule of thumb, the bigger and sparser the graph is, the smaller should α be, so that influences can back-propagate farther. Finally, we use $F[u, [s_1, \ldots, s_r]]$ as the node embedding of u. See Algorithm 2. We denote by $\mathbb{R}^{n \times n}_{\rho < 1}$ the set of real matrices with

spectral radius less than 1 (which also takes into account complex eigenvalues).

Algorithm 2 Tracer Propagation
$(A \in \mathbb{R}^{n\times n},\ S : \mathbb{R}^{n\times n} \longrightarrow \mathbb{R}^{n\times n}_{\rho<1},\ 0 \leq \texttt{τ} \leq 1,\ \texttt{sep} \in \mathbb{R}^{n\times n}_{+},\ \alpha > 0)$

$W := A^T$
$W_S := S(W)$
$L_0 := (I - W_S)^{-1} - I$
$L := \texttt{ReLU}\left(\frac{\rho(W)}{\rho(L_0)} L_0\right)$
$[s_1, s_2, \ldots, s_r] := \texttt{PCS}(L, \texttt{τ})$ ▷ Principal Component Selection
$\texttt{linkage} := \left(\frac{1}{\texttt{sep}+1}\right)^{\alpha}$ ▷ Component-wise operation
$F := L \cdot \texttt{linkage}$
return $F[:, [s_1, s_2, \ldots, s_r]]$

2.5 A word on efficiency

Algorithms that unavoidably rely on the use of the adjacency matrix of a graph are not practical to use on medium-sized and large sparse graphs due to their $O(n^2)$ or greater memory and time requirements, where n is the number of nodes. It should be noted, however, that, although we have been making use of the adjacency matrix in our theoretical explanations, constructing it is not strictly needed in implementations. If computing the squash function on the entries of the weight matrix does not rely on knowing all of its eigenvalues, but only on knowing its spectral radius (as is the case of the linear squash function), then selected entries of the long-term influence matrix of Equation 13 can be computed iteratively with Equation 4 by using the adjacency list of the graph and an estimate of the spectral radius that doesn't require one to construct the weight matrix (see, for example, Guo et al. (2019)). Furthermore, we can restrict ourselves to only computing the long-term influence from v to u if u is within the 2-hop set of v (i.e. if the shortest unweighted distance from v to u is at most 2). After all, if u is beyond the 2-hop, we know that $L[u, v]$ will be very close to 0 anyway. Similar approximations and iterative alternatives can be used on the remaining steps of Algorithm 2 so that we never actually have to build an $n \times n$ matrix.

3. Experiments

The common practice in the node embedding literature is to evaluate the embedding generated by the proposed algorithm by assessing its capacity to support downstream machine learning tasks on graphs, such as node classification and link prediction (Abu-El-Haija et al., 2018; Cavallari et al., 2017; Chen et al., 2020; Grover & Leskovec, 2016; Perozzi et al., 2014; Tang et al., 2015). However, as explained in Section 1.3, rather than using the embeddings to tackle further tasks, in this work we will be mainly preoccupied with how well Tracer Propagation is able to preserve the topological features of the graph

once its nodes get embedded into Euclidean space. This will be done by comparing the numerical clustering of the generated data points against the community detection of the graph. If the numerical clustering of the data points finds good communities when standard community detection algorithms do, this means our algorithm was successfully able to change the representation of nodes while maintaining their structural properties, which should already be an indicative of its future performance when used on downstream machine learning tasks. We delay the evaluation of the algorithm on such tasks to future work.

We ran our algorithm on the benchmarks given in Table 1 and then ran K-means on the resulting embeddings in order to obtain communities. In all cases, the number of clusters was set to the number of communities detected by Label Propagation. We then evaluated the quality of the detected communities using the scores explained in Section 1.2: internal edge density (Equation 1), external edge density (Equation 2), and conductance (Equation 3). We compared the results against the communities detected Label Propagation (Cordasco & Gargano, 2011) and Leiden (Traag et al., 2019). In particular, we used NMI to evaluate how close the communities detected by Tracer Propagation were to those detected by the other two algorithms. To find the best parameters of our algorithm on each benchmark, we performed multi-objective hyper-parameter optimization using Optuna (Akiba et al., 2019) with the same scores as used in evaluation. The results of the best parameters, as well as the parameters found, are reported in Table 2. The separation measure was not included in the hyper-parameter search and was set to shortest distance in all cases. Plots obtained by applying tSNE on the generated embeddings can also be visualized in Figure 3.

We see that our algorithm performs best or comparatively on small graphs, but its relative performance decreases as graphs get bigger. This is reinforced by the plots of the embeddings depicted in Figure 3. On all 5 small graphs, the embeddings have clearly been able to capture the underlying community structure and topological similarity of the nodes, and the cohesion is very strong between the clusters observed in the plots and the communities detected by Leiden or Label Propagation. However, this is clearly not the case on the bigger Power network, which almost seems like plotting nodes randomly around a center.

The hyper-parameter search also yields insights into the algorithm. Surprisingly, the best squash function on all benchmarks was the logistic function, which shows the importance of introducing non-linearity into the model. We can also see that the two graphs with the most pronounced community structure — Karate and Football — are also the ones with the smallest correlation threshold τ. We explain this by pointing out that, when a graph has strong community structure, the characteristics of a large portion of the nodes can be explained by their relation to a small portion of the nodes — the dominant nodes of each community. Additionally, we see from the plots of Figure 3 that the execution of PCS automatically detects which nodes are dominant in each community, as each com-

Table 1: Graph benchmarks used to carry out experiments with Tracer Propagation. All graphs are undirected and unweighted.

Name	Nodes	Edges	Description	Source
Karate	34	78	Social network of friendships between 34 members of a karate club at a US university in the 1970s.	Zachary (1977)
Dolphins	62	159	Social network of frequent associations between 62 dolphins in a community living off Doubtful Sound, New Zealand.	Lusseau et al. (2003)
Miserables	77	254	Coappearance network of characters in the novel Les Miserables.	Knuth (1993)
Football	115	613	Network of American football games between Division IA colleges during regular season Fall 2000.	Girvan & Newman (2002)
Netscience	379	914	Coauthorship network of scientists working on network theory and experiment.	Newman (2006)
Power	4941	6594	A network representing the topology of the Western States Power Grid of the United States.	Watts & Strogatz (1998)

munity always has a few nodes that were included as principal components. Finally, as expected, we see that larger and sparser graphs indeed require smaller linkage exponents, so that influences are able to back-propagate across more edges before vanishing — see the discussion on linkages of Section 2.4.

4. Conclusion

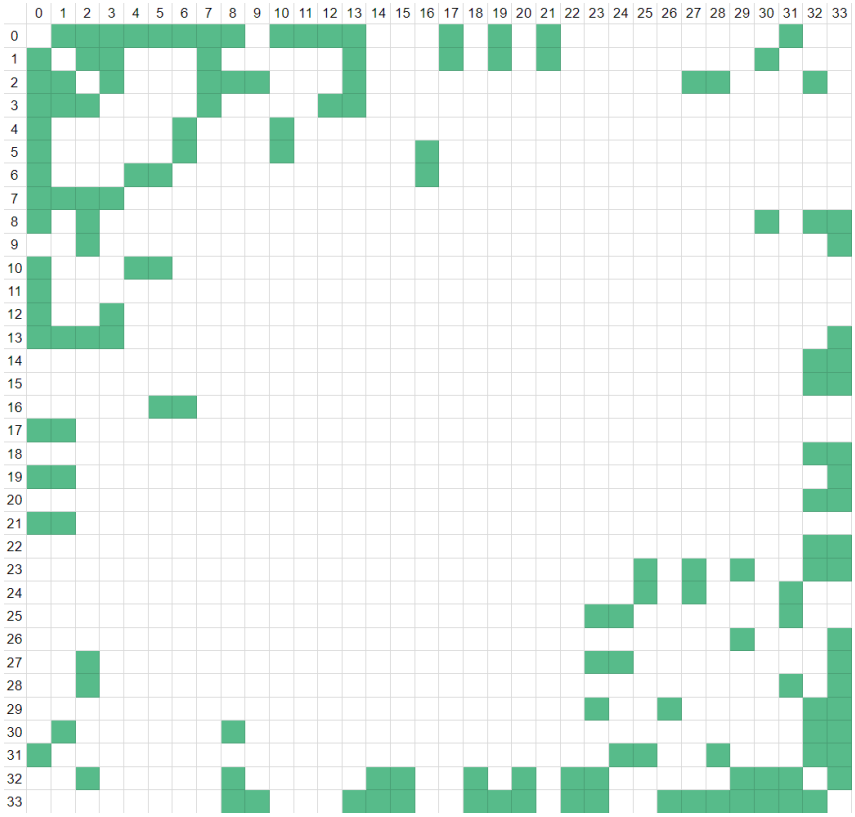
In this paper, we argued in favor of the connection between the problems of node embedding and community detection and introduced a novel node embedding algorithm called Tracer Propagation, aimed at preserving the topological features of graph nodes when they are embedded into Euclidean space. Instead of focusing solely on downstream machine learning tasks, as is common practice in the node embedding literature, our primary concern was to assess how effectively Tracer Propagation maintains the structural and community properties of graphs. We utilized K-means on the resulting embeddings to obtain communities and compared them against communities detected by two standard community detection algorithms: Label Propagation and Leiden. Additionally, we conducted multi-objective hyper-parameter optimization to find the best parameters for our algorithm on each benchmark.

Our experimental results revealed that Tracer Propagation performs comparatively to the state of start of community embedding algorithms on small graphs, where it effectively captures the underlying community structure and topological similarity of nodes.

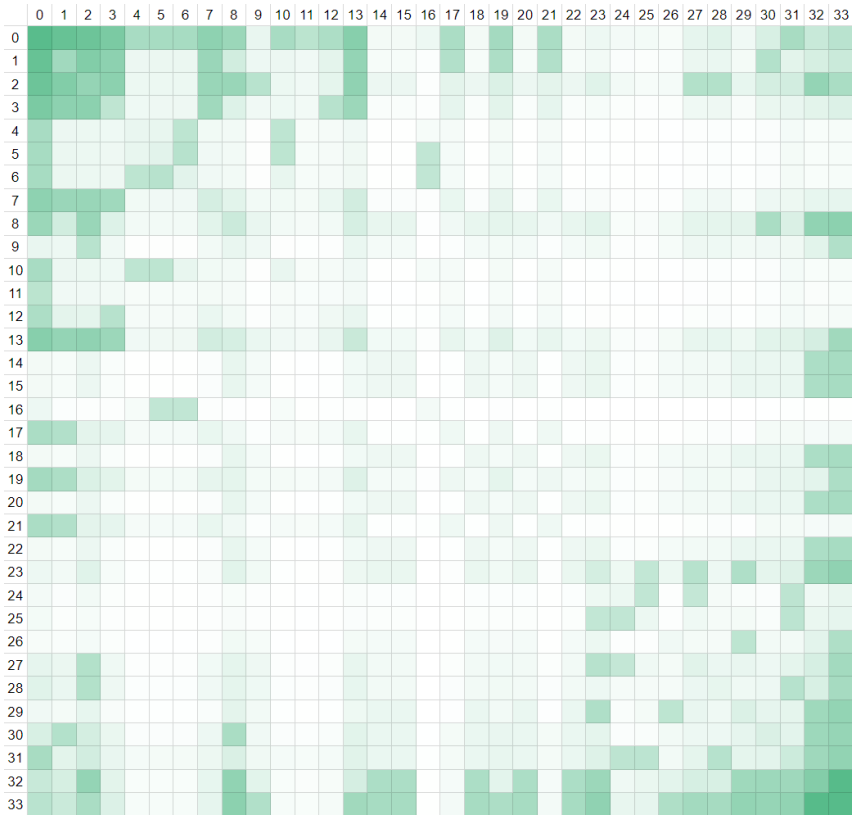
However, its relative performance diminishes as graphs become larger and sparser. Small graphs are usually easy to deal with and many community detection algorithms perform well on them while not yielding good results on larger graphs, so that medium-sized and large graphs are the most important to consider. We then conclude that the algorithm still requires improvement before it can be used in practice to support downstream machine learning problems. However, its great performance on the small graphs is a sign of a step in the right direction. In particular, we attribute the poor performance of Tracer Propagation on the Power network, in part, to the simplistic algorithm we used to derive separation values, which are merely the shortest distance between nodes. As pointed out, good linkage values are essential when one is dealing with larger networks, so that more sophisticated separation measurement algorithms might result in better performance. In this matter, we point out to the use of algorithms capable of detecting some form long-ranging node similarity, such as random walks, maximum flow, number of independent paths between two nodes, among others. We believe this to be a promising topic of further research.

Because Tracer Propagation is not an optimization-based node embedding algorithm, its components are highly interpretable: the j-th component of the embedding of node i can be interpreted as the influence of node j over node i. Thanks to this, rather than directly being used to support downstream machine learning tasks, the component values could be used to define node similarity measures to be subsequently fed to an optimization-based node embedding algorithm capable of generating denser embeddings. We conjecture that this approach might lead to better results than using the standard node similarity measures currently used in the literature, a hypothesis that could be tested in future work.

In the process of designing our algorithm, we also developed Principal Component Selection (PCS), a simple dimensionality reduction algorithm with the same intent as Principal Component Analysis (PCA). However, instead of creating new components, PCS is restricted to selecting components already present in the dataset. This preservation promotes interpretability while also detecting dominant components and reducing the redundancy of the dataset. We anticipate that further research could investigate the effectiveness and use cases of PCS in a variety of data analysis tasks, not necessarily related to graph analysis.

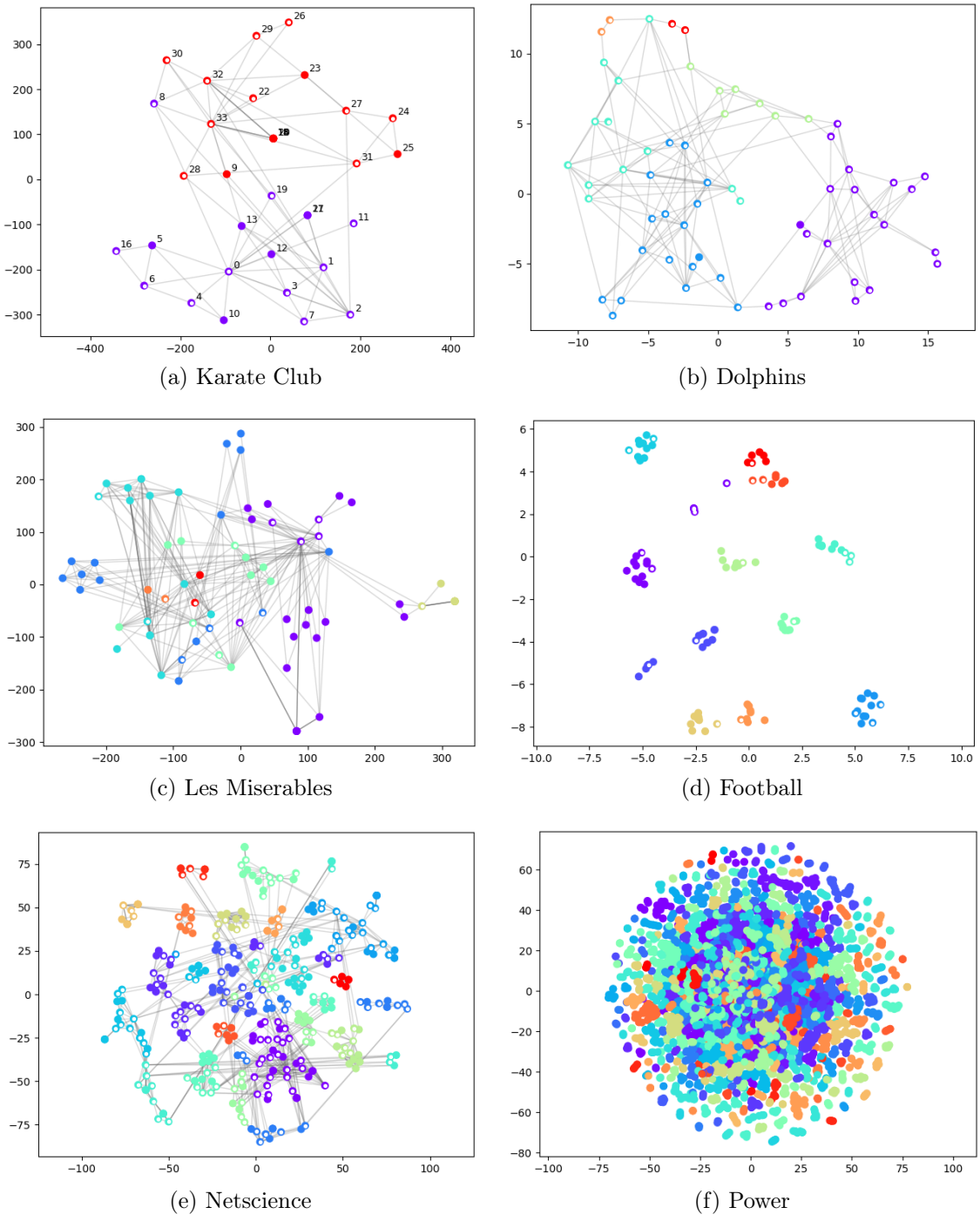


(a) Weight matrix



(b) Long-term influence matrix

Figure 2: The weight matrix and the long-term influence matrix of Zachary's Karate Club. The color intensity represents the magnitude of the value in a cell, from 0 (white) to 1 (completely green). The linear squash function (Equation 10) was used, with $a = 1$ and $b = 3$.



(a) Karate Club (b) Dolphins

(c) Les Miserables (d) Football

(e) Netscience (f) Power

Figure 3: Plots of the 6 graphs described in Table 1 using the node embeddings generated by Tracer Propagation and reduced in dimensionality by tSNE. Nodes with the same colors belong to the same community, as provided by ground-truth data (a), detected by Label Propagation (b, c, d), or detected by Leiden (e, f). Painting nodes with communities derived from other sources provides a means of comparing the results of Tracer Propagation with that of the other sources. Nodes with a white circle in the center correspond to principal components as detected by Principal Component Selection (Algorithm 1). The parameters used on each graph are given in Table 2

.

Table 2: Results of applying Leiden, Label Propagation (LP) and Tracer Propagation (TP) on the benchmarks given in Table 1. "cond." stands for "conductance".

KARATE	**# coms.**	**cond.**	δ_{ext}	δ_{int}	**NMI Leiden**	**NMI LP**	**NMI TP**
Leiden	4	0.288	0.049	0.451	1	0.542	0.637
Label Prop.	3	0.281	0.052	0.502	0.542	1	0.747
Tracer Prop.	3	**0.254**	**0.044**	**0.517**	0.637	0.747	1

squash = logistic | $a = 2.214$ | $b = 0.257$ | $\tau = 0.436$ | $\alpha = 6.611$

DOLPHINS	**# coms.**	**cond.**	δ_{ext}	δ_{int}	**NMI Leiden**	**NMI LP**	**NMI TP**
Leiden	5	**0.3**	0.03	0.362	1	0.738	0.687
Label Prop.	6	0.353	**0.026**	**0.567**	0.738	1	0.789
Tracer Prop.	6	0.323	0.029	0.378	0.687	0.789	1

squash = logistic | $a = 4.883$ | $b = 0.387$ | $\tau = 0.846$ | $\alpha = 4.302$

MISERABLES	**# coms.**	**cond.**	δ_{ext}	δ_{int}	**NMI Leiden**	**NMI LP**	**NMI TP**
Leiden	6	**0.237**	0.024	0.415	1	0.691	0.716
Label Prop.	7	0.283	**0.021**	**0.582**	0.691	1	0.751
Tracer Prop.	7	0.544	0.043	0.423	0.716	0.751	1

squash = logistic | $a = 4.783$ | $b = 0.321$ | $\tau = 0.209$ | $\alpha = 4.375$

FOOTBALL	**# coms.**	**cond.**	δ_{ext}	δ_{int}	**NMI Leiden**	**NMI LP**	**NMI TP**
Leiden	10	**0.294**	**0.03**	0.762	1	0.93	0.936
Label Prop.	11	0.334	0.034	**0.813**	0.93	1	0.938
Tracer Prop.	11	0.33	0.033	0.808	0.936	0.938	1

squash = logistic | $a = 0.902$ | $b = 0.951$ | $\tau = 0.847$ | $\alpha = 5.381$

NETSICENCE	**# coms.**	**cond.**	δ_{ext}	δ_{int}	**NMI Leiden**	**NMI LP**	**NMI TP**
Leiden	19	**0.077**	**0.001**	0.346	1	0.824	0.811
Label Prop.	55	0.271	0.003	**0.693**	0.824	1	0.881
Tracer Prop.	55	0.319	0.004	0.631	0.811	0.881	1

squash = logistic | $a = 4.921$ | $b = 0.75$ | $\tau = 0.82$ | $\alpha = 3.834$

POWER	**# coms.**	**cond.**	δ_{ext}	δ_{int}	**NMI Leiden**	**NMI LP**	**NMI TP**
Leiden	39	**0.034**	0	0.026	1	0.67	0.652
Label Prop.	1308	0.412	0	**0.711**	0.67	1	0.888
Tracer Prop.	1308	0.589	0	0.374	0.652	0.888	1

squash = logistic | $a = 3.372$ | $b = 0.283$ | $\tau = 0.853$ | $\alpha = 2.807$

References

Abu-El-Haija, S., Perozzi, B., Al-Rfou, R., & Alemi, A. A. (2018). Watch your step: learning node embeddings via graph attention. *neural information processing systems*, 31, 9180–9190. https://papers.nips.cc/paper/2018/file/8a94ecfa54dcb88a2fa993bfa6388f9e-Paper.pdf

Aggarwal, C. C. (2018, January). *Neural networks and deep learning*. https://doi.org/10.1007/978-3-319-94463-0

Akiba, T., Sano, S., Yanase, T., Ohta, T., & Koyama, M. (2019). OpTUNA: a next-generation Hyperparameter Optimization Framework. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1907.10902

Ana, L., & Jain, A. K. (2003). Robust data clustering. 3. https://doi.org/10.1109/cvpr.2003.1211462

Cai, H., Zheng, V. W., & Chang, K. C.-C. (2018). A comprehensive survey of graph embedding: problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9), 1616–1637. https://doi.org/10.1109/tkde.2018.2807452

Cavallari, S., Zheng, V. W., Cai, H., Chang, K. C.-C., & Cambria, E. (2017). Learning community embedding with community detection and node embedding on graphs. *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 377–386.

Chen, Y., Wu, L., & Zaki, M. J. (2020). Iterative Deep Graph learning for graph neural networks: better and robust node embeddings. *arXiv (Cornell University)*, 33, 19314–19326. https://arxiv.org/pdf/2006.13009

Cordasco, G., & Gargano, L. (2011). Community Detection via Semi-Synchronous Label Propagation Algorithms. *arXiv (Cornell University)*. https://doi.org/10.1504/..045103

Fortunato, S. (2010). Community detection in graphs. *Physics Reports*, 486(3-5), 75–174. https://doi.org/10.1016/j.physrep.2009.11.002

Fortunato, S., & Hric, D. (2016). Community detection in networks: A user guide. *Physics Reports*, 659, 1–44. https://doi.org/10.1016/j.physrep.2016.09.002

Girvan, M., & Newman, M. E. J. (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(12), 7821–7826. https://doi.org/10.1073/pnas.122653799

Grover, A., & Leskovec, J. (2016). node2vec. https://doi.org/10.1145/2939672.2939754

Guo, J.-M., Wang, Z., & Li, X. (2019). Sharp upper bounds of the spectral radius of a graph. *Discrete Mathematics*, 342(9), 2559–2563. https://doi.org/10.1016/j.disc.2019.05.017

Hall, B. C. (2015). *Lie groups, lie algebras, and representations: An elementary introduction* (2nd, Vol. 222).

Jolliffe, I. T., & Cadima, J. (2016). Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A*, 374(2065), 20150202. https://doi.org/10.1098/rsta.2015.0202

Kani, E., Pullman, N. J., & Rice, N. M. (2021). Powers of matrices. In *Algebraic methods* (pp. 311–370). Queen's University.

Knuth, D. E. (1993, November). *The Stanford GraphBase: a platform for combinatorial computing*. http://www.literateprogramming.com/sgbshort.pdf

Lusseau, D., Schneider, K., Boisseau, O., Haase, P. A., Slooten, E., & Dawson, S. M. (2003). The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, 54(4), 396–405. https://doi.org/10.1007/s00265-003-0651-y

Newman, M. E. (2006). Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3), 036104.

Perozzi, B., Al-Rfou, R., & Skiena, S. (2014). Deepwalk: Online learning of social representations. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 701–710.

Raghavan, U. N., Albert, R., & Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3). https://doi.org/10.1103/physreve.76.036106

Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., & Mei, Q. (2015). Line: Large-scale information network embedding. *Proceedings of the 24th international conference on world wide web*, 1067–1077.

The Editors of Encyclopædia Britannica. (2018). Isotopic tracer [Accessed January 27, 2024].

Traag, V., Waltman, L., & Van Eck, N. J. (2019). From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports*, 9(1). https://doi.org/10.1038/s41598-019-41695-z

Watts, D. J., & Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*, 393(6684), 440–442. https://doi.org/10.1038/30918

Xu, M. (2021). Understanding graph embedding methods and their applications. *Siam Review*, 63(4), 825–853. https://doi.org/10.1137/20m1386062

Zachary, W. (1977). An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33(4), 452–473. https://doi.org/10.1086/jar.33.4.3629752